Vector Pascal, an array language

Paul Cockshott, University of Glasgow, Imaging Faraday Partnership
Greg Michaelson, Heriot Watt University

January 9, 2002

Abstract

Vector Pascal is a language designed to support elegant and efficient expression of algorithms using the SIMD model of computation. It imports into Pascal abstraction mechanisms derived from functional languages having their origins in APL. In particular it extends all operators to work on vectors of data. The type system is extended to handle pixels and dimensional analysis. Code generation is via the ILCG system that allows retargeting to multiple different SIMD instruction sets based on formal descrition of the instruction set semantics.

1 Introduction

The introduction of SIMD instruction sets[13][1][25][14] to Personal Computers potentially provides substantial performance increases, but the ability of most programmers to harness this performance is held back by two factors. The first is the limited availability of compilers that make effective use of these instruction sets in a machine independent manner. This remains the case despite the research efforts to develop compilers for multi-media instruction sets[7][23][21][28]. The second is the fact that most popular programming languages were designed on the word at a time model of the classic von Neuman computer rather than the SIMD model.

Vector Pascal aims to provide an efficient and elegant notation for programmers using Multi-Media enhanced CPUs. In doing so it borrows concepts for expressing data parallelism that have a long history, dating back to Iverson's work on APL in the early '60s[16]. We are using the word elegant in the technical sense introduced by Chaiten[6]. By an elegant algorithm we mean one which is expressed as concisely as possible. Elegance is a goal that one approaches asymtotically, approaching but never attaining. APL and J[19] allow the construction of very elegant programs, but at a cost. An inevitable consequence of elegance is the loss of redundancy. APL programs are as concise, or even more concise than conventional mathematical notation[17] and use a special characterset. This makes them hard for the uninitiated to understand. J attempts to remedy this by restricting itself to the ASCII characterset, but still looks dauntingly unfamiliar to programmers brought up on more conventional languages. The aim of Vector Pascal is to provide the conceptual gains of Iverson's notation within a framework familiar to imperative programmers.

Pascal[20]was chosen as a base language over the alternatives of C and Java. C was rejected because notations like `x+y` for `x` and `y` declared as arithmetic operations are already overloaded for address operations in a way which precludes their use in map operations.

2 Array mechanisms for data parallelism

Vector Pascal extends the array type mechanism of Pascal to provide better support for data parallel programming in general, and SIMD image processing in particular. Data parallel programming can be built up from certain underlying abstractions[9]:

- operations on whole arrays
- array slicing
- conditional operations
- reduction operations
- data reorganisation

We will first consider these in general before moving on to look at their support in other languages, in particular J, Fortran 90[9] and NESL[3] then finally looking at how they are supported in Vector Pascal.

2.1 Operations on whole arrays

The basic *conceptual* mechanism is the *map*, which takes an operator and and a source array (or pair of arrays) and produces a result array by mapping the source(s) under the operator. Let us denote the type of an arry of Tas $T[]$. Then if we have a binary operator $\omega : (T \otimes T) \rightarrow T$, we automatically have an operator $\omega : (T[] \otimes T[]) \rightarrow T[]$. Thus if x, y are arrays of integers $k = x + y$ is the array of integers where $k_i = x_i + y_i$:

$\begin{array}{|c|c|c|}\hline 3 & 5 & 9 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline 2 & 3 & 5 \\ \hline\end{array} + \begin{array}{|c|c|c|}\hline 1 & 2 & 4 \\ \hline\end{array}$

Similarly if we have a unary operator μ:(T→T) then we automatically have an operator μ:(T[]→T[]). Thus z =sqr(x) is the array where $z_i = x_i^2$:

$\begin{array}{|c|c|c|}\hline 4 & 9 & 25 \\ \hline\end{array} = \mathrm{sqr}(\begin{array}{|c|c|c|}\hline 2 & 3 & 5 \\ \hline\end{array})$

The map replaces the *serialisation* or *for loop* abstraction of classical imperative languages. The map concept is simple, and maps over lists are widely used in functional programming. For array based languages there are complications to do with the semantics of operations between arrays of different lengths and different dimensions, but Iverson[16] provided a consistent treatment of these. Recent languages built round this model are J, an interpretive language[18][4][19], High Performance Fortran[9], F[24] a modern Fortran subset, NESL an applicative data parallel language. In principle though any language with array types can be extended in a similar way. The map approach to data parallelism is machine independent. Depending on the target machine, a compiler can output sequential, SIMD, or MIMD code to handle it.

Recent implementations of Fortran, such as Fortran 90, F, and High Performance Fortran provide direct support for whole array operations. Given that `A,B` are arrays with the same rank and same extents, the statements:

```
REAL,DIMENSION(64)::A,B
A=3.0
B=B+SQRT(A)*0.5;
```

would be legal, and would operate in a pointwise fashion on the whole arrays. Intrinsic functions such as SQRT are defined to operate either on scalars or arrays, but intrinsic functions are part of the language rather than part of a subroutine library. User defined functions over scalara do not automatically extend to array arguments.

J[1] similarly allows direct implementation of array operations, though here the array dimensions are deduced at run time:

```
 a=. 1 2 3 5
   a
1 2 3 5
   b=. 1 2 4 8
   a+b
2 4 7 13
```

[1]We will give examples from J rather than APL here for ease of representation in ASCII.

1	1	1	1
1	2	4	8
1	2	4	16
1	2	8	512

1	1	1	1
1	2	4	8
1	2	4	16
1	2	8	512

1	1	1	1
1	2	4	8
1	2	4	16
1	2	8	512

Figure 1: Different ways of slicing the same array

The pair `=.` is the assignment operator in J. Unlike Fortran, J automatically overloads user defined funtions over arrays. In what follows `^&2` is an expression binding the dyadic power function to the contant 2, `sqr` a user defined monadic operator.

```
 sqr=.^&2
   b=.1 2 4 8
   b+(sqr a)*0.5
1.5 4 8.5 20.5
```

NESL provides similar generality; the first J example above could be expressed as:

```
      {a+b: a in [1,2,3,5]; b in [1,2,4,8]};
=> [2, 4, 7, 13] : [int]
```

and the second example as:

```
      {b+ sqr(a)*0.5: a in [1,2,3,5]; b in [1,2,4,8]};
=> [1.5, 4, 8.5, 20.5] : [float]
```

Again user defined functions can be applied element wise, to arrays (or sequences as they are called in the language). The expressions in { } brackets termed the Apply-to-Each construct, are descended from the ZF notations used in SETL[27] and MIRANDA[30].

2.2 Array slicing

It is advantageous to be able to specify sections of arrays as values in expression. The sections may be rows or columns in a matrix, a rectangular sub-range of the elements of an array, as shown in figure 1. In image processing such rectangular sub regions of pixel arrays are called regions of interest. It may also be desirable to provide matrix diagonals[31].

The notion of array slicing was introduced to imperative languages by ALGOL 68[29]. In ALGOL 68 if x has been declared as [1:10]INT x, then x[2:6] would be a slice that could be used assignment or as an actual parameter.

Fortran 90 extends this notion to allow what it calls triplet subscripts, giving the start position end position and step at which elements are to be taken from arrays.

```
REAL,DIMENSION(10,10)::A,B
A(2:9,1:8:2)=B(3:10,2:9:2)
```

would be equivalent to the loop nest:

```
  DO 1,J=1,8,2
   DO 2, J=2,9
     A(I,J)=b(I+1,J+1)
2  CONTINUE
1 CONTINUE
```

J allows a similar operation to select subsequences. In what follows `i.` n is a function which produces a list of the first n starting with 0, and `{` is the sequence subscription operator.

```
   a=. 2*i.10
   a
0 2 4 6 8 10 12 14 16 18
   3{a
6
```

Selection of a subsequence is performed by forming a sequence of indices. Thus to select the 3 consecutive elements starting with the second, we first form the sequence `2 3 4` using the expression `2+i.3` and use this to subscript the array `a`:

```
   (2+i.3){a
4 6 8
```

2.3 Conditional operations

For data parallel programming one frequently wants to make an operation work on some subset of the data based on some logical mask. This can be thought of providing a finer grain of selection than subslicing, allowing arbitrary combinations of array elements to be acted on. For example one might want to replace all elements of an array A less than the corresponding element in array B with the corresponding element of B.

1	2	4	8	A
2	3	4	5	B
1	1	0	0	A<B
2	3	4	8	

Fortran 90 provides a mechanism to selectively update a section of an array under a logical mask, the `WHERE` statement.

```
REAL, DIMENSION(64)::A
REAL, DIMENSION(64)::B
WHERE (A>B)
  A=A
ELSE WHERE
  A= B
END WHERE
```

The `WHERE` statement is analogous to ALGOL 68 and C conditional expressions, but extended to operate on arrays. It can be performed in parallel on all elements of an array and lends itself to evaluation under a mask on SIMD architectures.

NESL provides a generalised form of Apply-to-Each in which a sieve can be applied to the arguments as in:

```
    { a+b : a in[ 1,2,3]; b in [4,3,2] |a<b}
=>  [5,5] : [int]
```

Notice that in NESL as in J values are allocated dynamically from a heap so that the length of the sequence returned from a sieved Apply-to-Each can be less than that of the argument sequences in its expression part. In Fortran 90, the `WHERE` statement applies to an array whose size is known on entry to the statement.

2.4 Reduction operations

In a reduction operation, a dyadic operator injected between the elements of a vector, the rows or columns of a matrix etc, produces a result of lower rank. Examples would be the forming the sum of a table or finding the maximum or minimum of a table. So one could use + to reduce $\boxed{1}\boxed{2}\boxed{4}\boxed{8}$ to 1+2+4+8=15

1	2	4
8	16	32

transposes to

1	8
2	16
4	32

Figure 2: Reorganising by transposition

1	2	4	8

convolve(

0.25	0.5	0.25

)=

0.5	1	2	2
0.5	1	2	4
0.25	0.25	0.5	1
1.25	2.25	4.5	7

Figure 3: Convolution by shifting

The first systematic treatment of reduction operations in programming languages is due to Iverson[16]. He introduced the notion of a reduction functional which takes a dyadic operator and, by currying, generates a tailored reduction function. In APL as in J the reduction functional is denoted by `/`. Thus `+/` is the function which forms the sum of an array.

```
    a
1 2 3 5
    +/a
11
```

The interpretation of reduction for comutative operators is simple, for non comutative ones it is slightly less obvious. Consider:

```
    -/a
_3
```

The `_3` here, is the J notation for -3, derived from the expansion $(1-(2-(3-4(-0))))$, just as 11 was derived from the expansion $(1+(2+(3+(4+0))))$. In J as in APL reduction applies uniformly to all binary operators.

Fortran 90, despite its debt to APL, is less general, providing a limited set of built in reduction operators on comutative operators: `SUM, PRODUCT, MAXVAL, MINVAL.` NESL likewise provides a limited set of reduction functions `sum, minval, maxval, any, all`. The last two are boolean reductions, `any` returns true if at least one element of a sequence is true, `all` if they are all true.

2.5 Data reorganisation

In both linear algebra and image processing applications, it is often desireable to be able to perform bulk reorganisation of data arrays.

One may want to transpose a vector or matrix, or to shift the elements of a vector. For example, one can express the convolution of a vector with a three element kernel in terms of multiplications, shifts and adds. Let $a = \boxed{1 \mid 2 \mid 4 \mid 8}$ be a vector to be convolved with the kernel $k = \boxed{0.25 \mid 0.5 \mid 0.25}$. This can be expressed by defining two temporary vectors

$$b, c = 0.25a, 0.5a = \boxed{0.25 \mid 0.5 \mid 1 \mid 2}, \boxed{0.5 \mid 1 \mid 2 \mid 4}$$

and then defining the result to the sum under shifts of b, c as shown in figure 3 This example replicates the trailing value when shifting. In other circumstances, when dealing with cellular automata for example, it is convenient to be able to define circular shifts on data arrays.

Fortran 90 provides a rich set of functions to reshape, transpose and circularly shift arrays. For instance given a 9 element vector v we can reshape it as a 3 by 3 matrix

```
V= (/ 1,2,3,4,5,6,7,8,9 /)
M=RESHAPE(V,(/3,3/))
```

gives us the array

```
1  2  3
4  5  6
7  8  9
```

We can then cyclically shift this along a dimension

```
M2=CSHIFT(M,SHIFT=2,DIM=2)
```

to give

```
3  1  2
6  4  5
9  7  8
```

NESL provides on sequences, similar operations to those provided on arrays by Fortran 90, so that if

```
v              =[ 1,2,3,4,5,6,7,8,9]
s              =[3,3,3]
partition(v,s)=[[1,2,3][4,5,6],7,8,9]]
rotate(v,3)    =[7,8,9,1,2,3,4,5,6]
```

3 Data parallelism in Vector Pascal

3.1 Assignment maps

Standard Pascal allows assignement of whole arrays. Vector Pascal extends this to allow consistent use of mixed rank expressions on the right hand side of an assignment. Given

r0:real; r1:array[0..7] of real; r2:array[0..7,0..7] of real

then we can write

1. r**1:= r2[3]; { supported in standard Pascal }**
2. **r1:= 1/2; { assign 0.5 to each element of r1 }**
3. **r2:= r1*3; { assign 1.5 to every element of r2}**
4. **r1:= \+ r2; { r1gets the totals along the rows of r2}**
5. **r1:= r1+r2[1];{ r1 gets the corresponding elements of row 1 of r2 added to it}**

The assignment of arrays is a generalisation of what standard Pascal allows. Consider the first examples above, they are equivalent to:

1. **for i:=0 to 7 do r1[i]:=r2[3,i];**
2. **for i:=0 to 7 do r1[i]:=1/2;**
3. **for i:=0 to 7 do for j:=0 to 7 do r2[i,j]:=r1[j]*3;**
4. **for i:=0 to 7 do begin t:=0; for j:=7 downto 0 do t:=r2[i,j]+t; r1[i]:=t; end;**
5. **for i:=0 to 7 do r1[i]:=r1[i]+r2[1,i];**

In other words the compiler has to generate an implicit loop over the elements of the array being assigned to and over the elements of the array acting as the data-source. In the above **i,j,t** are assumed to be temporary variables not refered to anywhere else in the program. The loop variables are called implicit indices and may be accessed using **iota**.

The variable on the left hand side of an assignment defines an array context within which expressions on the right hand side are evaluated. Each array context has a rank given by the number of dimensions of the array on the left hand side. A scalar variable has rank 0. Variables occuring in expressions with an array context of rank r must have r or fewer dimensions. The n bounds of any n dimensional array variable, with $n \leq r$ occuring within an expression evaluated in an array context of rank r must match with the rightmost n bounds of the array on the left hand side of the assignment statement.

Where a variable is of lower rank than its array context, the variable is replicated to fill the array context. This is shown in examples 2 and 3 above. Because the rank of any assignment is constrained by the variable on the left hand side, no temporary arrays, other than machine registers, need be allocated to store the intermediate array results of expressions.

Maps are implicitly and promiscously defined on both monadic operators and unary functions. If **f** is a function or unary operator mapping from type **r** to type **t** then if **x** is an array of **r** then **a:=f(x)** assigns an array of **t** such that **a[i]=f(x[i])**.

Functions can return any data type whose size is known at compile time, including arrays and records. A consistent copying semantics is used.

3.2 Slice operations

Image processing applications often have to deal with regions of interest, rectangular sub-images within a larger image. Vector Pascal extends the array abstraction to define sub-ranges of arrays. A sub-range of an array variable are denoted by the variable followed by a range expression in brackets.

The expressions within the range expression must conform to the index type of the array variable. The type of a range expression **a[i..j]** where **a: array[p..q] of t** is **array[0..j-i] of t.**

Examples

dataset[i..i+2]:=blank;

twoDdata[2..3,5..6]:=twoDdata[4..5,11..12]*0.5;

Subranges may be passed in as actual parameters to procedures whose corresponding formal parameters are declared as variables of a schematic type. Hence given the following declarations:

type image(miny,maxy,minx,maxx:integer)=array[miny..maxy,minx..maxx] of pixel;

procedure invert(var im:image);begin im:= - im; end;

var screen:array[0..319,0..199] of pixel;

then the following statement would be valid:

invert(screen[40..60,20..30]);

A particular form of slicing is to select out the diagonal of a matrix. The syntactic form **diag**<expression> selects the diagonal of the matrix to which it applies. A precise definition of this is given in section 3.5.

3.3 Conditional update operations

In Vector Pascal the sort of conditional updates handled by the Fortran `WHERE` statement, are programmed using conditional expressions. The Fortran code shown in 2.3 would translate to

var a:array[0..63] of real;

a:=if a>0 then a else -a

The **if** expression can be compiled in two ways:

1. Where the two arms of the if expression are parallelisable, the condition and both arms are evaluated and then merged under a boolean mask so the above assignment would be equivalent to:

   a:= (a and (a>0))or(not (a>0) and -a);

   were the above legal Pascal[2].

2. If the code is not paralleliseable it is translated as equivalent to a standard if statement, so the previous example would be equivalent to:

   for i:=0 to 63 do if a[i]>0 then a[i]:=a[i] else a[i]:=-a[i];

   Expressions are non parallelisable if they include function calls.

The dual compilation strategy allows the same linguistic construct to be used in recursive function definition and parallel data selection.

3.4 Operator Reduction

Maps take operators and arrays and deliver array results. The *reduction* abstraction takes a dyadic operator and an array and returns a scalar result. It is denoted by the functional form \. Thus if a is an array, \+a denotes the sum over the array. More generally $\backslash\Phi x$ for some dyadic operator Φ means $x_0\Phi(x_1\Phi..(x_n\Phi\iota))$ where ι is the identity element for the operator and the type. Thus we can write \+ for $\sum$, * for $\prod$ etc. The dot product of two vectors can thus be written as

x:=\+(y*z);
instead of
x:=0;
for i:=0 to n do x:= x+ y[i]*z[i];

A reduction operation takes an argument of rank *r* and returns an argument of rank *r-1* except in the case where its argument is or rank 0, in which case it acts as the identity operation. Reduction is always performed along the last array dimension of its argument.

Semantically reduction by an operator say $\odot$ is defined such that
var a:array[low..high] of t;x:t;
x:=\\$\odot$a;
is equivalent to:
var temp:t; i:low..high;
temp:= *identity*(t,$\odot$);
for i:= high downto low do temp:=a[i]$\odot$temp;
x:=temp;

Where *identity*(t,$\odot$) is a function returning the identity element under the operator $\odot$ for the type **t**. The identity element is defined to be the value such that $x = x\odot$*identity*(t,$\odot$). Identity elements for operators and types are shown in table 1.

3.5 Array reorganisation

By array reorganisation, we mean conservative operations, which preserve the number of elements in the orignal array. If the shape of the array is also conserved we have an element permutation operation. If the shape of the array is not conserved but it's rank and extents are, we have a permutation of the array dimensions. If the rank is not conserved we have a flattening or reshaping of the array.

[2]This compilation strategy requires that true is equivalent to -1 and false to 0. This is typically the representation of booleans returned by vector comparison instructions on SIMD instruction sets. In Vector Pascal this representation is used generally and in consequence, **true<false**.

Vector Pascal provides syntactic forms to access and manipulate the implicit indices used in maps and reductions. These syntactic forms allow the concise expression of many conservative array reorganisations.

The form **iota** i returns the *i*th current implicit index. Thus given the definitions

v1:array[1..3]of integer; v2:array[0..4]of integer;

the program fragment

v1:=iota 0; v2:=iota 0 *2;

would set v1 and v2 as follows:

```
v1= 1 2 3
v2= 0 2 4 6 8
```

whilst given the definitions

m1:array[1..3,0..4] of integer;m2:array[0..4,1..3]of integer;

then the program fragment

m2:= iota 0 +2*iota 1;

would set m2

```
m2=
2 4 6
3 5 7
4 6 8
5 7 9
6 8 10
```

The argument to iota must be an integer known at compile time within the range of implicit indices in the current context.

A generalised permutation of the implicit indices is performed using the syntatic form **perm[**<index-sel>[,<index-sel>]*] <expression>. The <index-sel>s are integers known at compile time which specify a permution on the implicit indices. Thus in e evaluated in context **perm**[i, j, k]e, then **iota 0 =iota** i, **iota 1=iota** j, **iota 2=iota** k.

An example of where this is useful is in converting between different image formats. Hardware frame buffers typically represent images with the pixels in the red, green, blue, and alpha channels adjacent in memory. For image processing it is convenient to hold them in distinct planes. The **perm** operator provides a concise notation for translation between these formats:

screen:=perm[2,0,1]img;

where:

type rowindex=0..479;colindex=0..639;channel=red..alpha;

screen:array[rowindex,colindex,channel] of pixel;

img:array[channel,colindex,rowindex] of pixel;

trans and **diag** provide shorthand notions for expressions in terms of **perm**. Thus in an assignment context of rank 2, **trans** = **perm[1,0]** and **diag** = **perm[0,0]**. The form **trans***x* transposes a vector [3]matrix, or tensor. It achieves this by cyclic rotation of the implicit indices. Thus if **trans** e , for some expression e is evaluated in a context with implicit indices

iota *0*.. **iota** *n*

then the expression e is evaluated in a context with implicit indices

iota'*0*.. **iota**'*n*

where

iota'*x* = **iota** ((*x*+*1*)**mod** *n*+*1*)

It should be noted that transposition is generalised to arrays of rank greater than 2.

[3]Note that **trans** is not strictly speaking an operator, as there exists no Pascal type corresponding to a column vector.

Examples Given the defintions used above, the program fragment:

m1:= (trans v1)*v2;
m2 := trans m1;
will set m1 and m2:

```
m1=
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
m2=
0 0 0
2 4 6
4 8 12
6 12 18
8 16 24
```

3.5.1 Array shifts

The shifts and rotations of arrays supported in Fortran 90 and NESL are not supported by any explicit operator, though one can of course use a combination of other features to achieve them. A left rotation can for instance be achieved as follows:

a:=b[(1+iota 0)mod n];
and a reversal by
a:=b[n-iota 0 -1];
Where in both examples:
var a,b:array[0..n-1] of integer;

3.6 Efficiency considerations

Expressions involving transposed vectors, matrix diagonals, and permuted vectors or indexing by expressions involving modular arithmetic on **iota**, do not paralellise well on SIMD architectures like the MMX, since these depend upon the fetching of blocks of adjacent elements into the vector registers. This requires that element addresses be adjacent and monotonically increasing. Assignments involving mapped vectors will usually have to be handled by scalar registers.

3.6.1 Element permutation

Permutations are widely used in APL and J programming, an example being sorting and array `a` into descending order using the J expression `\:a{a` which uses the operator `\:` to produce a permutation of the indices of `a` in descending order, and then uses { to index `a` with this permutation vector. If one is to use analogous constructs, one muse allow an array may be indexed by another array. If **x:array[t0] of t1** and **y:array[t1] of t2**, then in Vector Pascal, **y[x]** denotes the virtual array of type **array[t0] of t2** such that **y[x][i]=y[x[i]]**.

Example Given the declarations

const perm:array[0..3] of integer=(3,1,2,0);
var ma,m0:array[0..3] of integer;
then the statements
m0:= (iota 0)+1;
ma:=m0[perm];
would set the variables such that

```
m0= 1  2  3  4
perm= 3 1 2 0
ma= 4 2 3 1
```

4 Extensions to the Pascal Type System

4.1 Pixels

Standard Pascal is a strongly typed language, with a comparatively rich collection of type abstractions : enumeration, set formation, sub-ranging, array formation, cartesian product[4] and unioning[5]. However as an image processing language it suffers from the disadvantage that no support is provided for pixels and images. Given the vintage of the language this is not surprising and, it may be thought, this deficiency can be readily overcome using existing language features. Can pixels not be defined as a subrange 0..255 of the integers, and images modeled as two dimensional arrays?

They can be, and are so defined in many applications, but such an approach throws onto the programmer the whole burden of handling the complexities of limited precision arithmetic. Among the problems are:

1. When doing image processing it is frequently necessary to subtract one image from another, or to create negatives of an image. Subtraction and negation implies that pixels should be able to take on negative values.

2. When adding pixels using limited precision arithmetic, addition is nonmontonic due to wrap-round. Pixel values of $100 + 200 = 300$, which in 8 bit precision is truncated to 44 a value darker than either of the starting values. A similar problem can arise with subtraction, for instance $100 - 200 = 156$ in 8 bit unsigned arithmetic.

3. When multiplying 8 bit numbers, as one does in executing a convolution kernel, one has to enlarge the representation and shift down by an appropriate amount to stay within range.

These and similar problems make the coding of image filters a skilled task. The difficulty arises because one is using an inappropriate conceptual representation of pixels.

The *conceptual model* of pixels in Vector Pascal is that they are real numbers in the range $-1.0..1.0$. This representation overcomes the aforementioned difficulties. As a signed representation it lends itself to subtraction. As an unbiased representation, it makes the adjustment of contrast easier, one can reduce contrast 50% simply by multiplying an image by 0.5 [6]. Assignment to pixel variables in Vector Pascal is defined to be saturating - real numbers outside the range $-1..1$ are clipped to it. The multiplications involved in convolution operations fall naturally into place.

The *implementation model* of pixels used in Vector Pascal is of 8 bit signed integers treated as fixed point binary fractions. All the conversions necessary to preserve the monotonicity of addition, the range of multiplication etc, are delegated to the code generator which, where possible, will implement the semantics using efficient, saturated multi-media arithmetic instructions.

4.2 Dimensioned Types

Dimensional analysis is familiar to scientists and engineers and provides a routine check on the sanity of mathematical expressions. Dimensions can not be expressed in the otherwise

[4] The **record** construct.

[5] The **case** construct in records.

[6] When pixels are represented as integers in the range 0..255, a 50% contrast reduction has to be expressed as $((p - 128) \div 2) + 128$.

rigourous type system of standard Pascal, but they are a useful protection against the sort of programming confusion between imperial and metric units that caused the demise of a recent Mars probe. They provide a means by which floating point types can be specialised to represent dimensioned numbers as is required in physics calculations. For example:

kms =(mass,distance,time);
meter=real of distance;
kilo=real of mass;
second=real of time;
newton=real of mass * distance * time POW -2;
meterpersecond = real of distance *time POW -1;

The identifier must be a member of a scalar type, and that scalar type is then refered to as the *basis space* of the dimensioned type. The identifiers of the basis space are refered to as the dimensions of the dimensioned type. Associated with each dimension of a dimensioned type there is an integer number refered to as the power of that dimension. This is either introduced explicitly at type declaration time, or determined implicitly for the dimensional type of expressions.

A value of a dimensioned type is a dimensioned value. Let $\log_d t$ of a dimensioned type t be the power to which the dimension d of type t is raised. Thus for $t =$newton in the example above, and $d =$time, $\log_d t = -2$

If x and y are values of dimensioned types $t_x$and $t_y$respectively, then the following operators are only permissible if $t_x = t_y$: +, - ,<, >, =, <=, >=. For + and -, the dimensional type of the result is the same as that of the arguments. The operations. The operations *, / are permited if the types $t_x$and t_y share the same basis space, or if the basis space of one of the types is a subrange of the basis space of the other.

The operation **POW** is permited between dimensioned types and integers.

Dimension deduction rules

1. If $x = y * z$ for $x : t_1, y : t_2, z : t_3$ with basis space B then $\forall_{d \in B} \log_d t_1 = \log_d t_2 + \log_d t_3$.
2. If $x = y/z$ for $x : t_1, y : t_2, z : t_3$ with basis space B then $\forall_{d \in B} \log_d t_1 = \log_d t_2 - \log_d t_3$.
3. If $x = y$ **POW** z for $x : t_1, y : t_2, z : integer$ with basis space for t_2, B then $\forall_{d \in B} \log_d t_1 = \log_d t_2 \times z$.

5 Operators

5.0.1 Dyadic Operations

Dyadic operators supported are **+, +:, -:, -, *, /, div, mod , **, pow, <, >, >=, <=, =, <>, shr, shl, and, or, in, min, max**. All of these are consistently extended to operate over arrays. The operators **, pow denote exponentiation and raising to an integer power as in ISO Extended Pascal. The operators +: and -: exist to support saturated arithmetic on bytes as supported by the MMX instruction set.

5.0.2 Unary operators

The unary operators supported are **+, -, *, /, max, min, div, not, round, sqrt, sin, cos, tan, abs, ln, ord, chr, succ, pred** and @.

Thus the following are valid unary expressions**: -1, +b, not true, sqrt abs x, sin theta.** In standard Pascal some of these operators are treated as functions,. Syntactically this means that their arguments must be enclosed in brackets, as in **sin(theta)**. This usage remains syntactically correct in Vector Pascal.

Table 1: Identity element

type	operators	identity elem
number	+,-	0
set	+	empty set
set	-,*	fullset
number	*,/ ,div,mod	1
boolean	and	true
boolean	or	false

```
type
        complex = record data: array[0..1] of real;end;
var     complexzero,complexone:complex;

{ headers for functions onto the complex numbers }
function cmplx                  (realpart,imag:real):complex;
function complex_add            (A,B:Complex):complex;
function complex_conjugate      (A:Complex):complex;
function complex_subtract       (A,B:Complex):complex;
function complex_multiply       (A,B:Complex):complex;
function complex_divide         (A,B:Complex):complex;
function im                     (c:complex):real;
function re                     (c:complex):real;
{ Standard operators on complex numbers  }
{           symbol   function            identity element }
operator    + =     complex_add,        complexzero;
operator    / =     complex_divide,     complexone;
operator    * =     complex_multiply,   complexone;
operator    - =     complex_subtract,   complexzero;
```

Figure 4: Defining operations on complex numbers

Note that only the function headers are given here as this code comes from the interface part of the system unit. The function bodies and the initialisation of the variables complexone and complexzero are handled in the implementation part of the unit.

The dyadic operators are extended to unary context by the insertion of an identity element under the operation. This is a generalisation of the monadic use of + and - in standard pascal where **+a=0+a** and **-a = 0-a** with 0 being the additive identity, so too **/2 = 1/2** with 1 the multiplicative identity. For sets the notation **-s** means the complement of the set **s**. The identity elements inserted are given in table 1.

5.0.3 Operator overloading

The dyadic operators can be extended to operate on new types by operator overloading. Figure 4 shows how arithmetic on the type **complex** required by Extended Pascal [15] is defined in Vector Pascal. Each operator is associated with a semantic function and an identity element. The operator symbols must be drawn from the set of predefined vector pascal operators, and when expressions involving them are parsed, priorities are inherited from the predefined operators. The type signature of the operator is deduced from the type of the function[7]. When parsing expressions, the compiler first tries to resolve operations in terms

[7] Vector Pascal allows function results to be of any type.

of the predefined operators of the language, taking into account the standard mechanisms allowing operators to work on arrays. Only if these fail does it search for an overloaded operator whose type signature matches the context.

In the example in figure 4, complex numbers are defined to be records containing an array of reals, rather than simply as an array of reals. Had they been so defined, the operators +,*,-,/ on reals would have masked the corresponding operators on complex numbers.

The provision of an identity element for complex addition and subtraction ensures that unary minus, as in $-x$ for x :complex, is well defined, and correspondingly that unary / denotes complex reciprocal. Overloaded operators can be used in array maps and array reductions.

6 An example algorithm

As an example of Vector Pascal we will look at an image filtering algorithm. In particular we will look at applying a separable 3 element convolution kernel to an image. We shall initially present the algorithm in standard Pascal and then look at how one might re-express it in Vector Pascal.

Convolution of an image by a matrix of real numbers can be used to smooth or sharpen an image, depending on the matrix used. If A is an output image, K a convolution matrix, then if B is the convolved image

$$B_{y,x} = \sum_i \sum_j A_{y+i,x+j} K_{i,j}$$

A separable convolution kernel is a vector of real numbers that can be applied independently to the rows and columns of an image to provide filtering. It is a specialisation of the more general convolution matrix, but is algorithmically more efficient to implement. If $\mathbf{k}$ is a convolution vector, then the corresponding matrix K is such that $K_{i,j} = \mathbf{k}_i \mathbf{k}_j$.

Given a starting image A as a two dimensional array of pixels, and a three element kernel c_1, c_2, c_3, the algorithm first forms a temporary array T whose whose elements are the weighted sum of adjacent rows $T_{y,x} = c_1 A_{y-1,x} + c_2 A_{y,x} + c_3 A_{y+1,x}$. Then in a second phase it sets the original image to be the weighted sum of the columns of the temporary array: $A_{y,x} = c_1 T_{y,x-1} + c_2 T_{y,x} + c_3 Ty, x+1$. Clearly the outer edges of the image are a special case, since the convolution is defined over the neighbours of the pixel, and the pixels along the boundaries a missing one neighbour. A number of solutions are available for this, but for simplicity we will perform only vertical convolutions on the left and right edges and horizontal convolutions on the top and bottom lines of the image.

Figure 5 shows `conv` an implementation of the convolution in Standard Pascal. The pixel data type has to be explicitly introduced as the subrange -128..127. Explicit checks have to be inplace to prvent range errors, since the result of a convolution may, depending on the kernel used, be outside the bounds of valid pixels. Arithmetic is done in floating point and then rounded.

Image processing algorithms lend themselves particularly well to data-parallel expression, working as they do on arrays of data subject to uniform operations. Figure 6 shows a data-parallel version of the algorithm, `pconv`, implemented in Vector Pascal. Note that all explicit loops disappear in this version, being replaced by assignments of array slices. The first line of the algorithm initialises three vectors `p1, p2, p3` of pixels to hold the replicated copies of the kernel coefficients `c1, c2, c3` in fixed point format. These vectors are then used to multiply rows of the image to build up the convolution. The notation `theim[][1..maxpix-1]` denotes columns 1..maxpix-1 of all rows of the image. Because the built in pixel data type is used, all range checking is handled by the compiler. Since fixed point arithmetic is used throughout, there will be slight rounding errors not encountered with the previous algorithm, but these are acceptable in most image process-

```
type
    pixel = -128..127;
    tplain = array[0..maxpix ,0..maxpix] of pixel;

procedure conv(var theim:tplain;c1,c2,c3:real);
var tim:array[0..maxpix,0..maxpix]of pixel;
    temp:real;
    i,j:integer;
begin
        for i:=1 to maxpix-1 do
         for j:= 0 to maxpix do begin
              temp:= theim[i-1][j]*c1+theim[i][j]*c2+theim[i+1][j]*c3;
              if temp>127 then temp :=127 else
              if temp<-128 then temp:=-128;
              tim[i][j]:=round(temp);
         end;
        for j:= 0 to maxpix do begin
                tim[0][j]:=theim[0][j]; tim[maxpix][j]:=theim[maxpix][j];
        end;
        for i:=0 to maxpix do begin
            for j:= 1 to maxpix-1 do begin
                temp:= tim[i][j-1]*c1+tim[i][j+1]*c3+tim[i][j]*c2;
                if temp>127 then temp :=127 else
                if temp<-128 then temp:=-128;
                tim[i][j]:=round(temp);
            end;
            theim[i][0]:=tim[i][0]; theim[i][maxpix]:=tim[i][maxpix];
        end;
end;
```

Figure 5: Standard Pascal implementation of the convolution

```
procedure pconv(var theim:tplain;c1,c2,c3:real);
var tim:array[0..maxpix,0..maxpix]of pixel;
    p1,p2,p3:array[0..maxpix]of  pixel;
begin
     p1:=c1; p2:=c2; p3:=c3;
     tim [1..maxpix-1] :=
        theim[0..maxpix-2]*p1 +theim[1..maxpix-1]*p2+theim[2..maxpix]*p3;
     tim[0]:=theim[0]; tim[maxpix]:=theim[maxpix];
     theim[][1..maxpix-1]:=
        tim[][0..maxpix-2]*p1+tim[][2..maxpix]*p3+tim[][1..maxpix-1]*p2;
     theim[][0]:=tim[][0]; theim[][maxpix]:=tim[][maxpix];
end;
```

Figure 6: Vector Pascal implementation of the convolution

Table 2: Comparative Performance on Convolution

Algorithm	Implementation	Target Processor	Million Ops Per Second
conv	Vector Pascal	Pentium + MMX	61
	Borland Pascal	286 + 287	5.5
	Delphi 4	486	86
pconv	Vector Pascal	486	80
	Vector Pascal	Pentium + MMX	817

Measurements done on a 1Ghz Athlon, running Windows 2000.

ing applications. Fixed point pixel arithmetic has the advantage that it can be efficently implemented in parallel using multi-media instructions.

It is clear that the data-parallel implementation is somewhat more concise than the sequential one, 12 lines with 505 characters compared to 26 lines with 952 characters. It also runs considerably faster, as shown in table 2. This expresses the performance of different implementations in millions of effective arithmetic operations per second. It is assumed that the basic algorithm requires 6 multiplications and 6 adds per pixel processed. The data parallel algorithm runs 12 times faster than the serial one when both are compiled using Vector Pascal and targeted at the MMX instructionset. The `pconv` also runs a third faster than `conv` when it is targeted at the 486 instructionset, which in effect, serialises the code.

For comparison `conv` was run on two other Pascal Compilers, Borland Pascal and its successor Delphi. Both of these are extended implementations of Pascal supporting objects, but with no support for vector arithmetic. Delphi is a state of the art commercial compiler, as Borland Pascal was when released in 1992. In both cases range checking was enabled for consistency with Vector Pascal. The only other change was to define the type pixel as equivalent to the system type shortint to force implementation as a signed byte. Delphi runs `conv` 40% faster than Vector Pascal does, whereas Borland Pascal runs it at only 7% of the speed.

Further performance comparisons are given in table 3. The tests here involve vector arithmetic on vectors of length 640 and take the general form $v_1 = v_2 \phi v_3$ for some operator ϕ and some vectors v_1, v_2, v_3. The exception being the dot product operation coded as

```
r:=\+ r2*r3
```

in Vector Pascal, and using conventional loops for the other compilers. When targeted on a 486 the performance of the Vector Pascal compiler is roughly comparable with that of Delphi. The exception being dot product operations on which Delphi performs particularly well. When the target machine is a K6 which incorporates both the MMX and the 3DNow SIMD instruction sets, the acceleration is broadly in line with the degree of parallelism offered by the architecture: 8 fold for byte operands, 4 fold for 16 bit ones, and 2 fold for integers and reals. The speedup is best for the 8 bit operands, a sevenfold acceleration on byte additions for example. For larger operands it falls off to 60% for 32 bit integers and 33% for 32 bit reals.

For data types where saturated arithmetic is used, the accleration is most marked, a 12 fold acceleration being achieved for saturated byte additions and a 16 fold acceleration on pixel additions. These additional gains come from the removal of bounds checking code that would otherwise be required.

7 Implementation

At the heart of our implementation is the machine independent Intermediate Language for Code Generation (ILCG). ILCG[8] is strongly typed, supporting the base types common

Table 3: Performance on vector kernels

BP 286	DP 486	VP 486	VP K6	test
46	166	333	2329	unsigned byte additions
38	110	179	2329	saturated unsigned byte additions
47	285	291	466	32 bit integer additions
39	124	291	1165	16 bit integer additions
33	250	291	388	real additions
23	98	146	2330	pixel additions
39	99	146	1165	pixel multiplications
32	161	146	141	real dot product
33	440	161	166	integer dot product

BP 286 - Borland Pascal compiler with 287 instructions enabled range checks off.

DP 486 - Delphi

VP 486 - Vector Pascal targeted at a 486

VP K6 - Vector Pascal targeted at and AMD K6

All figures in millions of operations per second on a 1 Ghz Athlon.

in most programming languages along with type constructors for vectors, stacks and references. In particular, operators may be implicitly overloaded for vector operations. Its purpose is to act as an input to an automatically constructed code generator, working on the syntax matching principles described in [11]. Simple rules link high-level register transfer descriptions with the equivalent low-level assembly representations. ILCG may be used as a machine-oriented semantics of a high-level program or of a CPU. It may also be used as an intermediate language for program transformation.

7.1 Analogous work

There has been sustained research within the parallel programming community into the exploitation of SIMD parallelism on multi-processor architectures. Most work in this field has been driven by the needs of high-performance scientific processing, from finite element analysis to meteorology. In particular, there has been considerable interest in exploiting data parallelism in FORTRAN array processing, culminating in High Performance Fortran and F. Typically this involves two approaches. First of all, operators may be overloaded to allow array-valued expressions, similar to APL. Secondly, loops may be analysed to establish where it is possible to unroll loop bodies for parallel evaluation. Compilers embodying these techniques tend to be architecture specific to maximise performance and they have been aimed primarily at specialised super-computer architectures, even though contemporary general purpose microprocessors provide similar features, albeit on a far smaller scale.

There has been recent interest in the application of vectorisation techniques to instruction level parallelism. Thus, Cheong and Lam [7] discuss the use of the Stanford University SUIF parallelising compiler to exploit the SUN VIS extensions for the UltraSparc from C programs. They report speedups of around 4 on byte integer parallel addition. Krall and Lelait's compiler [21] also exploits the VIS extensions on the Sun UltraSPARC processor from C using the CoSy compiler framework. They compare classic vectorisation techniques to unrolling, concluding that both are equally effective, and report speedups of 2.7 to 4.7. Sreraman and Govindarajan [28] exploit Intel MMX parallelism from C with SUIF, using a variety of vectorisation techniques to generates inline assembly language, achieving speedups from 2 to 6.5. All of these groups target specific architectures. Finally, Leupers [23] 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.

There has been extensive research, initiated by Graham and Glanville, into the automatic production of code generators but predominantly for conventional rather than parallel instruction sets. There has also been research in the hardware/software co-design community into compilation techniques for non-standard architectures. Leupers [22] MIMOLA language allows the expression of both programs and structural hardware descriptions, driving the micro-code compiler MSSQ. Hadjiyiannis' [12] Instruction Set Description Language and Ramsey and Davidson's [26] Specification Language for Encoding and Decoding are also aimed at embedded systems, based on low level architecture descriptions.It is not clear whether MIMOLA, ISDL or SLED could readily be used for describing data parallelism through operator overloading as found in MMX extensions. Furthermore, ISDL and SLED's type systems will not readily express the vector types required for MMX.

To exploit MMX and other extended instruction sets it is desirable to develop compiler technology based on a richer meta-language which can express non-standard instruction sets in a succinct but architecture independent manner. Such a meta-language should support a rich set of types and associated operators, and be amenable to formal manipulation. It should also support a relatively high level of abstraction from different manufacturers' register level implementations of what are effectively the same MMX operations.

7.2 Intermediate Language for Code Generation

A Vector Pascal program is translated into an ILCG abstract semantic tree implemented as a Java datastructure. The tree is passed to a machine generated Java class corresponding to the code generator for the target machine. Code generator classes currently exist for the Intel 486, Pentium with MMX, and P3 and also the AMD K6. Output is assembler code which is assembled using the NASM assembler and linked using the gcc loader.

The code generators follow the pattern matching approach described in[2][5]and [11], 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 7.2. It is complex.

Consider first the path followed from a source file, the phases that it goes through are

i. The source file (1) is parsed by a java class PascalCompiler.class (2) a hand written, recursive descent parser[32], and results in a Java data structure (3), an ILCG tree, which is basically a semantic tree for the program.

ii. 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 transportations 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.

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

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

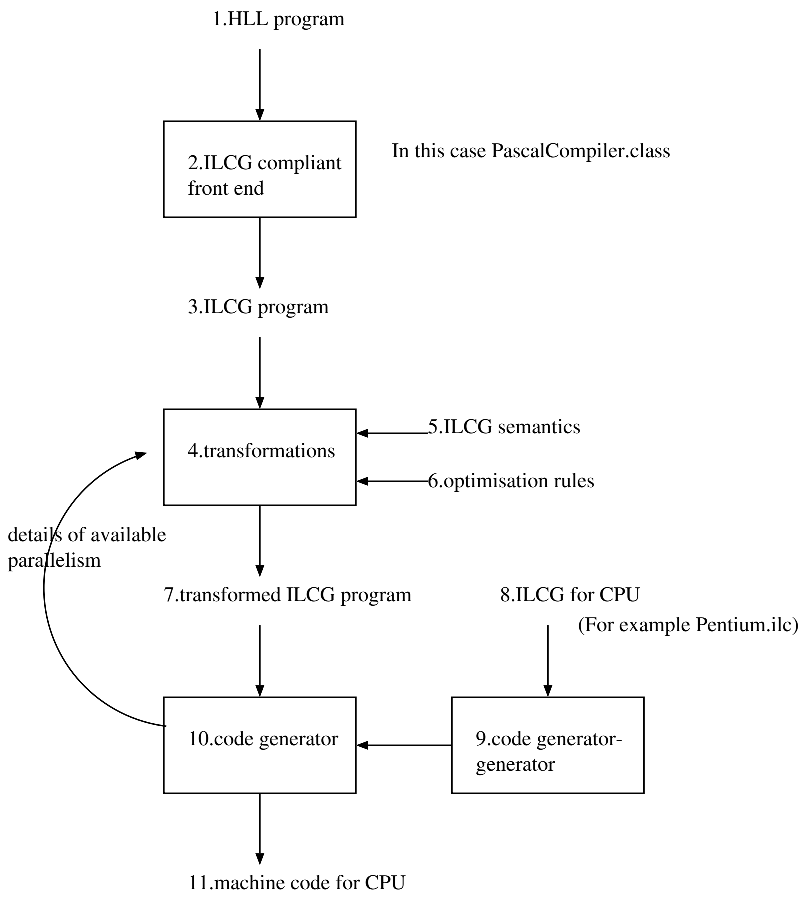

Figure 7: System Architecture

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

Figure 8: Sequential form of array assignment

```
{ 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]));
   };
}
```

Figure 9: Parallelised loop

7.3 Vectorisation

The parser initially generates serial code for all constructs. It then interogates 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 8 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 9. 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.

The result is shown in figure 10. When the `eval` method is invoked, constant folding causes the loop test condition to be evaluated to `if >(^i,8) then goto leb4af11b47f`.

```
{ var i:
  i:= 1;
  leb4af11b47e:
  if >( 2, 0) then if >(^i,8) then goto leb4af11b47f
                else null
                        fi
 else if <(^i, 8) then goto leb4af11b47f
 else null
         fi
  fi;
 (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)))));
  i:=+(^i,2);
 (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)))));
  i:=+(^i,2);
  goto leb4af11b47e;
  leb4af11b47f:
  i:=    9;
  v1[^i]:= +(^(v2[^i]),^(v3[^i]));
}
```

Figure 10: After applying `simplify` to the tree

7.4 Machine descriptions in ILCG

ILCG exists both as a tree language, defined as a set of Java classes, and as a textual notation that can be used to describe the semantics of machine instructions.

Pentium.class (10 in figure 7.2) is produced from a file Pentium.ilc (8), in ILCG, which gives a semantic description of the Pentium's instructionset. This is processed by a code generator generator which builds the source file Pentium.java.

A machine description typically consists of a set of register declarations followed by a set of instruction formats and a set of operations. This approach works well only with machines that have an orthogonal instruction set, ie, those that allow addressing modes and operators to be combined in an independent manner.

7.4.1 Registers

When entering machine descriptions in ilcg registers can be declared along with their type hence

register word EBX assembles['ebx'] ;
reserved register word ESP assembles['esp'];
would declare **EBX** to be of type **ref word**.

7.4.2 Aliasing

A register can be declared to be a sub-field of another register, hence we could write

alias register octet AL = EAX(0:7) assembles['al'];
alias register octet BL = EBX(0:7) assembles['bl'];
to indicate that **BL** occupies the bottom 8 bits of register **EBX**. In this notation bit zero is taken to be the least significant bit of a value. There are assumed to be two pregiven

registers **FP, GP** that are used by compilers to point to areas of memory. These can be aliased to a particular real register.

register word EBP assembles['ebp'] ;

alias register word FP = EBP(0:31) assembles ['ebp'];

Additional registers may be reserved, indicating that the code generator must not use them to hold temporary values:

reserved register word ESP assembles['esp'];

7.4.3 Register sets

A set of registers that are used in the same way by the instructionset can be defined.

pattern reg means [$EBP|EBX|ESI|EDI|ECX|EAX|EDX|ESP$**] ;**

pattern breg means[$AL|AH|BL|BH|CL|CH|DL|DH$**];**

All registers in an register set should be of the same length.

7.4.4 Register Stacks

Whilst some machines have registers organised as an array, another class of machines, those oriented around postfix instructionsets, have register stacks.

The ILCG syntax allows register stacks to be declared:

register stack (8)ieee64 FP assembles[' '] ;

Two access operations are supported on stacks:

PUSH is a void dyadic operator taking a stack of type ref t as first argument and a value of type t as the second argument. Thus we might have:

PUSH(FP,↑mem(20))

POP is a monadic operator returning t on stacks of type t. So we might have

mem(20):=POP(FP)

7.5 Instruction formats

An instruction format is an abstraction over a class of concrete instructions. It abstracts over particular operations and types thereof whilst specifying how arguments can be combined.

instruction pattern

RR(operator op, anyreg r1, anyreg r2, int t)

means[r1:=(t) op(↑((ref t) r1),↑((ref t) r2))]

assembles[op ' ' r1 ',' r2];

In the above example, we specify a register to register instruction format that uses the first register as a source and a destination whilst the second register is only a destination. The result is returned in register r1.

We might however wish to have a more powerful abstraction, which was capable of taking more abstract apecifications for its arguments. For example, many machines allow arguments to instructions to be addressing modes that can be either registers or memory references. For us to be able to specify this in an instruction format we need to be able to provide grammer non-terminals as arguments to the instruction formats.

For example we might want to be able to say

instruction pattern

RRM(operator op, reg r1, maddrmode rm, int t)

means [r1:=(t) op(↑((ref t)r1),↑((ref t) rm))]

assembles[op ' ' r1 ',' rm] ;

This implies that addrmode and reg must be non terminals. Since the non terminals required by different machines will vary, there must be a means of declaring such non-terminals in ilcg.

```
mov DWORD ecx,     1
leb4b08729615:
cmp DWORD ecx,      8
jg near  leb4b08729616
lea edi,[  ecx-(     1)]; substituting in edi with 3 occurences and score of 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)]; substituting in edi with 3 occurences and score of 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:
```

Figure 11: The result of matching the parallelised loop against the Pentium instruction set

An example would be:
pattern regindirf(reg r) means[↑(r)] assembles[r];
pattern baseplusoffsetf(reg r, signed s) means[+(↑(r) ,const s)] assembles[r '+' s];
pattern addrform means[baseplusoffsetf| regindirf];
pattern maddrmode(addrform f) means[mem(f)] assembles['[' f ']'];

This gives us a way of including non terminals as parameters to patterns. Instruction patterns can also specify vector operations as in:
instruction pattern PADDD(mreg m, mrmaddrmode ma)
means[(ref int32 vector(2)m:=(int32 vector(2))+((int32 vector(2))↑(m),(int32 vector(2))↑(ma)))]
assembles ['paddd 'm ',' ma];

Here vector casts are used to specify that the result register will hold the type **int32 vector(2)**, and to constrain the types of the arguments and results of the + operator.

7.6 Instructionsets

At the end of an ILCG machine description file, the instructionset is defined. This is given as an ordered list of instruction patterns. When generating code the patterns are applied in the order in which they are specified until a complete match of a statement has been achieved. If a partial match fails the code generator backtracks and attempts the next instruction. Care has to be taken to place the most general and powerfull instructions first. Figure 11 illustrates this showing the effect of matching the parallelised loop shown in figure 10 against the Pentium instruction set. Note that incrementing the loop counter is performed using load effective address (lea) since this, being more general, occurs before add in the instruction list.

This pattern matching with backtracking can potentially be slow, so the code generator operates in learning mode. For each subtree that it has sucessfully matched, it stores a string representation of the tree in a hash table along with the instruction that matched it. When a tree has to be matched, it checks the hash table to see if an equivalent tree has already been recognised. In this way common idioms only have to be fully analysed the first time that they are encountered.

8 Conclusions

Vector Pascal currently runs under Windows98 , Windows2000 and Linux. Separate compilation using Turbo Pascal style units is supported. C calling conventions allow use of existing libraries. Work is underway to port the BLAS library to Vector Pascal, and to develop an IDE and literate programming system for it.

Vector Pascal provides an effective approach to providing a programming environment for multi-media instruction sets. It borrows abstraction mechanisms that have a long history of sucessfull use in interpretive programming languages, combining these with modern compiler techniques to target SIMD instruction sets. It provides a uniform source language that can target multiple different processors without the programmer having to think about the target machine. Use of Java as the implementation language aids portability of the compiler accross operating systems.

References

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

[2] Aho, A.V., Ganapathi, M, TJiang S.W.K., Code Generation Using Tree Matching and Dynamic Programming, ACM Trans, Programming Languages and Systems 11, no.4, 1989, pp.491-516.

[3] Blelloch, G. E., NESL: A Nested Data-Parallel Language, Carnegie Mellon University, CMU-CS-95-170, Sept 1995.

[4] Burke, Chris, J User Manual, ISI, 1995.

[5] Cattell R. G. G., Automatic derivation of code generators from machine descriptions, ACM Transactions on Programming Languages and Systems, 2(2), pp. 173-190, April 1980.

[6] Chaitin. G., Elegant Lisp Programs, in The Limits of Mathematics, pp. 29-56, Springer, 1997.

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

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

[9] Ewing, A. K., Richardson, H., Simpson, A. D., Kulkarni, R., Writing Data Parallel Programs with High Performance Fortran, Edinburgh Parallel Computing Centre, Ver 1.3.1.

[10] Gagnon, E., SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK, School of Computer Science McGill University, Montreal , March 1998.

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

[12] Hadjiyiannis, G., Hanono, S. and Devadas, S., ISDL: an Instruction Set Description Language for Retargetability, DAC'97, ACM.

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

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

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

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

[17] Iverson, K. E. . Notation as a tool of thought. Communications of the ACM, 23(8), 444-465, 1980.

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

[19] Iverson, Kenneth E., J Introduction and Dictionary, Iverson Software Inc. (ISI), Toronto, Ontario, 1995.

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

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

[22] Leupers, R., Niemmann, R. and Marwedel, P. Methods for Retargetable DSP Code Generation, VLSI Signal Processing 94, IEEE.

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

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

[25] Peleg, A., Wilke S., Weiser U., Intel MMX for Multimedia PCs, Comm. ACM, vol 40, no. 1 1997.

[26] Ramsey, N. and Fernandez, M., ACM Transactions on Programming Languages and Systems, Vol. 19, No. 3, 1997, pp492-524.

[27] Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., and Schonberg, E., Programming with Sets: An Introduction to SETL (1986), Springer-Verlag, New York

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

[29] Tannenbaum, A. S., A Tutorial on ALGOL 68, Computing Surveys, Vol. 8, No. 2, June 1976, p.155-190.

[30] Turner, D., An overview of MIRANDA, SIGPLAN Notices, December 1986.

[31] van der Meulen, S. G.,ALGOL 68 Might Have Beens, SIGPLAN notices Vol. 12, No. 6, 1977.

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