FINAL REPORT: MERGING SINGLE ASSIGNMENT C AND VECTOR PASCAL TECHNOLOGIES.

PAUL COCKSHOTT

Introduction

This is a report on the Travel Grant EP/D/032423/1 provided by EPSRC to fund travel to investigate the merging of Single Assignment C and Vector Pascal technologies. The grant was of 3 months duration and was originally scheduled to run from July to September 2005. Due to difficulties in arranging the first visit, it actually ran from September to November.

I first present a historical background before going on to describe the work done as a result of the grant.

1. Historical background

There has always been a close relationship beween programming language design and computer design. Electronic computers and programming lanugages are both 'computers' in Turing's sense. They are systems which allow the performance of bounded universal computation. Each allows any computable function to be evaluated, up to some memory limit.

During the 1950's the first programming languges COBOL, Fortran and ALGOL came to be developed. The existence of these languages affected the design of the second and third generation computers and led to the invention of stack machines by Burroughs(Lonergan and King 1985) in 1960. These were designed to support block structured languages like ALGOL 60(Naur and Backus 1960). The influence of the B5000 and ALGOL persisted into the 1970s when the Intel 8086(Morse, Ravenel, Mazor and Pohlman 1985) was released.

During the 1970s attempts were made to directly implement high level languages in hardware most famously with the SYMBOL(Smith, Rice, Cheskey and andStephen Lundstrum 1985) computer which incorporated an interpreter and virtual storage manager for a heap based programming language directly in the hardware. In addition microcoded APL machines were proposed or implemented (Abrams 1970, Hassitt, Lageshulte and Lyon 1973, Hakami 1975). The aim of these machines was to narrow the semantic gap between machine code and the language used by applications programmers. Improved performance was a consideration but not the primary one. The invention of vector and SIMD super-computers aimed, above all, at getting the highest possible performance. Each represented a different approach to overcomming the bottleneck imposed on conventional computers by their need to fetch and update variables in memory a word at a time.

The vector processors pioneered by Cray(Russell 1985) based themselves on the principle of pipelining vector operations with the aim of delivering one floating point result every clock cycle. Instructions operated between vector registers, each of which could hold 64 floating point values. Thus a vector instruction like

V1=V2+V3

would add the corresponding elements of vectors V2 and V3 storing the result in the register V3. ICL's Distributed Array Processor (Reddaway 1973) or DAP. used 4096 small processors operating in parallel. The DAP architecture and that of Hillis's(Hillis

1986) derivative Connection Machine was refered to as SIMD - Single Instruction Multiple Datastream.

Experience with programming these classes of super-computer led to the development of new Fortran compilers (Perrott and Zarea-Aliabadi 1986). The Cray Fortran compiler took programs that were syntactically identical to standard Fortran and attempted to detect loops that could be vectorised. The DAP Fortran on the other hand allowed array parallel operations in APL style. This notation was later incorporated into standard Fortran90, High Performance Fortran(Ewing, Richardson, Simpson and Kulkarni 1998) and the Fortran90 subset F(Metcalf and Reid 1996).

The next generation of super-computers tended to be shared memory multi-processors. initially from makers like Cray but later from other manufacturers like Sun or IBM. These in turn sparked off another round of array language development with languages like ZPL (Snyder 1999), Nesl (Blelloch 1995) and SAC(Grelck and Scholz 2003, Scholz 2003) borrowing APL concepts but being targeted at multi-processors. The aim in these languages was to split array operations over a group of processors, with each processor working on part of each array.

From the late 90s, SIMD and Vector processing features started to be incorporated on commodity microprocessors. The significance of this development should be emphasised. What had once been an esoteric form of computer architecture, restricted to a few experimental super-computer sites was now the standard type of mass-produced PC. When operating on byte wide quantities, the P4 or Athlon architectures can speed their performance up by a factor of 10 or more by using the SIMD unit.

Programming these machines take two general approaches:

(1) Features of the low level machine architecture are made directly visible(Rojas and Leeser 2003). This is done either by allowing assembler macros in C code or by predefined data types which directly model types held in the vector registers: the Intel C compiler, Apple's G4 `gcc`, and Sony's Cell C++. This approach leads to efficient code, but the incursion of assembler concepts into the high level language impedes portability between processors.

(2) Other C and Fortran compilers(Cheong and Lam 1997) (Leupers 2000) (Krall and Lelait 2000) (Bik, Girkar, Grey and Tian 2002) (Srereman and Govindarjan 2000) have used classical vectorisation techiniques. Here, the compiler detects potential parallelism in ordinary C loops and attempts to map them onto vector operations. These techniques aremachine independent, effective at recognising straightforward array arithmetic, and in some cases can recognise and vectorise reduction constructs like dot-product and some constructs involving saturated arithmetic.

Another compiler, Vector Pascal (Cockshott 2004, Cockshott 2002) is a Pascal extension analogous to High Performance Fortran that has been targeted at SIMD PCs. However, like HPF, it requires the programmer to explicityly allocate and declare arrays which is markedly less flexible than what occurs in APL.

Since the development of the SIMD instructionsets to which these language developments responded, the demands for more realistic computer games has led chip designers to raise the level of parallelism further. The pioneer in this was Sony whose Emotion Engine, incorporated in the £100 Play Station 2 (PS2) included:

(1) A MIPS 64 bit core acting as the control processor.
(2) An integer SIMD unit analogous to that on the P4.
(3) A pair of Vector Processing Units (VPU), each of which could perform 4×32-bit floating point operations each clock cycle.

It incorporated both SIMD and distributed memory multi-processing. VPUs operate as independent processors fetching their own instructions. They have vector floating point registers and also scalar registers for address manipulation. Instead of cache, they each have a modest sized private high speed memory on chip. The Cell processor jointly developed by IBM and Sony for use in the PS/3 among other applications extends the model of the PS/2 by having 7 vector units each with 256K of memory on chip. These promise to be able to perform of the order of 50 Gigaflops provided that the VPUs can be kept busy. Similar architectures are being developed in other projects(Paulson 2005).

> At the crux of making supercomputer- on-a-chip systems effective ... will be the programming necessary to get the most out of the circuitry and architecture. However .... there are few programming tools and no programming languages optimized for this purpose.(Paulson 2005)

It seems plausible that only array languages which allow programmers to specify bulk operations in a data-parallel style, and which have full control over the allocation and placement of store will be able to meet these processing needs. But implementing efficient array language implementations on these machines will demand some of the most sophisticated compilers and interpreters so far developed. The aim of this research project has been to investigate a compilation technology for this new range of computer architectures. It is starting from the technologies already developed in SAC and Vector Pascal. If it were possible to combine the techniques used by the two languages 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 final significant difference relates to typing.

(1) 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.
(2) 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.

It would take the syntax of SAC but extend it with fixed point saturating types to target image processing.

It would use the SAC high level dependency analysis to exploit hyper-threading parallelism and the ILCG vectorisation technology to exploit SIMD parallelism.

2. Visits and resulting discussions

2.1. **Visit to Glasgow.** A visit was made by Sven Bodo Scholz and Clemens Grelk at the end of september to Glasgow University. Visit took place after it was initially planned due to the commitments of the visitors to other conferences etc over the summer.

A seminar was given to the Glasgow Department of Computing Science about SAC by the visitors. Discussions then took place between the visitors, Paul Cockshott and his PhD student Bin Li. Disucssion initially focussed on how it might be possible to translate between the intermediate format used by the SAC compiler - a C tree documented in XML and the intermediate format used by Vector Pascal - a Java tree structure. The aim would be to use the vectorising code generator of Vector Pascal to support SAC programs.

Considerable difficulties were encountered in this so an alternative approach was suggested by Grelck, that the inner loops of SAC be translated to Vector Pascal rather than into C as they currently were. This would enable them to be compiled in SIMD fashion by an unmodified Vector Pascal compiler.

As an intitial feasibility study the SAC compiler was modified whilst the visitors were here to allow it to put out the inner loops into distinct files allowing them to be replaced. As an experiment we hand translated an example inner loop to Vector Pascal and tested it. The example showed that the Vector Pascal was slightly faster than gcc on this example involving 32 bit integer array arithmetic. Note that this data-type is not optimal for MMX vectorisation which performs best for 8 bit data. At most an acceleration of two fold would be possible for 32 bit integers.

Examination of the assembler code produced for this example indicated that there were several optimisations possible and work was done subsequent to the visit at Glasgow University to implement these optimisations. By early november we were able to demonstrate the following improvement:

Version	Time
simd via Vector Pascal :	1.61s
no-simd via gcc :	2.25s

Test done on a P4m 2.00 GHz. This 40% improvement in performance was taken to validate the feasibility of the proposed approach, since it indicates that a substantially greater improvement will be possible once the range of data-types supported by SAC is extended.

Another achievement of the visit to Glasgow was the porting of Single Assignment C to the AMD Opteron - the standard machine used in our Lab.

2.2. **Meeting with Codeplay.** Representatives of the Compiler company Codeplay came to Glasgow for discussions with Clemens Grelck and I during his visit. Codeplay are the producers of the Vector C compiler, and are the one company in Europe whose commercial work most closely relates to the research carried out with Single Assignment C and Vector Pascal.

It was agreed that there was potential for the transfer of the technologies in developed by the academics to the contemporary compiler industry. A subsequent meeting then took place between myself, Codeplay and a representative of Ageia who are producing a novel vector processing chip for the computer games industry. Tentative agreement was reached for a Knowledge Transfer Partnership agreement to be started between Codeplay and Glasgow to transfer the code generation technology to industry. The preparation of this proposal is now underway.

2.3. **Work by Bin Li.** The strategy we have adopted for utilising the Vector Pascal(VP) codegenerator with Single Assignment C (SAC) is for SAC compiler to produce C files defining the inner loops of array expressions. These will be translated to VP and compiled to a library which will then be linked with the main SAC program. The VP compiler will have vectorised the inner loops.

A key stage in this is for us to be able to translate standard C for-loops into vectorisable VP statements (these are at a similar semantic level to APL statements). The PhD student Bin Li is currently engaged in this task and is utilizing the McGill University Sable toolkit for this purpose.

2.4. **Visit to Luebeck.** On the 10th of November a return visit to Luebeck was made by Paul Cockshott. A seminar was given on Vector Pascal and a more detailed strategy for the inter-language translation process was agreed between him Kai Trojahner and Clemens Grelck. This strategy may be summarised as follows:

(1) Interface between SAC and VP will be functional abstraction over the innermost SAC loops.
(2) The SAC compiler will generate a file containing a set of C function 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.
(3) A call will be placed to the function abstractions in the inner loops of the SAC code.
(4) There will be a translator from standard C to vector pascal which will generate a pascal library unit containing functions whose call interface is identical to the original C functions this file will be called A_simd.pas thus a function that was called foo will now be A_simd_foo
(5) The pascal code will be translated to assembler file called A_simd.asm which will be tranlated to a A_simd.o
(6) The function will have for each free array variable it will have 2 parametes 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 obriating the need for the C→Pascal translator to know about the descriptor structure.

3. Dissemination

We shall await the experimental results from the work currently underway prior to preparing papers for journals or conferences on the compilation technique being developed.

4. Grant Proposal

I am in the process of preparing an EPSRC respsonsive mode grant application for follow on funding to cover work over an beyond what can be done by a PhD student. This would include :

(1) Extending the type system of SAC to support the fixed point types needed for high performance image and graphics programming.
(2) Developing automatic partitioning of SAC array expressions and data to allow targeting to machines like the Sony Cell.
(3) Extending the machine description language ILCG to allow description of chips with multiple internal processors each of which may have a different instruction-set.
(4) Codeing a significant graphics application in SAC to evaluate performance. This is likely to derive from one of the computer vision applications in use at the computer graphics lab at Glasgow University.

5. Conclusion

The exporatory talks we have had, and our initial software experiments lead us to believe that there is a feasible route to merging the separately devised compiler technolgies.

When this is combined with an extension of the type system of SAC to better support graphics types, we should have a compilation system that is able to make use both of

Hyper-threading and SIMD extensions. We anticipate multiplicative gains in performance as a result of this synergy.

In addition the talks arising from the travel grant have opened up new possiblities for industrial collaboration that are now being actively pursued.

References

Abrams, P.: 1970, *An APL Machine*, Stanford Linear Accelerator Center, Stanford University, Stanford.

Bik, A. J. C., Girkar, M., Grey, P. M. and Tian, X.: 2002, Automatic intra-register vectorization for the intel architecture, *Int. J. Parallel Program.* **30**(2), 65–98.

Blelloch, G.: 1995, Nesl: A nested data-parallel language, Vol. CMU-CS-95-170, Carnegie Mellon University.

Cheong, G. and Lam, M.: 1997, An optimizer for multipmedia instructionsets, *2nd SUIF Workshop*, Stanford University.

Cockshott, P.: 2002, Vector pascal reference manual, *SIGPLAN Not.* **37**(6), 59–81.

Cockshott, P.: 2004, Efficient compilation of array expressions, *SIGAPL APL Quote Quad* **34**(2), 16–25.

Ewing, A., Richardson, H., Simpson, A. and Kulkarni, R.: 1998, *Writing Data Parallel Programs with High Performance Fortran*, Edinburgh ParallelComputing Centre.

Grelck, C. and Scholz, S.-B.: 2003, SAC — From High-level Programming with Arrays to Efficient Parallel Execution, *Parallel Processing Letters* **13**(3), 401–412.

Hakami, B.: 1975, *Efficient Implementation of APL in a Multilanguage Environment*, International Computers Ltd.

Hassitt, A., Lageshulte, J. and Lyon, L.: 1973, Implementation of a high level language machine, *Communications of the ACM* **16**.

Hillis, W. D.: 1986, *The connection machine*, MIT Press, Cambridge, MA.

Krall, A. and Lelait, S.: 2000, Compilation techniques for multimedia processors, *International Journal of Parallel Programming* **28**(4), 347–361.

Leupers, R.: 2000, Code selection for media processors with simd instructions.

Lonergan, W. and King, P.: 1985, Design of the b5000 system, *in* D. Sieworek, G. Bell and A. Newell (eds), *Computer Structures*, McGraw Hill.

Metcalf, M. and Reid, J.: 1996, *The F Programming Language*, Oxford Univesity Press.

Morse, S., Ravenel, B., Mazor, S. and Pohlman, W.: 1985, Intel microprocessors: 8008 to 8086, *in* D. Sieworek, G. Bell and A. Newell (eds), *Computer Structures*, McGraw Hill.

Naur, P. and Backus, J.: 1960, Report on the algorithmic language algol 60, Danish Academy of Technical Sciences, Copenhagen.

Paulson, L. D.: 2005, Squeezing supercomputers onto a chip, *Computer* **38**(1), 21–23.

Perrott, R. H. and Zarea-Aliabadi, A.: 1986, Supercomputer languages, *ACM Comput. Surv.* **18**(1), 5–22.

Reddaway, S. F.: 1973, Dap a distributed array processor, *SIGARCH Comput. Archit. News* **2**(4), 61–65.

Rojas, J. C. and Leeser, M.: 2003, Programming portable optimized multimedia applications, *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, ACM Press, New York, NY, pp. 291–294.

Russell, R.: 1985, The cray-1 computer system, *in* D. Sieworek, G. Bell and A. Newell (eds), *Computer Structures*, McGraw Hill.

Scholz, S.-B.: 2003, Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting, *Journal of Functional Programming* **13**(6), 1005–1059.

Smith, W., Rice, R., Cheskey, G. and andStephen Lundstrum, T. L.: 1985, The symbol computer, *in* D. Sieworek, G. Bell and A. Newell (eds), *Computer Structures*, McGraw Hill.

Snyder, L.: 1999, *A Programmer's Guide to ZPL*, MIT Press, Cambridge, Mass.

Srereman, N. and Govindarjan, G.: 2000, A vectorising compiler for multiimedia extensions, Vol. 28, pp. 363–400.

Dept Computing Science University of Glasgow
E-mail address: `wpc@dcs.gla.ac.uk`