

Towards Automatic Parallelisation for Multi-Processor DSPs

Björn Franke

Michael O'Boyle

Institute for Computing Systems Architecture (ICSA)
Division of Informatics, University of Edinburgh

Abstract

This paper describes a preliminary compiler based approach to achieving high performance DSP applications by automatically mapping C programs to multi-processor DSP systems. DSP programs typically contain pointer based memory accesses making automatic parallelisation difficult. This paper presents a new method to convert a restricted class of pointer-based memory accesses into array accesses with explicit index functions suitable for parallelisation. Different parallelisation approaches suitable for multi-processor DSPs are considered. We implemented our pointer conversion algorithm in the prototype Octave compiler where experimental results demonstrated that our technique increases the number of parallelisable loops from 6 to 24 for 11 of the DSPstone benchmarks. Furthermore our technique is shown to also improve the actual performance of DSP codes on single processor systems decreasing execution time by up to 33%.

1. Introduction

Typical DSP applications are becoming more demanding in terms of the amount of computation to be performed under hard real-time constraints. One obvious solution is to increase the amount of parallelism within DSP systems, providing increased performance without increasing clock speed (i.e. using the same integration technology). However, typical hardware solutions that rely on exploiting instruction level parallelism (ILP) provide diminishing returns as the amount of additional parallelism provided by greater number of functional units, for example, is not matched by the ability to detect ILP within the instruction stream.

One approach to overcome this difficulty, which many manufacturers have considered, is to offer solutions containing several conventional DSPs either on a board or integrated in a single package. Such an architecture allows the possible exploitation of a more coarse-grain parallelism at the program rather than instruction level. Unfortunately, the

programming tools for these multi-processor DSP systems are poor, giving little support in mapping the application to the multiple processing units. Usually, the system programmers are expected to parallelise their applications manually and to code inter-processor communications in their programs explicitly. Such an approach is error prone and time consuming. However, in the area of supercomputing there is a large body of work concerned with the automatic parallelisation and mapping of programs to multi-processor machines [9]. The parallelisation techniques developed rely on explicit array representations and are usually not applicable in the presence of pointers. However, pointers are widely present in C-based DSP programs under the assumption that they act as a “hint” to the compiler to generate code that utilises the Address Generation Units available in most DSPs. Such hints prevent auto-parallelisation.

This paper discusses the specific properties of multi-processor DSPs and existing pointer-based DSP C source codes. Problems encountered in automatic parallelisation of these codes are discussed before a method for the conversion of a restricted class of pointer-based memory accesses into array accesses with explicit index functions is outlined. This is followed by a discussion of how future auto-parallelisation techniques may be applicable to the particular characteristics of DSP applications. Experimental results combining our pointer clean-up scheme and the SUN E6500 automatic paralleliser show the large potential benefits of our approach. Furthermore, the usefulness of this technique for conventional single-processor DSPs such as the TriMedia TM-1 and the Texas Instruments TMS320C60 is demonstrated. The paper concludes with a summary and a discussion of future research in this area.

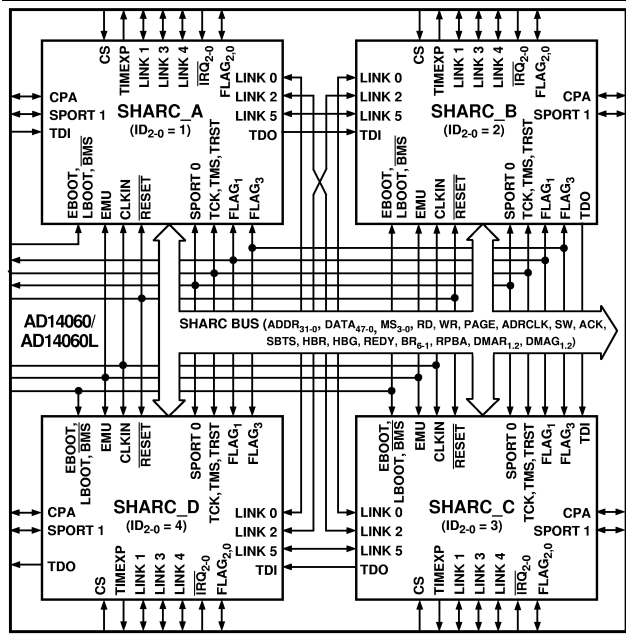
2. Background

This section briefly describes the type of architectures and programs we are interested in and presents an example of our compiler-based pointer conversion technique.

2.1. Multi-Processor DSPs

In this paper we consider multiple conventional DSPs linked via an interconnection network. The CPUs of such a multi-processor DSP can be contained in either separate ICs and linked together on one or more boards or be integrated into a single package. An example of an integrated multi-processor DSP is the Analog Devices Quad-SHARC shown in Figure 2.1. Each Quad-SHARC contains four ADSP-21060 DSPs which are conventional 32-bit DSPs. The built-in links of these four DSPs are used for interconnecting them on a shared bus that is also integrated into the Quad-SHARC.

Figure 2.1 Quad-SHARC AD14060 [1]



The Quad-SHARC is typical of the shared memory multi-processor DSPs we are interested in and supports synchronous multi-processing, i.e. all its internal DSPs can operate synchronously on the same clock signal. This is of special importance to parallelisation schemes that attempt to avoid unnecessary overhead for explicit (software-directed) synchronisation, since certain assumptions on the order of events at instruction level can be made.

2.2. DSP Codes

Programmers frequently used pointer-based accesses and pointer arithmetic within their programs in order to give “hints” to early compilers for on how and when to use post-increment/decrement addressing modes available in AGUs. For instance, consider example 2.1, a kernel loop of the *DSPstone* [7] benchmark *matrix2.c*. Here the pointer

Example 2.1 Original pointer-based array traversal

```
int *p_a = &A[0] ;
int *p_b = &B[0] ;
int *p_c = &C[0] ;

for (k = 0 ; k < Z ; k++)
{
    p_a = &A[0] ;
    for (i = 0 ; i < X; i++)
    {
        p_b = &B[k*Y] ;
        *p_c = *p_a++ * *p_b++ ;
        for (f = 0 ; f < Y-2; f++)
            *p_c += *p_a++ * *p_b++ ;
        *p_c++ += *p_a++ * *p_b++ ;
    }
}
```

increment accesses “encourage” the compiler to utilise the post-increment address modes of the AGU of a DSP.

While the use of pointer arithmetic is reasonable for irregular DSP architectures and can be automated [5], pointers impose some problems to the compilers of the more compiler-friendly DSPs with large, homogeneous register sets. If, however, further analysis for automatic parallelisation or optimisation is needed before code generation, then such a formulation is problematic as such techniques rely on explicit array index representations and cannot cope with pointer references. In order to maintain semantic correctness compilers use conservative strategies, i.e. many possible array access optimisations as well as loop restructuring and parallelisation schemes are not applied in the presence of pointers. Obviously, this limits the maximal performance of the produced code. It is highly desirable to overcome this drawback, but without adversely affecting AGU utilisation.

This paper develops a technique to collect information from pointer-based code in order to regenerate the original accesses with explicit indexes that are suitable for further analyses. Furthermore, this translation has been shown not to affect the performance of the AGU [3, 4].

Example 2.2 shows the loop after applying our compiler technique implemented in the Octave compiler. It now has explicit array indexes that are semantically equivalent to the previous loop in figure 2.1. Not only it is easier to read and understand for a human reader, but also easier to analyse by current compilers.

The transformed program with explicit array accesses is amendable to array data flow analyses, and compiler parallelisation. The program in figure 2.3 shows the parallel loops determined by the SUN auto-paralleliser. The same compiler was unable to detect any parallelism in the origi-

Example 2.2 After pointer conversion

```
for (k = 0 ; k < Z ; k++)
  for (i = 0 ; i < X; i++)
  {
    C[X*k+i] = A[Y*i] * B[Y*i];
    for (f = 0 ; f < Y-2; f++)
      C[X*k+i] +=
        A[Y*i+f+1] * B[Y*k+f+1];
    C[X*k+i] +=
      A[Y*i+Y-1] * B[Y*i+Y-1];
  }
```

Example 2.3 Parallel code

```
PAR for (k = 0 ; k < Z ; k++)
  PAR for (i = 0 ; i < X; i++)
  {
    C[X*k+i] = A[Y*i] * B[Y*i];
    for (f = 0 ; f < Y-2; f++)
      C[X*k+i] +=
        A[Y*i+f+1] * B[Y*k+f+1];
    C[X*k+i] +=
      A[Y*i+Y-1] * B[Y*i+Y-1];
  }
```

nal code.

3. Pointer Clean-up Conversion

This section is concerned with changing pointer-based programs typically found in DSP applications into an array based form amenable to current compiler analysis. In a sense we are reverse engineering “dusty desk” DSP applications.

3.1. Assumptions and Restrictions

The pointer conversion can only be applied if the resulting index functions of all array accesses are affine functions. These functions must not be dependent on any other variables apart from induction variables of some enclosing loops. If all pointer increments/decrements are constant, this can be ensured easily.

In order to facilitate pointer clean-up conversion and to guarantee its termination and correctness the overall affine requirement can be broken down further into the following restrictions:

1. structured loops

2. no pointer assignments apart from – maybe repeated – initialisation to some array start element
3. no data dependent pointer arithmetic
4. no function calls that might change pointers itself
5. equal number of pointer increments in all branches of conditional statements

3.2. Algorithm

Pointer clean-up conversion uses two stages during processing. In the first stage information on arrays and pointer initialisation, pointer increments and decrements as well as loop properties is collected. The second step then uses this information in order to replace pointer accesses by corresponding explicit array accesses and to remove pointer arithmetic completely. For further details please see [2].

4. Auto-Parallelisation

There is a large body of work describing auto-parallelisation of scientific array based codes that may be applied to array recovered DSP codes.

Traditionally, auto-parallelisation focused on loop transformations for parallelism and locality has been extensively studied in the context of shared memory parallel machines. Although frequently successful, they suffer from the fact that the analysis and transformations are inevitably local, since the unit of consideration is a loop nest rather than the entire program. Conversely, data transformations, such as alignment and partitioning, have received much attention in distributed memory compilation. As data layout has program wide impact, these techniques have, by necessity, been more global in their consideration. Though potentially determining good overall layouts, data transformations are unable to remedy any introduced poor code localised within a section of the program. Recent work has shown that combining these approaches is very effective in global optimisation. [8].

The typical stages of auto-parallelisation can be broken down as follows:

- *Transform program to uncover parallelism:* Loop skewing has been used to uncover wavefront parallelism, but this seems unnecessary in DSP applications where parallelism is normally apparent.
- *Determine parallelism using data dependence analysis:* Straightforward analysis can check if there are no cross loop iteration dependences and hence the loop is parallel. This can be extended to the use of cross-processor dependence analysis [8].

- *Partition and schedule program so as to map program parallelism to machine parallelism:* This is a critical stage as different mappings will have different communication, load balance and synchronisation costs.
- *Transform mapped code to exploit local processor resources:* Effective use of registers, and the memory hierarchy will require register tiling. Loop unrolling to expose ILP for the local processor has to be balanced against code growth.

There are three distinct characteristics, however, that distinguish DSP parallelism to that of scientific high performance computing. Firstly, many of the loop iterations are small and are unlikely to grow due to the underlying DSP algorithms. FIR filters, e.g., with very long filter kernels are quite rare. This means that typical loop or data based approaches will not scale as there is insufficient work within certain sections of the program. More aggressive techniques which go beyond doall parallelism will be required.

Secondly, DSP multi-processor may share a common clock. This allows much less pessimistic assumptions about the cost of inter-processor synchronisation and allows much tighter scheduling of the code.

Thirdly DSP codes have very static characteristics, so it is worthwhile investigating more extensive and expensive techniques, especially when the cost will be amortised over the number of products shipped.

5. Results

In order to quantify the benefit of applying pointer cleanup conversion for auto-parallelisation, a number of programs and platforms were selected for experimentation. We implemented the conversion technique in the Octave compiler and applied it to programs from the DSPstone benchmark suite. Both versions were then presented to the Sun E6500 C auto-paralleliser and the number of parallel loops discovered in each case recorded.

Table 5.1 summarises the results and gives additional information in those cases that the array-based program version could not be successfully parallelised.

In many of the pointer-based programs the compiler cannot parallelise any loop. In some programs a few loops can be parallelised, but after inspection of the codes it becomes apparent that these loops contain actually explicit array accesses and serve only as initialisation. With this knowledge, the result is that not a single loop with pointer accesses can be successfully parallelised! Whenever the compiler encounters a pointer access within a loop, it immediately assumes that this pointer will cause a data dependence and therefore conservatively avoids parallelisation.

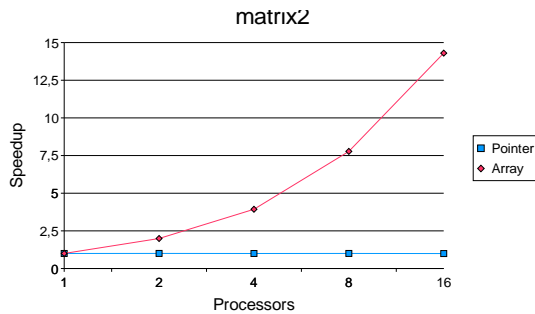
The number of parallelised loops in the programs with explicit array accesses is considerably larger, 19 versus 6.

Table 5.1 Parallelisable loops in DSPstone

Benchmark	#Loops	Pointer	Array	Comment
biquad_N_sect.	3	2	2	data dependent
convolution	2	0	2	
dot_product	1	0	0 (1)	not profitable
fir	3	0	3	
lms	3	0	0(2)	not profitable, data dependent
mat1x3	2	0	0 (1)	not profitable, data dependent
matrix1	5	2	3 (4)	not profitable, data dependent
matrix2	5	2	3 (4)	not profitable, data dependent
fir2dim	12	0	6 (7)	not profitable, data dependent
n_complex_up.	2	0	0	data dependent
n_real_updates	2	0	0 (1)	not profitable

There are two main reasons why the compiler avoids to convert loops into parallel forms: cross-iteration data dependences or expected performance penalties. Cross-iteration data dependences are dependences that exist between different loop iterations and prevent parallelisation. The Sun compiler can detect opportunities for using reduction operators, but not all cross-iterations can be resolved by this method. Although the loop iterations might be proven independent, there are situations when a parallelisation would result in an increased run-time and is deemed “not profitable” due to, among other costs, excessive synchronisation. However, synchronisation on the E6500 is more costly than on multi-processors with a common clock such as Quad-SHARC and thus potentially unprofitable parallelism may be profitably exploited in an embedded situation. If, therefore, we also consider loops that are parallelisable, regardless of profitability on the SUN (as shown in brackets), the number of loops with parallelism increases from 19 to 24.

In figure 5.1 the speedup of the `matrix2` benchmark is shown for the execution on 1,2,4,8 and 16 processors. The algorithm multiplies two matrices, and each matrix was chosen to have 1000×1000 elements. Whereas there is no speedup observable for the pointer-based version as the number of processors increases, the program with explicit array accesses scales nicely. The achieved speedup is close to the expected linear speedup, only some small overhead for synchronisation keeps the performance below the theoretical limit. Since many of the kernel loops of the DSPstone benchmark have comparatively small iteration ranges and short loop bodies, not all of the programs scale so well with the number of processors. This means that any DSP based auto-paralleliser must carefully consider program granularity.

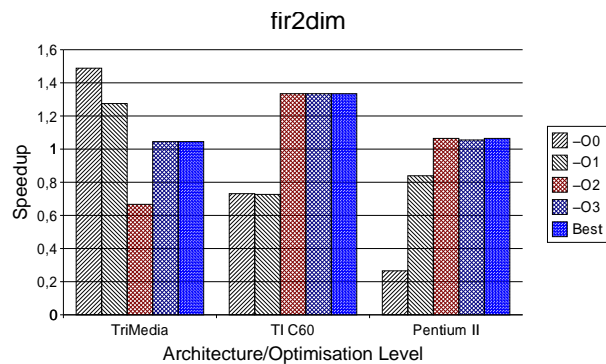
Figure 5.1 Speedup of the `matrix2` benchmark

The pointer cleanup conversion also increase the performance for existing single-processor DSPs and GPPs. The same set of DSPstone benchmarks has been compiled in the original and transformed version for the TriMedia TM-1, Texas Instruments C60 and Intel Pentium II. Figure 5.2 illustrates the results of the `fir2dim` benchmark as a typical example. The results are represented in terms of speedup. The run-times after pointer cleanup conversion are compared to those of the original program for all available optimisation levels of the compilers. Additionally, the speedup of the best explicit array access based version over the best pointer-based version is shown. On the TriMedia a significant speedup can be observed at the lower optimisation levels, whereas at the higher levels the differences in performance become smaller. However, at some intermediate optimisation level a speedup < 1 is achieved for the TriMedia. The transformed codes performs poorly without further optimisations on the TI C60 and PII, but with additional optimisations enabled some significant increase of up to 33% for the C60 can be observed. On the PII a speedup of 6% can be achieved.

Summarising, Sun's parallelising C compiler cannot parallelise any loops containing pointer accesses, but can handle explicit array accesses after applying the pointer cleanup-conversion. A substantial run-time benefit can be achieved by exploiting coarse-grain parallelism on loop level. Pointer cleanup is also profitable for conventional single-processor DSPs and their compilers, enabling further optimisation.

6. Conclusions and Outlook

This paper describes an approach to auto-parallelisation for multi-processor DSPs. It presented a pointer conversion technique as a pre-processing stage for auto-parallelisation. Empirical results demonstrate that such a phase is crucial for existing compiler technology in identifying suitable

Figure 5.2 Performance of `fir2dim` benchmark

program parallelism. The next stage is to develop auto-parallelisation techniques more applicable to DSP applications. In particular due to the small size of some applications, more aggressive techniques to discover and exploit program parallelism will be necessary. We are currently developing an auto-paralleliser for multi-processor DSPs and will present the current state of our research and experimental results at the workshop.

References

- [1] *Analog Devices, Inc.* AD14160 Datasheet <http://www.analog.com>, Norwood, MA, USA, 1997.
- [2] Franke B, and O'Boyle M., Compiler Transformation of Pointers to Explicit Array Accesses in DSP Applications, *Compiler Construction 2001*.
- [3] *de Araujo, Guido C.S.* Code Generation Algorithms for Digital Signal Processors Dissertation, Princeton University, Department of Electrical Engineering, June 1997.
- [4] *Leupers, R.* Novel Code Optimization Techniques for DSPs 2nd European DSP Education and Research Conference, Paris, France, 1998.
- [5] *Liem C., Paulin P., Jerraya A.* Address Calculation for Re-targetable Compilation and Exploration of Instruction-Set Architectures Proceedings of the 33rd Design Automation Conference, Las Vegas, Nevada 1996.
- [6] *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.
- [7] *Zivojnovic, V., J.M. Velarde, C. Schlager and H. Meyr* DSPstone: A DSP-Oriented Benchmarking Methodology *Proceedings of Signal Processing Applications & Technology*, Dallas 1994.
- [8] *O'Boyle M.F.P and P.M.W. Knijnenberg* Efficient Parallelisation using Combined Loop and Data Transformations, PACT '99, Parallel Architectures and Compiler Technology, *IEEE Press*, Newport Beach, October 1999.
- [9] *H. Zima* Supercompilers for Parallel and Vector Computers, *ACM Press*, 1991.