

From SIMD to Micro-Grids

Paul Cockshott

University of Glasgow, Department of Computing Science

I. COMPILING TO SIMD PARALLELISM

Most commodity microprocessors now support multi-media instructions. These instruction-set extensions are typically based on the Single Instruction-stream Multiple Data-stream (SIMD) model in which a single instruction causes the same mathematical operation to be carried out on many operands, or pairs of operands at the same time.

The multi-media instructions on these machines perform operations between vector registers. If u, v are vector registers, with each register having multiple sub-fields, i.e., u_0, u_1, u_2, \dots , then an instruction

PADD u, v

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

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

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

$$\begin{array}{r} 11001000 \\ + 10101111 \\ \hline = 1\ 01110111 \end{array}$$

dropping the leading 1 gives $01110111 = 119$, which is dimmer than either of the original pixels. Here it would be most appropriate to return 255, representing white.

To avoid such errors, image processing code using 8 bit values has contain tests to check if values are going out of range, and force all out of range values to the appropriate extremes of the ranges. This is shown in Figure 1. Such checks and coercions inevitably slow down the computation of inner loops. Besides introducing additional instructions, the tests involve conditional branches and pipeline stalls. Intel processors provide saturated arithmetic operations on the vector registers which keep arithmetic results within bounds. When implemented within the ALU hardware, saturated arithmetic can execute in a single clock cycle. The combined effect of the use of packed data and saturated types can be to produce a significant increase in code density and performance.

Consider the C code in Figure 1 to add 2 images in $v1$ and $v2$, storing the result in the image in $v3$. The code includes a check to prevent overflow. Compiled into assembler

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

Fig. 1. C code to add two images.

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

Fig. 2. Assembler version of the Figure 1 program.

code by the Intel C compiler the resulting assembler code has 18 instructions in the inner loop whereas the hand coded assembler inner loop in Figure 2 uses only 5 instructions. Note that this example assumes that registers `esi` is initialised to 0 and that `ecx` is initialised to `LEN/8`.

Furthermore, the MMX code processes 8 times as much data per iteration, thus requiring only 0.625 instructions per byte processed. The C code executes 29 times as many instructions. Whilst some of this can be put down to the superiority of hand assembled versus automatically compiled code, the combination of the SIMD model and the saturated arithmetic are a major factor.

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

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

This approach leads to efficient code, but the incursion

of assembler concepts into the high level language impedes portability between processors.

- 2) Other C compilers[6] [23] [22] [4] [28] have used classical vectorization techniques. Here, the compiler detects potential parallelism in ordinary C loops and attempts to map them onto vector operations. All of these compilers except Leupers' [23] target specific architectures. Leupers has reported a C compiler that uses vectorising optimisation techniques for compiling code for the multimedia instruction sets of some signal processors, but this is not generalised to the types of processors used in desktop computers.

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

High level language notations for expressing data parallelism have a long history, dating back to APL in the early '60s[17]. The key concept here is the systematic overloading of all scalar operators to work on arrays.

The most recent languages to be built round this model are J, an interpretive language[18], ZPL [27] an array language for multi-processors and F[25] a modern Fortran extension. Unlike the SIMD classes provided by the Intel C++ compiler, the array languages are machine independent. These can be implemented using scalar instructions or using the SIMD model without change of behaviour apart from speed differences.

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

Pascal is a classic imperative language whose first implementations were on CDC super-computers[20]. It has been an important language both in its own right, and as an influence on subsequent computer languages such as Ada[2], Modula[31] and Delphi [19]. It was originally developed for instructional purposes, and widely taught as a first programming language.

The British Standards Institute started work on standardising the language in the late 1970s, this led to a British Standard BS6192[3] for the language in 1982. The following year International Standard 7185[16], published by the ISO incorporated the British Standard for Pascal with minor amendments. This standardisation process coincided with a rapid growth in the use of the language. Its early implementation on microprocessors, gave it a large following among both Apple and PC programmers.

As a growing body of commercial code started to be developed in Pascal, weaknesses of the standard, particularly in the areas of support for separate compilation, variable sized arrays and in string handling became evident. These were addressed in two ways. On the one hand commercial implementations, in particular the influential Turbo-Pascal on the PC, adopted a separate compilation system based on Units, and a system of length encoded strings. On the other hand the standardisation effort led to International Standard 10206 :Extended Pascal[15]. Extended Pascal supported modules for

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

TABLE I
 EXTENDED PASCAL FEATURES SUPPORTED

separate compilation. It handled strings and variable sized arrays as part of an elegant system of parameterised types termed *schemata*. Today relatively few implementations of Extended Pascal exist [26], [21], whereas many compilers exist that use Turbo-Pascal style strings and Units.

No further Pascal standards have been agreed since 1990. Since then a number of object oriented implementation of Pascal have been released : [19], [26], [5] . In 1993 the Pascal Standards committee produced a draft proposal for object oriented extensions to Pascal, but this has not been formally adopted.

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

- 1) support the expression of data parallel operations in a manner independent of the instruction-set available.
- 2) develop a mechanism for automatically generating good machine code for data parallel operations.
- 3) enable the automatic re-targeting of the code generator to different SIMD and non-SIMD instruction-sets based on a formal description of these instruction-sets.
- 4) provide a toolkit that was operating system and machine independent.
- 5) to generate at least two code generators for scalar and SIMD instruction-sets on target architectures.

Vector Pascal is described in [8], [9], [10], so we will give only a summary account here. It constitutes a Pascal base language with array extensions and incorporates many, but not all, features of Extended Pascal. The features of Extended Pascal that Vector Pascal supports are given in table I In order to ease software migration the base language was designed to be upward compatible with Turbo Pascal, so the Turbo Pascal Unit system and string representations are used in preference to those given in [16].

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

The Vector Pascal compiler is written in Java. The lexical analyser is built using JLex while the syntax analyser uses the classic recursive descent parsing techniques employed in Wirth's original Pascal compiler. The syntax analyser builds an abstract program tree which is passed to a code generator built using the ILCG code-generator-generator system[7].

The code generators are automatically generated from ma-

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

TABLE II
 VECTOR PASCAL EXTENSIONS

chine specifications written in ILCG and follow the pattern matching approach described in [11]. ILCG is a strongly typed language which supports vector data types and the mapping of operators over vectors. It is well suited to describing SIMD instruction sets. The code generator classes export from their interfaces details about the degree of parallelism supported for each data-type. This is used by the front end compiler to iterate over arrays longer than those supported by the underlying machine. Where supported, parallelism is unitary; this defaults to iteration over the whole array.

Selection of target machines is by a compile time switch which causes the appropriate code generator class to be dynamically loaded. The structure of the Vector Pascal system is shown in figure 4.

The path followed from a source file is:

- The source file (1) is parsed by a Java class `PascalCompiler.class` (2), a hand written, recursive descent parser, and results in a Java data structure (3), an ILCG tree, which is basically a semantic tree for the program.
- The resulting tree is transformed (4) from sequential to parallel form and machine independent optimisations are performed. Since ILCG trees are Java objects, they can contain methods to self-optimize. Each class contains for instance a method `eval` which attempts to evaluate a tree at compile time. A further method `simplify` applies generic machine independent transformations to the code. Thus the `simplify` method of the class `For` can perform loop unrolling, removal of redundant loops etc. Other methods allow tree walkers to apply context specific transformations.
- The resulting ILCG tree (7) is walked over by a class that encapsulates the semantics of the target machine's instruction-set (10); for example `Pentium.class`. During code generation the tree is further transformed, as machine specific register optimisations are performed. The output of this process is an assembler file (11).
- The assembler file is then fed through an appropriate assembler and linker, assumed to be externally provided, to generate an executable program.

The Vector Pascal system currently supports several architec-

tures and both Windows and Linux. ILCG specifications exist for the Intel 486, the Intel Pentium with MMX, and for the Intel PIII and PIV, Amd K6 and Opteron, Sony PlayStation 2 processors. Code generators built from these specifications are distributed in the standard release.

The parser has to slightly modify standard recursive descent techniques [30]. The modifications arise from additional context sensitive features of the grammar and from the desire to collect information during the parsing that will aid vectorisation. Normally one would have a parameter-less recursive procedure for each non-terminal of the language and code generation would occur as a side effect. We have modified this technique as follows:

- 1) The procedures are replaced with recursive functions each of which returns an ILCG tree, or throws an exception on a parse error.
- 2) For expressions, each parsing function is passed a vector $[i, j, k, \dots]$ whose elements are implicitly declared variables to be used in indexing arrays.
- 3) A number of global variables are used to define the context within which an expression is being parsed.
- 4) A number of global flags and counters are used to collect information for vectorisation.

Array operations

Vector Pascal allows statements that apply binary operators to whole arrays to return whole arrays, through overloading of standard Pascal scalar notation. For example:

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

where

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

sets matrix `a` to the result of multiplying matrix `b` by vector `c`.

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

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

- 1) if `x` is of rank > 2 a type error is throw.
- 2) if `rank(x) = 2`, `x` is replaced with an ILCG tree for `x[i,j]`.
- 3) if `rank(x) = 1`, `x` is replaced with an ILCG tree for `x[j]`.
- 4) if `rank(x) = 0`, `x` is returned.

The combined effect of this is to replace `a:=b*c` with `a[i,j]:=b[i,j]*b[j]`. The equivalent sequential Pascal of the array assignment would be

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

ILCG is a relatively high level tree language and allows for loops. The parser initially generates serial code for all constructs. It then interrogates the current code generator class to determine the degree of parallelism possible for the types of operations performed in a loop, and if these are greater than

one, it vectorises the code. Simpler array operations can be passed directly to the code generator in APL style. Given the declaration in figure 3 A the assignment would be translated to the ILCG sequence shown in figure 3 B and then to the machine code in section C of the figure.

Graphics types

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

In the implementation pixels are represented as 8 bit fixed point signed binary fractions. Addition and subtraction of pixel vectors is performed using saturated arithmetic. Multiplication is more complex. The ILCG saturated multiplication pattern maps to a sequence of instructions that emulate an 8 way parallel multiplication, retaining the high order bits of the result. This has to be done using a couple of 4 way parallel multiplications. In the specification of the MMX instructionset registers 5,6 and 7 are declared as reserved. This prevents the code generator from allocating them to hold temporaries, and allows them to be used for this and other multi-instruction patterns.

An indication of the accelerations that can be achieved by the use of graphics data types and SIMD instructions is provided in Tables III and IV.

II. COMPILING TO MICROGRIDS

Machines like the Sony Ps2 and the Cell incorporate the sort of SIMD instructions to which Vector Pascal was targeted. However they also introduce two new levels of complexity.

- 1) Hetrogenous instructionsets - each chip contains several processing units which may run different instructionsets.
- 2) Disjoint memory spaces - in order to achieve high performance, the memory on some of these processing units takes the form of private on-chip random access store not shared with other processors.

Thus when working on the PS2 compiler we were faced with new problems. Since the capabilities of the processing units differed - some being better at vectorised floating point operations for example, it was desirable to run code fragments on the processors to which they would be best suited. Since the processors ran in parallel, a new form of parallelism quite distinct from SIMD data parallelism was being introduced. The fact that the vector units on the PS2 did not share memory space with each other or with the MIPs control processor, meant that some mechanism for partitioning code and data between the units had to be provided.

For this purposes we are investigating combining the technologies already developed in SAC[12] for multi-processor parallelism and in Vector Pascal for SIMD parallelism. If it were possible to combine the techniques they use very substantial gains in performance should be possible.

SAC works by carrying out multiple transformations of the source resulting in an output file that is in ISO C which is

then fed into a existing C compiler to generate machine code. The approach taken by SAC allowed its developers to abstract from the problems of different target machine architectures by taking advantage of the near universal availability of good C compilers.

Vector Pascal depends upon the ILCG code-generator-generator system. This uses a language, ILCG, which allows one to specify a typed semantics for machine instruction-sets. The type system of ILCG includes the most common base types of current machines, and also vectors over these types. The expression syntax of ILCG supports APL type operator overloading. These features allow it to be used to describe the instructionlevel parallelism of modern SIMD instruction-sets. An ILCG compiler then translates these machine specifications into code generator modules, expressed in either Java or Pascal. When an appropriately typed abstract syntax tree is passed to one of these code generators it will then produce vectorised assembler code targeted at the processor whose ILCG description originally specified it.

The base types of SAC are taken from ISO C. Those of Vector Pascal come from ISO Pascal. However because Vector Pascal is particularly targeted at image processing and signal processing applications a fixed point data type to represent pixels has been added. Arithmetic on this type is defined to be saturating to meet the requirements of parallel signal processing. It would probably be necessary to introduce such saturated fixed point types into SAC.

The array types of each language extend those of the parent language. SAC allows arrays whose shape and rank may both be unknown until run time. Vector Pascal uses the ISO Extended Pascal schematic type mechanism to allow arrays whose shape may be determined at run time, but their rank must still be statically specified. This makes the maximal abstraction level of Vector Pascal functions somewhat lower than those of SAC.

These similarities and differences between the two languages and their compilers suggests a strategy for the development of a new compilation system that would combine the strengths of each.

As an initial approach we are working on replacing the C compiler which performs the final code generation for SAC with the Vector Pascal compiler. This strategy may be summarised as follows:

- The existing SAC multiprocessor parallelism will be used to target the multiple cores on a processor.
- The interface between SAC and VP will be functional abstraction over the innermost SAC loops.
- The SAC compiler now generates a file containing a set of C functions that are SAC callable, and which contain all the free vars of the inner loops in the function signatures. Call this file A_SIMD.c where A is a name the name of the SAC file.
- A call will be placed to the function abstractions in the inner loops of the SAC code.
- There is a translator from standard C to Vector Pascal which generates a Pascal library unit containing functions whose call interface is identical to the original C functions this file is called A_SIMD.pas thus a function that was

called foo will now be A_SIMD_foo

- The Pascal code is translated to assembler file called A_SIMD.asm which is translated to a A_SIMD.o This phase will bring into play the SIMD optimiser of the Vector Pascal compiler.
- The function will have for each free array variable 2 parameters the first will be a pointer to the value of the array, the second will be a pointer to the descriptor of the array as a *int. Any information in the descriptor that is used will be copied out to local variables by C code planted by the SAC compiler obviating the need for the C→Pascal translator to know about the descriptor structure.

Status

Work on the PS2 compiler has reached the stage that allows programs to run in SIMD mode on the MIPS control processor using VPU0, but the DMA transfer of parameters to units has not yet been implemented. We can demonstrate the translation of SAC code to Vector Pascal which uses SIMD instructions on Intel processors, but we do not yet have performance data on the combined code.

Conclusion

The language Vector Pascal already provides a number of the key elements needed to support microgrids. In particular it has

- 1) Support for the SIMD model of programming supported by machines like the Cell.
- 2) A readily retargetable backend that allows configuration to new instructionsets.
- 3) Dynamically loadable code generators so that heterogeneous code can be output.

In conjunction with the SAC compiler it offers the possibility to make use of both SIMD and multi-core parallelism on a range of new processors.

REFERENCES

- [1] Advanced Micro Devices, 3DNow! Technology Manual, 1999.
- [2] ANSI, The Programming Language Ada Reference Manual, Springer-Verlag, February, 1983.
- [3] BSI, Specification for the Computer programming language Pascal, British Standard BS6192, 1982.
- [4] Bik, A. J. C., Girkar, M., Grey, P. M., Tian, X., Automatic Intra-Register Vectorization for the Intel Architecture, International Journal of Parallel Programming, Vol 30, No. 2, 2002, pp. 65..97.
- [5] Canneyt, Michael, Free Pascal Reference Guide, Jan 2001, (<http://www.freepascal.org/docs-html/user/user.html>).
- [6] Cheong, G., and Lam, M., An Optimizer for Multimedia Instruction Sets, 2nd SUIF Workshop, Stanford University, August 1997.
- [7] Cockshott, P., Direct Compilation of High Level Languages for Multimedia Instruction-sets, Department of Computer Science, TR-2000-72, University of Glasgow, Nov 2000.
- [8] Cockshott, P., Vector Pascal Reference Manual, Department of Computer Science, TR-2002-16, University of Glasgow, Feb 2002.
- [9] Cockshott, P., The Abstraction Mechanisms of Vector Pascal, Vector, Vol. 18 No. 4, pp 100-112, April 2002.
- [10] Cockshott, P., Vector Pascal: an Array Language for Multimedia Code, APL2002, Madrid, July 2002.
- [11] Susan L. Graham, Table Driven Code Generation, IEEE Computer, Vol 13, No. 8, August 1980, pp 25..37.
- [12] Grelck, C and Scholz, S-B., SAC – From High-level Programming with Arrays to Efficient Parallel Execution, Parallel Processing Letters, 2003, VOL 13, 3, 401?412
- [13] Intel, Intel Architecture Software Developers Manual Volumes 1 and 2, 1999.
- [14] Intel, Willamette Processor Software Developer's Guide, February, 2000.
- [15] ISO, Extended Pascal ISO 10206:1990, 1991.
- [16] ISO, Pascal, ISO 7185:1990, 1991.
- [17] K. E. Iverson, A Programming Language, John Wiley & Sons, Inc., New York (1962), p. 16.
- [18] Iverson K. E. A personal View of APL, IBM Systems Journal, Vol 30 , No 4, 1991.
- [19] Kerman, Mitchel, Programming and Problem Solving with Delphi, Addison Wesley, 1001.
- [20] Jensen K., and Wirth N., Pascal User Manual and Report, Springer, 1978.
- [21] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., Ullrich, Ch.: PASCAL-XSC - Language Reference with Examples. Springer-Verlag, New York, 1992.
- [22] Krall, A., and Lelait, S., Compilation Techniques for Multimedia Processors, International Journal of Parallel Programming, Vol. 28, No. 4, pp 347-361, 2000.
- [23] Leupers, R., Compiler Optimization for Media Processors, EMMSEC 99/Sweden 1999.
- [24] Leupers, R., Code Selection for Media Processors with SIMD instructions, DATE 2000.
- [25] Metcalf, M., and Reid., J., The F Programming Language, OUP, 1996.
- [26] Prospero Software, 'Extended Pascal Language Reference Manual', Prospero Development Software, 2000.
- [27] Snyder, L., A Programmer's Guide to ZPL, MIT Press, Cambridge, Mass, 1999.
- [28] Srereman, N., and Govindarajan, G., A Vectorizing Compiler for Multimedia Extensions, International Journal of Parallel Programming, Vol. 28, No. 4, pp 363-400, 2000.
- [29] Texas Instruments, TMS320C62xx CPU and Instruction Set Reference Guide, 1998.
- [30] Watt, D. A., and Brown, D. F., Programming Language Processors in Java, Prentice Hall, 2000.
- [31] Wirth, N., Recollections about the development of Pascal, in *History of Programming Languages-II*, ACM-Press, pp 97-111, 1996.
- [32]

<pre> program vecadd; type byte=0..255; var v1,v2,v3:array[0..6399]of byte; A: i:integer; begin v3:=v1 + v2; end. </pre>	<pre> cmp DWORD[ebp+-19208],6399 jg NEAR 14847d577 mov ecx, DWORD [ebp+-19208] movq MM1, [ecx+ebp +-6400] C: paddb MM1, [ecx+ebp +-12800] movq [ecx+ebp +-19200],MM1 add DWORD [ebp+-19208], 8 jmp 14843d577 14847d577: </pre>
↓	↑
<pre> B: (ref uint8 vector (6400))mem(+((ref int32)ebp),-19200)):= +((ref uint8 vector (6400))mem(+((ref int32)ebp),-6400))), ((ref uint8 vector (6400))mem(+((ref int32)ebp),-12800)))) </pre>	

Fig. 3. Translation of an array expression via ILCG to MMX assembler. Compare figs 1 and 2.

TABLE III
COMPARATIVE COMPILER PERFORMANCES ON CONVOLUTION.

Program	Implementation	Target Processor	Million $\frac{Ops}{Second}$	Cpu
conv	Vector Pascal	Pentium + MMX	61	1 Ghz Athlon
	Borland Pascal	286 + 287	5.5	1 Ghz Athlon
	Delphi 4	486	86	1 Ghz Athlon
	DevPascal	486	62	1 Ghz Athlon
	Free Pascal	386	138	1.6Ghz Centrino
pconv	Vector Pascal	486	80	1 Ghz Athlon
	Vector Pascal	Pentium + MMX	817	1 Ghz Athlon
	Vector Pascal	Intel P3	1240	1.6Ghz Centrino

The program conv is written in standard Pascal, pconv uses vector operations and graphics types of Vector Pascal.

TABLE IV
SHOWING THE COMPARATIVE PERFORMANCE OF DIFFERENT PASCAL IMPLEMENTATIONS ON THE SIEVE OF ERASTOSTENES PROGRAM AS A FUNCTION OF SET SIZE

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

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

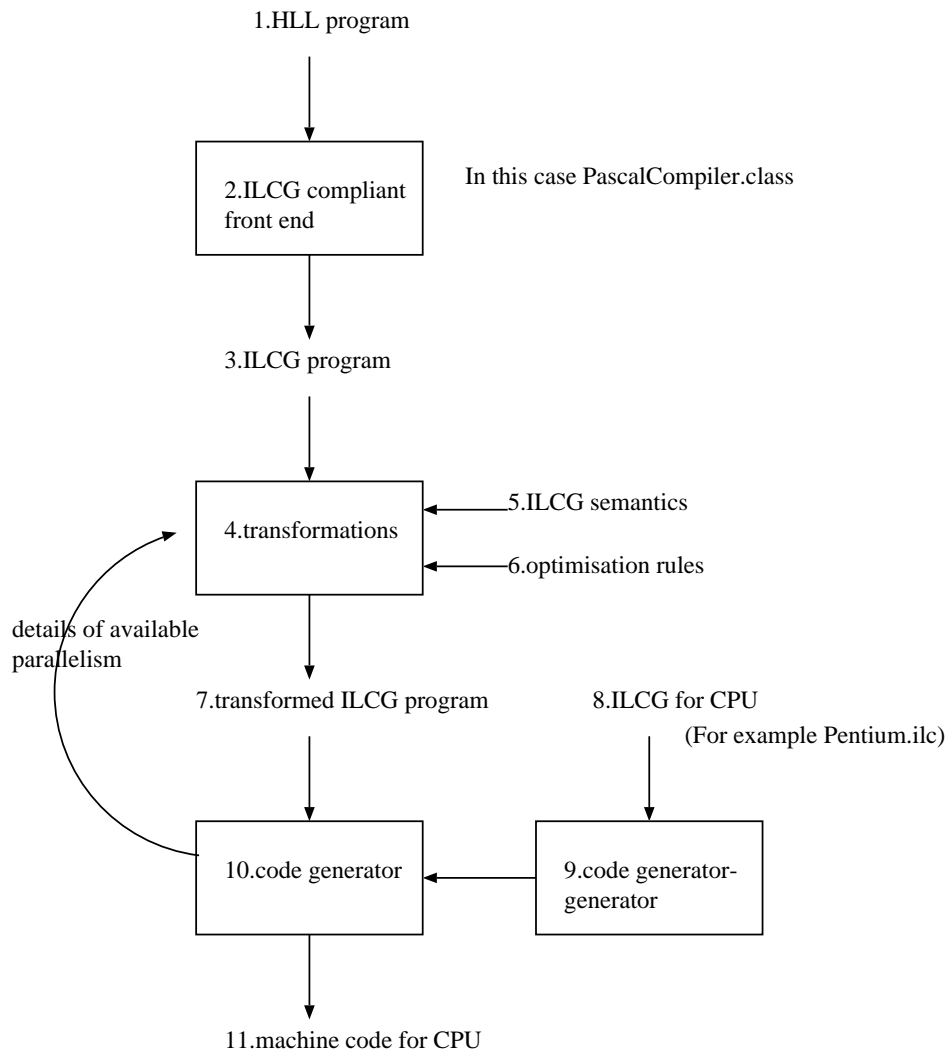


Fig. 4. Vector Pascal System Architecture

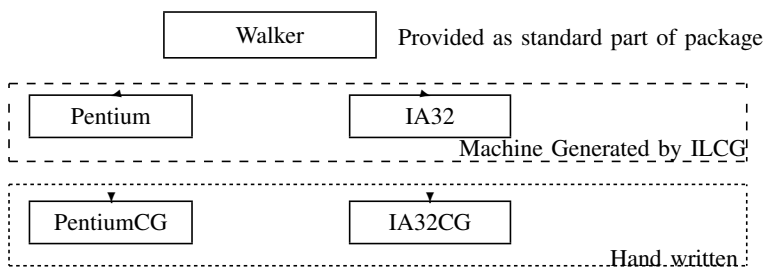


Fig. 5. How machine generated and hand written extensions descend from `ilcg.tree.Walker`.