Several companies have recently introduced DSP-oriented microprocessors embodying a computer architecture design style called VLIW (very long instruction word architecture). The VLIW style, which makes processors potentially much faster through the use of ILP (instruction-level parallelism), has accompanied hardware built for DSP for several years. However, recent research has brought a significantly greater understanding of these architectures and their associated software tools, especially VLIW compiler techniques. Now that silicon is dense enough to allow us to build cost-effective single-chip VLIWs for embedded DSP applications, these new techniques are changing the field dramatically. At the same time, the need for DSP compute cycles has exploded, driven primarily by media and communications processing.

This article is about the technologies behind the resurgence of DSP-oriented microprocessors and the techniques that allow us to use them well. After an overview of the VLIW architectural style, we discuss three main areas: VLIW architectural features relevant to DSP applications; their associated compiler techniques; and...
coding techniques that allow the application programmer, while still coding in a high-level language, to best exploit the architecture.

Overview

When designing a DSP VLIW architecture, one is presented with several choices not typically encountered in microprocessors designed for a typical workstation workload. The most important of these deal with the structure of the data-storage parts of the processor. These include registers and their connectivity, dedicated memory elements, as well as the rest of the memory hierarchy. Other important topics include subword parallelism, branch structure, hardware data types, and the effect of these artifacts on code size. As each of these is discussed, the related compiler technology is considered.

One of the most appealing results of the past decade of research into VLIW architectures is that we now can build DSP-oriented chips that are stand-alone microprocessors. Previously, we built inner-loop-oriented engines that acted as slave processors to a CPU able to run the rest of the application and the operating system. Progress in the fields of compiling and architecture has made it possible to write the actual compute-intensive loops as part of the whole application, and to allow the compiler to do the fine-grained scheduling necessary to achieve the available instruction-level parallelism. While the application is written in a high-level language, the programmer can do a lot to enable the compiler to find parallelism. Sometimes these are very small changes, sometimes they involve the use of sophisticated techniques—still far easier than the former paradigm of writing very tightly knit, low-level code to fit the DSP engine and aligning that code with the host processor. The final section of the article is a survey of valuable coding techniques and their effect. The use of these techniques can greatly improve performance on a DSP-oriented VLIW.

DSP-oriented hardware has been built for years, but it’s been hard to use, programmed primarily through libraries and excruciatingly tailored hand code. We are now adding significant ILP into the mix, which makes hand-coding far more difficult still. Additionally, the industry has changed toward a DSP chip-family orientation, making the engineering cost of starting almost anew for each new chip prohibitive. Techniques like those put forward here are required for these families to succeed.

Instruction-Level Parallelism

VLIW, like RISC and CISC, is a processor design philosophy. It is a logical extension of the RISC philosophy, but one geared to adding ILP to a processor. ILP is a set of design techniques that speed up programs by executing in parallel several RISC-style operations, such as memory loads and stores, integer additions, or floating-point multiplications. These operations are taken from a single stream of execution, rather than from parallel tasks. Only at the finest-grained level are these operations assigned to function units—adders and memory units, for example—for execution in parallel. This parallelism is largely invisible to the user, though the sophisticated user may be very aware of its operation and may restructure code or carry out other actions, described later in this article, to enhance ILP.

We distinguish between the latent or "available" ILP inherent in a region of code and the realizable or "achievable" ILP provided by the hardware. Given ILP hardware, a DSP programmer tries to produce code with an instruction mix that matches the CPU's achievable ILP. On the other hand, the CPU designer studies application code to design processor hardware that offers ILP to match the application space.

ILP Hardware

Hardware can offer ILP in several different ways. For example:

1. Several of the functional units found in a processor can execute at the same time,
2. Multiple copies of functional units, possibly accessing different register files to add register bandwidth, can be added for the purpose of executing in parallel, and/or
3. Functional units with latency longer than one cycle can be pipelined.

These all enable the parallelism of the same RISC operations that are executed sequentially in the view of the programmer. For example, consider the integer functional unit, floating functional unit, and registers of a typical microprocessor. As shown in Fig. 1, we might add ILP to this processor by corresponding to the list above:

1. Allowing operations to execute simultaneously on each of the functional units. Having separate register banks for the integer and floating-point data can help us do this by reducing potential hardware resource conflicts (this has other important benefits, as we will see later),
2. Adding to that a second integer unit, so that it can do two integer operations per cycle, and finally,
3. Pipelining the floating point and cache operations so that one can be initiated each cycle, even though each might take several cycles to finish.

On ILP hardware with more ILP than the example above, a fragment of code that might execute sequentially as:

```
cycle 1:  add t3 = t1, t2
cycle 2:  store [addr0] = t3
cycle 3:  fmul f6 = f7, f14
cycle 4:  ...waiting...
cycle 5:  ...waiting...
cycle 6:  fmul f7 = f7, f15
cycle 7:  ...waiting...
cycle 8:  ...waiting...
cycle 9:  add t1 = p2, p7
cycle 10: add t5 = p2, p10
cycle 11: add t4 = t1, t5
cycle 12: store [addr1] = t4
```

Might instead execute as:

```
cycle 1:  add t3 = t1, t2
      add t5 = p2, p10
      add t1 = p2, p7
      fmul f6 = f7, f14
ncycle 2:  add t4 = t1, t5
      fmul f7 = f7, f15
      store [addr0] = t3
```

Notice that the operations must be staged so that no operation is issued until its operands have been computed.

We have been discussing what is called the execution hardware of an ILP processor. Microprocessors have control hardware as well, which rather than carrying out the operations that contribute to the desired computation, control the flow of operations instead. For example, RISC execution hardware might examine a condition bit and decide which operations to execute next based on that bit, thus performing a branch specified in the program.

Depending upon the behavior of the control unit, modern processors embodying ILP are typically classified into one of two styles:

▲ Superscalar Processors. Most modern general-purpose microprocessors embody a form of ILP called superscalar. In any ILP processor the execution hardware is enhanced in one or more of the ways we’ve just discussed. But in a superscalar, control hardware is made more complex. It does the job of deciding, while the program runs, which RISC-level operations should be executed on which functional units when there is a choice, and when to execute them. This more complex hardware is often called scheduling hardware, which must determine which operations are legally executed in each cycle. Sometimes the scheduling hardware will rearrange operations to take advantage of a free functional unit or some other resource, thus speeding up the computation. A superscalar processor is handed ordinary code, compiled for a sequential model of computation, and the scheduling hardware produces the ILP. A detailed analysis of superscalar hardware is beyond the scope of this article, but for a detailed explanation see [21]

▲ VLIW Processors. These are very similar to superscalars. They use the same or similar execution hardware to make ILP available, but typically have extremely simple control units. In a VLIW, it’s the compiler that does the job of determining ILP and scheduling it on the functional units. It communicates the information about where, how, and when things are done via the program itself, by specifying directly in each cycle exactly what each function unit is to do, where it gets its data from, etc. Thus each instruction looks much like the ILP version of the code shown above. Since this design philosophy, in the extreme, mandates the simplest possible control hardware, it is common to put no-ops in each instruction corresponding to those functional units, making their task in that cycle explicit. In that case, the instructions from our code fragment might look like Table 1 (which, for space reasons, only shows the opcodes). The objects listed as Instruction 1, etc., are what the code itself looks like when handed to the hardware. Practical VLIWs have been built having instructions that were 1,024 bits long, with each cycle being able to issue 16 integer operations (eight of which could optionally be loads), four floating-point operations, and four branches [10, 32]. These systems, built by Multiflow Computer in the late 1980s and called the Trace 28/300, could issue 28 operations each cycle. A more detailed survey of instruction-level parallelism can be found in [38]. Some early DSP processors embodied this same style of computation in the form of user-programmable microcoded engines, typically found in embedded military applications.

<table>
<thead>
<tr>
<th>Function Unit</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
<th>FMUL/FADD</th>
<th>LD/STORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1:</td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>FMUL</td>
<td>NOP</td>
</tr>
<tr>
<td>Instruction 2:</td>
<td>ADD</td>
<td>NOP</td>
<td>NOP</td>
<td>FMUL</td>
<td>STORE</td>
</tr>
<tr>
<td>Instruction 3:</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>STORE</td>
</tr>
</tbody>
</table>
VLIWs Used in DSP Applications

VLIWs have been extremely popular in the DSP world for several reasons. One important reason is the ability to take advantage of the abundant ILP that's available in typical DSP codes. Because the control hardware becomes enormous when it must control large numbers of functional units, the elimination of the extra control hardware becomes a very big performance and cost factor and makes it practical to deliver the available ILP.

It's also worth pointing out that a major disadvantage of VLIWs in the general-purpose microprocessor world is the issue of object code compatibility: programs for different VLIWs are likely to look different, and thus code must be recompiled for each architecture. This tends not to be an impediment in the DSP world, since recompilation and unique hardware have been the norm. Similarly, because its scheduling decisions are made in advance, sometimes a VLIW can do poorly because of its inability to adjust to dynamic conditions in a program. The relatively static flow of control in DSP and multimedia applications has made VLIW a good choice.

Past VLIW-like Hardware for DSP

Until recently, the many uses of VLIW technology for DSP have involved host/slave processor systems, in which a general-purpose host controls the DSP hardware, doing the rest of the application and supplying the "glue." During the most compute-intensive DSP inner loops the host initiates a computation on the special-purpose VLIW hardware. This article concerns instead an important new capability that VLIWs can bring to DSP: a VLIW can be built so that it is a general-purpose processor in the sense that you can compile any program onto it (for example, it runs the operating system), but is as fast as special-purpose hardware when carrying out the DSP applications it was intended for. The host and the DSP processor are one in the same. VLIW as a design philosophy enables this more "normal" processing, built with compiling in mind, yet does not sacrifice performance on DSP and multimedia applications.

In the historical host/slave arrangement the same concept of large instruction words controlling functional units has been a constant in the DSP world for many years. This style of parallelism was most often found in mainstream processors in their horizontal microcode, in which complex operations that appeared in CISC instruction set were emulated by a fetch-execute engine that operated on a level down from the CISC architecture. Thus, to execute a string move, for example, the hardware would fetch a small "microprogram" subroutine that would do the small pieces of a string move—fetching the string elements and moving them to their destinations, as well as handling the looping, etc.—and so on. Since these were routines written while the hardware was being designed, and not open to the user, or to being changed by anyone but hardware designers later, it was very tempting to notice that several of these "micro-

operations" could be done simultaneously. Thus, horizontal microcode was very much like VLIW, except that it involved individual hand-written routines operating at a lower level (often doing bit manipulations, for example). It was a natural step to store these microprograms in a writeable control store and open them to the user to add special operations. Given this kind of capability, it was again natural to contemplate processors with their operations particularly suited to DSP applications, having this kind of instruction-level speedup.

In the 1970s and early 1980s there were many horizontally microcoded slave engines that were used in such diverse applications as CAT scanners, oil exploration, mechanical CAD, general image processing, and so on. The best known of these were probably the Floating Point Systems FPS AP120 and FPS 164. Other popular systems included models made by Analogic, Numerix, Masscomp, CSPI, and CDC, STAR Technologies, and others. A bridge to the current microprocessor designs was the Advanced Micro Devices 2900 family of chips. These were "bit slice processors" and were modular parts of ALUs that were sold in 4-bit slices. There were signals on these chips that allowed them to be ganged together easily, so that one could build one's own horizontally microprogrammable processor. The family included parts that made it easy to build the control unit, which, since these followed the VLIW style, were quite simple. Many "home-brew" VLIW-style DSP- and media-oriented processors were built from these parts, as well as commercial processors.

In the embedded DSP world some early 16-bit DSP microcontrollers, such as the TI C2x and C5x families, were VLIW-style, but with only 2 or 3 functional units. For example, one might perform a MAC or other data operation, while simultaneously incrementing an address register or changing the "current" address register. Amusingly enough, the lack of good compilation techniques forced many programmers to resort to "programming by spreadsheet," a spreadsheet being the best tool available for capturing the two dimensional nature of the code.

Compiling for VLIWs: Trace Scheduling and Software Pipelining

Compiling for VLIW DSP processors (or horizontal microcode, for that matter) was not done effectively before the 1980s, except for compiling the surrounding code on the FPS attached processors. Until then, the only way a manufacturer could offer the user relief from laborious hand coding was through canned libraries. This has the following disadvantages:

- Laborious work must be done to splicе the code in the library into the rest of the application, especially in the placement and accessing of data.
- Applications must be changed to meet the requirements of what is available in the libraries, often to the detriment of the application.
Performance often has to be compromised to meet the library's need for generality.

A high-level language approach was clearly more desirable, and the processors' user communities asked for compilers. Unfortunately, until the 1980s, there were no approaches that matched, or even came close to, the performance that hand code could provide. The major barrier faced, even by experienced compiler writers, was that available practical compiler technology only enabled the scheduling of operations that came from single straight-line segments of code, called "basic blocks" by compiler writers. Whenever the compiler encountered a jump of any form, or even a branch target, the compiler would stop any further scheduling, declare that section done, and start all over with the next section. Unfortunately, there was little performance to be gained that way: studies have repeatedly shown that most ILP to be gained in basic block scheduling is about a factor of 2 (the classic study by Tjaden and Flynn [40]). If operations could somehow be moved between blocks—that is, globally, as experienced handcoders routinely did—rather than locally, far higher degrees of ILP were available.

In the early 1980s, two sets of techniques emerged that offered the prospect of compiler-driven performance that could match or even beat handcode: trace scheduling and software pipelining. Today, each of these has been implemented many times, and several commercial optimizing compilers embody variants of them, though they are still often regarded as experimental techniques. These techniques free DSP processors from their previous role as functional black boxes and bring them into the mainstream of computer systems.

**Trace Scheduling and Related Techniques**

The global scheduling algorithm known as trace scheduling [14] is centered on traces. Rather than a single basic block, the compiler focuses on loop-free sequences of basic blocks embedded in the control flow graph; that is, a path through the program that could conceivably be taken for some set of input data. Traces are selected and scheduled in order of their frequency of execution. The selected trace is then scheduled as if it were a single basic block, i.e., giving no special consideration to branches. By scheduling the trace all at once, the compiler implicitly moves operations between blocks. Sometimes this produces a schedule that would not correctly carry out the program's semantics, in which case additional operations are scheduled off the trace. After a given trace has been scheduled, the next most frequently executed trace is selected from among the remaining unscheduled operations, including those added in the process of scheduling earlier traces. This continues until the entire program has been scheduled. Trace scheduling is too complex to describe in detail—far too much so to describe here—and complex to implement (for a good description, see [12]), and many simplifications have been proposed and implemented. There is no consensus about the effectiveness of the various alternatives techniques, but all follow the basic philosophy initiated by trace scheduling: Do code motions implicitly as part of scheduling a region much larger than a basic block.

Often, especially in DSP and multimedia codes, the compiler is presented with a single small loop. Since trace scheduling does not proceed beyond a back edge, there would be little opportunity for the compiler to do the global code motions that allow trace scheduling to pay off. The usual solution for this is to unroll the code, and trace scheduling compilers typically do a sophisticated job of turning a small piece of code into a long chain of blocks without back edges and without the artificial data dependencies that a naive job of unrolling would introduce. Unrolling, which we discuss more later, can be very effective, but has the problems that it generates much extra code and sometimes loses performance because there is always a start up and close down cost at the beginning and end of each series of unrolled iterations. When possible, it is sometimes beneficial to apply a technique called software pipelining, which, when it can be applied, does not have these problems.

**Software Pipelining and Modulo Scheduling**

Software pipelining [3, 7, 37] is a technique in which the compiler rearranges a loop so that iterations of the loop can be issued repeatedly, far more frequently that the time it takes for a single iteration to execute. Like trace scheduling, the implementation of software pipelining is too complex to describe here in detail. To understand the central ideas, however, imagine that the compiler, having been handed a simple loop, determines by examining it that it could be rearranged so that it issues a new iteration every N cycles.

To accomplish this, the compiler rearranges the loop so that the VLIW's resources used in cycle i will complement those used in cycle (i + N). That is, both of these operations could be initiated together (or, more generally, all operations that are in the same position in the loop mod N can be issued together without violating the resource availability of the processor—thus "modulo scheduling"). Software pipelining works well for very simple loops, and continuing research has broadened its applicability considerably. Some DSP-oriented processors have had "rotating registers," which allow a form of register renaming as the loop runs, thus allowing a compact representation of the schedule, since single operations can refer to the same registers in the code while actually involving different sets of registers.

**DSP and General-Purpose Computing Compared**

A DSP processor is specifically designed to support regular computation-intensive tasks. The basic characteristics that distinguish a DSP processor from a general-purpose processor can be summarized in three categories:

1. Special-purpose operations that permit the execution of frequently used combinations of instructions in a
single cycle. The most common dedicated operation is the multiply-accumulate, very commonly used in most of the filter kernels based on dot-product structures (of the form a = a + b • c).

2. Multiple concurrent accesses to memory in a single cycle, with one or more dedicated address-generation units that usually include increment capabilities decoupled from the main control flow. For instance, post-increment, bit-reversed, and modulo addressing are commonly used to access buffers. Memory to support multiple accesses is typically either partitioned or interleaved.

3. Special support for efficient, low-overhead counted loops. This usually comes in the form of repeat or decrement-and-branch instructions.

Recently, general-purpose superscalar processors have narrowed the gap in these areas and they are slowly eroding some fraction of the DSP market with the major enablers being:

- Better technology (driven by higher volumes), yielding higher clock rates.
- Added DSP functionality, such as multimedia extensions, that can make performance comparable to DSP processors in important areas.
- Simpler systems where one single processor can run both DSP and non-DSP code.
- Better compiler and code optimization technology that makes high-level languages more appealing than assembler programming and improves code portability and reusability.
- Simplicity of use and wider selection of available development environments.

The ILP in these general-purpose microprocessors has and will continue to further soften the distinction between general-purpose and DSP architectures. What ILP brings to the table is a very effective way of providing performance without compromising cost or generality.

VLIsWs targeted to DSP applications are now combining the cost/performance ratio of a specialized DSP processor and the flexibility of a general-purpose processor. The next section of this article presents an overview of the different architectural features that can be exploited in ILP processors in general and how they match the characteristics of DSP applications.

**DSP Workloads**

The growing demand of multimedia and DSP applications is fundamentally changing the mix of applications that represent the typical workload of a general-purpose workstation.

Until recently, general-purpose microprocessors have been built to handle a "traditional" mix of applications—the mix that the Spec Benchmark attempts to characterize. We can characterize this traditional workload as tasks that:

- Have limited ILP
- Are very branch intensive
- Have very unpredictable and data-dependent branch behavior

- Have a memory access pattern that is well suited for a traditional multiple-level cache memory hierarchy.

On the other hand, applications composing a "media-oriented" workload:

- Exhibit a high degree of ILP. For most applications, this is a repeated set of independent operations carried out on large quantities of data (e.g., the pixels of an image).
- Have a very predictable control flow, as most of their time is typically spent in relatively small innermost loops. Also, this control flow regularity can usually be detected at compile time, either through heuristics or via profile information.
- Have a memory access pattern that does not fit very well with a traditional cache hierarchy. This is due to the dual nature of most applications that tend to interleave accesses with no temporal locality and accesses with no spatial locality in the same piece of code (more details are presented later).

Similar characteristics are typical of other DSP applications: software modems, packet-switching tasks, etc. These are usually multiple streams of data being processed in identical ways.

It is becoming harder and harder to draw the distinction between a DSP application and a general-purpose application, and it appears that both components are usually present in the applications that drive the market today.

**Floating-Point vs. Integer Arithmetic**

Before discussing the architectural properties of these processors, there is one auxiliary issue worth discussing. Most DSP processors and applications use fixed-point arithmetic, where real numbers are represented as fractions between −1.0 and 1.0. This architectural decision is usually made for cost reasons, as floating-point functional units use a lot of die area. DSP architectures have been used so often in cost-sensitive applications that algorithms for them are typically characterized in fixed-point terms. This adds a coding burden compared to using floating-point arithmetic, where values are represented through mantissa and exponent. With fixed-point arithmetic, the programmer must carefully avoid overflows by appropriately scaling intermediate results of the computation. The range of a floating-point number is much larger and the programmer can almost always ignore precision constraints.

If we consider multiple-issue processors, the cost argument increases even more. While it is conceivable in today's technology to build a processor with 8-16 integer units at reasonable costs, the same issue width for a floating-point data-path would certainly be too expensive in terms of die size.

However, if a floating-point unit is available (such as in general-purpose processors with DSP capabilities), it is sometimes convenient to code in floating-point for performance reasons. For example, many processors with a floating-point unit turn integer multiplication into a sequence of:
- Convert the integer operands to FP operands
- Do a FP multiply
- Convert the FP result to integer result

In this case it is evident that re-thinking the code in terms of FP values would yield further performance benefits. Unfortunately, these techniques are highly nonportable and only apply to a particular class of processors. For this reason, the rest of the article assumes fixed-point arithmetic as our default, unless otherwise noted.

**DSP-Oriented VLIW Architectures and their Compilers**

This section highlights some of the most promising evolutionary directions of DSP/media architectures. In particular, we focus on five different architecture aspects that we consider particularly important:
- Instruction-level parallelism
- Memory hierarchy
- Registers
- Branch support
- Code size

For each of these, we are particularly interested in covering:
- The motivation for adding the new architecture feature, particularly in relation to some characteristics of DSP applications.
- Where it successfully applies and where it does not, or even where it hurts performance and cost.
- The impact of the feature on the programming model, on compiler algorithms and design, and on microarchitecture design and overall cost.

**ILP Execution Unit Architecture**

As we discussed previously, typical DSP applications spend most of their time in innermost loops. These are traditionally good candidates for transformations that expose instruction-level parallelism, such as loop transformations, unrolling, and software pipelining. Typical examples of this are:
- Signal processing filters (e.g., FIR, IIR)
- Frequency transforms (e.g., Fourier, Cosine)
- Image processing algorithms (e.g., filters, edge manipulation)

In general, whenever the computation of a “new” value (or pixel) does not affect the computation of “future” values (or neighboring pixels) we can usually extract significant amounts of ILP. Even when a dependency exists, often the nature of DSP algorithms allows a considerable amount of overlap among different loop iterations.

Traditional DSP processors have exploited these characteristics by identifying the most commonly executed groups of instructions and by mapping them to dedicated instructions, such as multiply-accumulate. Additionally, as we have seen, ILP allows us to issue several operations simultaneously, thus increasing performance, sometimes dramatically. It is worth mentioning that another alternative is to use multiple DSP engines in parallel, although this usually requires a significant amount of hardware and software engineering. Multiple processor parallelism is an important and very large subject, which we will not cover here.

We have already seen how VLIW architectures offer the prospect of exploiting ILP. Another form of ILP that complements VLIW and that is very often an important factor in achieving high performance is a form of parallelism called SIMD (for single instruction, multiple data). This is particularly useful when applied to “subword parallelism.”

**Subword Parallelism: SIMD**

SIMD instructions are a very efficient and compact way of representing multiple occurrences of the same operation on different data items. SIMD instructions were used in the past mainly in the form of special-purpose vector instructions. They were quite powerful but did require dedicated hardware and memory addressing modes, and they were abandoned mainly because of their lack of flexibility and ease of use.

Today, SIMD instructions are appearing again in both DSP and general-purpose processors, mostly in the form of multimedia instructions. These exploit a limited amount of ILP by packing smaller precision data in larger containers and by setting up the ALUs to perform multiple smaller-precision slices of the same operations on the individual subwords of the inputs. The containers (normally 32 or 64 bit) are usually already available in the architecture; for example, as double-precision floating-point registers.

This feature is often called subword parallelism, packed-arithmetic, or multimedia extension [29]. The most widely used general-purpose microprocessor platforms are adopting particular implementations of this concept, including:
- INTEL: MMX instructions
- SUN: VIS instructions
- HP: PA-RISC Multimedia Extensions
- PPC, ALPHA, MIPS:
  - High-performance DSP processors are starting to adopt this paradigm, including:
    - Texas Instruments’ TMS320C6xxx
    - Philips’ Tri-Media
    - Chromatic’s MPACT
    - Mitsubishi’s V30

Current implementations are usually exposed at the programmer level and require code rewriting, either at assembler level or through dedicated libraries of intrinsic functions. For example, a simple loop to add two arrays traditionally written (in C) as the following:

```c
int size = 1000;
byte out[size], in1[size], in2[size];
```
for(i = 0; i < size; i++) {
    out[i] = in1[i] + in2[i];
}

Must be rewritten as:

int size = 1000;
byte out[size], in1[size], in2[size];
for(i = 0; i < size; i++ ) {
    packed4 t1 = packed4_load(in1+i);
    packed4 t2 = packed4_load(in2+i);
    packed4 t3 = packed4_add(t1,t2);
    packed4_store(out+i, t3);
}

Here, the packed4_add operation actually performs four additions in parallel on the four pairs of packed input data items and generates four packed output data items. The packed4_load and packed4_store operations are in charge of reading and writing the data from/to memory with the proper packing and unpacking. In the particular example, they can just read memory words (32 bits instead of bytes). In general this operation can be more complicated due to alignment constraints and more complex packing schemes.

For example, a possible pseudo-assembler code segment for the loop body could be:

cycle 0: packed4_load t1 = [in1 + i]
cycle 1: packed4_load t2 = [in2 + i]
cycle 2: add4 t3 = t1,t2
cycle 3: packed4_store[out + i] = t3

In this case, the advantage of the add4 operation is evident and gives a nice factor of four speedup in execution time, with no code size penalty at all. The additional execution hardware for this functionality is minimal:

- More control logic in the ALU to perform four parallel 8-bit additions instead of a single 32-bit addition.
- A few more instruction opcodes for the new SIMD operations and the corresponding decoding logic.

Even from this simple example, however, the limitations of the technique start to emerge:

- We had to make a set of assumptions based on the knowledge of the code to do the rewriting, such as: we know that size is divisible by 4; we know that the arrays in1,in2, and out are aligned 0 modulo 4; and we know that an 8-bit addition is sufficient for our target precision.
- Attempting to automatically operate the same type of transformation in a compiler can turn out to be prohibitively complicated (and often impossible) in the absence of user guidance, since: loop bounds are usually not known at compile time; alignment is usually not known at compile time; and the C language mandates 32-bit precision for arithmetic operations (for a 32-bit machine).

When interprocedural and whole-program analysis and optimization are possible and affordable, some of these problems can be mitigated, especially for very small applications.

Since data has to be packed into larger containers, there are unpacking penalties to pay if we need to access single items individually and not as a whole group. For example, the following code would require unpacking a single element from the packed word in order to execute the comparison:

```c
int size = 1000;
byte out[size], in1[size], in2[size];
for(i = 0; i < size; i++) {
    out[i] = in1[i] + in2[i];
    if(out[i] > 200) x = 1;
}
```

Most multimedia instruction extensions include compare operations that generate masks of “predicates” (discussed later) to drive conditional move operations (or other types of “predicated” operations, in which an operation is executed or not depending upon the value of a single condition bit already produced). However, the applicability of this approach is not general and often forces the programmer to think in a nonintuitive way.

Supporting a full set of opcodes for all the possible combinations of operations is impractical. This means that if the need of doing nonhomogeneous operations arises, more code to unpack and repack starts appearing. For example, the following code does not map well to a simple sequence of SIMD operations:

```c
int size = 1000;
byte out[size], in1[size], in2[size];
for(i = 0; i < size; i++) {
    if (i & 1) out[i] = in1[i] + in2[i];
    else out[i] = in1[i] - in2[i];
}
```

Subword Parallelism: VLIW

An alternative method of exploiting subword parallelism is to use a set of heterogeneous functional units whose operation can be independently addressed by distinct instruction fields. In this method the instruction width grows with the number of units—we have seen that VLIWs can indeed be very long.

With a VLIW paradigm, the fact that the parallelism is limited to subword elements diminishes its importance as far as the number of operations is concerned. It can, however, affect register allocation if proper hardware support for subword containers exists and the compiler is able to identify the precision of some values. This is discussed in more detail later.

A VLIW approach presents obvious advantages and disadvantages in comparison to the packed SIMD approach. For example, let us consider the same simple vector add loop of the previous section:
int size = 1000;
byte out[size], in1[size], in2[size];
for(i = 0; i < size; i++) {
    out[i] = in1[i] + in2[i];
}

An optimizing compiler targeting a VLIW architecture with enough functional units could unroll the loop (four times in the example) and generate a code section for the loop body such as:

**cycle 0:** load t1_0 = [in1 + i0]
load t1_1 = [in1 + i1]
load t1_2 = [in1 + i2]
load t1_3 = [in1 + i3]
load t2_0 = [in2 + i0]
load t2_1 = [in2 + i1]
load t2_2 = [in2 + i2]
load t2_3 = [in2 + i3]

**cycle 1:** add t3_0 = t1_0, t2_0
add t3_1 = t1_1, t2_1
add t3_2 = t1_2, t2_2
add t3_3 = t1_3, t2_3

**cycle 2:** store [out + i0] = t3_0
store [out + i1] = t3_1
store [out + i2] = t3_2
store [out + i3] = t3_3

**cycle 3:**

This would not be possible with a SIMD approach (at least without a major rethinking) due to the rigidity of the instruction set.

To summarize the disadvantages of a pure VLIW approach (and some mitigating considerations):
- VLIW functional hardware is more expensive. A packed SIMD architecture exploits the same ALU hardware to execute operations on subwords; a VLIW requires multiple functional units with full precision to achieve the same level of ILP. (While the VLIW approach is more expensive, it is more flexible. One could argue that the silicon area occupied by functional units normally represents a small fraction of the silicon budget of a modern microprocessor, but this factor nonetheless has to be taken into account.)
- VLIW register connectivity is more expensive. A packed SIMD architecture uses the same register file port to read and write more logically distinct values; a VLIW requires multiple ports, one per functional unit. However, while a fully connected VLIW (with a single register file) could be impossible to implement, alternative structures such as clustering mitigate this effect and reduce the cost to affordable levels. This is discussed in more detail later.
- VLIW instruction encoding is more expensive. A packed SIMD architecture encodes in the same instruction multiple independent operations; a VLIW requires multiple independent instruction to drive the same level of ILP. This is a potential showstopper for the embedded DSP market, since code size is a major concern. However, memory compression techniques can actually reduce VLIW code to a level that is very competitive to RISC/DSP instruction sets. This is discussed in more detail later.

And the advantages:
- A VLIW target does not require major rewriting of the application. As we previously discussed, using SIMD instructions requires rewriting the application to manually identify where the ILP is and to insert calls to proper library functions using the SIMD instructions.

could be mapped to a very similar piece of code, such as:
Alignment constraints do not have to be identified. If the hardware provides enough flexibility for multiple concurrent memory accesses, the compiler can issue memory operations to the smaller data type and ignore alignment considerations.

Precision constraints do not have to be identified. The compiler generates code that obeys the semantic of the language. In particular, C defines arithmetic operations to be executed in 32-bit precision (for a 32-bit architecture). As a consequence, the programmer does not have to identify and isolate operations that could be executed with a smaller precision. This task could be quite hard for real-life algorithms that are rarely as clean as the shown examples.

Packing and unpacking overhead is minimal. Since data items maintain their individual identity, it is not required to always treat them as a group. Of course, it is always possible to "manually" pack and unpack data when the memory architecture can take advantage of that (e.g., to do one wide memory access instead of many narrow ones).

VLIW instructions are more flexible. At the cost of a more complex decoding scheme, VLIW architectures do not impose limitations on the combination of operations that can be executed in parallel. This is again very important when the nature of the algorithm is not regular.

Conclusions
In this section we have presented what we believe are the two most promising techniques to expose and achieve ILP for DSP applications. Both of them present advantages and disadvantages, and both of them are well suited for different aspects that are quite common in typical DSP code.

To summarize our view, we believe that:
- Traditional DSP instruction sets are not a good match for multiple-issue processors, mainly because of the rigidity and extreme specialization of the instruction set. This makes it very hard for a compiler to generate efficient parallel code. At the same time, writing assembler code for a multiple-issue variable-latency problem is certainly one of the most time-consuming and unfriendly tasks for a programmer.
- Packed arithmetic itself is important, but a pure SIMD instruction set is not fully adequate to achieve the performance demanded by multimedia applications for the most common algorithms.
- An ambitious pure VLIW architecture could have significant cost impacts that may result to be unacceptable for an embedded application.

What seems to emerge from this is that a combined approach could be the winning solution. The multimedia-style SIMD approach does not conflict with a VLIW architecture, and a good compiler could mix both worlds when properly guided.

A possible solution to get the best of both approaches could be represented by a moderately wide VLIW architecture where some of the functional units embed packed arithmetic functionality. An expert programmer could exploit such architecture where code size considerations are the first priority and where the regularity of the algorithms makes a SIMD type of parallelism obvious. On the other hand, the VLIW component of the architecture is a good target for an optimizing compiler in the code sections that are more complicated and would require major rewriting to fit a SIMD model, were it even possible.

Register File Architecture

Quantity
In general, applications with large amounts of ILP require a large number of registers, and the most effective techniques to expose ILP add register pressure, such as:
- Speculative Execution: speculatively generated values have to be retained until committed or discarded.
- Unrolling/Software Pipelining: compiling $N$ iterations of a loop usually implies that the lifetime of loop values is multiplied by $N$ and the register requirements are higher.
- Predication: values along both paths have to be kept around until the merge points.

This effect is somewhat mitigated with an SIMD approach, where fractions of a register are used to pack smaller size values, but the nature of the problem does not substantially change.

Having many architecturally visible registers helps reduce the memory traffic to the bare minimum and also enables more complex optimizations.

Register Connectivity and ILP
As we mentioned earlier, register connectivity is potentially a cost and performance barrier for wide-issue machines.

This is mainly due to the fact that a VLIW architecture with $N$ 2-input/1-output functional units requires reading 2$N$ values and writing $N$ values per cycle to sustain the achievable throughput. For a single register file, this translates to 2$N$ read-ports and $N$ write-ports.

If we take a look at typical DSP applications and we consider today's bandwidth to memory, it is reasonable to expect ILP levels around a factor of 8. If we want to build a fully connected eight-wide VLIW, we need a 24-port register file. That is very likely to be prohibitively expensive and slow in today's technology:
- The area of a register file grows approximately with the square of the number of ports.
- The read access time of a register file grows approximately linearly with the number of ports.

In practice, this means that alternative structures need to be considered when the width of a VLIW architecture grows. The threshold that still makes a single register file a viable structure is hard to quantify and changes with the technology generation. Today's technology allows us to build register files with around 10-15 ports at reasonable cost and speed, and this means that they can feed VLIW architectures with a width between about two and four.
A packed-SIMD approach diminishes the need for ports since the access to a single register moves multiple data items to the functional units. This is one of the major microarchitectural savings of the SIMD paradigm.

**Partitioned Register Files**

One of the most promising techniques for overcoming the register port problem involves partitioning the register space in physically separated register banks, each of them with a limited connectivity to a subset of the available functional units [6, 12]. Figure 2 shows the structure of an execution unit with clustered register files.

Once we decide to partition the register file, we can still have a few different architectural scenarios:

- **Register file clustering is architecturally invisible.** This means that the compiler still sees a logically unified register space. When the operands of an operation are in the “right” place, then the operation can take place regularly. When they are not, the hardware must provide support to stall the pipeline and automatically move the missing values in the right place. This solution has the advantage that the compiler task is greatly simplified, but the disadvantage that the “copy overhead” to move the operands cannot be hidden. This adds a potentially high dynamic penalty if the code is not scheduled properly. Also, this penalty is not easily quantifiable, and real-time DSP application could suffer from this unpredictability and be forced to use very conservative worst-case estimates.

- **Register file clustering is architecturally visible.** Here we have two more choices:

  1. Functional units have architecturally complete connectivity with different latencies for “local” and “remote” writes (or reads). In this configuration, a “remote” operation is composed by the operation itself, followed by (or preceded by) a register copy to the remote destination (from the remote operands). The advantage of this solution is that the compiler can try to hide the copy overhead in the schedule. However, the fact the operation-copy (or copy-operation) pair is “atomic” still does not allow it to schedule copies as flexibly as the compiler could do.

  2. Functional units have architecturally limited connectivity, and copy operations have to be explicitly scheduled. This has the advantages of the previous alternative, but with the additional benefit that the compiler is not constrained to keep the copy and operation together and can potentially generate a better schedule. The additional cost of this solution is code space to hold the code for the copies in the object code.

**Clustering Algorithms**

For all the previously described approaches, the compiler’s task is to partition the code to minimize the overhead of copy operations. If the clustering is invisible, the compiler can optimize the code but is not responsible for the correctness that is enforced by the hardware. If the clustering is visible, the compiler is responsible for generating correct code that satisfies the latency and limited connectivity constraints.

In both cases, compiling for a clustered register file greatly complicates the code-generation phase for a VLIW compiler. The work involved falls into the category of task graph scheduling problems [1, 30]. Too complex to consider here, these algorithms allocate a set of parallel operations represented by an edge-weighted direct acyclic graph (DAG) to a set of heterogeneous resources, with the objective of minimizing the completion time. They are quite effective in practice.

**Exploiting Subword Precision**

As we mentioned earlier, one typical characteristic of DSP applications is to require many operations with reduced precision. Independent of the adoption of packed-SIMD instructions, we can think of exploiting this property as motivation to increase the number of compiler-visible registers.

A few conditions are necessary to exploit subword precision:

- **The hardware must provide support to address individual parts of a register.** For instance, we can add hardware support to view a register file of 32 32-bit registers also as a register file of 64 16-bit registers. In this case, the compiler has the possibility to see twice the number of “containers” when the precision is smaller. This involves:
  1. Reserving bits in the instruction to address the larger address space.
  2. Adding decoding logic to be able to read and write subparts of a register without affecting the rest.
  3. Adding routing logic to shift and align the accessed subpart of a register with the proper bits of the execution units.

One possible organization that minimizes the hardware costs is, for example, a register file physically built from smaller subbanks that are combined when larger containers are required. For instance, we can build two 16-bit banks with 32 registers each that are combined together when we want to access a 32-bit register, or sepa-
3. Example of a register file organized to exploit subward precision.

Memory Architecture
Characterization of Data Memory Accesses

DSP applications are generally very demanding of the memory architecture of the system. If we try to characterize the type of data memory accesses, two classes emerge:

1. Memory accesses with spatial locality but no temporal locality. These are typically regular accesses to large data structures (such as images or a bit streams) that are traversed with a very structured and predictable pattern. These accesses:
   ▲ Have high spatial locality, since the access pattern is usually sequential or with a small stride.
   ▲ Have negligible temporal locality, since the arrays are usually traversed once (as in a bit-stream) or the data to buffer between two visits to the same location is too large for practical purposes (as in a multipass image filter).

2. Memory accesses with temporal locality but no spatial locality. These are typically data-dependent accesses to tables, such as: compression dictionaries, interpolation tables, complex function look-up tables, and so on. These accesses:
   ▲ Have high temporal locality, since the array is usually small enough to be buffered or is repeatedly accessed in neighboring areas.
   ▲ Have negligible spatial locality, since the accesses are usually data-dependent and not sequential.

This mix of memory accesses for DSP applications is quite different from a general-purpose program. As a consequence, structures such as caches, which work well for a general-purpose architecture, are often of limited utility in DSP applications. A look at today’s DSP market, where data caches are rarely present, confirms this fact.

The reason an indiscriminate use of a data cache is typically not very effective in DSP applications derives from the mix of memory accesses that we described. In theory, a data cache works well for both temporal locality (reuse of the same location multiple times) and spatial locality (use of neighboring location in the same cache line). However, large data structures with only spatial locality tend to “blow away” smaller data structures that could benefit from temporal locality. This is unfortunately true for DSP applications.

There are a few possible workarounds for this problem. The basis for all of them is that accesses must be distinguished, and this can be done at different levels: in the source program through annotations, in the compiler level through analysis, in the hardware through profiling.
A first solution that fits in a traditional memory hierarchy and requires a very limited amount of hardware support is by bypassing. If we identify structures with no temporal locality and bypass the data cache for them, we minimize the pollution effect described before. However, a bypassed access always pays the latency of a random access to main memory and cannot benefit from DRAM bursts. This can only work if the bypassed accesses are not in the critical path of the computation.

In the following sections, we consider two approaches to this problem. In particular, we look at the use of local memories to exploit temporal locality and the use of prefetch buffers to exploit spatial locality.

**Local Memories**
Both DSP processors and general-purpose processors have traditionally made large use of local data memories. The main difference is that cache memory is the model of choice for general-purpose processors, while local memory banks are usually preferred in DSP architectures.

- Data cache memory buffers parts of the main shared address space and is very effective for temporally local data whose access pattern and size is predictable, does not require programmer's support to be used, has a non-predictable access time, and an access missing in the cache is usually one (or even two) orders of magnitude longer than a cache hit.

- Local memory is mapped to an address space that is separated from main memory and is very effective for predictable accesses to small or medium-size data, requires explicit programmer or sophisticated compiler support to be used, and has a known and constant access time.

The real-time requirement of many DSP applications is an important reason that DSP architectures traditionally contain local memory rather than cache. Since real-time requires designing for the worst case, the large mismatch between hit and miss access time of a cache makes the worst-case performance practically unacceptable for most applications. Local memory, on the other side, does require some more effort by the programmer, but potentially gives a much higher improvement and a more predictable execution time.

For all these reasons, local memories and VLIW architectures are also a good match. The major difference between the "traditional" DSP usage of local memories and VLIW architectures is the role of the compiler.

In a VLIW architecture, the compiler is responsible for the management of all the memory accesses, their latency, and resource utilization. The programmer is not supposed to handle the details of a memory operation explicitly. Unfortunately, for languages like C, it is hard for the compiler to automatically identify data structures that can be safely moved to a different address space. This is mainly due to the ambiguity of memory references through pointers and the limitation of the current techniques for pointer alias analysis.

An alternative method for exploiting local memory within high-level language programming is through compiler annotations or language extensions:

- **Language extensions** are normally in the form of attributes or qualifiers added to the declarations of types and variables. Extensions have been widely used in the past to express platform-dependent features that the language does not natively support. An example of a familiar language extension is the `far` modifier for pointers in a 16-bit machine (such as an 80286) to access memory areas beyond the physical address space.

- **Compiler annotations** are normally in the form of `#pragma` commands in the source code. The use of `#pragma` forms has the advantage of maintaining compatibility across compilers. Unlike extensions, a compiler that does not know how to interpret annotations can safely ignore them. In this way, the code remains portable across different platforms and compilers.

One possible way of mapping data structures to a local memory bank is through an extension of the C type system. The programmer "decorates" certain types and declarations with properties that instruct the compiler to allocate the object in a different memory address space. For instance, the following code can be used to copy an array from main memory to local memory:

```c
int main_mem_array[1000];

#pragma local_memory
int local_memory
int main_mem_array[1000];
...
for (i = 0; i < 1000;++i) {
   local_mem_array[i] =
   main_mem_array[i];
}
```

The `#pragma local_memory` directive instructs the compiler to tag the type of `local_mem_array` with a property that turns all the accesses into accesses to local memory. Of course, the whole system tool-chain must be aware of the new memory address space and behave accordingly.

The use of annotations:

- Allows programmers to exploit local memories within a high-level language environment. The programmer is still responsible for guaranteeing coherency among the different address spaces.

- Allows compilers to identify the local memory accesses without a big effort. This enables a set of optimizations and compaction algorithms that can take different latencies and bandwidth into account.

- Maintains language compatibility, in the sense that the code remains legal regardless of whether local memory is used or not.

- Enables an early debugging phase with the aid of type system checking rules.
Prefetch Buffers

Local memories are effective for storing small to medium sized data structures that get reused with a certain frequency, (i.e., data with temporal locality). Unfortunately, in a typical DSP application large portions of memory accesses are used only once or a few times at most and traverse very large data structures. Local memories are of little use here.

We can consider two alternatives to speed up these accesses:

1. Use dedicated off-chip memory with a fast access time. This solution is not always viable for a couple of reasons. First of all, an off-chip access is always an order of magnitude slower. In addition, if we use off-chip static (SRAM) memories, the cost of storing large data structures can become prohibitive. Finally, if we use off-chip dynamic (DRAM or SDRAM) memories, we have to pay for a nonburst, random access every time.

   2. Use an on-chip prefetching mechanism. This solution works very well when the access pattern is predictable and the references are within a loop. Under these conditions, we can easily add prefetch operations to prepare values for the future.

Prefetching is a well-known technique and has been widely used in the past, both in hardware and in software (see, for example, [8], [9], [4], [23], and [35]). Here, we present an overview of possible software techniques and their implication on DSP algorithms, hardware, and compilers.

We have a few different choices of granularity to add software-level prefetching instructions:

1. Loop-level prefetching. It operates at a coarse granularity level to preload the values to be used in a loop, before the loop itself starts. This requires a symbolic analysis of the program at a global level. For instance, the loop to add two arrays could be rewritten as:

   ```c
   int size = 1000;
   byte out[size], in1[size], in2[size];
   for(i = 0; i < size; i++) {
       prefetch(out[i]); /* Even written lines must be valid */
       prefetch(in1[i]); /* in cache to prevent stalls */
       prefetch(in2[i]);
   }
   for(i = 0; i < size; i++) {
       out[i] = in1[i] + in2[i];
   }
   ```

In this case we preload the data cache with the whole data set that we are going to use in the near future. The idea is that a `prefetch()` operation does not stall the machine, but runs in the “background” and hopefully brings in data before its use is scheduled. This kind of prefetching requires the compiler (or the programmer) to think at

a coarse level of granularity, requires the data cache to be large enough to store the required arrays, does not require a tight synchronization between the prefetch and the use, and it well tolerates any variation of prefetch latency due to unpredictable events.

2. Iteration-level prefetching. It operates within a loop body to preload the values to be used in the next iterations of the loop. For example:

   ```c
   int size = 1000;
   byte out[size], in1[size], in2[size];
   for(i = 0; i < size; i++) {
       prefetch(out[i + prefetch_stride]);
       prefetch(in1[i + prefetch_stride]);
       prefetch(in2[i + prefetch_stride]);
       out[i] = in1[i] + in2[i];
   }
   ```

In this case, the `prefetch()` operation will load the values of the arrays at the current address offset by a certain stride (`prefetch_stride`). For regular access, this technique is probably the most effective, since it requires the minimum amount of analysis and can be easily integrated in a compiler; it requires the compiler (or the programmer) to identify regular accesses within inner loops and their strides; it does not require large areas of the cache, since the distance between prefetch and use is usually short, but does require some limited amount of buffering; does not require a tight synchronization between prefetch and use; and it well tolerates small variations of prefetch latency due to unpredictable events.

3. Instruction-level prefetching. It operates within a scheduling region (such as a loop body) and tries to prefetch a memory value just before its use. The net effect is very similar to assuming a longer latency for the memory operation.

   ```c
   int size = 1000;
   byte out[size], in1[size], in2[size];
   for(i = 0; i < size; i++) {
       prefetch(out[i]);
       prefetch(in1[i]);
       prefetch(in2[i]);
       out[i] = in1[i] + in2[i];
   }
   ```

The difference between this and iteration-level prefetching is subtle but very important. Iteration-level prefetching attempts to bring in values for future iterations, while instruction-level prefetching brings in values for future instructions in the current iteration. Instruction-level prefetching is at the finest possible level of granularity and requires the compiler to schedule the prefetch well enough in advance to be useful. This means that a variation of the prefetch latency might cause the prefetch to last more than what was assumed, probably causing a performance loss. To
summarize, instruction-level prefetching does not require any complicated analysis in the compiler, does not require additional storage, requires a tight synchronization between prefetch and use, and does not tolerate variations of prefetch latency.

It is hard to say what level of prefetch granularity is the best for a given application. This usually depends upon many factors, including bandwidth/latency requirements and impact on the programming model. A good tradeoff is represented by the iteration-level prefetch mechanism, since this allows compiler optimizations to automatically add prefetch operations, requires a minimum amount of user intervention, and still achieves near-optimal performance for simple regular loops.

Hardware support for prefetching is usually not too expensive. In the simplest scenario, we can think of the data cache as our prefetch storage and use prefetch instructions to bring lines in the cache before the use. This, however, causes a cache set to be completely overwritten for each data stream when data sizes are larger than the cache. If we are not going to use the data we prefetch more than once (little temporal locality), it is possible to conceive better structures to hold prefetch data, such as a separate prefetch buffer.

The use of a small separate prefetch buffer has a few interesting advantages:

- It avoids thrashing the whole data cache for large data sets.
- It allows defining different prefetch lengths and strides, while the cache is limited to the line length. Considering that the ratio between a random and a burst access is significantly increasing with the new generation of memories, this can be an important advantage. For example, in the extreme case of parts like the Rambus memories, a random access can cost as much as 10 to 15 times the cost of a burst access. This pushes us toward longer bursts and longer cache lines. Alternatively, too long lines in the data cache reduce its effectiveness for accesses whose spatial locality is limited. The choice of a separate hardware structure for prefetching seems natural in this scenario.
- It allows multiple streams at a time, while a cache is limited to the set associativity of the cache. Since a prefetch buffer does not need to be large, it is possible to imagine a fully associative structure.

Compiler support for prefetching has been widely investigated in the past (4, 33, 34). In addition to following user hints, the compiler’s task is to identify accesses that are likely to be predictable and belong to critical sections of the code. For most practical purposes, we can limit the analysis to inner loops and try to compute the strides for each access and decide—using a heuristic—whether the access is a good candidate for prefetching. Alternatively, profiling can be used to replace the heuristics.

Branch Architecture

Low-Overhead Looping

In the last decade, designers of high-performance general-purpose processors have spent significant amounts of energy and resources trying to decrease the impact of branches on performance. Many sophisticated techniques such as hardware branch prediction are now commonly adopted in most off-the-shelf microprocessors. These techniques tend to be quite complex and expensive to implement.

Conversely, traditional DSP processors usually incorporate very simple and efficient ways of handling the control flow of simple loops.

This discrepancy can be explained if we look at typical DSP applications, where most algorithms spend a large amount of their time in simple innermost loops. Moreover, these loops are usually iterated a constant and predictable amount of time and do not involve complex control flow in the loop body. This is partially motivated by the fact that most DSP applications have some kind of real-time constraint and must thus have a predictable worst-case running time.

- Inner loops are usually handled through repeat instructions that allow the execution of a block of instructions for a predefined number of iterations. This approach eliminates the requirement for a compare, an increment, and a branch in the loop body and achieves near-optimal performance with little code.
- Outer loops (or more complicated inner loops) can take advantage of decrement-test-and-branch instructions. These are again single-cycle operations that combine the most frequently used combination of RISC-style operations necessary to iterate over simple loops. Some DSPs allow nested repeats to address the same problem.

When a processor has shallow execution pipelines, these approaches maintain their effectiveness, since it is possible to prepare the target address of the branch well ahead of the branch itself. In this way, once the operation is decoded the branch can take place with little or no penalty at all.

When execution pipelines become deep (and the clock cycle decreases) the decrement-test-and-branch approach

![Diagram](image)

4. Example of a traditional branch pipeline (F: fetch, D: decode, R: read, E: execute, B: branch) with two delay slots.
starts losing its appeal. This is due to the fact that even a simple branch operation requires fetching, decoding, and executing the instruction, and this could cause a few cycles to get lost before the new program counter is valid.

For example, Fig. 4 shows a possible branch pipeline where, even if we can execute a branch without reading values (i.e., we can produce the new program counter in the read stage), we still pay a penalty of two “delay” cycles before the branch target can start its pipeline. The two “no-operation” (Nop) cycles of the figure can be exposed to the compiler, which may attempt to fill them with useful operations, or can be quashed by the hardware, in which case they are always lost.

The problem with having “delayed”-branches is that the branch operation becomes a pipelined operation which can cause significantly more complexity in the compiler.

A possible alternative technique is to decouple the various stages of a branch operation and to expose them to the compiler before the branch happens. This can be done by adding a separate “prepare-to-branch” operation [26] at an earlier stage (see Fig. 5), which has the following advantages:

- The branch pipeline is exposed to the compiler, which can attempt to move the “prepare” operations ahead of the branch itself if the schedule allows it.
- The branch itself requires a very simple decoding and can be detected very early in the pipeline to minimize the cycle to quash (or delay).

Predication
The presence of branches, especially inside a loop body, is a limiting factor for the compiler’s exploitation of available ILP. The compiler’s ability to generate a compact schedule (that is a lower number of cycles) for a computational DAG is limited by the presence of a branch. In fact, a VLIW compiler gets most of its code-compaction capabilities from performing extensive code motion and operation reordering. Despite techniques like trace scheduling and software pipeline, the presence of a branch is a barrier to code motion, especially if memory operations are involved. To get better results, it is often the case that II’s can be converted to operations or series of operations not requiring a branch. A predicated instruction has an additional operand that determines whether or not the operation is executed. These predicates are typically stored in a special register bank that holds the set of registers used by the processor. Other processors store predicate bits in normal register locations. Finally, some processors have the equivalent of one predicate flag as part of their program status register bits.

This doesn’t come for free: often the resulting total number of operations is increased, and there are other problems associated with predication. Nonetheless, there is an advantage when the overhead introduced is outweighed by the advantages of better ILP.

Code Size
Requirements and Constraints
Embedded applications and DSP have traditionally put a big emphasis on code size. The cost of ROM has decreased considerably (2-3 $/MB), but applications have grown and the increasing use of compilers rather than hand-coded assembly also add code size. Code size is still a major concern for DSP applications.

VLIW architectures are traditionally not the best choice for code size. This comes from a combination of factors:

- VLIW compilation techniques (such as trace scheduling) make large use of inlining, unrolling, and code-motion techniques that cause code to be duplicated when operations are moved beyond branches and join points.
- VLIW instructions are very “sparse,” since they contain all the fields to drive the various functional units. When only part of the machine is used, a lot of space is wasted.

However:

- Aggressive compilation is only necessary in the critical sections of code. These are normally limited to a small fraction of the code in a large application. For example, if the compiler expands 5% of the code by a factor of 4, the overall expansion is about 15%
- We can think of clever encodings of VLIW instructions so that main memory only stores useful parts of an instruction, and not the empty fields. These techniques fall into the category of instruction compression.

Techniques for Instruction Compression
Machines with a lot of ILP must be able to encode the widest combination of resources in a VLIW instruction. This potentially wastes a lot of instruction space and is widely considered to be one of the main disadvantages of VLIW architectures.

There are two types of inefficiencies in a pure VLIW instruction stream:

- Vertical no-ops. These come from the exposed latency of multicycle operations and are represented by empty cy-
cles. Vertical no-ops can be reduced by means of multicycle no-op operations, inserted by the compiler when empty cycles are present.

▲ Horizontal no-ops. These come from the sparse encoding of the VLIW instructions and are represented by empty slots. Horizontal no-ops can be reduced by means of instruction-compression techniques.

The goal of instruction-compression techniques is to reduce the number of bits necessary to represent a VLIW instruction [11]. Due to the sparse nature of VLIW instructions, there is usually a lot of room for compression. The basic idea is to keep the instructions compressed in a higher level of the memory hierarchy and to decompress them when they are about to be executed. In this way, the compression part is run when the program is built and can be expensive. The decompression part is run during execution and must be simple and fast.

We can classify decompression techniques in two categories:

▲ Decompression on cache refill. Here, main memory is compressed, but the instruction cache is uncompressed. This was done in the Multiflow Trace systems mentioned earlier. The advantage of decompressing on refill is that the operation is not time-critical since it has to be executed only when the I-cache misses. As a consequence, a more complicated decompression algorithm can be implemented. However, the disadvantages include diminished effectiveness of the cache since it has to store uncompressed instructions. In addition, nonsynchronized instruction addresses in main memory and in the cache since main memory stores variable-length instructions and the I-cache stores fixed-length instructions. This causes contiguous addresses in main memory to be noncontiguous in cache and requires a more complicated decoding scheme to get to the next instruction [10].

▲ Decompression on instruction fetch. Here, main memory and I-cache are both compressed. The advantage is that the cache utilization is optimal and the address translation mechanism is simple. The disadvantages are that the decompression is on the critical path of the instruction execution pipeline since it has to be executed at every I-cache hit. This means a few decompression stages are very likely necessary before decoding the instruction. This increases the cycle penalty for taken branches. Also, the hardware is likely to be more expensive due to the fact that it is on the critical path.

Applications and Coding

DSP application development has historically involved the use of a high-level programming language along with system and library calls for the parts of the application that deal with data management and partitioning, I/O, control flow, communications, and possibly system interface. For the actual data-processing kernels, however, the programmer has had two choices:

1. Using pre-existing libraries, when available, or else

2. Coding those kernels in the assembly language of the DSP.

The second option typically requires direct interaction with architectural details of the processor's internals: the allocation of registers to values, exploitation of special operations (e.g., multiply/add, accumulation, packed and saturated arithmetic), synchronization, etc. These details are often very complex to code and, unfortunately, changes in processor architecture, even within DSP families, cause most of the effort to be discarded. These complex features and asymmetries have historically been too difficult for compilers to handle well. As a result, compilers have traditionally been unable to generate code even nearly as efficient as hand-optimized assembly.

Application partitioning is another area of DSP programming difficulty. In some application areas, using shared memory, message passing, or both has sometimes resulted in a great deal of speedup due to coarse-grained parallelism in multi-DSP systems. But it has posed serious limitations as well for those applications that tend to have a more easily exploited fine-grained parallelism and are not usually easy to partition on a coarse level.

As pointed out earlier, an advantage of VLIWs used as DSPs is that there exist compiler techniques which, though complex, allow the programmer to write effective code in a high-level language. The advantages of this are obvious when compared to the use of hand assembly code:

▲ The code is usable across different members of the same DSP family.

▲ To a great extent, the code is effective even when changing DSP families.

▲ Flexibility, code reusability in other applications, and maintainability are greatly improved.

▲ There is little penalty (and often an improvement) in performance.

The state-of-the-art in compiling for DSPs in the late 1990s does not yet place the programmer off the hook completely. The programmer still has a role in meeting these goals. Sometimes the differences from the usual software engineering are few, but often more radical solutions have to be used. However, unlike previous alternatives, the work is done in a high-level language like C or FORTRAN, with the advantages listed above. At the outset, the programmer must have the mindset that, in the VLIW world, performance is achieved by exposing ILP. This leads to different programming solutions that depend upon the structure and needs of the specific applications. Considerations like code transformation, data partitioning, memory bandwidth analysis, dependence analysis, variable localization, memory disambiguation, and many others, play an important role to maximize performance. Sometimes the programmer provides additional information or hints to the compiler in the form of pragmas, or structured comments in the high-level language that the compiler understands. D-cache (if one is present) performance optimization and coding style are
also related. Again the compiler can be instructed to use different addressing modes for different kinds of data to reduce D-cache misses or to preserve D-cache/memory coherence when needed.

In general the first steps in generating code involve profiling and application analysis. It is important to use good visual tools for this, including:

▲ Profilers
▲ Tools that help the programmer visualize the flow of control (usually represented as a control-flow graph, or CFG).
▲ Tools that help the programmer visualize the data dependencies (usually represented as a directed acyclic graph, or DAG).

This can lead to a thorough understanding of two important aspects of the application: in which (usually small) kernels of code the time is spent, and the structure of those kernels. From the point of view of ILP, these computational kernels can be classified into two broad categories according to the complexity of the control flow graph and the parallelism evident on the surface:

▲ Regular structure, for which the compiler can usually expose the existing ILP very effectively, provided that the programmer used some guidelines when writing the code. Examples include mono and multidimensional transforms (FFT's, DCT's, wavelets, etc.), filtering, color space conversions/calibrations, pattern matching, motion detection, image warping, etc. Often, these kernels have data-independent control flow (e.g., an FFT computation control flow doesn't depend on the data being transformed, at least for a classical implementation), and thus contain almost unlimited ILP.
▲ Irregular structure, in which a great deal of effort is typically required to make any progress.

Although regularly structured algorithms are very common, we are not always so lucky, especially when starting from an already existing implementation. Often, the control flow is complex and data dependent and the available ILP is not very high. In those cases, the programmer has many choices for solving the various problems; however, each problem must be faced taking its relative importance into account.

Coding Style

When porting or writing new code for any DSP, the specific coding style appropriate for that architecture should be used. In the case of VLIW DSPs, there are some guidelines that ought to be followed in order to ease the compiler's task by exposing ILP.

Many of the following guidelines are related to each other. For example, when optimizing loop execution it is often the case that one must eliminate or convert all the "if" statements present in the loop body, or one must allocate values and data by partitioning the memory hierarchy in some way specific to that loop.

Variable Localization

Variable localization is the process of selecting a storage location for values needed by a given computational kernel or function. This can make a crucial difference in performance if the value must be accessed many times (i.e., if it is referenced in an innermost loop). Historically this has been a well-known problem for standard DSPs and processors, but the difference now is that different levels of memory (registers, internal, external, etc.) give more choices to the programmer than simply declaring a variable to be a "register."

One of the main limits to ILP is the memory disambiguation capability of the compiler. Imagine a global variable being accessed in a critical section of the code. Without an extensive interprocedural analysis (not always possible or even feasible) it's not easy to understand if the variable value gets modified (e.g., by a function call). What happens is that the compiler most of the time has to be conservative and doesn't copy the value to a local temp (a register), but it generates accesses to it all the time its value is needed. This is only a simple example of what can happen, but generally speaking memory disambiguation is a key point in exposing ILP. Later, we'll see how giving "hints" to the compiler could solve it.

When possible, local variables should be used since (at least in C) they are semantically on the stack, so their addresses cannot conflict with global variables. This will help the compiler to allocate those variables into registers and to identify independent memory operations. Also, the coding style can be somewhat different from the usual C programming style. In fact the availability of a great number of registers may affect dramatically the way some algorithms can be implemented. We'll see how this may become really important in many cases and we'll show an example of a possible use of the available registers to implement an 8x8 DCT.

Converting IFs to SELECTS

As an alternative to predication, a compiler can usually convert simple IFs into something called SELECT operations. This can make it possible to pipeline the loop containing them and can often increase the available ILP by a large factor. For this purpose, however, it is necessary to transform conditionals so that they operate and write to local data and then assign the result to the memory location. The reason for this is possible side effects. C pointers may point to any region of memory, so the compiler may not be able to determine if dereferenced pointers in the code are actually pointing to aliased memory locations. To generate correct code, the compiler must be conservative and issue a conditional branch, thus maintaining the correct order of evaluation.

The user may know this is not the case and modify the code as in the following example. If the IF is used to conditionally assign values to group of variables or memory location, then the compiler cannot convert the IF to a SELECT (especially if the IF arms contain memory refer-
ences the compiler isn’t able to disambiguate). In this case it is better to transform the if to a series of conditional assignments, as in:

```
if (ifexpr) {
    a = aexpr1;
    b = bexpr1;
}
else {
    a = aexpr2;
    b = bexpr2;
}
```

becomes:
```
cond = ifexpr;
a = (cond ? aexpr1 : aexpr2);
b = (cond ? bexpr1 : bexpr2);
```

Again this enables the compiler to pipeline the loop (remember expressions must not contain memory references). This increase in ILP may or may not lead to faster execution, depending upon the ratio between the improvement obtained by better scheduling the loop and the overhead due to the speculative execution of the exprs. Remaining IFs may be optimized by means of profiling, or when possible they should be eliminated (or moved outside the loop) by using loop transformations and/or data partitioning. Suppose a loop contains an IF that conditionally executes some code depending on the loop trip counter directly or indirectly (through a memory indirection):

```
for i {
    code1;
    if (expr) code2;
}
```

Unrolled, the loop will look like:

```
for i {
    code1; /* i */
    if (expr) code2; /* i */
    code1; /* i + 1 */
    if (expr) code2; /* i + 1 */
    code1; /* i + 2 */
    if (expr) code2; /* i + 2 */
    ............... code1; /* i + N */
    if (expr) code2; /* i + N */
}
```

The problem here is that the compiler has to guess statically the probability of branches being taken for it to make good decisions about code motions. Profile-based recompilation may help, but even so, if the statistical probability of code2 being executed is around 50%, then the compiler can generate optimized code, but it will be executed only part of the time. The rest of the time, the code will generally have a low degree of ILP, since presence of the IF prevents code motion and ultimately ILP exposition. In the end, very little improvement or no improvement at all may occur. Alternatively, if the probability in one direction is low, the compiler may be still able to compact the unrolled loop by assuming the branch always goes in the predicted direction, as in Fig. 6.

![6. One of the possible control flows for the previous example unrolled multiple times, where the probability of the branch being taken must be much lower than the fall through. In the case of miss-prediction, the penalty will be high because the compiler has to insert compensation code to compensate for the code motion across branches.](image)

Indeed, the condition evaluation is always carried out, but now code motion can occur above and under branches. The price to pay here is higher in the case of branch misprediction, because compensation code must be inserted to undo the operations being moved above the branch. To avoid this drawback or to convert the IF in those cases where the branch statistics are balanced, it is possible sometimes to split the loop into two (or more) separate loops without IFs. For example using data partitioning and precomputing the condition for each loop before execution and storing the results into intermediate arrays), we have:

```
k = j = 0;
for i {
    f = expr;
p = Part2[k];
Part2[k] = (f ? i : p);
k = (f ? k+1 : k);
p = Part1[k];
```
Part1[k] = (f ? p : i);
    i = (f ? j : j+1);
}
for i {
    j = Part1[i];
    code1;
}
for i {
    j = Part2[i];
    code1;
    code2;
}

The two arrays, Part1 and Part2, are partitions of the 
loop trip counter range, for cases when only code1 is ex- 
ecuted or when both code1 and code2 are executed respec-
tively. Even if there is some dependence between dif-
ferent iterations of the loop, this may nonetheless at-
tain a good amount of parallelism, depending on how 
complex "expr" is to evaluate. The two loops replace the 
original loop.

Now the two loops can potentially be scheduled in a 
more efficient way because there are no longer any 
branches in their bodies. As to the previously mentioned 
relations between different optimizations and bottle-
necks, note that the above methods might potentially 
have negative effects on the D-cache performance. In fact, 
if the original loop was sequentially accessing (streaming) 
data in main memory, now the accesses will show non-
sequential behavior and possibly even nonconstant strides. A possible solution to this problem is prefetching, 
as we will see later on.

**Partitioning Data in the Memory Hierarchy**

In C every memory reference is (semantically) in the same 
memory space. However, as we saw earlier, there are no 
problems involved in extending the physical memory to 
different levels. For example a VLIW DSP may have on-
chip memory as well as interfaces to private memory 
buses for specialized storage or I/O operations.

Specifying that a given data structure should use some 
partial class of memory is not possible within legal C 
programming, except when using dedicated libraries and/or system or hardware support. Even in this case, the 
granularity of memory operations performed on the special 
memory is necessarily coarse to avoid the huge over-
head of calling a function for every memory access. So the 
use of such interfaces in C is often limited to system-level 
block I/O and little more. In order to exploit effectively 
aditional memory levels, and thus expose more ILP, the 
compiler should be able to address different kinds of 
memory directly and schedule for their respective laten-
cies. One possible solution is to use code annotations to 
inform the compiler of the physical location of data and 
how to access it. From the programmer's point of view, it 
is a matter of analyzing the application needs in terms of 
bandwidth and the spatial and temporal locality of mem-
ory accesses.

DSP applications (especially multimedia) show very 
mixed sets of data-locality characteristics. Typically, one 
or more main streams of data (the image or signal) are 
processed by the DSP, which may have only spatial local-
ity. Those memory accesses often completely disrupt the 
data cache, polluting other data structures having tempo-
ral locality.

To avoid D-cache pollution and to reduce the memory 
latency and increase bandwidth, the compiler can gen-
erate different memory accesses for streaming data out of 
main memory and for accessing data with temporal local-
ity. Even more ILP can be attained if the DSP has an in-
ternal very low latency memory. Necessarily, this 
memory will be limited, but it can be very useful for stor-
ing static (and sometimes dynamic) structures such as 
look-up tables, arrays of temporary variable, and so on. 
The memory latency is completely exposed to the com-
piler, so it can take full advantage of it and generate com-
 pact schedule.

An important advantage of having different levels of 
memory is the increased memory bandwidth available. 
When writing an application for a VLIW DSP, many of 
the memory access that for a conventional processor or 
DSP would normally be accessing main memory can be 
turned into access to internal or private (off chip) mem-
ory. It is often straightforward to understand which data 
should be stored where and what kind of access should be 
used. For example, when implementing an FIR filter, the 
main stream of data (either mono or multidimensional) 
will be accessed though streaming hardware support. 
Since the mask and filter coefficients have a high degree of 
temporal locality, they can be stored in main memory and 
accessed through the D cache, or much more efficiently 
be stored in the on-chip memory for faster access. From 
the programmer's point of view, this doesn't require deal-
ing with obscure assembly features or library calls, but it is 
plain C code with some annotations for the compiler. For 
example, a code for a mono-dimensional FIR filtering is:

```c
#pragma local_memory
int Mask[N];

void FIR_mono(int *src, int *dst, int len, int mask[n])
{
    int i,j;
    int sum;
    #pragma stream
    int *Src,*Dst;

    Src = src + N/2;
    Dst = dst + N/2;
    copymask(Mask,mask); /* copy filter
    coeff from main to
    on-chip memory */
```
for (i=0; i < len; i++) {
    dst[i] = src[i] + src[i+1];
}

This directive will have a loop-wide scope. If indeed there are real dependencies between some pair of pointers, then it cannot be used. An alternative solution in this case would be to use assertions (such as, say #pragma assert_no_alias(dst, src)) for only those pointers that do not alias. Usually this will be an unsafe optimization for the compiler to carry out; thus the need for the programmer to specify it.

Another way to remove memory disambiguation problems for the compiler and expose more ILP is via taking advantage of the large number of registers generally available in a VLIW. This could be achieved by transforming the loop; we’ll see an example in the next section.

**Source Transformations**

FORTRAN is much more suited to the application of extensive automatic source-code transformation than C. This is due to many reasons, and we refer the reader to [42] for more extensive discussions on this subject. However, even with C, the programmer may apply source transformations that expose ILP to the compiler. We have already seen examples of source transformations. There are many more types, but in general these transformations usually modify:

- The storage of data (and the way it is accessed)
- The program’s control flow
- Loop order

The proper transformation to be applied depends upon many different factors—it is very hard to give firm guidance to the programmer on this. Worse still, some transformations are beneficial in some ways and potentially degrade performance in others. Clearly, the application of transformations is something of an art. That notwithstanding, we will show how two in particular may be of great importance for a VLIW DSP: loop unrolling and loop jamming or fusion.

**Using Extra Registers to Reduce Memory Traffic**

Loop unrolling, as already addressed, is an effective way to expose more ILP. Particularly when used in combination with IF conversion, memory disambiguation, etc., unrolling yields bigger basic blocks (or traces), giving the compiler more opportunity to perform efficient scheduling via code motions.

Software pipelining (discussed earlier) is often an alternative or complement to loop unrolling. Especially when a processor has register-renaming support, it can be a very effective way of representing loops very efficiently, with extremely good use of the available hardware. Even when software pipelining itself is not appropriate, there is an alternative technique that very often helps in disambiguating memory as well as improving ILP—especially in the absence of register renaming. This technique involves using registers as a kind of memory storage. Consider one of the most extensively studied (and implemented) multimedia operations, the 8 x 8 quantized in-
verse DCT—the forward version would behave similarly. It is the most compute-intensive operation in DCT transform-based image and video compression algorithms such as JPEG, MPEG, H261 [16-20], and others. Although 2D native versions of it have been derived [13], the most used schemes exploit the separability of the 2D-DCT transform kernel by carrying it out through 1D-DCT on rows and columns. A classical implementation will use a matrix to store the initial quantized DCT coefficient values and the intermediate results, as well as the quantizer steps values. The control flow will have three or more loops, one performing dequantization of DCT and two carrying out the row and column 1D-IDCT.

The amount of available ILP in all three of these loops may be fairly large—it might completely use all VLIW resources. Unfortunately, the extensive memory bandwidth required either by DCT coefficients and dequantization matrix entries will probably negate this. Additionally, memory disambiguation problems may arise, depending upon the specific implementation. To actually get the most out of this code we can observe that the number of registers needed to store DCT coefficients is 64 and that the computation could be done in place into those registers as previous values are never reused (see Fig. 7). Also we can observe that if the application requires that we perform an inverse transform on multiple blocks, as is often the case, the dequantization matrix usually stays the same for all of them at least in most, though not in all, compression schemes. Therefore, it can be efficiently stored into the internal on-chip memory or, better still, into 64 other registers if the overall number of registers suffices. This is feasible only because the control flow of the IDCT does not depend on the data, and even more so because the relations between the elements of the matrices used to store them stay fixed (for a given IDCT algorithm the elements involved in a given butterfly at a given stage are always the same). This means we can associate matrix elements to register values with a one-to-one correspondence. To do that we need to fully unroll all the three loops and, of course, have enough register to hold all the required values (including temps required to carry out expression computation). Otherwise we will incur undesirable amounts of register spilling to memory.

The advantages of doing this are twofold. First, we completely remove any memory read/store for intermediate results between reading in DCT coefficients and writing out pixels values, effectively reducing the number of operations but also avoiding all memory disambiguation problems (namings values directly through registers does not lead to any ambiguity). This also has the effect of eliminating memory latency from the picture. The second important advantage is the complete elimination of pointer arithmetic to address values if matrices were used. Indeed, the complexity of algorithms as typically defined is only a part of the operations that must be done—pointer arithmetic sometimes accounts for a significant number of operations too. This implementation of the IDCT can easily improve performance from 2-10x on the same VLIW architecture, depending upon the memory system speed, the number of registers, the width of the machine (number of simultaneous operations that can be performed in one cycle), and so on. Figure 8 shows two possible schedules for an 8 x 8 decsalted IDCT along with de-zigzag and memory-related operations (coefficient loads and pixel output) being part of the JPEG image-compression standard.

The first schedule (left side of Fig. 8) is obtained by optimally (in a minimum cost sense) preallocating opera-
tions to computational units in order to minimize intercluster copies, while the second schedule (right side of Fig. 8) uses a more conventional heuristic allocation strategy to perform the allocation. The results show clearly that the first schedule is more compact due to the reduced amount of communication between clusters (compare the number of magenta blocks). This shows that to effectively schedule for DSP applications, classical compilation and scheduling techniques must be sometimes revised.

Loop Jamming or Fusion

Sometimes, dependencies are real in the sense that they cannot be eliminated (for example, in an IIR filter). At those times, a potentially helpful technique can sometimes be applied when the kernel is part of a computa-
tional pipeline made up of different stages. If, after all other improvements have been made, each pipeline stage still leaves significant extra available ILP, it may be possible to jam or fuse together all or part of those stages through loop jamming or "fusion." This technique can also be generalized to control flow-graph fusion and can even be automated if the functions used fit an appropriate model (that is, they have a producer/consumer relationship) (see [22] and [36]).

As a simple illustration of loop jamming, we’ll consider a simple 3 taps IIR monodimensional filter:

```c
void IIR_3(int *data, int len, 
    int m1, int m2, int m3)
{
    int i,j;
    int sum;
    #pragma stream
    int *Data;
    int v1, v2, v3;

    Data = data + 1;
    len = len - 2;
    v1 = data[0]; v2 = data[1];
    for(i=0;i<len;i++) {
        v3 = Data[i+1];
        sum = v1 * m1 + v2 * m2 + v3 * m3;
        Data[i] = sum;
        // next iteration v1 value has been modified by the line above */
        v1 = sum;
        v2 = v3;
    }
}
```

Notice that we have already used the register storage concept to avoid unnecessary memory read operations (this is called redundant load elimination), and, indeed, the implementation of this filter requires reading each input value only once. However the available ILP will not be very high (at least for a wide enough VLIW). Unrolling the loop will not help because the dependency across loop iterations is real this time and we cannot apply the memory disambiguation pragma to this example. But if this filter is followed or preceded by some other kernels, then we may be able to apply loop jamming. For the sake of simplicity, suppose we have to apply a cascade of two IIR filters. In that case, a jammed implementation of them may look like:

```c
void IIR_IIR_3(int *data, int len, 
    int m1, int m2, int m3, int n1, int n2, 
    int n3)
{
    int i,j;
    int sumv, sumw;
    #pragma stream // assuming iteration-level prefetching
    int *Data;
    int v1, v2, v3;
    int w1, w2, w3;

    Data = data + 2;
    len = len - 3;
    v1 = data[0]; v2 = data[1];
    // three peeled up iterations of the loop are required to produce w1, w2 and w3, and v1, v2 and v3 */
    i = 0;
    ...
    ...
    for(; i < len; i++) {
        v3 = Data[i+1];
        sumv = v1 * m1 + v2 * m2 + v3 * m3;
        v1 = sumv;
        v2 = v3;
        sumw = w1 * n1 + w2 * n2 + w3 * n3;
        Data[i] = sumw;
        w1 = sumw;
        w2 = w3;
        w3 = sumv;
    }
}
```

Notice that the dependency is still there, but the number of operations inside one loop iteration is doubled. What’s more, they are independent and can be carried out in parallel. One additional advantage of loop fusion is that intermediate storage is not needed and a memory read/store can be avoided. This can result in a huge benefit in the presence of high D-cache miss ratios when many stages (and many temporary buffers) are present.

As already mentioned, there is a broad class of functions falling into the category of producer/consumer, or, more generally, "filters." For those functions the process of loop jamming may be extended and automated to the whole control flow graph of the functions involved. The advantages of doing this automatically are many, not least the reusability of the single functions. In fact, if loop jamming is done by hand, inevitably it will incur drastic changes as soon as one of the loops involved changes significantly. Figure 9 shows the control flow graphs for the previous examples (the fused CFG was generated by an automatic tool based on concepts found in [36]).

**Profile-Driven Restructuring**

Further optimizations can be achieved by using profiling information. By profiling we mean not only the classic profiling of function calls and relative statistics (number of calls, time spent) and the construction of the call graph, but also more advanced statistics, such as the temporal/spatial/value locality of data.
Classic profiling can be used to improve static branch prediction. An especially useful technique is called “path profiling” [14], in which the relationships between the directions of different branches are determined at profile time. Using these techniques, the compiler can do speculative code motions and other optimizations in a way that is more likely to be profitable. This is especially useful for applications with complex control flow. Profiling done appropriately, can also allow an analysis of data locality, which can then lead to more effective application restructuring. Some data structures may have very high spatial locality but at the same time be too large to fit into the on-chip memory. If the D-cache is not present or already heavily used, it can be useful to modify the application by using a software data cache for that data structure. A profile of the application can often determine the likelihood of the data being in the software cache, which could then be stored in a fast low-latency on-chip memory. The miss ratio versus the ILP increase due to the faster memory accesses will determine the threshold for the effective use of this approach. Value locality can also be exploited. For example it is well known that quantized DCT coefficients distribution is closer to zero for high frequencies (this principle is exploited in compression algorithms). In order to speed up execution it is possible to classify DCT blocks into N categories, with each one having zero-valued coefficients in given positions. Multiple implementations of the IDCT can be done in order to save operations involving coefficients having a zero value. Then to perform the inverse transform of multiple DCT blocks the appropriate IDCT function will be used. The advantage is that most of the time the version of the IDCT with many zero coefficients will be used, speeding up execution significantly.

This IDCT implementation is known as statistical IDCT; however, with a classic processor or DSP, one can rarely produce such a big improvement because the implementation requires many branches to check for the coefficients being zero valued.
Summary

The techniques presented here are often complex and sometimes require expert knowledge of the application at hand. However, there are several points that are important to remember. These techniques are not always necessary, and, in any case, a VLIW DSP microprocessor will perform well without them—they are there to allow the application to be represented in a high-level language, which is good for many reasons, not the least of which is scalability across different models of a DSP family, as well as across potentially very different DSP chips altogether. Finally, the effort involved in doing this cannot be compared to the Herculean effort required to write DSP assembly code.

These techniques can produce enough performance to deliver on the promise of a single chip, programmed in a high-level language, that does the DSP processing on the main CPU—a tremendous advantage over older ways of doing this processing.

Acknowledgments

We would like to acknowledge the tremendous help given by the editor of the special issue, as well as our reviewers, especially reviewer "number T."

Paolo Faraboschi, Giuseppe Desoli, and Joseph A. Fisher are research scientists with Hewlett-Packard Laboratories Cambridge in Cambridge, Massachusetts.

References


19. ISO/IEC 11172, Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, 1993.


