Compilation Techniques for Multimedia Processors

Andreas Krall and Sylvain Lelait
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A–1040 Wien
{andi,sylvain}@complang.tuwien.ac.at

Abstract

The huge processing power needed by multimedia applications has led to multimedia extensions in the instruction set of microprocessors which exploit subword parallelism. Examples of these extended instruction sets are the Visual Instruction Set of the UltraSPARC processor, the AltiVec instruction set of the PowerPC processor, the MMX and ISS extensions of the Pentium processors, and the MAX-2 instruction set of the HP PA-RISC processor. Currently, these extensions can only be used by programs written in assembly language, through system libraries or by calling specialized macros in a high-level language. Therefore, these instructions are not used by most applications.

We propose two code generation techniques to produce native code using these multimedia extensions for programs written in a high level language: classical vectorization and vectorization by unrolling. Vectorization by unrolling is simpler than classical vectorization since data dependence analysis is reduced to acyclic control flow graph analysis. Furthermore, we address the problem of unaligned memory accesses. This can be handled by both static analysis and dynamic run time checking. Preliminary experimental results for a code generator for the UltraSPARC VIS instruction set show that speedups of up to a factor of 4.8 are possible, and that vectorization by unrolling is much simpler but as effective as classical vectorization.

Keywords

compiler, multimedia processors, vectorization, loop unrolling

1 Introduction

Nowadays multimedia applications requiring high processing power are widely used. The trend of using multimedia will increase in the future. Current processors do not have the capabilities for efficient handling of multimedia information (such as audio/video data). Therefore, computer architects try to provide solutions for the fast growing market of media processing. A very promising solution is the use of special instruction sets aimed at exploiting the subword parallelism available when handling specific multimedia data. These special instruction sets can either be implemented by special processors [6] or existing instruction sets can be enhanced by a multimedia instruction set. Four widely used processors currently offer multimedia extensions: the UltraSPARC with the Visual Instruction Set [9], the PowerPC with the AltiVec extension [13], the Pentium with the MMX extension [8] and the HP PA-RISC with the MAX-2 instruction set [7].

The present means for exploiting this new functionality are not yet satisfactory. Programmers have to code in assembly language, use provided system libraries or call macros in the high-level code. All these solutions have drawbacks, among which portability problems and high cost of software development are the more obvious. A better solution would be to compile a program coded in a high-level language into multimedia instructions. The code produced would be more efficient and safe. Furthermore, the new functionalities of such processors can also improve the performance of other applications by exploiting their parallelism in a better manner.

The following example illustrates a C implementation of the vector product, which forms the inner loop of many signal-processing operations:

```
int dotprod(int n, short int *a, short int *b) {
    int i, sum;
    for (sum = 0, i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
    }
```

Using the VIS-instruction set of the UltraSPARC processor four 16-bit values can be hold in a 64-bit variable (`ad4x16` and `bd4x16`). Two 32-bit sized result values can be hold in a 64-bit variable (`rdh2x32` and `rdl2x32`). A rewritten C-program using VIS-instruction macros for the inner loop (see figure 1) can execute the dot vector product 7.5 times faster for an aligned vector of length 1024. From the complexity of this code it is evident, that such transformations should be done by the compiler, not by the programmer.

We are developing a research compiler which translates C source programs into native code for the SPARC processor exploiting the VIS instruction set. Our compiler is based on the CoSy compiler generation framework from ACE. This framework includes a front end with all classical optimizations like constant propagation, common subexpression elimination, strength reduction, loop invariant code motion and similar techniques and the BEG [5] code generator-

generator with tree pattern matching instruction selection, graph-coloring register allocation and instruction scheduling. The extensions for exploiting the VIS instructions are described in the following sections. Classic vectorization is presented in section 3, alignment problems are handled in section 5 and vectorization by unrolling is described in section 6.

2 Related Work

Little published work exists which directly deals with compilation techniques for multimedia processors. Cheong and Lam have given a presentation at the second SUIF compiler workshop [4]. The SUIF vectorizer is used in a two phase source to source optimizer for multimedia instruction sets. In the first phase parallel loops are identified, and instructions in the the loop bodies are converted to vector instructions working on infinite length vectors. In the second phase, the vector operations are transformed into function calls of SPARCs VIS instructions. The article also describes their approach for handling unaligned load and stores of vectors.

In [12] Govindarajan presented an implementation of a vectorizing compiler for the MMX extensions of the Pentium processors. The compiler is based on the SUIF compiler using C source to source translation with inline assembly instructions. The compiler identifies data parallel section of code. It enhances the scope of applications by performing code transformations like strip mining, scalar expansion, grouping and reduction, loop fission and distribution. The compiler prototype gives performance improvements up to a factor of 7.

Multimedia instruction sets put multiple values in a register and operate on all values at the same time. Therefore, a multimedia processor can be viewed either as a VLIW processor with a set of combinable operations, or as a vector processor with very short vectors. Work on vector processors took place mainly at the end of the 80's and the beginning of the 90's. Concerning vector processors, Allen and Kennedy [2] developed global register register allocation techniques for Fortran90 on vector processors. They used program transformations to increase data locality and vector operations are modified to be executed without problem on the processor. The vector operations must be subdivided into sections that fit the hardware of the target machine. They describe sectioning methods like strip mining, sectioning transformations such as loop reversal, input prefetching, loop splitting, loop interchange and loop fusion. Prefetching is very useful to optimize data locality as well.

A good introduction into compilation techniques for parallel computing are the books by Zima [15] and Wolfe [14]. A survey on optimizations techniques suited for vector processors, including the optimizations described above, has been done in [3].

3 Classic vectorization

When viewing a multimedia processor as a vector processor with short fixed length vectors, techniques known from classic vectorization can be used to generate code for multimedia instructions. In our compiler this is accomplished in many smaller phases (called engines in CoSy terminology) executed sequentially.

- loop analysis
- loop normalization
- scalar expansion
- dependence analysis
- vectorization
- alignment management
- strip mining
- constant expansion
- lower iteration space
- lower alignment
- instruction selection and register allocation

Loop analysis determines loops and loop normalization adjusts the iteration count to start at 0 by a step of 1. Scalar expansion transforms a scalar variable used inside the loop into a vector. Dependence analysis computes the data dependence graph for a loop. Vectorization transforms the loop body into vector instructions. Constant expansion changes scalar constants to vector constants. Alignment management handles unaligned vector load/stores. Strip mining reduces variable length vectors to word length vectors. During instruction selection multimedia instructions are used for the short vector operations and floating point registers are allocated to the vectors.

3.1 Data dependence analysis

The data dependence analysis is based on the Janus system by Sogno [11]. Janus contains an inequation solver over the integers, to be used for dependence computing in automatic parallelization. The input to the solver is a system of linear inequations resulting from the data dependence problem from loop parallelization. This system represents the set of constraints which have to be verified by the variables of the loop. The output gives information on the data dependence direction vectors which are compatible with the input system. The result of the dependence analysis is the possibly cyclic data dependence graph of the loop.

3.2 Vectorization

Vectorization is done using Allen and Kennedys algorithm for vector code generation [1, 2]. Input to vectorization is the loop with its corresponding data dependence graph. The algorithm computes the strongly connected components in the data dependence graph and recursively vectorizes the statements if possible. The result is a vectorized loop in intermediate representation with assignments replaced by vector assignments. Following example demonstrates vectorization of a simple loop.

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000; i++)
    a[i] = b[i] + c[i];
```

is transformed into

```
short int a[1000], b[1000], c[1000];

a[0:999] = b[0:999] + c[0:999];
```

3.3 Strip mining

The Sparc processors can store up to four shorter data values in one register. Therefore, the vectorized loop using infinite length vectors has to be transformed using only fixed length vectors by strip mining. The size of the vector and the number of the elements of the vector depends on the type of the data. The following example shows strip mining of the previous vectorized loop using vectors of length four.

```
short int a[1000], b[1000], c[1000];

a[0:999] = b[0:999] + c[0:999];
```

is transformed into

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000; i += 4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];
```

3.4 Lowering of the Iteration Space

In order to ease the code generation process, several data are computed and directly attached as attributes to the instructions. An attribute containing the number of parallel operations, NumOp, is added to the statement. It helps to choose between single or double precision versions of VIS instructions. Another attribute, FromType, containing the type of the array elements of the vector

instruction is computed and propagated to the operators of the instruction. It is used to choose between 16- or 32-bit versions of the VIS instructions.

Furthermore, to be able to generate partial store instruction, a new intermediate representation operator, edge, is introduced. It is used for the left hand-side of vector statements when vectors smaller than the word size have to be stored. An example is the tail of a big vector which is not a multiple of the wordsize.

The iteration space itself is removed, because it is not needed anymore during the code generation process itself (BEG). Let us take the following example after strip-mining:

```
short int a[23], b[23], c[23];

for (i = 0; i < 20; i+=4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];

a[20:22] = b[20:22] + c[20:22];
```

is lowered into:

```
for (i = 0; i < 20; i+=4)
   a[i] = b[i] + c[i] {NumOp = 4, FromType = short int};

mask = edge(a[20], a[22]) ;
a[20] = b[20] + c[20] {NumOp = 3, FromType = short int};
store_mask(a[20], a[22]);
```

The vector statement performing the remaining 3 operations outside of the loop is translated into several instructions. The first one computes a mask for a partial store instruction. All 4 elements are loaded by the addition statement, and the addition itself is performed on all 4 elements. The correct result is achieved by the partial store which writes only 3 elements.

3.5 Code Generation for Conditional Statements

In the case of conditional statements in the loop body, the vectorizer introduces masks in the iteration space of each vector statement. A vector comparison producing a mask is generated for each atomic condition. The atomic masks are combined by logical operations. This mask is then used immediately as operand of a partial store, or, if the vector assignment already needed a partial store, the mask is combined with the mask produced by the edge statement. This final mask is then used by the partial store. The following loop

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000;i++)
    if (b[i] > c[i])
      a[i] = b[i] + c[i];
```

is first transformed into the following vector statement:

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000; i = i+4)
    a[i:i+3] = b[i:i+3] + c[i:i+3] {b[i:i+3] > c[i:i+3]};
```

During the lowering step a vector compare instruction is introduced which produces a mask. The dependent partial store takes the produced mask as operand.

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000; i = i+4) {
    mask = b[i] > c[i]; {NumOp = 4, FromType = short int}
    a[i] = b[i] + c[i]; {NumOp = 4; FromType = short int}
    store_mask(a[i], a[i+3]); /* partial store */
```

3.6 Instruction selection and register allocation

The code generator-generator BEG is controlled by tree pattern matching rules which describe the instruction selection. The operators have attributes which contain the type of the operands. Depending on the operands rules are selected which either generate scalar code or multimedia instructions. Vector operands are assigned to floating point registers which are allocated automatically by the graph coloring register allocator.

4 Loop Transformations

Several loop transformations help in vectorizing loops. Vectorization is simpler if the loop body contains only one assignment statement. Loop distribution splits a loop with more statements into several loops with one statement. Loop fusion can then be used to merge these split loops again into one loop after strip mining has been applied. Sometimes it would be more useful to vectorize the outer loop in a loop nest. Loop interchange is then applied to exchange the outer loop with the inner loop.

4.1 Loop Normalization

Loop normalization ensures that the iteration space of the loops is regular. For a C compiler the lower bounds are set to 0 and the step is set to 1. The index expressions and the lower bounds are modified accordingly. The following small loop with lower bound 2

```
short int a[1000], b[1000], c[1000];
```

```
for (i = 2; i < 1000; i++)
    a[i] = b[i] + c[i];
```

is transformed into:

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 998; i++)
    a[i+2] = b[i+2] + c[i+2];
```

4.2 Scalar and Constant Expansion

Scalar expansion is needed when a scalar variable appears in a vector expression. The scalar variable is copied to each element of a temporary vector which replaces the scalar variable in the vector expression.

Constant expansion is a "special case" of scalar expansion. It is needed for handling constants in vector statements. In the case of the UltraSPARC processor, VIS instructions can only handle registers as operands (no immediate values). Therefore scalar constants are replaced by constant arrays allocated to registers.

The following loop demonstrates scalar and constant expansion:

```
short int a[1000], b[1000], c;

for (i = 0; i < 1000; i++)
    a[i] = b[i] + c + 2;
```

The integer constant is replaced by an array access to a constant array (`const`) which is initialized in its declaration. The scalar constant `c` is also replaced by an array access. Initialization of the array `c4` is made just after the declaration. The size of the new arrays is computed according to the number of operations performed in parallel. Thus the loop is transformed into:

```
short int a[1000], b[1000], c, c4[4], const[4] = {2, 2, 2, 2};

c4[0] = c; c4[1] = c; c4[2] = c; c4[3] = c;

for (i = 0; i < 1000; i += 4)
    a[i:i+3] = b[i:i+3] + c4[0:3] + const[0:3];
```

5 Alignment Problems

The SPARC like all other RISC processors allows loading and storing of short vectors only if they are aligned correctly (on a 4 byte boundary for 4 byte vectors and on an 8 byte boundary for 8 byte vectors). The VIS instruction set added instructions for partial load and store and instructions for merging

partial loaded data. Special instruction sequences are needed to implement an unaligned load (4 instructions) or an unaligned store (up to 12 instructions).

Due to separate compilation of C and aliasing of pointers static alignment analysis is expensive and imprecise [10]. Therefore, the prototype compiler includes only a simple analysis which detects arrays declared as static and aligns them on 8 byte boundaries during the lowering phase.

If the alignment of accesses cannot be proven statically, dynamic alignment checks are inserted into the code. Depending on the result of the checks either fast vectorized code or slower non vectorized code is used. The alignment checker scans all operands of the instructions in the loop body and – depending on a compiler switch and the number of operands – either generates a single test with two code versions or a whole tree with $2^n - 1$ code versions (n being the number of operands in the loop).

In the following example two code versions are generated. The condition includes a test over all operands of the loop.

```
short int *a, *b, *c;

for (i = 0; i < 1000; i += 4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];
```

is transformed into

```
short int *a, *b, *c;

if ((a & 7) || (b & 7) || (c & 7))
    for (i = 0; i < 1000; i += 4) {
       a[i]   = b[i]   + c[i];
       a[i+1] = b[i+1] + c[i+1];
       a[i+2] = b[i+2] + c[i+2];
       a[i+3] = b[i+3] + c[i+3];
       }
else
    for (i = 0; i < 1000; i += 4)
        a[i:i+3] = b[i:i+3] + c[i:i+3];
```

An unaligned vector load only takes 4 instructions. It makes sense to optimize loops where one or two vectors are unaligned. If code size is unimportant (controllable by a compiler switch), different versions for partially unaligned loops are supported. We do not generate code for unaligned vectors which are stored to memory. The following example shows a code tree with 5 different versions for unaligned array accesses.

```
short int *a, *b, *c;

if (a & 7)      non vectorized loop
else if (b & 7)
```

```
    if (c & 7)  b and c unaligned
    else        b unaligned
else if (c & 7) c unaligned
    else        a, b and c aligned
```

6 Implementation of vectorization by loop unrolling

The implementation of vectorization by loop unrolling is again done in a sequence of phases implemented by engines of the CoSy framework. To keep the development effort of these engines smaller many engines of the classical vectorization are reused. An example is the engine which does dependence analysis and computes the data dependence graph. For vectorization by unrolling most of the analysis could be done using acyclic analysis, but reusing the old engines was simpler.

- loop analysis
- compute unrolling degree
- loop unrolling
- dependence analysis
- dependence verification
- generation of vector instructions
- alignment management
- lower iteration space
- lower alignment
- instruction selection and register allocation

The idea of this method is to avoid a costly dependency analysis and the entire vectorization process. The loop is first unrolled the correct number of times depending on the type of the operands. In the example below the loop is unrolled 4 times for short int operands, so that the final operands fit into the 64-bit registers. Then the loop body is inspected and acyclic instruction scheduling is performed in order to have all instances of the same loop instruction grouped together. Then these instances are replaced by a vector instruction operating directly on the correct number of instances. The last step is the lowering phase of the former method.

Let us detail this method on a simple example. The first step is to compute the unrolling degree of the loop. This is done by scanning each operand of the loop body instructions. Depending on their type, the unrolling degree is set to

2 or 4. For instance if they are all 8 or 16-bit data types, then the loop can be unrolled 4 times, if one of them is a 32-bit data type, the loop can only be unrolled twice.

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000; i++)
    a[i] = b[i] + c[i];
```

is transformed into

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000; i += 4) {
    a[i]   = b[i]   + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
    }
```

After the loop has been unrolled, a data dependence graph is built to check the validity of the forthcoming transformations. There should not exist any true or output dependence between different instances of the statements of the loop body. Once this has been verified, vector statements are generated directly by removing all the instructions of the loop body but the first unrolled iteration, and by converting them into vector statements.

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000; i += 4) {
    a[i]   = b[i]   + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
    }
```

is transformed into

```
short int a[1000], b[1000], c[1000];

for (i = 0; i < 1000; i += 4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];
```

Similar to dynamic alignment checking we are working on dynamic data dependence checking. If static data dependence analysis does not give precise results for a possible dependence of load and stores, but gives enough information about the increment values of the induction variable, a check for overlapping vector loads/stores can be done before the execution of the loop. A drawback is that this check is more costly both in run time and code size than the dynamic alignment check.

7 Results

The previous presented algorithms have been implemented in the CoSy compiler framework and we evaluated both the implementation complexity and the speedup gained by using multimedia instructions for short integer data loops.

The complexity has been evaluated counting the number of source code lines of each engine (see table 1). Classical vectorization has about one and a half times the number of lines that vectorization by unrolling has, despite the favoring of classical vectorization by reusing the complex data analysis engine.

Table 2 gives the execution times and the speedups for some benchmark programs. viscc - our compiler which generates VIS multimedia instructions - is compared with the standard compiler which does not use these instructions. A speedup greater than 4 is possible since the standard compiler does not unroll the loops and the loop overhead is reduced to one quarter. The third benchmark shows the speedup which is possible when one operand is unaligned due to an index variable shift. The last benchmark shows some speedup for a conditional statement. In that case, the speedup depends on the distribution of the conditions. In the normal version, the computation `a[i] = b[i] + c[i+1]` is only performed if the test is true. In the vectorized version, this instruction is always executed. The highest speedup is reached if the test is always true. Both classic vectorization and vectorization by unrolling produce exactly the same code. Therefore, only the results for the unrolling vectorizer are presented.

8 Conclusion and future work

The experimental results show that exploiting the UltraSPARC VIS instruction set in a code generator gives speedups of up to a factor of 4.8. Vectorization by unrolling is much simpler but as effective as classical vectorization.

We are working on vectorization for loop bodies with conditionals and more than one basic block. For this problem vectorization by unrolling is much more promising than the classical vectorization.

Acknowledgement

We express our thanks to David Gregg for his comments on earlier drafts of this paper. We would also like to thank the reviewers for their helpful suggestions. Sylvain Lelait was supported by the FWF grant P-12574-INF (Austrian Science Foundation).

References

[1] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM TOPLAS*, 9(4):491–542, October 1987.

```
afh2x16 = vis_read_hi(ad4x16);
bfh2x16 = vis_read_hi(bd4x16);
afl2x16 = vis_read_lo(ad4x16);
bfl2x16 = vis_read_lo(bd4x16);
tuh2x32 = vis_fmuld8sux16(afh2x16, bfh2x16);
tlh2x32 = vis_fmuld8ulx16(afh2x16, bfh2x16);
tul2x32 = vis_fmuld8sux16(afl2x16, bfl2x16);
tll2x32 = vis_fmuld8ulx16(afl2x16, bfl2x16);
tdh2x32 = vis_fpadd32(tuh2x32, tlh2x32);
tdl2x32 = vis_fpadd32(tul2x32, tll2x32);
rdh2x32 = vis_fpadd32(rdh2x32, tdh2x32);
rdl2x32 = vis_fpadd32(rdl2x32, tdl2x32);
```

Figure 1: Inner loop of dot product using VIS-macros

compiler phase (engine)	line count vectorizing	line count unrolling
loop normalization	1252	1252
scalar expansion	510	510
compute unrolling degree	–	218
loop unrolling	–	4485
dependence analysis	11760	11760
dependence verification	–	191
vectorization	12799	237
alignment management	1417	749
strip mining	632	–
constant expansion	430	430
lower iteration space	691	691
lower alignment	104	104
sum	29595	20627

Table 1: Line counts

loop body	viscc (secs)	cc (secs)	Speedup
a[i] = b[i] + c[i]	1.07	4.90	4.58
a[i] = b[i] * c[i]	1.77	8.44	4.77
a[i] = b[i] + c[i+1]	1.70	4.72	2.78
if (b[i] > c[i]) a[i] = b[i] + c[i]	2.29	6.34	3.67

Table 2: Speedups

[2] Randy Allen and Ken Kennedy. Vector Register Allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):325–420, December 1994.

[4] Gerald Cheong and Monica S. Lam. An optimizer for multimedia instruction sets. In *Second SUIF Compiler Workshop*, Stanford, August 1997.

[5] Helmut Emmelmann, Friedrich Schröer, and Rudolf Landwehr. BEG - a generator for efficient back ends. In *Conference on Programming Language Design and Implementation*, volume 24(7) of *SIGPLAN*, pages 227–237, Portland, 1989. ACM.

[6] Craig Hansen. MicroUnity's MediaProcessor archtecture. *IEEE Micro*, 16(4):34–41, August 1996.

[7] Ruby B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[8] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[9] André Seznec and Fabien Lloansi. Étude des architectures des microprocesseurs MIPS R10000, UltraSparc et PentiumPro. Technical Report 1024, IRISA, Rennes, May 1996.

[10] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensituve points-to analysis. In *POPL'97: The 24th Symposium on Principles of Programming Languages*, pages 1–14, Paris, January 1997. ACM.

[11] Jean-Claude Sogno. The Janus test: a hierarchical algorithm for computing direction and distance vectors. In *29th Hawaii International Conference on System Sciences*, Maui, Hawaii, January 1996.

[12] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. Dagstuhl Seminar on Instruction Level Parallelism and Parallelizing Compilation, April 1999.

[13] J. Tyler, J. Lent, A. Mather, and Huy Nguyen. AltiVec: bringing vector technology to the PowerPC processor family. In *International Performance, Computing and Communications Conference*, pages 437–444. IEEE, 1999.

[14] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[15] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1990.