An Empirical Evaluation of High Level Transformations for Embedded Processors

Björn Franke Michael O'Boyle
Institute for Computing Systems Architecture (ICSA)
Division of Informatics
University of Edinburgh

ABSTRACT

Efficient implementation of DSP applications are critical for many embedded systems. Optimising compilers for application programs written in C, largely focus on code generation and scheduling which, with their growing maturity, are providing diminishing returns. This paper empirically evaluates another approach, namely high level source to source transformations. High level techniques were applied to the DSPstone benchmarks on 3 platforms: TriMedia TM-1000, Texas Instruments TMS320C6201 and the Analog SHARC ADSP-21160. On average, the best transformation gave a factor of 2.43 improvement across the platforms. In certain cases a speedup of 5.48 was found for the SHARC, 7.38 for the TM-1 and 2.3 for the C6201. These preliminary results justify further investigation into the use of high level techniques for embedded compilers.

1. INTRODUCTION

Digital Signal Processing (DSP) and media processing are performance critical applications for embedded processors. This demand for performance has led to the development of specialised architectures, with application programs hand coded in assembly. More recently as the cost of developing an embedded system becomes dominated by algorithm and software development, there has been a move towards the use of high level programming languages, in particular C. As in other areas of computing, programming in C is much less time consuming than hand-coded assembler but this comes at the price of a less efficient implementation when compared to hand-coded approaches [9].

To bridge the gap between application development time on the one hand and performance on the other there has been much interest in optimising compiler technology, where the compiler is responsible for automatically "tuning" the program [3, 14, 20, 19, 5]. This work has primarily focused on efficient code generation or scheduling of the low level instructions.

However, code generation and to a lesser extent scheduling are platform specific. More significant is the fact that they are relatively mature techniques and there is a diminishing rate of return for increasingly sophisticated approaches. In [20], for instance, a scheduler is developed for a particular in-house core that is optimal in the majority of cases. Thus, if we wish to continue to increase performance, it is worth considering alternative approaches.

One such approach is to examine high level transformations. These are inherently portable and have been shown to give significant performance improvement for general purpose processors [12], yet there is little work on their impact on embedded applications perhaps due to the historical bottom-up approach to embedded systems.

One major difficulty in the use of high level transformations is that the preferred application language is C which is not well suited to program transformation. Extensive usage of pointer arithmetic [13, 22, 16] prevents the application of well developed array based dataflow analysis and transformations. However, in previous work [8] we developed a technique to transform pointer based programs into an equivalent array based form which opens the opportunity for application of more extensive high level transformations.

There has been limited work in the evaluation of transformations on embedded systems performance. In [6], the tradeoff between memory space and execution time with loop unrolling has been investigated and in [10] the impact of tiling on power consumption has been evaluated. Although power consumption and memory size are also very important issues for embedded systems they are not the focus of this work. Rather, we focus on techniques to improve execution time assuming a fixed amount of embedded memory. We empirically evaluate the impact of several high level transformations on the DSPstone [22] benchmark suite on 3 different embedded processors. It is shown that by selecting the appropriate transformation, we can on average improve the execution time by a factor of 2.43, justifying further investigation of high level transformations within embedded compilers.

The paper is organised as follows. Section 2 provides a motivating example illustrating the application and effect of high level transformations. Section 3 describes the DSPstone benchmarks, the three embedded processors and the transformations investigated and is followed in section 4 by a description and analysis of the results. Section 5 discusses the incorporation of high level transformations into a com-

```
for (i = 0 ; i < N ; i++, p_a++)
  {
    *p_d    = *p_c++ + *p_a++ * *p_b++ ;
    *p_d++ -=          *p_a   * *p_b-- ;

    *p_d    = *p_c++ + *p_a-- * *p_b++ ;
    *p_d++ +=          *p_a++ * *p_b++ ;
  }
```

Figure 1: Original pointer-based array traversal

```
for (i  =  0; i < N; i++)
  {
    D[2*i]  =  C[2*i] + A[2*i] * B[2*i] ;
    D[2*i]  -= A[2*i+1] * B[2*i+1] ;
    D[2*i+1] =   C[2*i+1] + A[2*i+1] * B[2*i] ;
    D[2*i+1] +=  A[2*i] * B[2*i+1] ;
  }
```

Figure 2: After conversion to explicit array accesses

piler and section 6 concludes the paper.

2. EXAMPLE

The excessive use of pointers within DSP programs is one possible reason for preventing the wider adoption of high level transformations within existing compilers. For instance, consider the example in figure 1, a kernel loop of the *DSPstone* benchmark `n_complex_updates.c`. Although the pointer increment accesses may help the compiler to utilise the post-increment address modes of the Address Generation Unit (AGU) [13] of a DSP, they are a severe impediment to program analysis and transformation.

Although general array access and pointer analysis are without further restrictions intractable [15], it is easier to find suitable restrictions of array data dependence problem while keeping the resulting algorithm applicable to real-world programs. In [8] we developed a technique to translate a restricted class of pointer-based programs into an array based form. Applying this technique to figure 1 we have the code shown in figure 2. It is both easier to read and has been shown [3, 14, 13] that such a program form can be translated back into a pointer based program without adversely affecting AGU utilisation.

Once in this form we may apply apply further optimisations. Figure 3 shows the example loop after application of de-linearisation [17] and redundant load and store elimination via scalar replacement [7] which gives a 20% improvement in execution time on the TI C6201

3. EXPERIMENTAL FRAMEWORK

In this paper we focus on the execution time behaviour of the DSPstone benchmark suite on 3 distinct processors with respect to high level transformations. Here we briefly describe the salient points of the processors, benchmarks and transformations considered.

3.1 Processors

```
for (i  =  0; i < N; i++)
  {
    temp3  = A[i][0];
    temp4  = B[i][0];
    temp5  = B[i][1];
    temp6  = A[i][1];

    temp1  =  C[i][0] + temp3 * temp4 ;
    temp1  -=  temp6 * temp5 ;
    D[i][0]  = temp1;
    temp2  =  C[i][1] + temp6 * temp4 ;
    temp2  +=  temp3 * temp5 ;
    D[i][1] = temp2;
  }
```

Figure 3: Delinearisation and Scalar Replacement

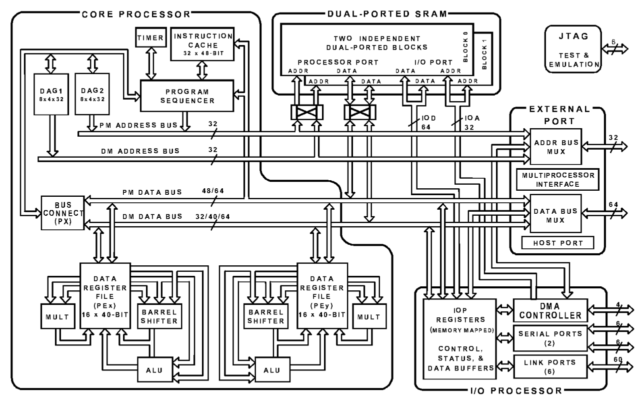

Figure 4: Functional Block Diagram of the ADSP-21160

3.1.1 *Analog Devices ADSP-21160*

The Analog Devices ADSP-21160 (see figure 4) is a 32-bit DSP with dual-ported on-chip SRAM and two 32-bit floating point processing elements (PEs) [2]. It has 4M bits of internal memory and no external memory. Internal memory is organised in two blocks of 2M bits, giving 38K words of code space and 6K/32K of Program Memory/Data Memory, respectively. In sequential mode (SISD), only one of the two PEs is active. In SIMD mode the second PE always performs the same operations as the first one, but with different data. Basically, the ADSP-21160 a vector processor with a vector size of two. In order to utilise the SIMD mode, the PEs must simultaneously access different data banks, i.e. program and data memory, via two independent buses. Switching to and from SIMD modes requires special instructions and involves some runtime overhead.

3.1.2 *Philips TriMedia TM-1*

The TriMedia TM-1 (see figure 5) is a 32-bit floating point VLIW Multimedia processor [18]. During each cycle it issues up to 5 operations in order to exploit instruction level parallelism. Data and instruction caches with 16kB and 32kB, respectively, are on-chip and speed up the data transfer be-

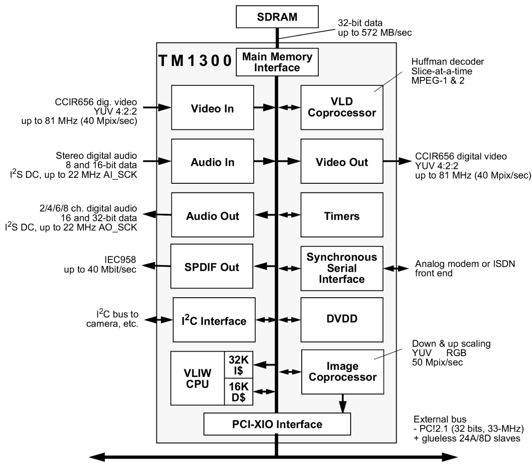

Figure 5: Functional Block Diagram of the TriMedia TM-1300

tween the processor core and the external RAM. To achieve highest performance it is important to have a compiler that schedules as many operations as possible in an instruction and to use the caches efficiently.

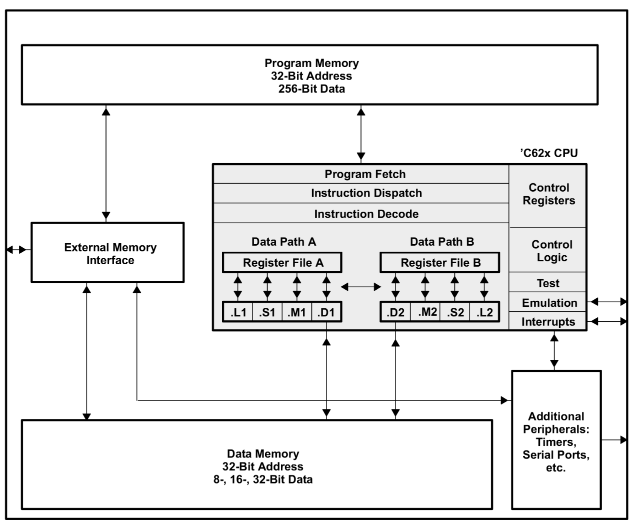

Figure 6: Functional Block Diagram of the TMS320C6201B

3.1.3 *Texas Instruments TMS320C6201*

In contrast to the other two processors, the TMS320C6201 (sse figure 6) is a 32-bit fixed-point processor [21]. Like the TriMedia the TI processor is a VLIW CPU with two data paths each containing four functional units and a register file with 16 registers. The on-chip memory is structured as a 64kB block of program memory and two 32kb blocks of data memory that can accessed in parallel. External memory can be connected via the external memory interface.

3.2 Benchmarks

The benchmarks selected are from the well-known DSP-stone [22] suite. 11 of the benchmarks containing iterative program structures were selected. Each benchmark is a small kernel found in typical DSP applications and makes extensive use of pointer arithmetic.

3.3 Transformations

The transformations investigated were selected based on the characteristics of the processors and the benchmark suite. As the benchmarks are loop and array based, transformations extensively studied in the area of scientific computation were selected [4]. *Loop unrolling* was selected as it can increase the size of the loop body allowing more instruction level parallelism (TM-1,C6201) to be exposed. *Arrayification* was used as a pre-processing transformation and has been shown to aid the low-level native compiler [8]. *Delinearisation, tiling and padding* allow cache to be optimised (TM-1). *Vectorisation* allows the exploitation of SIMD parallelism on the SHARC. Finally *scalar replacement* reduces the number of accesses to memory (SHARC, TM-1, C6201).

An automatic source-to-source tool was used for arrayification, developed in our previous work [8]. All other transformations were applied manually.

After the application of the transformation on source-to-source level the resulting code is then used as an input for the C compilers of the SHARC (VisualDSP++, Release 3.0.1.3), the Philips TriMedia TM-1 (compiler version 5.3.4) and the Texas Instruments TMS320C6x (compiler version 1.10). Performance data was collected by executing the programs on three simulators.

4. RESULTS

Since we are interested in high-performance signal processing the emphasis of our benchmarking is focused on execution time speedup given a fixed memory size. Code size and power consumption are important constraints but are beyond the scope of this paper.

4.1 Transformation-oriented Evaluation

In this section we examine the effect of each transformation in isolation on the benchmarks' behaviour shown in figures 9,10 and 11.

Arrayification. This often helps the native compiler to perform more accurate dependence testing. The TriMedia often benefits from the transformation, while the SHARC consistently performs worse. It has a variable but usually beneficial impact on the C6201. Its main benefit, however, is that it enables further transformations discussed below.

Unrolling. This can increase ILP for VLIW machines but has a variable impact on the SHARC's performance. Here loop unrolling of the pointer based versions of the `lms` and `n_real_updates` programs deteriorates performance. Small programs can benefit from total unrolling but have little or no beneficial effect on the other benchmarks. Usually, the array based programs slow down after loop unrolling; the exceptions being `matrix1`, `matrix2`, `fir2dim` and `mat1x3`. On the TriMedia and the C6201 it generally improves performance and does best on the array form of the program. The best unroll factors are considered here but performance does vary with respect to unroll factor as can be seen in figure 8. Unrolling increases code size and is limited by the amount of instruction memory.

Loop after arrayification only

```
    lcntr=1024, do(pc,_L$x-1) until lce;
    r4=dm(i3,m6);
    F12=F2*F4, r11=dm(i1,m6);
    F1=F11+F12, r2=dm(i2,m6);
    dm(i0,m6)=r1;
_$Lx:
```

Loop after arrayification and loop unrolling

```
    lcntr=512, do(pc,_L$x-1) until lce;
    r2=dm(i2,2);
    r4=dm(i3,2);
    i4=dm(-5,i6);
    F12=F2*F4, r11=dm(i1,2);
    i5=dm(-6,i6);
    F1=F11+F12, r13=i4;
    r15=r13+r5, dm(i0,2)=r1;

    r0=dm(i4,m5);
    r6=dm(i5,m5);
    F14=F0*F6, r8=i5;
    i4=dm(-4,i6);
    r12=R8+r5, r11=dm(i4,2);
    i5=dm(-3,i6);
    F7=F11+F14, r4=i4;
    dm(-4,i6)=r4;
    dm(i5,2)=r7;

    r10=i5;
    dm(-3,i6)=r10;
    dm(-5,i6)=r15;
    dm(-6,i6)=r12;
_$Lx:
```

Figure 7: Assembly code generated for `n_real_updates` after arrayification and loop unrolling

In order to explain the impact of unrolling on low level code generation and performance, figure 7 juxtaposes the ADSP-21160 assembly codes generated from the `n_real_updates` benchmark after arrayification only and arrayification and loop unrolling, respectively. Only the loop body of the loop under inspection performing the computation `D[i] = C[i] + A[i]*B[i]` for all `i` is shown. Whereas the loop body after arrayification only just takes six instructions and also utilises the ADSP-21160's capabilities to execute an arithmetic operation in parallel with a memory operation, the unrolled version is more complex. With its 26 instructions the code size has grown by a factor of 4.33. Although the compiler is able to identify the sequential traversal of the arrays and therefore generates a loop with a fixed number of iterations and uses post-increment mode for memory accesses, it also generates unnecessary code for the increment of the loop induction variable by two. Repeated loading, changing and storing of index registers additionally wastes cycles. The resulting performance of the unrolled loop is far worse than of the loop after arrayification only. This example shows that loop unrolling even with small unroll factors is not always beneficial, especially on the ADSP-21160.

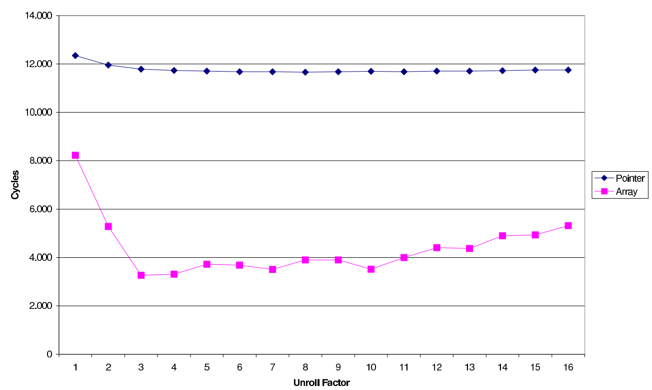

Figure 8: Influence of the Unroll Factor (`n_real_updates`, TriMedia)

SIMD vectorisation. This is only applicable on the ADSP-21160 with its two functional units. Where applicable it generally gives good performance except in the two cases where the overhead of changing to SIMD mode is grater than the work available. Arrayification is necessary for this transformation even though, on its own, arrayification decreases performance on the 21160.

Delinearisation. This is applicable to just four of the benchmarks shown in figure 9. It aids dependence analysis and allows further loop and data transformations. Overall it is generally beneficial for the TriMedia and C6201, but costly for the SHARC.

Array padding. This is used to reduce data cache conflicts and is suitable only for the TriMedia where it improves execution time in those case where it is applicable.

Loop tiling. This increases cache utilisation for the TriMedia. It also improves some codes on the TI C6201, although it does not contain any data cache. In effect we are matching the working set to that of the local memory.

Scalar Replacement. It generally improves the performance on the TriMedia for those benchmarks where it is applicable,

		fir2	mat1	mat2	comp
Delin	21160	0.90	0.98	1.00	0.59
	TriMedia	1.20	1.02	1.00	2.16
	C6201	0.98	1.40	1.63	1.00
Pad	TriMedia	1.20	1.11	1.08	
Tile	TriMedia	1.21	1.00	1.00	
	C6201	0.99	2.11	0.68	

Figure 10: Speedup due to Delinearisation, padding and tiling

		fir2	iir	mat1	mat3	comp
Scalar	21160	1.66	0.95	0.99	0.99	0.91
	TriMedia	1.43	3.13	1.28	0.88	2.16
	C6201	0.98	1.00	1.40	0.96	1.20

Figure 11: Speedup due to Scalar Replacement

but is more variable for the other two architectures.

Summary. The above results show that transformations can have a significant impact on performance. However, this impact is not always beneficial and varies depending on machine and benchmark. Furthermore, combinations of transformations are not considered. In the next section we look at combining transformations and their impact on performance.

4.2 Benchmark-oriented Evaluation

In figure 12 the results for the selected set of benchmarks are summarised. For each benchmark program and each architecture the maximum speedup achieved and the set of applied transformations is shown.

Highlighting the best performance is justified by the fact that an embedded programmer or a feedback directed compiler system would try several different options before selecting the best one. This is discussed further in section 5.

matrix1 (mat1). The `matrix1` benchmark computes the product of two matrices. After arrayification of the original program several different transformations and analyses can be applied to this program. For the ADSP-21160 delinearisation and subsequent SIMD-style parallelisation result in a speedup of 5.48. The transformed program utilises both datapaths of the Analog Devices processor and also its two memory banks can be used in parallel whereas the original program makes poor usage of the available resources. The speedup achieved is still higher than the expected speedup of 4 due to usage of both datapaths and memory banks. The additional performance gain is due to improved code generation in the presence of array based code.

The TriMedia benefits most from a delinearised version of the array-based program to which scalarisation, loop unrolling and padding have been applied. Successful scalarisation depends on an array representation of memory accesses, whereas padding requires some knowledge about the data layout of the program and the cache design of the processor. Each transformation applied on its own already increases the performance, but in combination a speedup of 3.82 can be observed. loop unrolling increases the flexibility of the instruction scheduler to fill the five issue slots of the TM-1300 with instructions since the new loop body has more instructions to chose from. Scalarisation reduces the number of memory accesses whereas padding reduces the number of cache conflicts. The situation is similar for the TI C6201B, although the best performance is achieved with just arrayification and loop unrolling together. The execution speed can be more than doubled on the C60 architecture (Speedup 2.11).

matrix2 (mat2). This is based on the same matrix multiplication algorithm as `matrix1`, but in this implementation the first and last iteration of the inner loop are peeled off. Originally intended as a hint to the compiler to create efficient code for the available AGUs and to avoid the otherwise necessary accumulator clear operation before the loop, this transformation prevents the exploitation of SIMD parallelism of the ADSP-21160. Since the required double-word alignment of array data in SIMD loops is violated, the `matrix2` benchmark cannot take benefit of parallel loops unless it is "re-transformed" back into the more regular form of `matrix1`. However, arrayification and loop unrolling can be applied and yields in a speedup of 1.16. For VLIW architectures the situation is different, because parallelism is exploited on instruction level rather than loop level. On the TriMedia a speedup of 3.52 can be achieved after arrayification, padding, scalarisation and 5-fold unrolling. Also the TI DSP benefits from the differences of the `matrix2` implementation, after arrayification and 3-fold unrolling a speedup of 2.39 is possible. This example shows that a certain code transformation has some positive effect for one class of DSP architectures whereas it inhibits successful usage of available resources on others. However, after arrayification the resulting code can be analysed with existing methods and easily transformed back by loop fusion into its "original" form of `matrix1`, so that the SIMD parallelism of the ADSP-21160 may be exploited.

matrix1x3 (mat3). This program computes the matrix product of a 3×3 matrix and a 3×1 vector in a simple loop. For the ADSP-21160 arrayification and total loop unrolling of the very small loop can speed up the execution by a factor of 1.93 However, an interesting observation is that since the number of iterations is an odd number some cleanup code must be generated when SIMD parallelisation is applied. When a further array element and an "artificial" void iteration are introduced the number of cycles can be reduced, because the cleanup overhead can be avoided. The largest speedup on the TriMedia can be achieved with total loop unrolling of either the pointer or array based version of the program. Although loop unrolling in general increases the code size, it is well justified in this case as the loop iteration range as well as the loop body are very small. On the C6201B total loop unrolling of the array based code also can be accounted for the largest possible speedup, but the performance gain on this architecture is smaller than on the TriMedia.

fir. This program is am implementation of a Finite Response Filter and contains a single loop. After arrayification the loop is amendable to loop reversal and loop splitting which in turn allow for using SIMD parallelisation of one of the resulting loops. Although just a single loop can

		conv.	dot	fir	fir2	iir	lms	mat1	mat2	mat3	comp	real
array	21160	1.00	0.80	1.00	0.90	0.92	1.00	0.99	0.96	0.78	0.64	0.99
	TriMedia	1.00	2.71	1.02	1.15	3.13	1.30	1.02	1.00	0.82	2.09	1.50
	C6201	1.00	0.99	1.00	0.98	(1.00)	1.11	1.40	2.15	0.96	1.01	1.00
Unroll	21160, Pointer	1.00	0.66	1.00	1.03	1.00	0.67	1.00	1.00	1.40	1.00	0.80
	21160, array	1.00	0.80	0.80	1.43	0.82	0.71	1.19	1.16	1.93	0.40	0.38
	TriMedia, Pointer	3.56	4.60	1.56	1.16	2.57	1.21	1.01	1.10	2.04	1.01	1.06
	TriMedia, array	2.40	4.60	2.20	1.16	4.41	3.16	3.01	1.10	2.04	2.52	3.78
	C6201, Pointer	1.82	0.99	1.05	0.97	1.00	1.56	1.14	1.50	0.92	1.02	1.02
	C6201, array	1.82	1.00	1.08	0.97	1.00	1.56	2.11	2.39	1.05	1.02	1.00
SIMD	21160	3.86	0.28	1.32			1.70	5.48		0.65	2.21	3.91

Figure 9: Speedup due to arrayification, unrolling and SIMD vectorisation

Benchmark		**Architecture** SHARC 21160	TriMedia TM-1300	TI C6201B
matrix1	max. S	5.48	3.42	2.11
	Transf.	array, Delin.,SIMD	array, Delin, Scalar, Pad (1), Unroll (9)	array, Unroll (3)
matrix2	max. S	1.00	3.52	2.39
	Transf.	(Original)	array, Pad (1), Unroll (5), Scalar	array, Unroll (3)
matrix1x3	max. S	1.93	2.0	1.04
	Transf.	(Original)	array, Unroll (total)	array Unroll (total)
fir	max. S	1.32	2.20	1.08
	Transf.	array, SIMD (part.)	array, Unroll (8)	array, Unroll(15)
fir2dim	max. S	1.66	7.38	1.02
	Transf.	array, Scalar	array, Delin., Scalar, Unroll	array, Scalar, Unroll, Pointer
convolution	max. S	3.86	3.56	1.82
	Transf.	array + SIMD	Unroll (10)	Unroll (8)
n_real_updates	max. S	3.91	3.78	1.02
	Transf.	array, SIMD	array, Unroll (3)	Unroll (9)
n_complex_updates	max. S	2.21	2.52	1.20
	Transf.	array, SIMD (part.)	array, Delin., Scalar, Unroll (2)	array, Delin., Scalar
dot_product	max. S	1.00	4.60	1.00
	Transf.	(Original)	array, Unroll (total)	(Original)
iir_biquad_N_sections	max. S	1.14	4.41	(1.0)
	Transf.	array, Delin., Scalar	array, Unroll (5)	Original
lms	max. S	1.70	3.16	1.56
	Transf.	array, SIMD (part.)	array, Unroll (11)	array, Unroll (7)
Average Speedup		2.31	3.69	1.39
Maximum Speedup		5.48	7.38	2.3

Figure 12: Speedup summary for the DSPstone benchmark suite

be parallelised for the ADSP-21160, a speedup of 1.32 is achieved. Since the other loop contains a memory access to a non double-word aligned array element, this loop must be executed sequentially. Further transformations in order to overcome this restriction are possible, but are beyond standard compiler analysis. In order to find an upper bound of speedup, the available library function of an FIR filter was used as a reference. The library function is approx. 4 times as fast as the original program and approx. 2.88 as fast as the SIMD-parallelised loop. Loop unrolling of the array-based loop gives the best results on the TriMedia. 8-fold unrolling results in a speedup of 2.20. Again on the C6201 the same set of transformations accounts for the largest speedup, which is 1.08. Once again the achieved speedup is smaller on the TI processor than the TriMedia.

fir2dim (fir2). This is a 2 dimensional Finite Response Filter. In theory, the `fir2dim` benchmark could be parallelised for the ADSP-21160, but the compiler is overly restrictive with the use of the `SIMD_loop` directive. In sequential execution mode, scalarisation can be applied after arrayification of the program and a speedup of 1.66 obtained. On the TriMedia a speedup of 7.38 is possible after arrayification, Delinearisation, scalarisation and total unrolling of the three small inner loops. In this example loop tiling increases locality and also improves performance, whereas the influence of array padding is negligible. For the C6201 a pointer-based version of the program achieved the best performance, but only after it has been converted into the array-based representation which allowed for the application of array dataflow analysis and scalarisation. After scalarisation and loop unrolling the program was converted back into the pointer-based form which gave an overall speedup of 1.02. Although the speedup is rather small, this example shows that the major advantage of the arrayification is the possibility to apply program analyses and transformations that could not be applied to the pointer-based program. In addition, it is possible to go back to pointer-based code when this appears to improve the overall performance.

convolution (conv). This routine for performs the convolution operation. Basically, it is a small loop with two memory accesses and a MAC operation in it. This loop can easily be parallelised for the ADSP-21160 after arrayfication and the execution time is reduced to 25.9% of the original time. For the TriMedia 10-fold loop unrolling does best, it results in a speedup of 3.56. Similar, the largest speedup ($S = 1.80$) on the C6201 is achieved with 8-fold unrolling. The increased size of the loop body provides the TriMedia and TI compiler with an increased flexibility for instruction scheduling and reduces the number of NOP-operations.

n_real_updates (real). This contains a small loop that performs n iterations each computing the statement $d = c+a\times b$ on real numbers. This loop can be easily parallelised for the ADSP-21160 and a speedup of 3.91 is achieved. The TriMedia benefits most of arrayification and unrolling. Arrayification helps the compiler to prove independence of different memory accesses, whereas loop unrolling increases the number of instructions in the loop body. Therefore, it gives the instruction scheduler more instructions to chose from when trying to fill the five issue slots. The maximum speedup achievable on the TriMedia is 3.78. On the contrary, the maximum speedup on the C6201 is rather small ($S = 1.02$) and due to 9-fold loop unrolling of the pointer-based code.

n_complex_updates (comp). This implements the n operations of the type $d = c + a \times b$ on complex numbers. Full SIMD parallelisation fails due to an overly restrictive compiler, but it is still possible to take advantage of the two functional units of the ADSP-21160 after arrayification, loop splitting and parallelisation of one of the resulting loops. A speedup of 2.21 could be achieved on the Analog Devices DSP. For the TriMedia arrayification once again was proven to be useful supporting other transformations like Delinearisation and Redundant Load/Store Elimination. Together with 2-fold loop unrolling, a speedup of 2.52 was obtained. The same set of transformations but without unrolling also achieved on the C6201 the largest speedup of 1.20.

dot_product (dot). This implements the computation of the dot product of two vectors with two elements each. This simple loop offers few opportunities for applying high-level transformations. Arrayification, SIMD parallelisation and unrolling are the three transformations that were compared. The original pointer-based program performs best on the ADSP-21160. SIMD parallelisation is applicable, but the overhead involved in switching from sequential to SIMD mode and back is larger than the benefit obtained from parallel processing. Also loop unrolling is not beneficial as it increases the execution time. In contrast, loop unrolling of either the pointer or array based program results in a speedup of 4.6 on the TriMedia. This VLIW architecture clearly benefits from the larger number of instructions in the loop body during instruction scheduling. The TI compiler handles the loop as well as the unrolled straight-line code, so the run-time of the original program cannot be decreased by unrolling.

iir_biquad_N_sections (iir). This is a benchmark that implements a Infinite Impulse Response Filter with N biquad sections. SIMD parallelisation cannot be applied to this program due to a loop-carried data dependence in its loop body. However, the sequential version for the ADSP-21160 can be improved by arrayification, delinearisation and scalarisation and speeded up to a factor of 1.14. Arrayification and 5-fold unrolling gives the best performance with a speedup of 4.41 on the TriMedia. Experiments on the C6201 were less successful due to technical problems with this program in the available simulation environment. However, initial results show only small chances of achieving a significant speedup.

lms. This is the kernel of a least mean square filter. The `lms` program contains two loops that can bothbenefit from SIMD parallelisation. After arrayification, loop reversal is applicable, so that the Analog Devices compiler accepts the first loop as a SIMD loop. An overall speedup of 1.70 is achieved on the ADSP-21160. The TriMedia and also the C6x architecture both benefit most from arrayification and loop unrolling and speedups of 3.16 and 1.56, respectively, are possible.

5. DISCUSSION

This paper has shown that selecting an appropriate transformation combination can lead to good performance. However, selecting the right transformation is a much more difficult task. There has been much work investigating the use of static analysis to determine the best transformation [11]. This approach is highly attractive in the context of general purpose computing as the analysis is typically a small fraction of the compilation time. Transformation selection based on static analysis is a fast but unfortunately a frequently inaccurate approach. Profile-directed approaches have proved promising where actual runtime data is used to improve optimisation selection. Recently, we have investigated the use of iterative compilation, where different optimisations are selected and evaluated, with the eventual best performing one selected [12]. Such an approach has a much longer compilation time, but this is not a major issue in embedded systems. Using such an approach, a compiler should automatically find the best speedups shown in figure 11 and is the subject of ongoing research.

6. CONCLUSION

This paper has empirically demonstrated the usefulness of applying high level transformations to DSP applications. Selecting the appropriate transformation, gives on average a 2.43 speedup across the three platforms investigated. The programs considered are relatively straightforward kernels and future work will investigate larger applications where the scope for high level optimisations is potentially greater. Given the empirical evidence justifying the use of high level transformations, the next step is to build a compiler strategy exploiting such transformations. Future work will investigate both static and iterative approaches to optimisation selection and consider the integration of high level optimisation with low level code selection and scheduling.

7. REFERENCES

[1] Allen, Randy and Steve Johnson, Compiling C for Vectorization, Parallelization, and Inline Expansion, *Proceedings of the SIGPLAN '88 Conference of Programming Languages Design and Implementation*, pp. 241-249, Atlanta, Georgia, June 22-24, 1988

[2] Analog Devices Inc.
ADSP-21160 SHARC DSP Hardware Reference
`http://www.analog.com`, 1999.

[3] de Araujo, Guido C.S., Code Generation Algorithms for Digital Signal Processors, Dissertation, Princeton University, Department of Electrical Engineering, June 1997.

[4] Bacon, D.F., Graham, S.L., Sharp, O.J.
Compiler Transformations for High-Performance Computing
ACM Computing Surveys, Vol. 26, Issue 4, 1994.

[5] Bhattacharyya, S.S., Leupers, R., Marwedel, P.
Software Synthesis and Code Generation for Signal Processing Systems
IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 47, No. 9, September 2000.

[6] Bodin, F., Chamski, Z., Eisenbeis, C. Rohou, E., Seznec, A., GCDS: A Compiler Strategy for Trading Code Size Against Performance in Embedded Applications INRIA Tech Report RR-3346, 1998.

[7] Duesterwald, E., R. Gupta and M. Soffa, A Practical Data Flow Framework for array Reference Analysis and its Use in Optimizations, *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*, 28(6), pp. 67-77, Albuquerque, New Mexico, 1993.

[8] Franke, B., O'Boyle, M.
Compiler Transformation of Pointers to Explicit array Accesses in DSP Applications
Proceedings of ETAPS CC 2001, Genova, Italy, 2001.

[9] Frederiksen, A., Christiansen, R., Bier, J., Koch, P.
An Evaluation of Compiler-Processor Interaction for DSP Applications
34th IEEE Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, October 2000.

[10] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and H. S. Kim., Experimental evaluation of energy behavior of iteration space tiling, Languages and Compilers for high Performance Computing (LCPC'00), Yorktown Heights, NY, August 2000.

[11] M. Kandemir, J. Ramanujam, and A. Choudhary, Improving cache locality by a combination of loop and data transformations, IEEE Transactions on Computers, Vol. 48, No. 2, February 1999.

[12] T. Kisuki, P.M.W. Knijnenburg and M.F.P. O'Boyle, Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation, Proc. PACT 2000, Parallel Architectures and Compiler Technology, *IEEE Press*, October 2000.

[13] Liem, C., Paulin, P., Jerraya, A. Address Calculation for Retargetable Compilation and Exploration of Instruction Set Architecture
33rd ACM Design Automation Conference (DAC), Las Vegas, USA, 1996.

[14] Leupers, Rainer, Novel Code Optimzation Techniques for DSPs, 2nd European DSP Education and Research Conference, Paris, France, 1998.

[15] Maydan, Dror.E., John L. Hennessy, and Monica S. Lam., Effectiveness of Data Dependence Analysis, *International Journal of Parallel Programming*, 23(1):63-81, 1995.

[16] Numerix-DSP Digital Signal Processing Web Site, `http://www.numerix-dsp.com/c_coding.pdf`, 2000.

[17] O'Boyle M.F.P and Knijnenberg P.M.W., Integrating loop and Data Transformat ions for Global Optimisation, *PACT '98, Parallel Architectures and Compiler Technology*, IEEE Press, October 1998.

[18] Philips Semiconductors
TriMedia TM-1300
`fttp://www.semiconductors.philips.com`, 1999.

[19] Sair, S., Kaeli, D.K., Meleis, W.
A Study of loop unrolling for VLIW-based DSP Processors
Proceedings of the 1998 IEEE Workshop on Signal Processing Systems (SiPS '98), October 1998, pp 519-527.

[20] Timmer A., Strik M., van Meerbergen J. and Jess J., Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores., DAC, 1995.

[21] Texas Instruments
SM320C6201B Digital Signal Processor
`http://www.ti.com`, 2001.

[22] Zivojnovic, V., J.M. Velarde, C. Schlager and H. Meyr, DSPstone: A DSP-Oriented Benchmarking Methodology, *Proceedings of Signal Processing Applications & Technology*, Dallas 1994.