Orthogonal Parallel Processing in Vector Pascal

Paul Cockshott [a,*] Greg Michaelson [b]

[a] *Imaging Faraday Partnership, Department of Computing Science, University of Glasgow, Scotland, +44 141 330 3125*

[b] *School of Mathematical and Computer Sciences, Heriot Watt University, Edinburgh, Scotland +44 131 451 3422*

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. To fully exploit the SIMD model of computation such operations offer, programmers depend on CPU specific machine code or implementation dependent libraries.

Here we present Vector Pascal (VP), a language designed to enable the elegant and efficient expression of SIMD algorithms. VP 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. The type system is also 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 formalised descriptions of the instruction set semantics.

Key words: Pascal, Parallelism, SIMD, Image Processing, Vector Processing, Data Parallel

1 Introduction

The subject of the principles and laws of notation is so important that it is desirable, before it is too late, that the scientific academies of the world should each contribute the results of their own examination and conclusions, and that some congress should assemble to discuss them. Perhaps it might

* Corresponding Author

Email addresses: `wpc@dcs.gla.ac.uk` (Paul Cockshott), `greg@macs.hw.ac.uk` (Greg Michaelson).

be better still if each academy would draw up its own views, illustrated by examples, and have a sufficient number printed to send to all other academics. Charles Babbage, Passages from the Life of a Philosopher, 1864, in P. Morrison and E. Morrison (Eds), Charles Babbage and his Calculating Engines, Dover, 1961, p72.

There has long been a tension between programming language and processor design. Languages may offer constructs that seem hard to realise in the machine code supported by processors. In turn, processors, and related hardware may offer functionality whose exploitation in languages appears non-trivial.

For example, where early languages allowed the expression of decimal fraction and floating point arithmetic, early processors only offered integer arithmetic based on sequential decimal or parallel binary operations. For many years, floating point units were relatively expensive add ons, only becoming integral parts of microprocessors in the 1980's.

Similarly, heap storage with garbage collection first came to prominence for list processing in LISP in the late 1950's. BASIC, a widely used language on 70's and 80's mini- and micro-computers, had implicit heap management for strings. Ada from the 80's and Java from the 90's bother offered implicit generalised heap storage management. Nonetheless, contemporary CPUs still do not support automatic heaps.

This disjunction is most stark when type abstractions are considered. Programming languages have been evolving with increasingly rich type systems since Algol 68, supporting sequence and discriminated union structures with varying degrees of polymorphism over control as well as data. However, despite experiments in supporting more complex types in processors, typically through tag-data architectures such as the various LISP machines and the object oriented Rekursiv[1], CPUs remain stubbornly rooted in untyped, flat, binary representations.

In contrast, all contemporary personal computers contain graphics cards but no significant language provides graphics abstractions. Instead, graphics operations are still supported by low level calls to driver routines or through implementation dependent libraries.

Most recently, single instruction/multiple data (SIMD) extensions have been added to common CPU instruction sets [2–5]. These are popularly termed multi-media extensions (MMX) as they were provided to support, and are widely used for, real-time interactive games. MMX instructions are based on standard digital signal processing (DSP) repertoires, supporting parallel multi-word [1] arithmetic and logic. Where the instruction operand width is the same

[1] Here we regard a word, measured in bits, as the basic unit of processing within

as, or a small multiple of, the CPU/memory bus width, such MMX extensions offer close to linear speedup over the equivalent single word operations.

The attraction of MMX instructions for computer games is that pixel representations tend to be small multiples of eight bit bytes and so several pixels can fit into a single MMX operand. In principle, however, such instructions could be used wherever logical and arithmetic operations are performed over base types represented as multiple bytes.

However, the wider exploitation of MMX instructions has been held back by two factors. First of all, few commercial compilers make effective use of these instruction sets in a machine independent manner, despite considerable research [6–9]. Secondly, most popular programming languages were designed on the word at a time model of the classic von Neuman computer rather than on the SIMD model.

We have been exploring the design and implementation of an efficient and elegant notation for exploiting SIMD operations on enhanced CPUs. We are using the word *elegant* in the information theoretic sense introduced by Chaiten[10], implying maximal conciseness. Elegance is a goal that one approaches asymtotically; approaching but never attaining. However, an inevitable consequence of elegance is a loss of redundancy which reduces human intelligability.

We have borrowed concepts for expressing data parallelism that have a long history, dating back to Iverson's work on APL in the early '60s[11]. APL programs are as concise, or even more concise, than conventional mathematical notation[12] and use a special character set. However, this makes them hard for the uninitiated to understand. Iverson's J[13] attempts to remedy this by restricting itself to the ASCII character set, but still looks dauntingly unfamiliar to programmers brought up on more conventional languages. Thus, a central aim in developing our new language Vector Pascal[14,15], has been to provide the conceptual gains of Iverson's notation within a framework familiar to imperative programmers.

Subsequent sections discuss abstractions for data parallelism and their expression in other languages. Vector Pascal's treatment of SIMD operations is then introduced, and shown to be considerably more concise and efficient than Pascal without these orthogonal extensions. Finally, the architecture independent expression and architecture dependent realisation in the ILCG compiler for Vector Pascal are considered.

---

a CPU. Thus we refer to a 16/32/64 bit word processor, rather than a 2/4/8 byte processor.

2 Array mechanisms for data parallelism

Data parallel programming can be built up from certain underlying abstractions[16]:

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

We will next consider these in more detail and look at their support in other languages, in particular J, Fortran 90[16] and NESL[17].

In the following, we will denote a vector of n elements x_i as:

$\{x_1, x_2, ...x_n\}$

and an m by n array of elements a_{ij} as:

$\{\{a_{11}, a_{12}, ...a_{1n}\},$
$\{a_{21}, a_{22}, ...a_{2n}\},$
$...$
$\{a_{m1}, a_{m2}, ...a_{mn}\}\}$

2.1 Operations on whole arrays

The basic *conceptual* mechanism for whole array operations is the *map*, which takes an operator and one or more source arrays, and produces a result array by mapping the source(s) under the operator. Let us denote the type of an array of elements of type T as $T[]$. 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 $k_i = x_i + y_i$, for example:

$\{1, 2, 3, 5\} + \{1, 2, 4, 8\} \rightarrow \{2, 4, 7, 13\}$

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 $z_i = x_i^2$, for example:

$sqr(\{1, 2, 4, 8\}) \rightarrow \{1, 4, 16, 64\}$

Map replaces the *bounded iteration* 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. Iverson[11] provided a consistent treatment of these. Recent languages built round this model are J, an interpretive language[18,19,13], High Performance Fortran[16], F[20] a modern Fortran subset and NESL an applicative data parallel language. In principle 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 for it. In particular, *map* may be exploited through implementation independent *algorithmic skeletons* [21] based on parallel templates for process farms which are instantiated with appropriate sequential arguments from the original source program[22].

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:

1. `REAL,DIMENSION(64)::A,B`
2. `A=3.0`
3. `B=B+SQRT(A)*0.5`

would be legal, and would operate in a pointwise fashion on the whole arrays. Thus, line 1 initialises every element of array `A` to `3.0` and line 2 sets each element of array `B` to `0.5` times the corresponding element of `A`.

Intrinsic functions, such as SQRT, are defined to operate either on scalars or arrays, but are part of the language rather than part of a subroutine library. User defined functions over scalars do not automatically extend to array arguments.

J [2] similarly allows direct implementation of array operations, though here the array dimensions are deduced at run time:

1. > `a=. 1 2 3 5`
2. > `a`
3. `1 2 3 5`
4. > `b=. 1 2 4 8`
5. > `a+b`
6. `2 4 7 13`

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

The pair `=.` is the assignment operator in J so line 1 initialises a new array `a` of length 4 and line 4 initialises a new array `b` of length 4. Line 2 displays the value of `a` and line 5 calculates and displays the array formed by summing corresponding elements of `a` and `b`.

Unlike Fortran, J automatically overloads user defined functions over arrays:

```
7. > sqr=.^&2
8. > c=.1 2 4 8
9. > c+(sqr a)*0.5
10. 1.5 4 8.5 20.5
```

Here, line 7 defines a new monadic function `sqr` by partially applying the binary power function `^` to the exponent 2. Line 8 then initialises array `c` and line 3 calculates and displays the array formed by adding each element of `c` to half the square of the corresponding element of `a`.

The functional language NESL provides similar generality. The first J example above could be expressed as:

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

and the second example as:

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

The expressions in { } brackets termed Apply-to-Each constructs, also known as comprehensions, are descended from the ZF notations used in SETL[23] and MIRANDA[24]. Thus line 1 finds the sum of the successive elements of the *sequences* `[1,2,3,5]` bound to `a` and `[1,2,4,8]` bound to `b`. Similarly, line 3 finds the sum of successive elements of `b` and half the square of the successive elements of `a`.

Again user defined functions can be applied element wise to sequences.

2.2 Array slicing

It is advantageous for many applications to be able to specify sections of arrays as values in expression. The sections may be rows or columns in a matrix, or 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[25].

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

Fig. 1. column, row and sub-array slices

The notion of array slicing was introduced to imperative languages by ALGOL 68[26]. In ALGOL 68 if `x` has been declared as `[1:10]INT x`, then `x[2:6]` would be a slice consisting of the second through the sixth elements inclusive that could be used on the right of an 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. For example:

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

would be equivalent to the nested loop:

```
  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. For example:

```
1.> a=. 2*i.10
2.> a
3.  0 2 4 6 8 10 12 14 16 18
4.> 3{a
5.  6
```

Here, `i.`n is a function which produces a list of the first n elements of an array starting with element 0. Line 1 constructs an array where each element is double its subscript. "{" is the sequence subscription operator so line 4 selects the element at index `3`.

Selection of a subsequence is performed by forming a sequence of indices. For example:

```
6. > (2+i.3){a
7.   4 6 8
```

In line 6, the expression `2+i.3` forms the sequence `2 3 4` which then subscripts the array `a`.

NESL does not offer a direct equivalent to slicing.

2.3 *Conditional operations*

Much data parallel programming is based on the application of some operation to a subset of the data selected through a mask. This can be thought of as 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 that element of `B`:

$$\begin{array}{cc} \text{A} & \{1,2,4,8\} \\ \text{B} & \{2,3,4,5\} \\ \text{A<B} & \{1,1,0,0\} \\ \hline & \{2,3,4,8\} \end{array}$$

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

```
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. For example:

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

In line 1, `a` and `b` are constrained by the requirement that each element of `a` must be less than the corresponding element of `b`.

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 is injected between the elements of a vector or the rows or columns of a matrix to produce a result of lower rank. Examples include forming the sum or finding the maximum or minimum of a table. For example, `+` would reduce $\{1, 2, 4, 8\}$ to $1 + 2 + 4 + 8 = 15$.

The first systematic treatment of reduction operations in programming languages is due to Iverson[11]. His *reduction functional* takes a dyadic operator and, by currying, generates a tailored reduction function. In APL and J the reduction functional is denoted by `/`. Thus `+/` is the function which forms the sum of an array:

```
1. > a
2.    1 2 3 5
3. > +/a
4.    11
```

In line 3 the reduction `+/a` expands to $(1 + (2 + (3 + (4 + 0))))$.

The interpretation of reduction for non commutative operators is slightly less obvious. Consider:

```
5. > -/a
6.    _3
```

In line 6, _3 is the J notation for -3, derived from the expansion of $(1 - (2 - (3 - 4(-0))))$ from `-/a` in line 5. 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 commutative operators: `SUM, PRODUCT, MAXVAL, MINVAL`. NESL likewise provides a limited set of reduction functions `sum, minval, maxval, any, all`. `any` and `all` are boolean reductions: `any` returns `true` if at least one element of a sequence is true i.e. disjunctive reduction; `all` returns `true` if they are all true i.e. conjunctive reduction.

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, for example to transpose a vector or matrix or to shift the elements of a vector. For example:

$\{\{1, 2, 4\}, \{8, 16, 32\}\}$

transposes to:

$\{\{1, 8\}, \{2, 16\}, \{4, 32\}\}$

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

$b = 0.25a = \{0.25, 0.5, 1, 2\}$

$c = 0.5a = \{0.5, 1, 2, 4\}$

The result is then defined as the sum under shifts of b, c:

$\{1, 2, 4, 8\}convolve(\{0.25, 0.5, 0.25\}) \rightarrow$
$\{0.5, 1, 2, 2\} + \{0.5, 1, 2, 4\} + \{0.25, 0.25, 0.5, 1\} \rightarrow$
$\{1.25, 2.25, 4.5, 7\}$

This example replicates the trailing value when shifting. In other circumstances, for example when dealing with cellular automata, 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 example, 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/))
```

to give 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 similar operations on sequences to those provided on arrays by Fortran 90. For example, if:

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

then:

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

is equivalenet to the Fortran above.

3 Data parallelism in Vector Pascal

3.1 Language Decisions

In seeking to exploit new programming concepts one may either design a new language or adapt an existing language. Designing a new language is high risk in terms of the effort to be expended in developing new tools and promoting a core community before any wider take up is likely.

Occam[27] represents a salutary object lesson. This language was intended for a novel architecture, the transputer, and had its own formal logic, CSP. However, occam was never made adequately available on non-transputer archictectures, and the transputer was overpriced and complex compared with the Intel/Motorola hegemony. Now only CSP survives, having found a niche as as a language- and architecture-independent formal notation.

For data parallelism, APL and J represented radical breaks from their contemporaries, introducing novel notations. We think that this was an important factor in limiting their wider use. Overall, experience suggests that new concepts gain provenence if they are presented in a familiar guise and if their use involves low additional cost for the benefits it brings.

An existing language may be adapted through the introduction of new notation or through the overloading of existing notation. Both approaches involve modifications to existing language processors or the development of new ones. Furthermore, both approaches may lose backwards compatibility with the original. Finding a principled basis for adding a new notation to an extant language is problematic.

For example, the late 80's and 90's saw a variety of attempts to extend C and C++ with parallel programming concepts. Johnston[28] lists:

- CC++ with `par` and `parfor` constructs;
- C** with aggregate classes and concurrent element nomination;
- Mentat with aggregate classes and explicit parallel methods;
- pC++ with concepts from High Performance Fortran.

In the same period. Lattice Logic Limited (3L) developed their Parallel C based on occam-like constructs[29]. All of these represent well thought through extensions but none of these languages has gained widespread acceptance. We speculate that, in part, this was because the extensions did not build naturally on existing constructs.

NESL was strongly grounded in the functional language tradition. For example, it's sequences and APPLY-TO-EACH are effectively overloadings of lists and list comprehensions. NESL has influenced recent research into extending Standard ML for data parallelism. However, because overall the functional paradigm is far less familar than the imperative paradigm, functional languages in general have still to gain wider currency beyond their academic constituencies.

High Performance Fortran (HPF) is based on a combination of overloading standard Fortran notation for arrays and operators, and the introduction of new notation, for example for conditional operations and slicing. HPF provides a relatively transparent extension to the widely used Fortran and represents the most successfull data parallel language to date, enjoying wide use in the scientific and technical communities.

Our Vector Pascal extends the array type mechanism of Pascal to provide support for data parallel programming in general, and SIMD processing in particular. Wherever possible,rather than introducing new constructs, we have sought to increase orthogonality in Strachey's sense[30] by overloading extant notation. As most MMX extensions support arithmetic and logical operations over byte sequences, a central concern in choosing a host language was the degree to which the corresponding operators were already overloaded.

Pascal[31] was chosen as a base language over the alternatives of C and Java. C overloads arithmetic operators to include address manipulation, often with implicit type coercions. Thus, these operators could not also be used to express data parallelism over structures. For example, Java overloads `+` both for string concatenation and to coerce other base types to string when they are `+`ed with strings. This precludes the use of `+` as a data parallel operation for combining, as opposed to joining, arrays.

Pascal has other advantages in providing additional notations which can be overloaded consistently for data parallelism. For example, the sub-range notation is a natural basis for slicing.

3.2 Assignment maps

Standard Pascal allows assignment of whole arrays. Vector Pascal extends this to allow consistent use of mixed rank expressions on the right hand side of an assignment. For example, given:

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

then we can write:

1. `r1:=1/2;`
2. `r2:=r1*3;`
3. `r1:=rdu + r2;`
4. `r1:=r1+r2[1];`

Line 1 assign `0.5` to each element of `r1`. Line 2 assign `1.5` to every element of `r2`. Line 3 uses the reduce operator `rdu` to set each element of `r1` to the totals along the coresponding row of `r2`. Line 4 increments each element of `r1` with the corresponding elements of row 1 of `r2`.

These may be translated directly to standard Pascal through iteration:

1'. `for i:=0 to 7 do r1[i]:=1/2;`

2'. `for i:=0 to 7 do for j:=0 to 7 do r2[i,j]:=r1[j]*3;`

3'.
```
for i:=0 to 7 do
begin
     t:=0;
     for j:=7 downto 0 do t:=r2[i,j]+t;
     r1[i]:=t;
end;
```

4'. `for i:=0 to 7 do r1[i]:=r1[i]+r2[1,i];`

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 the operator `ndx`.

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 1 and 2 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, and `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.

3.3 Slice operations

Vector Pascal extends the array abstraction to define sub-ranges of arrays. A sub-range is denoted by the array 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`.

For example:

```
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. For example:

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

inverts all of `screen`s pixels in the subarray from (40,60) to (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 3.6.

3.4 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 above 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. Thus, the above assignment would be equivalent to:

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

were the above legal Pascal[3].

(2) If the code is not parelleliseable it is translated as equivalent to a standard if statement. Thus, 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 definitions and parallel data selection.

3.5 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 `rdu`. Thus if `a` is an array, `rdu + a` denotes the sum over the array. More generally `rdu`Φx 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 `rdu+` for $\sum$, `rdu*` for $\prod$ etc[4].

The dot product of two vectors can thus be written as:

```
x:= rdu +(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 of 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:=rdu⊙a;
```

[3] 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`.

[4] For those who prefer a more APL style, `rdu` can also be written \ as in `\+a` to sum the elements of an array

is equivalent to:

```
var temp:t;
    i:low..high;
temp:= identity(t,⊙);
for i:= high downto low do temp:=a[i]⊙temp;
x:=temp;
```

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

3.6 Array reorganisation

Array reorganisation involves *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.

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

The form `ndx` i returns the ith current implicit index. Thus, the sequence:

```
v1:array[1..3] of integer;
v2:array[0..4] of integer;
...
v1:= ndx 0;
v2:= ndx 0  * 2;
```

would set:

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

In contrast, given the sequence:

```
m1:array[1..3,0..4] of integer;
m2:array[0..4,1..3]of integer;
m2:= ndx 0 + 2 * ndx 1;
```

would set m2 to:

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

The argument to ndx 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 permutation on the implicit indices. Thus in e evaluated in context `perm[`i, j, k`]`e, then:

`ndx 0 = ndx` i, `ndx 1= ndx` j, `ndx 2= ndx` k

This is particularly useful 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:

```
type rowindex=0..479;
      colindex=0..639;
var channel=red..alpha;
     screen:array[rowindex,colindex,channel] of pixel;
     img:array[channel,colindex,rowindex] of pixel;
...
screen:=perm[2,0,1]img;
```

`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, matrix, or tensor. [5] 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:

`ndx` *0* `..` `ndx` *n*

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

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

```
ndx'0 .. ndx'n
```

where:

```
ndx'x = ndx ( (x+1) mod n+1)
```

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

For example, 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
```

3.6.1 Array shifts

The shifts and rotations of arrays in Fortran 90 and NESL are not supported by any explicit Vector Pascal operator, though one can use a combination of other features to achieve them. For example, given:

```
var a,b:array[0..n-1] of integer;
```

a left rotation can achieved as:

```
a:=b[(1+ ndx 0)mod n];
```

and a reversal by:

```
a:=b[n-1 - ndx 0];
```

3.6.2 Element permutation

Permutations are widely used in APL and J programming, an example being sorting an array `a` into descending order using the J expression `\:a{a`. This 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. The use of analogous constructs requires the ability to index one array by another. 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]]`.

For example, given the sequence:

```
const perm:array[0..3] of integer=(3,1,2,0);
var ma,m0:array[0..3] of integer;
...
m0:= ( ndx 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
```

3.7 Efficiency considerations

Expressions involving transposed vectors, matrix diagonals, and permuted vectors, or indexing by expressions involving modular arithmetic on `ndx`, do not parallelise well on SIMD architectures like the MMX. These depend upon the fetching of blocks of adjacent elements into the vector registers which requires that element addresses be adjacent and monotonically increasing. Assignments involving re-mapped vectors are usually handled by scalar registers.

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 [6] and unioning [7]. However for image processing it lacks support for pixels and images. Given the vintage of the language this is not surprising and can be readily overcome using existing language features. Thus pixels may be defined as a subrange 0..255 of the integers, and images may be modeled as two dimensional arrays.

However, 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 non-montonic due to wrap-round. Pixel values of $100 + 200 = 300$, is in 8 bit precision 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, for example in executing a convolution kernel, it is necessary 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 through the use of 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 above difficulties. As a signed representation it lends itself to subtraction. As an unbiased representation, it makes the adjustment of contrast easier. For example, one can reduce contrast 50% simply by multiplying an image by 0.5 [8]. 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.

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

Dimensional analysis is familiar to scientists and engineers and provides a routine check on the sanity of mathematical expressions. Dimensions cannot be expressed in the otherwise rigorous 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. In particular, 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 type identifier must be a member of a Pascal 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 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 `*` and `/` 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.

The rules for deducing dimensions are:

(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 and Overloading

5.1 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 most multi-media instruction-sets.

5.1.1 Inner Product of Vectors

Given the `v`, `w` are one dimensional arrays then `v.w` is the scalar product of the two vectors.

If `M` is a two dimensional array and `v` a vector, `M.v` produces the vector transformed by the matrix M. This has obvious applications in graphics. If `a,b` are two dimensional arrays then `a.b` applies the standard equation for matrix multiplication:

$$c_{ik} = \sum_{s=1}^{p} a_{is} b_{sk} \tag{1}$$

where A is of order $(m \times p)$and B is of order $(p \times n)$ to give a resulting matrix C of order $(m \times n)$. Vector and matrix multiplication are implemented with in-line code allowing context specific optimisations to be used and extend to any types for which addition and multiplication operators are defined.

5.2 Unary operators

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

In standard Pascal some of these operators are treated as functions. Syntactically this means that their arguments must be enclosed in brackets, as in

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

`sin(theta)`. This usage remains syntactically correct in Vector Pascal.

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 the divide operator can be used monadicaly, for example `/2` meaning `1/2` with 1 as the multiplicative identity element. Similarly, for sets the notation `-s` means the complement of the set `s`. The identity elements inserted are given in table 1.

5.3 Operator overloading

The dyadic operators can be extended to operate on new types by operator overloading. Figure 9 shows how arithmetic on the type `complex` required by Extended Pascal [32] 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 [9] . When parsing expressions, the compiler first tries to resolve operations in terms 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 9, 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

[9] Vector Pascal allows function results to be of any non-procedural type.

```
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  }
operator + = complex_add,complexzero;
operator / = complex_divide,complexone;
operator * = complex_multiply,complexone;
operator - = complex_subtract,complexzero;
```

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.

Fig. 2. Defining operations on complex numbers

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 *map*s and array reductions.

6 Example: image filtering

As an example of Vector Pascal we will consider 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 k is a convolution vector, then the corresponding matrix K is such that $K_{i,j} = k_i k_j$.

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: $A_{y,x} = c_1 T_{y,x-1} + c_2 T_{y,x} + c_3 T_{y,x+1}$.

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 are 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 6 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 4 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 processing applications. Fixed point pixel arithmetic has the advantage that it can be efficently implemented in parallel using multi-media instructions.

The data-parallel implementation is considerably 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 op-

```
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;
```

Fig. 3. 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;
```

Fig. 4. Vector Pascal implementation of the 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
	DevPascal	486	62
pconv	Vector Pascal	486	80
	Vector Pascal	Pentium + MMX	817

Table 2
Comparative Performance on Convolution

erations per second on a 1GHz Athlon. 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 instruction set. `pconv` also runs a third faster than `conv` when it is targeted at the 486 instruction set, which in effect, serialises the code.

For comparison `conv` was run on other Pascal Compilers [10], DevPascal 1.9, Borland Pascal and its successor Delphi [11]. These are extended implementations, but with no support for vector arithmetic. Delphi is a state of the art commercial compiler, as Borland Pascal was when released in 1992. DevPas is a recent free compiler. In all 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, and DevPascal is roughly comparable to Vector Pascal.

Further performance comparisons are given in table 3. Here:

- DevP - Dev Pascal version 1.9
- TMT - TMT Pascal version 3
- BP 286 - Borland Pascal compiler with 287 instructions enabled range checks off.
- DP 486 - Delphi version 4
- VP 486 - Vector Pascal targeted at a 486
- VP K6 - Vector Pascal targeted at an AMD K6

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

[10] In addition to those shown the tests were perfomed on PascalX, which failed either to compile or to run the benchmarks. TMT Pascal failed to run the convolution test.
[11] version 4

DevP	TMT	BP 286	DP 486	VP 486	VP K6	test
71	80	46	166	333	2329	unsigned byte additions
55	57	38	110	225	2329	saturated unsigned byte additions
85	59	47	285	349	635	32 bit integer additions
66	74	39	124	367	1165	16 bit integer additions
47	10	33	250	367	582	real additions
49	46	23	98	188	2330	pixel additions
67	14	39	99	158	998	pixel multiplications
47	10	32	161	164	665	real dot product
79	58	33	440	517	465	integer dot product

Table 3
Performance on vector kernels

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 is the dot product operation coded as

```
r:=rdu + 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 for vector arithmetic is consistently better than that of other compilers. The exception to this is 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.

For an indicator of the performance of Vector Pascal on other instruction mixes, the Dhrystone Pascal benchmark was used, shown in Table 4. Due to timing quantization all figures are accurate to only about 4%. These tests indicate that Vector Pascal on such instruction mixes is comparable to other Pascal compilers. The two outliers are Pascal X and Delphi. The Pascal X

Compiler	Dhrystones per sec
Pascal X	4317
Turbo Pascal 4	142373
TMT Pascal	159461
Borland Pascal	183144
Vector Pascal	190246
DevPascal	201612
Delphi	357142

Table 4
Dhrystone performance. All measurements were performed on a 266 Mhz Intel Pentium.

```
Find L the minimum of the lengths of x and y.
Allocate a new array R of length L.
For i = 0 to L-1 set Rᵢ= primeval( xᵢ, yᵢ, ω)
Return R
```

Fig. 5. Interpretive array expression evaluation

system is a p-code interpreter, and as such is slower. The Delphi compiler is faster than the others, possibly due its use of registers rather than the stack for parameter passing in procedure calls.

7 Integrating Array Optimisations

An implementation of array valued expressions poses significant design choices. These will affect the efficiency of the compiled code. In order to bring out some of the issues involved in compiling such expressions let us first consider how an interpretive language might handle them. Initially consider evaluating $x\omega y$ where ω is a primitive operator like addition or multiplication. If the interpreter is written in C then the array expression can be evaluated by a generalised function of the form:

```
arrayt eval(arrayt x, arrayt y, char omega)
```

where `arrayt` is a type representing a pointer to an array descriptor on the heap. When the function `eval` returns it will have allocated a new array on the heap into which it will have written the result of applying the operation `omega` to corresponding elements of `x` and `y`. If we abstract from the problem of handling arrays of different rank, and consider intially the case where `x` and `y` are vectors then this can be done with the algorithm in Figure 5.

Here *primeval* is a function that evaluates the scalar expression $a\omega b$, for scalars a, b. We chose an interpretation of $x\omega y$ that returns an array whose length is the minimum of the lengths of x, y for simplicity of explanation.

This is clearly a very naive interpretive algorithm since it costs a function call for each primitive evaluation. Efficiency can be greatly improved by using a case statement to branch on ω prior to for loops:

```
switch(omega){
case ('+'):for(i=0;i<L;i++)R[i]=x[i]+y[i]; break;
case ('-'):for(i=0;i<L;i++)R[i]=x[i]-y[i]; break;
... }
```

In this case, the efficiency of the interpretive code can be quite high, indeed it appears to tends to the limiting efficiency of compiled code as $L \to \infty$. This sort of optimisation explains why the performance of good interpretive array languages can be comparable with compiled code.

However the final run time performance of a program though crucially dependent on the number of arithmetic instructions in inner loops, the optimisation addressed above, also depends upon other factors. Among these are the efficient use of the storage hierarchy and the ability to exploit vector instructions.

Suppose we evaluate the assignment statement `z←x+y`. Prior to the execution of the statement we have 3 buffers in memory corresponding to the arrays referenced by the three variables. We disregard the case where `z` has no initial value. The operation results in the discarding of the original buffer for `z` which becomes garbage. In a long running algorithm this will have to be recovered. This is storage overhead becomes more significant when we consider more complex array expressions like

```
z ← a*(b-c)
```

In this case a temporary array will be created for the result of `b-c` prior to the array multiply. The first order cost of recovering the space will depend on the sort of storage management algorithm used, and in particular on whether the algorithm always clears any buffer that it allocates. If it is not cleared, then the allocation and recovery will be independent of the buffer size. If the buffer is cleared, then this imposes an overhead of at least one memory store for each pair of primitive operands evaluated in an array expression. By itself, this is a relatively small overhead which can be avoided if there is a non-initialising call to the buffer allocation API. But it becomes more significant once we consider the interaction of buffer allocation with the memory hierarchy.

The levels to consider are registers, cache and main memory and in very large applications virtual memory. The interpretive implementation makes no

```
t1=memalloc(b_min_c);
for(i=0;i<b_min_c;i++)t1[i]=b[i]-c[i];
t2=memalloc(t1_min_a);
for(i=0;i<t1_min_a;i++)t2[i]=a[i]*t1[i];
dispose(z);
z=t2;
```

Fig. 6. Naive compiled algorithm.

```
for(i=0;i<zlen;i++)
 z[i]=a[i]*(b[i]-c[i]);
```

Fig. 7. Optimal version.

attempt to exploit registers and makes very inefficient use of the cache. Since the result of each primitive binary operation is written to memory, and since the memory block is always newly allocated, it will tend not to be present in cache. When it is written to for the first time, and in this approach buffers tend to be written to only once, the result will be stored in cache. By the time the multiplication in `a*(b-c)`is done, the result of the subtraction is in cache, so the multiplication proceeds relatively fast. But a cost of storing the result of subtraction in cache is that an equivalent number of bytes of other data must be purged from the cache. If the arrays are sufficiently large relative to cache sizes, then the result of the subtraction could flush the array `a` out of the cache. Given that access to main memory can be 20 times slower than cache access such cache flushing could seriously affect performance.

A naive array expression compiler would generate code whose dynamic execution would model the dynamic execution of an interpreter. We will model the compilation process by producing equivalent C code. Clearly one route to compilation is to translate the array code into C and leave the machine dependent part of the compilation to the code generator. We will see later that this is not always optimal. The statement `z` $\leftarrow$ `a*(b-c)` would map to the code shown in Figure 7.

For simplicity we assume that the sizes of the result arrays `b_min_c` etc, have been precomputed either at compile time or in an earlier code fragment. This C code, whilst it has removed the interpretive overheads still retains the same cache penalties as the interpretive code and memory allocation penalties as the original interpretive version.

An alternative approach would be to decompose the array operations so that the whole expression was evaluated as a single loop equivalent to the C code shown in Figure 7. This approach, termed pull-through, has a number of advantages. In this example we have actually carried out two transformations, we have removed the redundant creation of temporary vectors, and also reorganised the code so that the final result is written into the original buffer z,

vector length	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
naive μs	0.053	0.047	0.056	0.082	0.088	0.105
optimal μs	0.036	0.032	0.037	0.048	0.057	0.071
naive/optimal	1.48	1.48	1.49	1.70	1.53	1.48

Table 5
Crusoe performance on `x:=a*(b-c)` implemented in registers versus naively.

vector length	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
naive μs	0.016	0.02	0.021	0.029	0.029	0.03
optimal μs	0.009	0.012	0.0019	0.013	0.014	0.015
naive/optimal	1.66	1.61	1.11	2.2	2.13	2.03

Table 6
P4 performance on `x:=a*(b-c)` implemented in registers versus naively.

rather than a newly created buffer z. This optimisation is possible so long as z already references a buffer of the appropriate size, something that a compiler can usually determine.

The expression `a[i]*(b[i]-c[i])` is a scalar one, and as such can be effectively computed within registers. This gives us two advantages:

(1) Since the temporary result of `b[i]-c[i]` is in a register, it can be accessed without any store accesses when needed for the subsequent multiply.
(2) Since no cache writes occur for the vector result of b-c, the elements of the array a are more likely to be in cache when needed.

Table 5 shows the Crusoe performance on `x:=a*(b-c)` implemented in registers compared with the naive implementation. Code was generated using 16 bit numbers with MMX vectorisation enabled. The total number of integer operations were held constant as the vector length varies by altering the number of times the operation is repeated. The table shows times in microseconds to perform the arithmetic to produce each 16 bit result averaged over a large number of runs on a 750MHz Crusoe processor with a 64KB primary cache. The program was compiled using the Vector Pascal compiler. The naive version was emulated by splitting the statement into two parts thus `f:=b-c;x:=a*f;` Table 6 shows P4 performance on `x:=a*(b-c)` implemented in registers compared with the naive implementation. Vectorisation was enabled. Again, the table shows times in microseconds to perform the arithmetic to produce each 16 bit result. A 1300MHz P4 processor with a 8KB primary cache and 256k secondary cache was used. Compilation was as described in Table 5.

The broad features to note are :

- As vector lengths rise, the time taken to produce each byte of the result rises, this is due to generally poorer locality of access and thus generally poorer cache utilisation for larger vectors.

- In all cases the optimal code which evaluates temporary results in registers performs better than the naive code which allocates temporaries to memory buffers. In some cases it performs twice as well.
- The degree of advantage of the optimal code varies non-linearly with vector length as a result of interaction of the vector block sizes with the levels of the cache hierarchy. With certain combinations of sizes of vector and cache, the temporaries produced by the initial subtraction do not interfere with the source data. In these cases the performance penalty for the naive implementation is less.

We have argued against evaluating each binary array operator to produce a new array value, because it is more efficient to combine all subexpressions within a loop operating on a scalar expression. So long as one is dealing with vector or matrix expressions of the form $\mathbf{M} \leftarrow \mathbf{A}\omega_1\mathbf{B}\omega_2\mathbf{C}$ where ω_1, ω_2 are pairwise scalar operators, such as + or -, which are mapped over the matrices, no problems arise. The semantics of both implementation approaches are the same. When we allow other operators such a matrix transpose or matrix multiply, problems can arise.

Suppose we transpose the matrix $\begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$ using the assignment $\mathbf{A} \leftarrow \mathbf{B}^T$, then $\mathbf{A}$ ends up with $\begin{matrix} 1 & 3 \\ 2 & 4 \end{matrix}$, and if we evaluated $\mathbf{B} \leftarrow \mathbf{B}^T$ we would expect $\mathbf{B}$ to also equal $\begin{matrix} 1 & 3 \\ 2 & 4 \end{matrix}$. However if we have translated $X \leftarrow Y^T$ for 2 by 2 matrices X, Y, as:

```
for(i=0;i<2;i++)
 for(j=0;j<2;j++)
  X[i,j]=Y[j,i]
```

using this approach $\mathbf{B} \leftarrow \mathbf{B}^T$ would result in the corruption of B as the transpose was being evaluated giving $\begin{matrix} 1 & 2 \\ 2 & 4 \end{matrix}$. A similar corruption would occur with the evaluation of some matrix products by simple nested loops: $\mathbf{A} \leftarrow \mathbf{B}.\mathbf{C}$ would be safe but $\mathbf{A} \leftarrow \mathbf{B}.\mathbf{A}$would not. It is to obviate these dangers that Fortran 90 defines the semantics of array assignments as having to evaluate the entire array valued expression on the right hand side of the assignment prior to performing the assignment itself. Vector Pascal takes the opposite approach, specifying that the semantics of array expressions is defined in terms of the equivalent loop construct with the compiler signaling an error if it encounters a data flow hazard in an array expression. Since, in the presence of parameter aliasing, it is not always possible to detect such data-flow hazards at compile time, one can argue that the Fortran 90 approach is safer. The additional cost incurred by the Fortran 90 semantics can be avoided in those cases where data-flow analysis can prove that no hazard exists - for instance where the arrays are not function parameters.

It is highly desirable that an array language have some system of array bounds checking to detect errors. The simplest way to do this is to plant explicit tests on array bounds whenever an array is indexed. Using C as an algorithmic notation this would take the general form:

```
if(i<lwb\_a)~boundsfault();
if(i>upb\_a)boundsfault();
b=a{[}i{]};
```

This is clearly expensive and would dominate the cost of actual array access. But this can be done with as little as one extra instruction on an Intel processor using the:

```
BOUND r,pair
```

instruction where `r` is a register and pair is reference to a two element vector in memory. This checks whether a register `r` is between the values `pair[0]`, and `pair[1]` generating an interrupt if it fails. Since a modern processor will be able to fetch 64 bits at time from memory, both bounds can be accessed in a single memory fetch, but the extra instruction and extra memory fetch can still represent an overhead of 50% or more on the original cost of array access. It is thus desirable to try and minimise the number of array bounds checks that are done.

Suppose we start with a loop with naive bounds checking:

```
for(i=x;i<=y;i++){
 if(i<lwb\_a) boundsfault();
 if(i>upb\_a) boundsfault();
 total+=a[i];
}
```

Then it is clear that $x \le i \le y$ throughout the loop. Thus if an array bounds fault is to occur, either $x < lwb\_a$ or $y > upb\_a$. We can safely move the bounds check to before the start of the loop:

```
if(x<lwb_a) boundsfault();
if(y>upb_a) boundsfault();
```

	code	μs	ratio
optimal array bounds	vector	0.03	
naive array bounds	vector	0.036	
naive/optimal	vector		1.2
optimal array bounds	scalar	0.076	
naive array bounds	scalar	0.18	
naive/optimal	scalar		2.36

Table 7
Crusoe performance on x:=a*(b-c) in registers with no intermediate buffer assignment comparing naive with optimised array bounds checking.

```
for(i=x;i<=y;i++)
 total+=a[i];
```

So long as $x < y$ this formulation will have a lower overhead. Indeed in many cases it allows the bounds check to be totally eliminated. If the array bounds and loop bounds are known at compile time, then the tests can be evaluated during compilation. If they pass, the compiler can elide the bounds checking code, if they fail, it can signal a type error at compile time.

In an array language, most of the for loops that the compiler deals with are automatically generated ones arising from array assignments. These lend themselves very well to the optimisations described above. For statically sized arrays, which are the predominant ones in classical Fortran or Pascal programming, most array bounds checks can be performed at compile time. For dynamically sized arrays most bounds checks can precede the loop. An exception comes in scatter-gather statements such as: `b:=a[c]`, where c is an array. In this case we must plant code to check that each `c[i]` is within the bounds of `a`.

Table 7 shows Crusoe performance on `x:=a*(b-c)` in registers with no intermediate buffer assignment comparing naive with optimised array bounds checking. The same test conditions apply as in Table 5. The vector lengths are 1024.

The propagation of array bounds checking outside of loops can make modest but very worthwhile improvement to performance for vectorised code, but makes a very big difference to performance on non-vectorised code. For the vectorised code, array bounds are checked for only 25% of the entries anyway, reducing the advantage of optimising the checks.

Other obvious optimisations applied by the compiler are loop unrolling and vectorisation. If we now consider the combined effects of all of the optimisa-

	improvement contribution	cumulative improvement
registers versus temporary array results	1.48	1.48
vectorisation	4.74	7.01
optimal array bound checks	1.2	8.41
Loop unrolling	1.55	13.0

Table 8
The cumulative effects on performance of all the array optimisations.

tions we can see that they speed array expressions up more than 10 times (table 8). The gains from each optimisation combine multiplicatively. The exact figures given for each of the optimisations will depend on the machine it is being run on, the vector lengths and the efficiency of other aspects of the compilation process. In our examples we have used 16 bit arithmetic because this vectorises on any common-denominator Pentium compatible machine. The vectorisation gains for floating point code can be less.

8 Implementation

At the heart of the Vector Pascal implementation is the machine independent Intermediate Language for Code Generation (ILCG)[33]. Its purpose is to act as an input to an automatically constructed code generator, working on the syntax matching principles described in [34]. 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.

ILCG is strongly typed, supporting the base types common in most programming languages along with type constructors for vectors, stacks and references. Of particular relevance to Vector Pascal, operators may be implicitly overloaded for multi-word operations.

8.1 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 such exploitation involves two approaches. First of all, operators are overloaded to allow array-valued expressions, as discussed above. Secondly, loops are 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. Cheong and Lam [6] 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 [8] 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 [9] exploit Intel MMX parallelism from C with SUIF, using a variety of vectorisation techniques to generate inline assembly language, achieving speedups from 2 to 6.5. All of these groups target specific architectures. Larsen [35] and Amarasinghe report also using SUIF to detect general parallelism in C code using the Motorola Alti-vec intructions. Finally, Leupers [7] 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' [36] MIMOLA language allows the expression of both programs and structural hardware descriptions, driving the micro-code compiler MSSQ. Hadjiyiannis' [37] Instruction Set Description Language and Ramsey and Davidson's [38] 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, we thought it 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 conceptually very similar MMX operations.

8.2 Intermediate Language for Code Generation

Our Intermediate Language for Code Generation (ILCG) is at the heart of a code-generator-generator system, driven by specifications of both machines and languages.

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

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.

8.2.1 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`.

8.2.2 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'];
```

8.2.3 Register sets

A set of registers that are used in the same way by the instruction set 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.

8.2.4 Register Stacks

Whilst some machines have registers organised as an array, another class of machines, those oriented around postfix instruction sets, 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)`

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 specifications 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, addrmode 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 is a means of declaring such non-terminals in ILCG.

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.

8.4 Instruction Sets

At the end of an ILCG machine description file, the instruction set 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 12 illustrates this showing the effect of matching the parallelised loop shown in figure 11 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.

see end of document

Fig. 8. System Architecture

8.4.1 Instruction set deficiencies

The SIMD instruction-sets show signs of having been designed from the standpoint of assembler programmers. Deficiencies become apparent when they are treated as targets for more general array languages. Our experience in writing formal descriptions of the SIMD instruction sets provided by Intel and AMD leads to the conclusion that there are serious non-orthogonalities in the instructions. These non-orthogonalities cause problems for the efficient parallelisation of code.

By far the most serious of these is the lack of scalar to vector instructions. The processors have reasonably efficient means of operating on scalar quantities and on vector quantities, but there is no means by which scalars can be added to vectors, vectors multiplied by scalars etc. In the absence of built in scalar to vector operations, the code generator has to replicate scalar quantities at run time before operationg on a vector with them. Since no means of addressing the individual words of a vector register are provided, the replication has to be done in memory with consequent store access costs.

Another lack of orthogonality is the requirement in the PIII that the operands of a floating point vector instruction be aligned on 16 byte boundaries. This can not in general be guaranteed in a language that supports array slicing, which in turn forces the use of register to register instructions and relatively slower un-aligned load and store instructions.

8.5 Vector Pascal Compilation

A Vector Pascal program is translated into an ILCG abstract semantic tree implemented as a Java data structure. The tree is then passed to a machine generated Java class corresponding to the code generator for the target machine. 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, it defaults to iteration over the whole array.

The structure of the Vector Pascal system is shown in figure 8.5. 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[39], 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 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.
iii. The resulting ilcg tree (7) is walked over by a class that encapsulates the semantics of the target machine's instruction set (10). Code generators, automatically generated from machine specifications written in ILCG, follow the pattern matching approach described in[40,41,34].

    For example Pentium.class is produced from a file Pentium.ilc (8), in ILCG, which gives a semantic description of the Pentium's instruction set. This is processed by a code-generator generator which builds the source file Pentium.java. 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.

Selection of target machines is by a compile time switch which causes the appropriate code generator class to be dynamically loaded. Code generator classes currently exist for the Intel 486, Pentium with MMX, and P3 and also the AMD K6. Output assembler code is processed using the NASM assembler and linked using the gcc loader.

8.6 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;
```

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

Fig. 9. 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]));
 };
```

Fig. 10. Parallelised loop

would first be translated to the ILCG sequence shown in figure 9.

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 10.

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 11. 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]));
```

Fig. 11. Application of `simplify` to the tree

9 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. It has its own IDE and literate programming environment[14]. It has been ported to the vector instruction-sets of the AMD Athlon, and Intel MMX, SSE and SSE2 processor models. Ports are underway to the Opteron and Sony Play-station.

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.

```
 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:
```

Fig. 12. Matching the parallelised loop against the Pentium instruction set

References

[1] Harland, D., "Rekursiv: Object-Oriented Computer Architecture", Ellis Horwood, 1991,(ISBN: 0-7458-0396-2).

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

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

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

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

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

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

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

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

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

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

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

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

[14] Cockshott, P., Renfrew, K., SIMD Programming Manual for Linux, Springer Verlag, 2004.

[15] Cockshott, P., Vector Pascal an Array Language for Multimedia Code, Proc. APL 2002 Conf., pp 83-91.

[16] 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.

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

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

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

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

[21] Cole, M., Algorithmic Skeletons: Structured Management of Parallel Computation, Research Monographs in Parallel and Distributed Computing, Pitman, 1989.

[22] Michaelson, G., Scaife, N., Bristow, P., and King, P., Nested algorithmic skeletons from higher order functions, Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing, Vol. 16, pp 181-206, Aug 2001

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

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

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

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

[27] Inmos Ltd, occam2 Reference Manual, prentice-Hall, 1988.

[28] Johnston, D., C++ Parallel Systems, ECH: Engineering Computing Newsletter, No. 55, Daresbury Laboratory/Rutherford Appleton Laboratory, March 1995,pp 6-7.

[29] 3L Limited, Parallel C V2.2, Software Product Description, 1995.

[30] Strachey, C., Fundamental Concepts of Programming Languages, University of Oxford, 1967.

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

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

[33] Cockshott, P, Direct Compilation of High Level Languages for Multi-media Instruction-sets, TR2000-72, Department of Computer Science, University of Glasgow, Nov 2000.

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

[35] Larsen, S., Amarasinghe, S, Exploiting Superword Level Parallelism with Multimedia Instruction Sets, ACM Conf., Programming Language Design and Implementation, Vancouver, 2000, pp 145-156.

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

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

[38] Ramsey, N. and Fernandez, M., Specifying Representations of Machine Instructions, ACM Transactions on Programming Languages and Systems, Vol. 19, No. 3, 1997, pp492-524.

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

[40] 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.

[41] 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.

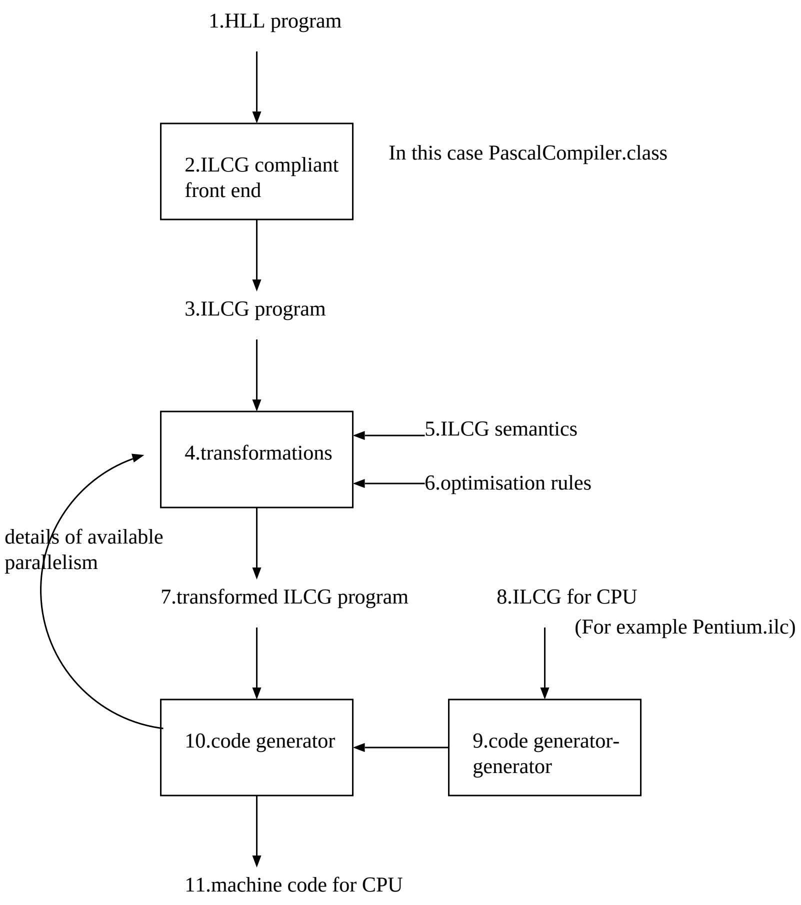
[42] Gagnon, E., SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK, School of Computer Science, McGill University, Montreal, March 1998.-

Biographical Sketches

Dr Paul Cockshott is a Reader in Computing Science at the University of Glasgow in Scotland. His research interests include the design and implementation of programming languages, 3D computer vision, data compression and novel hardware.

Dr Greg Michaelson is a Senior Lecturer and Head of Computer Science at Heriot-Watt University in Edinburgh, Scotland. His research interests include formally motivated computing, functional programming, parallel computing, and programming language design and implementation.

Drawings



the above is figure 8.5