SIMD and other optimisations in Vector Pascal

Paul Cockshott
Greg Michaelson

December 12, 2002

Abstract

Despite the widespread adoption of parallel operations in contemporary CPU designs, their use has been restricted by a lack of appropriate programming language abstractions and development environments. Vector Pascal is a language designed to enable the elegant and efficient expression of SIMD algorithms. It imports into Pascal abstraction mechanisms derived from functional languages, in turn having their origins in APL. In particular, it extends all operators to work on vectors of data. Arithmetic is extended to support multi-media datatypes. Code generation is via the ILCG system that allows retargeting to multiple different SIMD instruction sets based on formalised descriptions of the instruction set semantics. We discuss some of the specific program transformations required for the efficient generation of SIMD Pascal code.

1 Background

Vector Pascal, is a Pascal implementation that has been extended to make efficient use of multi-media instruction-sets.

Pascal is a classic imperative language whose first implementations were on CDC super-computers[17]. It has been an important language both in its own right, and as an influence on subsequent computer languages such as Ada and Modula[28]. It was originally developed for instructional purposes, and widely taught as a first programming language.

The British Standards Institute started work on standardising the language in the late 1970s, this led to a British Standard BS6192 for the language in 1982. The following year International Standard 7185, published by the ISO incorporated the British Standard for Pascal with minor amendments. This stanardisation process conicided with a rapid growth in the use of the language. Its early implementation on microprocessors, gave it a large following among both Apple and PC programmers.

As a growing body of commercial code started to be developed in Pascal, weaknesses of the standard, particularly in the areas of support for separate compilation, variable sized arrays and in string handling became evident. These were addressed in two ways. On the one hand commercial implementations, in particular the influential Turbo-Pascal, on the PC adopted a separate compilation system based on Units, and a system of length encoded strings. On the other hand the standardisation effort led to International Standard 10206 :Extended Pascal. Extended Pascal supported modules for separate compilation. It handled strings and variable sized arrays as part of an elegant system of parameterised types termed *schemata*. Today relatively few implementations of Extended Pascal exist [23], [18], whereas many compilers exist that use Turbo-Pascal style strings and Units.

No further Pascal standards have been agreed since 1990. Since then a number of object oriented implementation of Pascal have been released : [16], [23], [3] . In 1993 the Pascal Standards committee produced a draft proposal for object oriented extensions to Pascal, but this has not been formally adopted.

1.1 Multi-media instruction-sets

A number of widely used contemporary processors have instructionset extensions for improved performance in multi-media applications. The aim is to allow operations to proceed on multiple pixels each clock cycle. Such

instructionsets have been incorporated both in specialist DSP chips like the Texas C62xx[26] and in general purpose CPU chips like the Intel PIV[10][11] or the AMD Athlon[1].

These instructionset extensions are typically based on the Single Instruction-stream Multiple Data-stream (SIMD) model in which a single instruction causes the same mathematical operation to be carried out on many operands, or pairs of operands at the same time.

The multi-media instructions on these machines perform operations between vector registers. If u, v are vector registers, with each register having multiple sub-fields, i.e., $u_0, u_1, U2, ...$, then an instruction

```
PADD u,v
```

would produce the effect $u_0 \leftarrow u_0 + v_0, u_1 \leftarrow u_1 + v_1, ...$

Current SIMD instructions for multi-media applications have vector register sizes ranging from 32 bits (Texas C62xx) to 128 bits (Intel PIV, Motorola G4). The vector elements range from 8-bit integers to 64-bit floating point numbers. Intel and AMD processors support saturated integer types which are designed to obviate the worst effects of overflows when using limited precision arithmetic.

Suppose we are adding two images represented as arrays of bytes in the range 0..255 with 0 representing black and 255 white. It is possible that the results may be greater than 255. For example 200+175 = 375 but in 8 bit binary

```
      11001000
+     10101111
=   1 01110111
```

droping the leading 1 we get $01110111 = 119$, which is dimmer than either of the original pixels. The only sensible answer in this case would have been 255, representing white.

To avoid such errors, image processing code using 8 bit values has to put in tests to check if values are going out of range, and force all out of range values to the appropriate extremes of the ranges, see figure 1. This inevitably slows down the computation of inner loops. Besides introducing additional instructions, the tests involve conditional branches and pipeline stalls. Intel processors provide saturated arithmetic operations on the vector registers which keep arithmetic results within bounds. When implemented within the ALU hardware, saturated arithmetic can execute in a single clock cycle. The combined

```
main()
{
unsigned char v1[LEN],v2[LEN],v3[LEN];
 int i,j,t;
 for (j=0;j<LEN;j++){
  t=v2[j]+v1[j];
  v3[j]=(unsigned char)(t>255?255:t);
 }
}
```

Figure 1: C code to add two images. Code compiled on the Intel C compiler version 4.0.

```
l1: movq mm0,[esi+ebp-LEN]
    paddusb mm0,[esi+ebp-2*LEN]
    movq [esi+ebp-3*LEN],mm0
    add esi,8
    loop l1
```

Figure 2: Assembler version of the test program. The example assumes that registers `esi` is initialised to 0 and that `ecx` is initialised to LEN/8.

effect of the use of packed data and saturated types can be to produce a significant increase in code density and performance.

Consider the C code in figure 1 to add 2 images in `v1` and `v2`, storing the result in the image in `v3`. The code includes a check to prevent overflow. Compiled into assembler code by the Intel C compiler the resulting assembler code has 18 instructions in the inner loop whereas the hand coded assembler inner loop in figure 2 uses only 5 instructions. Furthermore, the MMX code processes 8 times as much data per iteration, thus requiring only 0.625 instructions per byte processed. The C code executes 29 times as many instructions. Whilst some of this can be put down to the superiority of hand assembled versus automatically compiled code, the combination of the SIMD model and the saturated arithmetic are obviously a major factor.

1.2 Programming multi-media SIMD

A number of C programming systems have been targeted at multi-media instructionsets. These take two general approaches:

1. Features of the low level machine architecture are made directly visible. This is done either by allowing assembler macros in C code or by allowing new data types are declared which directly model the data types held in the vector registers. For example,Intel supply a C compiler that has low level extensions allowing the extended instructions to be used. Intel terms these extensions 'assembler intrinsics'. The Intel C compiler also comes with a set of C++ classes that correspond to the types held in the MMX and XMM registers. Apple support a release of GCC that has similar vector data types for the G4 vector instructions.

   This approach leads to efficient code, but the incursion of assembler concepts into the high level language impedes portability between processors.

2. Other C compilers[4] [20] [19] [2] [25] have used classical vectorisation techiniques. Here, the compiler detects potential parallelism in ordinary C loops and attempts to map them onto vector operations. All of these compilers except [20] target specific architectures. Leupers has reported a C compiler that uses vectorising optimisation techniques for compiling code for the multimedia instruction sets of some signal processors, but this is not generalised to the types of processors used in desktop computers.

   Nonetheless, these techniques are in principle machine independent. They are effective at recognising straightforward array arithmetic, and in some cases can recognise and vectorise reduction constructs like dot-product. They can have difficulty in recognising constructs involving saturated arithmetic, since there is no standardised way to represent this in C.

1.3 Array Languages

High level language notations for expressing data parallelism have a long history, dating back to APL in the early '60s[14]. The key concept here is the systematic overloading of all scalar operators to work on arrays.

The most recent languages to be built round this model are J, an interpretive language[15], ZPL [24] an array language for multi-processors and F[22] a modernised Fortran. Unlike the SIMD classes provided by the Intel C++ compiler, the array languages are machine independent. They can be implemented using scalar instructions or using the SIMD model. The only difference is speed.

2 Vector Pascal

Vector Pascal takes Pascal as the base language and extends it to support data parallel operation by the systematic overloading of operators. The aim of its development has been:

1. Provide a means of expressing data parallel operations that was independent of the instructionset available.

2. To provide a mechanism for automatically generating good machine code for data parallel operations.

3. To provide a means by which the code generator can be automatically retargeted to different SIMD and non-SIMD instructionsets based on a formal description of these instructionsets.

4. To provide a toolkit that was operating system and machine independent, (it runs under Windows and Linux).

5. To produce at least two code generators one for a scalar instructionset and one for a SIMD instructionset (there are currently 5 machines targeted).

Vector Pascal is described in [6], [7], [8], so we will give only a summary account here. It constitutes a Pascal base language with array extensions. It incorporates many, but not all, features of Extended Pascal. The supported features are given in table 1 In order to ease software migration the base language designed to be upward compatible with Turbo Pascal, so the Turbo Pascal Unit system and string representations are used in preference to those given in [13].

Table 1: Extended Pascal Features Supported

Schematic arrays
Otherwise in case statements
Protected Parameters
Sets of arbitrary size
Extended succ and pred
Non decimal integer literals
Complex numbers
Relaxed declaration order
Compile time constant expressions
Operators **, POW, ><
Functions return any type
For i in s do

The extended features added by Vector Pascal are summarised in table 2. It can be seen that these go somewhat beyond those required for SIMD applications.

Table 2: Vector Pascal Extensions

Overloading of all operators to array types
Functions map over arrays
Matrix transpose operator
Array permutation operators
Array slices
Generalised reduction operations
Saturated arithmetic operators +:, -:
Operators MIN, MAX
Conditional expressions
Operator overloading
Polymorphic functions
Dimensioned types
Pixels as fixed point type
Input and output of scalar types
Input and output of arrays
Optional garbage collection
Literate programming support

3 Performance

Figure 3 shows a Vector Pascal program equivalent to those shown in figures 1 and 2. The vector intention of the operation is clearer, and more concise. The assembler code generated is also shown. Loop unrolling makes it look verbose, but only 0.537 instructions are executed per byte processed, compared with 0.625 for figure 2 and 18 for figure 1. Performances on this, admitedly trivial, example are shown in table 3 which compares Vector Pascal with the assembler and C versions run on different compilers.

It can be seen that the Vector Pascal version is substantially faster than the hand-written assembler and much faster than C.

Table 3 compares performance on a simple vector operation between Vector Pascal and implementations in other languages. Table 5 presents performance data on kernels using a number of different data-types. For comparison the performance of a number of other Pascal compilers is presented.

In these tests the clock speed $c = 1$Ghz is held constant, and the number of base operations is known for each row of the table. All figures are in terms of millions of base operations per second measured on a 1 Ghz Athlon. Different rows of the table have different effective data type

Table 3: Speeds of different implementations

Implementation	Iteration time μs	MOPS
Vector Pascal	2.9	2207
Assembler	5.2	1250
Intel C Ver4	53	120
gcc	131	48

Timings on a 1Ghz Athlon. Measured over 100000 iterations of inner loop. Both Vector Pascal and the Intel C compiler targeted at the Pentium with MMX.

widths w. Variations in speed going down a column show the effects of w, and also measure the relative efficiency, u, of the compilers for different data types.

The bit width of the registers available b, varies between the columns since one compiler was targeted on the 286 instructionset giving $b = 16$, another was targeted on the K6 instructionset with $b = 64$ and the others on the 486 instructionset with $b = 32$. The resulting variations in performance along the rows measure the effect of b and u varying between the compilers.

It can be seen that the combined effects of variations in bu can ammount to a performance variation of 100 to 1 along the rows.

4 Implementation

The Vector Pascal compiler is written in Java. The lexical analyser is built using JLex. The syntas analyser uses the classic recursive descent parsing techniques employed in Wirth's original Pascal compiler. The syntax analyser builds an abstract program tree which is passed to a code generator built using the ILCG code-generator-generator system[5].

The code generators follow the pattern matching approach described in [9], and are automatically generated from machine specifications written in ILCG . ILCG is a strongly typed language which supports vector data types and the mapping of operators over vectors. It is well suited to describing SIMD instruction sets. The code generator classes export from their interfaces details about the degree of parallelism supported for each data-type. This is used by the front end compiler to iterate over arrays longer than those supported by the underlying machine. Where supported parallelism is unitary, this defaults to iteration over the whole array.

Selection of target machines is by a compile time switch which causes the appropriate code generator class to be dynamically loaded. The structure of the Vector Pascal system is shown in figure 7.

The path followed from a source file is:

- The source file (1) is parsed by a java class PascalCompiler.class (2) a hand written, recursive descent parser, and results in a Java data structure (3), an

```
program vecadd;
var v1,v2,v3:array[0..6399]of byte;
begin
      v3:=v1 :+ v2;
end.
labelee832b2b32a:
 cmp DWORD ebx,      6399
 jg near  labelee832b2b32c
 movq MM4, [  ebx+v1]
 paddusb MM4, [  ebx+v2]
 movq  [  ebx+v3],MM4
 lea ebx,[  ebx+     8]
 movq MM4, [  ebx+v1]
 paddusb MM4, [  ebx+v2]
 movq  [  ebx+v3],MM4
 lea ebx,[  ebx+     8]
 movq MM4, [  ebx+v1]
 paddusb MM4, [  ebx+v2]
 movq  [  ebx+v3],MM4
 lea ebx,[  ebx+     8]
 movq MM4, [  ebx+v1]
 paddusb MM4, [  ebx+v2]
 movq  [  ebx+v3],MM4
 lea ebx,[  ebx+     8]
 jmp  labelee832b2b32a
labelee832b2b32c:
```

Figure 3: Example program in Vector Pascal, including the assembler produced for the inner loop. The :+ operator performs saturated arithmetic.

ILCG tree, which is basically a semantic tree for the program.

- The resulting tree is transformed (4) from sequential to parallel form and machine independent optimisations are performed. Since ILCG trees are java objects, they can contain methods to self-optimise. Each class contains for instance a method `eval` which attempts to evaluate a tree at compile time. Another method `simplify` applies generic machine independent transformations to the code. Thus the `simplify` method of the class `For` can perform loop unrolling, removal of redundant loops etc. Other methods allow tree walkers to apply context specific transformations.

- The resulting ilcg tree (7) is walked over by a class that encapsulates the semantics of the target machine's instruction-set (10); for example Pentium.class. During code generation the tree is further transformed, as machine specific register optimisations are performed. The output of this process is an assembler file (11).

- This is then fed through an appropriate assembler and linker, assumed to be externally provided to generate an executable program.

4.1 Parser extensions

The parser has to slightly modify standard recursive descent techniques[27]. The modifications arise from additional context sensitive features of the grammar and from the desire to collect information during the parsing that will aid vectorisation. Normally one would have a parameterless recursive procedure for each non-terminal of the language. Code generation would occur as a side effect. We have modified this technique as follows:

1. The procedures are replaced with recursive functions each of which returns an ILCG tree, or throws an exception on a parse error.

2. For expressions each parsing function is passed a vector $[i, j, k, ...]$ whose elements are implicitly declared variables to be used in indexing arrays.

3. A number of global variables are used to define the context within which an expression is being parsed.

4. A number of global flags and counters are used to collect information for vectorisation.

4.2 Array expressions

Vector Pascal allows statements of the form:

```
a:=b * c;
```

where

```
var a,b: array[1..n,1..m] of real;
c: array[1..m] of real;
```

The function that parses assignment statements, after having recognised the sequence `a:=` will inspect the type of the object defined by the ILCG tree for `a`. Since this is a two dimensional array, it will declare a couple of hidden variables i, j to be used in evaluating the expression on the right. The variable `a` is then replaced by `a[`i, j`]`. The indices, along with 2, the rank of the array, are then passed to the function `Node expression(int rank, Node[] indices)`.

If `expression` encounters a variable `x` it applies the following rules:

1. if x is of rank > 2 a type error is throw.

2. if rank(`x`)$= 2$ it rreplaces it with an ILCG tree for `x[`i, j`]`.

3. if rank(`x`) = 1 it replaces it with an ILCG tree for `x[`j`]`.

4. if rank(`x`) = 0 it returns `x`.

The combined effect of this is to replace a:=b*c with `a[`i, j`]:=b[ `i, j`]*b[`j`]`. ILCG is a relatively high level tree language and allows for loops, so the final stage of semantic translation is to expand this to:

```
for i:= 1 to n do
for j:= 1 to m do a[i,j]:=b[ i,j]*b[j]
```

Figure 4: Seive Program, typeset using the VPTEXsub-system of the Vector Pascal Compiler.

```
program seive ( output );
const
   maxlim =10000;
type
   range = 1..maxlim ;
   intset =set of range ;
var
   Let primes ∈ intset;
   Let i, k, j ∈ integer;
begin
   primes← [2..maxlim];
   k← 1;
   for i in primes do
   begin
      j← i × (k + 1);
      while j ≤ maxlim do
      begin
         primes← primes - [j];
         j← j + i;
      end ;
   end ;
   primes← primes + [1];
   for i in primes do WRITELN(i);
end .
```

4.3 Set handling

Pascal is relatively unusual in that it directly supports sets. These are restricted to sets of ordinal types, and are implemented as bitmaps. The grammar for set expression can not be type checked in a context free fashion. Given the following type declarations

```
Type t1=set of 1..10;
t2= set of 5..20;
```

then the set expression `[8,w]-[x,y]` for `var w,x,y:integer` is of ambigous type. This is not a problem in most Pascal compilers which, restrict sets to a relatively small maximum cardinality and represent them with fixed sized bitmaps. The types t1 and t2 would be represented by bitmaps of the same length. Bit 8 would represent the presence of the integer 8 in either set type.

Vector Pascal allows sets of cardinality up to 1+maxint. In this context it is no longer practicable to represent all sets by bitmaps of the same size. In practice the size of the result of a set expression can always be deduced from the assignment context. Given `a:=[8,w]-[x,y],` we can deduce the representation required for the sub expressions from the type of `a`. This prevents unecessarily large bitmaps being allocated and operated on in the expression.

The overall efficiency of set algorithms also depends crucially on the efficiency of the set insertion and deletion operations. These are expressed in Pascal in terms of addition or subtraction of singleton sets. Unless these are recognised as special cases, the compiler will generate code to perform boolean operations on what can be large bit-maps. If the addition or subtraction of singleton sets is recognised as a special case, then the compiler can generate code to toggle an individual bit, which will be much faster for large bitmaps.

Consider the program in figure 4 to compute the first 10,000 primes using an algorithm due to Erastostenes. It makes use of aribitrary sized sets, with the set size given by the constant *maxlim*. The algorithm removes multiples of each prime from an intially full set of primes. The key step is the operation *primes*← *primes* - [j] which subtracts the set [j]from the primes. If implemented naively this involves performing an AND operation between two bitmaps 10,000 bits long. This is clearly inefficient as only one bit in the bitmap has to be cleared. The parser therefore recognises singleton sets as a special case and compiles them differently.

The statement *primes*← *primes* - [j] is translated to:

```
mov eax,DWORD  [j]
sub eax, 1
mov esi, eax
shr esi,3
lea edi,[esi+ primes]
movzx ebx,BYTE[edi]
mov esi,1
mov ecx,eax
and ecx,7
shl esi,cl
xor esi,255
and ebx,esi
mov BYTE[edi],bl
```

This optimisation not only makes set operations very fast compared to other Pascal implementations, but also alters the complexity order of algorithms. Table 6 compares the run times on `seive` of two Pascal compilers: Vector Pascal and Prospero Extended Pascal\footnote{Prospero Pascal is probably the only complete implementation of ISO10206 available for Intel processors. Other Pascal compilers for PC's will generally not handle sets of arbitrary size. }. It can be seen that Vector Pascal is between 40 and 300 times faster than Prospero Pascal. Column 4 of the table shows that for Vector Pascal the algorithm is $< \mathbf{O}n$, whereas Column 5 shows that for Prospero Pascal it is $\approx \mathbf{O}n^2$.

4.4 Schemas and slices

Vector Pascal allows subranges of arrays to be operated on and passed as parameters. One can write convolution statements of the form

```
a[i..j]:= 0.5*(b[i..j]+b[i-1..j-1]);
```

Schematic type definitions of the form :

```
Type matrix(i,j:integer)=
                array[1..i,1..j] of real;
```

are also allowed. This contrasts strongly with Standard Pascal where all array bounds are known at compile time. In Vector Pascal a static array is represented simply by the number of bytes required to store the array being allocated in the global segment or on the stack.

Slices and schematic arrays are both handled by array descriptors. A dynamic array is always stored on the heap. Since its rank is known to the compiler what needs to be stored at run time are the bounds and the means to access it. For simplicity we make the format of dynamic and conformant arrays the same. Thus for schema

```
s(a,b,c,d:integer)= ar-
ray[a..b,c..d] of integer
```

whose run time bounds are evaluated to be 2..4,3..7 we would have the following structure:

address	field	value
x	base of data	address of first element
x+4	a	2
x+8	b	4
x+12	step	20
x+16	c	3
x+20	d	7

The base address for a schematic array on the heap, will point at the first byte after the array header show. For a conformant array, it will point at the first data byte of the array or array range being passed as a parameter. The step field specifies the length of an element of the second dimension in bytes. It is included to allow for the case where we have a conformant array formal parameter

```
x:array[a..b:integer,c..d:integer] of in-
tegeր
```

to which we pass as actual parameter the range

```
p[2..4,3..7]
```

as actual parameter, where `p:array[1..10,1..10] of integer`

In this case the base address would point at `@p[2,3]` and the step would be 40 - the length of 10 integers.

4.4.1 Alignment

A consequence of the use of generalised slices, is that data alignment is not generally known at compile time. The Intel SSE instructionset on the Pentium IV uses 128 bit vector registers. Load instructions exist for both the 16 byte aligned and the unaligned cases. Memory to register arithmetic instructions all demand aligned variables.

Since ILCG neither provides any notation for memory alignment restrictions, nor can these be guaranteed in the presence of slicing, the code generator for the P4 only uses unaligned loads and stores. This entails 2 performance penalties:

1. More instructions are needed as only register to register arithmetic is supported.

2. The unaligned loads and stores are slower than the aligned ones.

The combined effect is to more than halve the speed of vectorised code using 128 bit registers on the P4. Unaligned vectorised code actually runs faster using the 64 bit MMX vector registers than it does in the 128 bit XMM registers. In consequence it has been found preferable to only use the XMM registers for the vectorisation of floating point operations. Their use here both removes a resource clash (the MMX registers are aliased with the FPU stack) and simplifies the caching of floating point variables in registers.

4.4.2 Bounds checking

Pascal mandates that bounds be checked on all array acceses. However, since most array accesses occur in implicit or explicit for loops, there is room for substantial optimisations. If all arrays that will be accesed are known at the start of a for loop it is possible to simply check the loop limits against the array bounds before the loop is entered. The parser maintains a table, `iteratorset`, indexed by loop iterators for each for loop currently being parsed. This maps the iterators to a pair whose frist element is a vector of lower bounds against which the iterator must be checked, and whose second element is vector of upper bounds against which it must be checked. When an array indexing expression is encountered where one of these iterators is the index, an appropriate pair of entries is made in the tables.

A code trees to perform the tests are prepended to the loop tree once the parser exits the loop. Subsequent constant folding and dead code removal result in most bounds tests being optimised away.

Figure 5: Sequential form of array assignment

```
{ var  i;
  for  i=1 to 9 step 1 do {
   v1[^i]:= +(^(v2[^i]),^(v3[^i]));
  };
}
```

4.5 Vectorisation

4.5.1 Array arithmetic

The parser initially generates serial code for all constructs. It then interrogates the current code generator class to determine the degree of parallelism possible for the types of operations performed in a loop, and if these are greater than one, it vectorises the code.

Given the declaration
var v1,v2,v3:array[1..9] of integer;
then the statement
v1:=v2+v3;
would first be translated to the ILCG sequence shown in figure 5 In the example above variable names such as `v1` and `i` have been used for clarity. In reality `i` would be an addressing expression like: `(ref int32)mem(+(^((ref int32)ebp), -1860))`, which encodes both the type and the address of the variable. The code generator is queried as to the parallelism available on the type `int32` and, since it is a Pentium with MMX, returns 2. The loop is then split into two, a portion that can be executed in parallel and a residual sequential component, resulting in the ILCG shown in figure 8.

In the parallel part of the code, the array subscriptions have been replaced by explictly cast memory addresses. This coerces the locations from their original types to the type required by the vectorisation. Applying the `simplify` method of the For class the following generic transformations are performed:

1. The second loop is replaced by a single statement.

2. The parallel loop is unrolled twofold.

3. The For class is replaced by a sequence of statements with explicit gotos.

Figure 6: The result of matching the parallelised loop against the Pentium instruction set

```
 mov DWORD ecx,     1
leb4b08729615:
 cmp DWORD ecx,      8
 jg near  leb4b08729616
 lea edi,[  ecx-(     1)];
 movq MM1, [  ebp+edi* 4+     -1620]
 paddd MM1, [  ebp+edi* 4+     -1640]
 movq  [  ebp+edi* 4+     -1600],MM1
 lea ecx,[  ecx+     2]
 lea edi,[  ecx-(     1)];
 movq MM1, [  ebp+edi* 4+     -1620]
 paddd MM1, [  ebp+edi* 4+     -1640]
 movq  [  ebp+edi* 4+     -1600],MM1
 lea ecx,[  ecx+     2]
 jmp  leb4b08729615
leb4b08729616:
```

4.5.2 Reduction

Reduction operations involve performing arithmetic on vectors to reduce them to scalars. They inject a dyadic operator between the vector elements. The most commonly used reduction operation is probably dot product. Vector Pascal provides a generalise syntactic form for reduction:

`s := RDU` $\oplus e$

where $\oplus$ is some dyadic operator and e is a vector valued expression. Thus one might have

`s:= RDU+ a*b;`

for `a,b:array[0..n] of real; s:real;`

On a machine with vector registers of legnth 4 the parser translates this as follows:

```
{
var t:ieee32 vector(4);
t[0]:=0;t[1]:=0;t[2]:=0;t[3]:=0;
{ var i:int32;
  for i:= n-3 to 0 step -4 do{
  t:=a[i..i+3]*b[i..i+3]+t;
  }
  s:=t[0]+t[1]+t[2]+t[3];
}
```

Since variables `t, i` are declared as local to blocks they are marked by the code-generator as being cacheable in registers, provided that registers of the appropriate sort exist. On a PIII the resulting code, with loop unrolling elided is:

```
 mov DWORD esi,     n
 labeleea15864f9a:
  cmp DWORD esi,      0
 jl near  labeleea15864f9c
 movups  Xmm1,[  esi* 4+ a]
 movups  Xmm2,[  esi* 4+ b]
 mulps  Xmm1,Xmm2
 addps  Xmm0,Xmm1
 lea esi,[  esi -4]
;---- unrolled copies go here
 jmp  labeleea15864f9a
 labeleea15864f9c:
 movups   [  t], Xmm0
 movss  xmm1,[  t]
 addss  xmm1,[  t+4]
 addss  xmm1,[  t+8]
 addss  xmm1,[  t+12]
 movss   [  s],xmm1
```

4.6 Pixel arithmetic

In Vector Pascal pixels are *conceptually* represented as real numbers in the range -1 .. 1, with -1 representing black and 1 representing white, for monochrome images. This representation allows unbiased contrast adjustment and lends itself to the formation of difference images - a common task in image processing.

In the implementation pixels are represtented as 8 bit fixed point signed binary fractions. Addition and subtraction of pixel vectors is performed using saturated arithmetic. Multiplication is more complex. The ILCG specification for the saturated multiply instruction pattern on the MMX is:

```
instruction pattern
PMULLSSB(im8reg m1,
         mrmaddrmode m2,
         mrmaddrmode ma)
/* m1 is an mmx reg typed as
   holding 8 bit integers,
```

```
   m2,ma are either mmx registers
   or memory locations */
means[m1:=*:(octoct(^(m2)),
             octoct(^(ma)))]
assembles[
/* clear registers */
   'pxor MM7,MM7'
'\n pxor MM5,MM5'
/* load 8 bytes as 8 shorts */
'\n punpckhbw MM7,'ma
'\n pxor MM6,MM6'
'\n punpckhbw MM6,'m2
'\n pxor 'm1','m1
'\n punpcklbw 'm1','ma
/* multiply the shorts */
'\n pmulhw MM7,MM6'
'\n psraw MM7,7'
'\n punpcklbw MM6,'m2
'\n pmulhw 'm1',MM6'
'\n psraw 'm1',7'
/* pack result in m1 */
'\n packsswb 'm1',MM7'];
```

4.7 Optimisation cache

Given that the implementation language is Java, and that a code generators use a unification based pattern matcher with backtracking, inspired by Prolog, code generation can be slower than with table driven techniques. To compensate, the codegenerators build optimisation tables at run time to accelerate translation. Two types of table are used :

1. Disjunction Tables: corresponding to each disjunction in the pattern rules for the processor there is a hash table, indexed on strings mapping to the first sucessfully matched term within the disjunction.

   When matching an ILCG sub-tree to a disjunction, the signature of the tree as a string is used to index the hash table, if an entry is found, the term specified is matched, skipping all earlier ones. If no entry is eturned, each term in the disjunction is tried in sequence until a match is found. Once a match is found the hash table is updated.

   The signatures used are lexical flattenings of ILCG trees with literal constants replaced by tokens indicating the type but not the value of the constants. Thus all trees of the form:

   `mem[`x`]:=` y

   for x, y integer literals would map to the same signature string.

2. Statement Table: the disjunction tables are maintained by the machine specific code of each codegenerator class. All code generator classes descend from a base class ILCG.Walker which contains machine independent strategies and tactics used in mapping trees to assembler. The public interface to the code generator is provided by a method `codeGen(Node n)`, exported by Walker. This in turn calls an abstract method `match(Node n)`, which is instantiated by machine specific subclasses. The method `match` can in turn make recursive calls on `codeGen` to handle sub-trees that it encounters or synthesises. The class Walker maintains a master statement table that maps ILCG statements to assembler sequences. Prior to calling match, it checks to see if the same statement has been translated before, and simply outputs the appropriate assembler sequence if it has.

   Unlike the disjunction tables the statement table requires an exact match to fire. The purpose of the disjunction tables is to speed up the matching but not to generate code directly. Matching is still needed to perform parameter substitution. The statement table directly maps intermediate code to source code, and so can not handle unbound variables.

On sucessfully compiling a program the instance of the code generator used is serialised using java serialisation to a `.vwu` file in the current directory. LZ encoding is used to reduce the size of the serialised code generators. The optimisation cache tables are saved along with the codegenerator. The next time a program is compiled in that directory, the codegenerator is loaded from the `.vwu` file so that subsequent compilations can take advantage of previous shortcuts. Given that during a normal development process only a few source lines are changed between successive compilations, most code generation is handled by the cache.

The acceleration in compile times from the optimisation cache depend upon the complexity of the code. Com-

Table 4: Optimisation cache performance

Program	Compile time First Compilation secs	Compile time Next Compilation secs
Dhrystone	11.1	9.8
Konv	14.8	7.4

pilation of highly vectorised code is accelerated more than that of scalar code. Table 4 shows the gains on two programs: the Dhrystone benchmark, and an image convolution program Konv. The latter generates highly vectorisable code, the former does not.

References

[1] Advanced Micro Devices, 3DNow! Technology Manual, 1999.

[2] Bik, A. J. C., Girkar, M., Grey, P. M., Tian, X., Automatic Intra-Register Vectorization for the Intel Architecture, International Journal of Parallel Programming, Vol 30, No. 2, 2002,pp. 65..97.

[3] Canneyt, Michael, Free Pascal Reference Guide, Jan 2001, (http://www.freepascal.org/docs-html/user/user.html).

[4] Cheong, G., and Lam, M., An Optimizer for Multimedia Instruction Sets, 2nd SUIF Workshop, Stanford University, August 1997.

[5] Cockshott, P., Direct Compilation of High Level Languages for Multi-media Instruction-sets, Department of Computer Science, TR-2000-72, University of Glasgow, Nov 2000.

[6] Cockshott, P., Vector Pascal Reference Manual, Department of Computer Science, TR-2002-16, University of Glasgow, Feb 2002.

[7] Cockshott, P., The Abstraction Mechanisms of Vector Pascal, Vector, Vol. 18 No. 4, pp 100-112, April 2002.

[8] Cockshott, P., Vector Pascal: an Array Language for Multimedia Code, APL2002, Madrid, July 2002.

[9] Susan L. Graham, Table Driven Code Generation, IEEE Computer, Vol 13, No. 8, August 1980, pp 25..37.

[10] Intel, Intel Architecture Software Developers Manual Volumes 1 and 2, 1999.

[11] Intel, Willamette Processor Software Developer's Guide, February, 2000.

[12] ISO, Extended Pascal ISO 10206:1990, 1991.

[13] ISO, Pascal, ISO 7185:1990, 1991.

[14] K. E. Iverson, A Programming Language, John Wiley & Sons, Inc., New York (1962), p. 16.

[15] Iverson K. E, A personal View of APL, IBM Systems Journal, Vol 30 , No 4, 1991.

[16] Kerman, Mitchel, Programming and Problem Solving with Delphi, Addison Wesley, 1001.

[17] Jensen K., and Wirth N., Pascal User Manual and Report, Springer, 1978.

[18] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., Ullrich, Ch.: PASCAL-XSC - Language Reference with Examples. Springer-Verlag, New York, 1992.

[19] Krall, A., and Lelait, S., Compilation Techniques for Multimedia Processors, International Journal of Parallel Programming, Vol. 28, No. 4, pp 347-361, 2000.

[20] Leupers, R., Compiler Optimization for Media Processors, EMMSEC 99/Sweden 1999.

[21] Leupers, R., Code Selection for Media Processors with SIMD instructions, DATE 2000.

[22] Metcalf, M., and Reid., J., The F Programming Language, OUP, 1996.

[23] Prospero Software, 'Extended Pascal Language Reference Manual', Prospero Development Software, 2000.

[24] Snyder, L., A Programmer's Guide to ZPL, MIT Press, Cambridge, Mass, 1999.

[25] Srereman, N., and Govindarajan, G., A Vectorizing Compiler for Multimedia Extensions, International Journal of Parallel Programming, Vol. 28, No. 4, pp 363-400, 2000.

[26] Texas Instruments, TMS320C62xx CPU and Instruction Set Reference Guide, 1998.

[27] Watt, D. A., and Brown, D. F., Programming Language Processors in Java, Prentice Hall, 2000.

[28] Wirth, N., Recollections about the development of Pascal, in *History of Programming Languages-II*, ACM-Press, pp 97-111, 1996.

Table 5: Performance on vector kernels

b	16	32	32	32	32	64	
w	BP	DevP	TMT	DP	VP 486	VP K6	test
8	46	71	80	166	333	2329	unsigned byte +
8	38	55	57	110	225	2329	saturated unsigned byte +
8	23	49	46	98	188	2330	pixel +
8	39	67	14	99	158	998	pixel $\times$
16	39	66	74	124	367	1165	short integer +
32	47	85	59	285	349	635	long integer +
32	33	47	10	250	367	582	real +
32	32	47	10	161	164	665	real dot prod
32	33	79	58	440	517	465	integer dot prod

Measures on a 1Ghz Athlon.
The following compilers were used

1. BP - Borland Pascal compiler with 287 instructions enabled range checks off, $b = 16$, release of 1992
2. DevP - Dev Pascal version 1.9, $b = 32$
3. TMT - TMT Pascal version 3, $b = 32$, release of 2000.
4. DP - Delphi version 4, $b = 32$, release of 1998
5. VP 486 - Vector Pascal targeted at a 486, $b = 32$, release of 2002
6. VP K6 - Vector Pascal targeted at an AMD K6, $b = 64$, release of 2002

Figure 7: Vector Pascal System Architecture

Table 6: Showing the comparative performance of different Pascal implementations on the Seive program as a function of set size

	1	2	3	4	5
	secs	secs		μs per integer	μs per integer
Maxlim	Vector	Prospero	ratio	Vector	Prospero
20000	0.73	42	57 to 1	0.1217	6.96
25000	0.91	63	69 to 1	0.1213	8.40
40000	1.30	315	242 to 1	0.1083	26.25

Measurements taken using a 700 Mhz Trans-Meta Crusoe processor. Vector Pascal compiled to the MMX instructionset. Columns 1 and 2 give total run time in seconds to find the primes excluding time to print them. Column 3 shows the speed ratio between the two compilers. Columns 4 and 5 show how the time to process each integer changes as the set size grows.

Figure 8: Parallelised loop

```
{ var i;
   for i=    1 to     8 step     2 do {
    (ref int32 vector ( 2 ))mem(+(@v1,*(-(^i,1),4))):=
       +(^((ref int32 vector ( 2 ))mem(+(@v2,*(-(^i,1),4)))),
         ^((ref int32 vector ( 2 ))mem(+(@v3,*(-(^i,1),4)))));
   };
   for i=    9 to     9 step     1 do {
      v1[^i]:= +(^(v2[^i]),^(v3[^i]));
   };
}
```