Combining Program Recovery, Auto-parallelisation and Locality Analysis for C programs on Multi-processor Embedded Systems

Björn Franke M.F.P. O'Boyle
Institute for Computing Systems Architecture (ICSA)
School of Informatics
University of Edinburgh

Abstract

This paper develops a complete auto-parallelisation approach for multiple-address space digital signal processors (DSPs). It combines a pointer conversion technique with a new modulo elimination transformation. This is followed by a combined parallelisation and address resolution approach which maps array references without introducing message-passing. Furthermore, as DSPs do not possess any cache structure, an optimisation is presented which transforms the program to both exploit remote data locality and local memory bandwidth. This parallelisation approach is applied to the DSPstone and UTDSP benchmark suites, giving an average speedup of 3.78 on a four processor Analog Devices TigerSHARC.

1 Introduction

Multi-processor DSPs provide a cost effective solution to embedded applications requiring high performance. Although there are sophisticated optimising compilers and techniques targeted at single DSPs [17], there are no successful parallelising compilers. The reason is simple, the task is complex. It requires the combination of a number of techniques to overcome the particular problems encountered with compiling for DSPs, namely the programming idiom used and the challenging multiple-address space architecture.

Applications are written in C and make extensive use of pointer arithmetic [23]. This alone will prevent most auto-parallelising compilers from attempting parallelisation. The use of modulo addressing prevents standard data dependence analysis and will also cause parallelisation failure. This paper outlines two *program recovery* techniques that will translate restricted pointer arithmetic and modulo addresses into a form suitable for optimisation.

Multi-processor DSPs have a multiple-address memory model which is globally addressable, similar to the Cray T3D/E [22]. This reduces the hardware cost of supporting a single-address space, eliminating the need for hardware consistency engines, but places pressure on the compiler to either generate message-passing code or some other means to ensure correct execution. This paper describes a *mapping* and *address resolution* technique that allows remote data to be accessed without the need for message-passing. It achieves this by developing a baseline mechanism, similar to that used in generating single-address space code, whilst allowing further optimisations to exploit the multiple-address architecture.

As there is no cache structure, the compiler cannot rely on caches to exploit temporal re-use of remote data nor on large cache line sizes to exploit spatial locality. Instead, multiple-address space machines rely on effective use of Direct Memory Access (DMA) transfers [22]. This paper describes multiple-address space specific *locality optimisations* that improve upon our baseline approach. This is achieved by determining the location of data and transforming the program to exploit locality in DMA transfers of remote data. It also exploits the increased bandwidth that is typically available to data that is guaranteed to be on-chip. These location specific optimisations are not required for program correctness (as in the case of message-passing machines [9, 19]) but allow a safe, incremental approach to improving program performance.

This paper develops new techniques and combines them with previous work in a manner that allows, for the first time, efficient mapping of standard DSP benchmarks written in C to multiple address space embedded systems.

This paper is organised as follows: section 2 provides a motivating example and is followed by four sections on notation, program recovery, data parallelisation and

Original code (1)

```
int a[32], b[32], c[32];
int *p_a,*p_b,*p_c;

p_a = &a[0]; p_b = &b[0];
for (i= 0;i<=31;pa++,pb++,i++){
 p_c = &c[0];
 for (j = 0 ; j <=31 ; j++){
1:   *p_a  += *p_b * *p_c++;}
}

for (i = 0 ; i <=31 ; i++)
 for (j = 0 ; j <=31 ; j++)
2:   e[i][j] = f[i] * g[i][j%8]
                  * h[i][j%4];
```

Array recovery and strip-mine (2)

```
int a[32], b[32], c[32];
int *p_a,*p_b,*p_c;

p_a = &a[0]; p_b = &b[0];
for (i=0;i <=31;pa++,pb++,i++){
 p_c = &c[0];
 for (j = 0 ; j <=31 ; j++){
     p_c++;
1:   a[i]  += b[i] + c[j];}
};
for (i = 0 ; i <=31 ; i++)
 for (j1 = 0 ; j1 <=3 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
2:   e[i][8*j1+j2] = f[i]
       * g[i][j2] * h[i][j2%2];
```

Remove pointers and strip-mine (3)

```
int a[32], b[32], c[32];

for (i = 0 ; i <=31 ; i++)
 for (j = 0 ; j <=31 ; j++)
1:   a[i]  += b[i] * c[j];

for (i = 0 ; i <=31 ; i++)
 for (j1 = 0 ; j1 <=3 ; j1++)
  for (j2 = 0 ; j2 <=1 ; j2++)
   for (j3 = 0 ; j3 <=3 ; j3++)
2:   e[i][8*j1+4*j2+j3] = f[i]
       * g[i][4*j2+j3] * h[i][j3];
```

Figure 1. Example showing program recovery.

locality optimisations. Section 7 provides an overall algorithm and evaluation of our approach. This is followed by a brief review of the extensive related work and some concluding remarks.

2 Examples

To illustrate the main points of this paper, two examples are presented allowing a certain separation of concerns. Example 1 demonstrates how program recovery can be used to aid later stages of parallelisation. Example 2 demonstrates how a program containing remote accesses is transformed incrementally to ensure correctness and, wherever possible, exploits the multiple-address space memory model.

Example 1 The code in figure 1, box (1), is typical of C programs written for DSP processors. The use of post-increment pointer traversal is a well know idiom [23] as is circular buffer access using modulo expressions. Such non-linear expressions defeat most data dependence techniques and prevent further optimisation and parallelisation. In our program recovery scheme, the pointers are first replaced with array references based on the loop iterator and the modulo access removed by applying a suitable strip-mining transformation to give the new code in box (2), figure 1. Removing the pointer arithmetic and repeated strip-mining gives the code in box (3). The new form is now suitable for parallelisation and although the new code contains linear array subscripts, these are easily optimised by code hoisting and strength reduction in standard native compilers.

Example 2 Consider the first, array recovered loop nest in box(3) of figure 1. Assuming four processors, single address space compilers would simply partition the iteration space across the four processors. To access remote data in multiple address space machines, however, the processor location of partitioned data needs to be explicitly available. Our scheme achieves this by data strip-mining [20] each array to form a two dimensional array whose inner index is to correspond with the four processors. Box(1), figure 2, shows the program after partitioning by data strip-mining and applying a suitable automatically generated loop recovery [20] transformation. Assuming the `z` loop is parallelised, array `a` is now partitioned such that `a[0][0...7]` is written on processor 0 and `a[1][0...7]` written on processor 1 etc. Similarly for arrays `b` and `c`. For a multiple-address space machine, we now need to generate a separate program for each processor, i.e. explicitly enumerate the processor ID loop, `z`. For example, the partitioned code for processor 0 (as specified by `z`) is shown in figure 2, box (2). Multiple address space machines require remote, globally-accessible data to have a distinct name to local data[1]. Thus, each of the globally-accessible sub-arrays are renamed as follows: `a[0][0...7]` becomes `a0[0...7]` and `a[1][0...7]` becomes `a1[0...7]` etc. On processor 0, `a0` is declared as a variable residing on that processor while `a1,a2,a3` are declared `extern` (see the second and third lines of box (3), figure 2) .

To access both local and remote data, a local pointer array is set up on each processor. We use the original name of the array `a[][]` as the pointer array `*a[]` and then initialise the pointer array to point to the

[1] Otherwise they are assumed to be private copies

Partition to expose proc ID, z (1)

```
int a[4][8],b[4][8],c[4][8];

for (z = 0 ; z <=3 ; z++)
 for (i = 0 ; i <=7 ; i++)
  for (j1 = 0 ; j1 <=3 ; j1++)
   for (j2 = 0 ; j2 <=7 ; j2++)
    a[z][i] += b[z][i]
            * c[j1][j2];
```

One program per processor (2)

```
#define z 0

int a[4][8],b[4][8],c[4][8];

 for (i = 0 ; i <=7 ; i++)
  for (j1 = 0 ; j1 <=3 ; j1++)
   for (j2 = 0 ; j2 <=7 ; j2++)
    a[z][i] += b[z][i]
            * c[j1][j2];
```

Address resolution (3)

```
#define z 0
int a0[8]; /* local */
extern int a1[8], a2[8], a3[8];

int *a[4] ={a0,a1,a2,a3};

for (i = 0 ; i <=7 ; i++)
 for (j1 = 0 ; j1 <=3 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
    a[z][i] += b[z][i]*c[j1][j2];
```

Isolate local and remote refs (4)

```
#define z 0

/* a0,b0 local, c remote */

for (i = 0 ; i <=7 ; i++)
 for (j1 = 0 ; j1 <=z-1 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
    a0[i] += b0[i] * c[j1][j2];

/* a0,b0,c0 all local*/

for (i = 0 ; i <=7 ; i++)
 for (j1 = z ; j1 <=z ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
    a0[i] += b0[i] * c0[j2];

/* a0,b0 local, c remote */

for (i = 0 ; i <=7 ; i++)
 for (j1 = z+1 ; j1 <=3 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
    a0[i] += b0[i] * c[j1][j2];
```

Introduce load loops (5)

```
for (i = 0 ; i <=7 ; i++)
 for (j1 = 0 ; j1 <=z-1 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
     temp[j1][j2] = c[j1][j2];

for (i = 0 ; i <=7 ; i++)
 for (j1 = 0 ; j1 <=z-1 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
   a0[i] += b0[i]* temp[j1][j2];

for (i = 0 ; i <=7 ; i++)
 for (j1 = z ; j1 <=z ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
   a0[i] += b0[i] * c0[j2];

for (i = 0 ; i <=7 ; i++)
 for (j1 = z+1 ; j1 <=3 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
     temp[j1][j2] = c[j1][j2];

for (i = 0 ; i <=7 ; i++)
 for (j1 = z+1 ; j1 <=3 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
   a0[i]+= b0[i]* temp[j1][j2];
```

DMA remote access (6)

```
for (j1 = 0 ; j1 <=z-1 ; j1++)
 get(&temp[8*j1], &c[j1][0],8);

for (i = 0 ; i <=7 ; i++)
 for (j1 = 0 ; j1 <=z-1 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
    a0[i] += b0[i]
          * temp[8*j1];

for (i = 0 ; i <=7 ; i++)
 for (j1 = z ; j1 <=z ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
    a0[i] += b0[i]
          * c0[j2];

for (j1 = z+1 ; j1 <=3 ; j1++)
 get(&temp[8*j1], &c[j1][0],8);

for (i = 0 ; i <=7 ; i++)
 for (j1 = z+1 ; j1 <=3 ; j1++)
  for (j2 = 0 ; j2 <=7 ; j2++)
    a0[i] += b0[i]
          * temp[8*j1];
```

Figure 2. Example showing data transformation, address resolution scheme and locality optimisation applied to first loop in box(3), figure 1. Not all declarations are shown

four distributed arrays `int *a[4] = {a0,a1,a2,a3}` (see box(3), figure 2). Using the original name means that we have exactly the same array access form in all uses of the array as in box(2). This has been achieved by using the property that multi-dimensional arrays in C are defined as containing an array of pointers to sub-arrays.

While this program provides a baseline code each array reference requires a pointer look-up and, as the native compiler does not know the eventual location of the data, it must schedule load/stores that will fit on an external interconnect network or bus. As bandwidth to on-chip SRAM is greater, this will result in under-utilisation of available bandwidth. It is straightforward to identify local references and replace the indirect pointer array access with the local array name, by examining the value of the partitioned indices to see if it equals the local processor ID, `z`.

Data references that are sometimes local and sometimes remote can be isolated by index splitting the program section and replacing the local references with local names. This is shown in box (4), figure 2. Only references to pointer array `c` occur in the first and last loop, all other references are local and transformed to the local name `c0`. Element-wise remote access is expensive and therefore group access to remote data, via DMA transfer, is an effective method to reduce startup overhead. In our scheme remote data elements are transferred into a local temporary storage area. This is achieved by inserting load loops for all remote references as shown in box(5), figure 2. The transfers are performed in such a way to exploit temporal and spatial locality and map potentially distinct multi-dimensional array references, occurring throughout the program, into a single dimension temporary area, which is reused. This is shown in figure 2, box(6).

3 Notation

Before describing the partitioning and mapping approach, we briefly describe the notation used. The loop *iterators* can be represented by a column vector $J = [j_1, j_2, \ldots, j_M]^T$ and the loop ranges are described by a system of inequalities defining the *polyhedron* or *iteration space* $BJ \leq b$. The array *indices* are represented as $\mathcal{I} = [i_1, i_2, \ldots, i_N]^T$ and describe the *array index space* given by $A\mathcal{I} \leq a$. We assume that the subscripts in a reference to an array can be written as $\mathcal{U}J + u$, where $\mathcal{U}$ is an integer matrix and u a vector. Thus in figure 1, box(3) the array declaration `a[32]` is represented by

$$\begin{bmatrix} -1 \\ 1 \end{bmatrix} [i_1] \leq \begin{bmatrix} 0 \\ 31 \end{bmatrix} \tag{1}$$

i.e. the index i_1 ranges over $0 \leq i_1 \leq 31$. The loop bounds are represented in a similar manner and the subscript of `a`, `a[i]`, is simply

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} \tag{2}$$

When discussing larger program structures, we introduce the notion of computation sets where $Q = (BJ \leq b, (s_i | Q_i))$ is a computation set consisting of the loop bounds, $BJ \leq b$ and either enclosed statements $(s_1, \ldots, s_n)$ or further loop nests $(Q_1, \ldots, Q_n)$.

4 Program recovery

This section outlines two program recovery techniques to aid later parallelisation.

4.1 Array Recovery

Array recovery consists of two main stages. The first stage determines whether the program is in a form amenable to conversion and consists of a number of checks. The second stage gathers information on arrays and pointer usage so as to replace pointer references with explicit array accesses and remove pointer arithmetic completely. For more details see [7].

Pointer assignments and arithmetic Pointer assignment and arithmetic are restricted in our analysis. Pointers may be initialised to an array element whose subscript is an affine expression of the enclosing iterators and whose base type is scalar. Simple pointer arithmetic and assignment are also allowed. Pointers to pointers are prohibited in our scheme. An assignment to a dereferenced pointer may have side effects on the relation between other pointers and arrays that is difficult to identify and, fortunately, rarely found in DSP programs.

Dataflow Information Pointer initialisations and updates are captured in a system of dataflow equations which is solved by an efficient one-pass algorithm [7].

Pointer Conversion The index expressions of the array accesses are now constructed from the dataflow information (see loop 1, figure 1, box(2)). In a separate pass pointer accesses and arithmetic are replaced and removed as shown in loop 1, box (3), figure 1.

4.2 Modulo Removal

Modulo addressing is a frequently occurring idiom in DSP programs. We transform the program into an equivalent linear form, if one exists by using the rank-modifying transformation framework [20] which manipulates extended linear expressions including `mods` and `divs`.

We restrict our attention to simple modulo expressions of the form $(a_j \times i_j)\%c_j$ where i_j is an iterator, and a_j, c_j constants and $j \in 1, \ldots, m$ is the reference containing the modulo expression. More complex references are highly unlikely but may be addressed by extending the approach below to include skewing.

Let l be the least common multiple of c_j. In figure 1, box(1), loop 2, we have $c_1 = 8$, $c_2 = 4$ from the access to `g` and `h` and hence $l = 8$. We then apply a loop strip-mining transformation S_l based on l to the loop nest. As $l = 8$,

$$S_l = \begin{bmatrix} 1 & 0 \\ 0 & (.)/8 \\ 0 & (.)\%8 \end{bmatrix} \tag{3}$$

When applied to the iterator, $J' = SJ$, we have the new iteration space:

$$B'J' \leq b' = \begin{bmatrix} S & 0 \\ 0 & 1 \end{bmatrix} BS^{\dagger}SJ \leq \begin{bmatrix} S & 0 \\ 0 & 1 \end{bmatrix} b \tag{4}$$

where $S^{\dagger}$ is the pseudo-inverse of S, in this case:

$$S^{\dagger} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 8 & 1 \end{bmatrix} \tag{5}$$

When applied to the second loop nest in box(1), figure 1 we have the new iteration space:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} j_1 \\ j_2 \\ j_3 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 31 \\ 3 \\ 7 \end{bmatrix} \tag{6}$$

or the loop nest shown in loop nest 2, figure 1, box (2). The new array accesses are found by $\mathcal{U}' = \mathcal{U}S^{\dagger}$ giving the new access shown in the second loop of box(2), figure 1. This process is repeated until no modulo operations remain. The one modulo expression in the array `h` subscript remaining in the second loop of box (2) is removed by applying a further strip-mining transformation to give the code in box (3), figure 1.

5 Data Parallelism

This section briefly describes the baseline parallelisation approach of our scheme.

5.1 Partitioning

We attempt to partition the data along those aligned dimensions of the array that may be evaluated in parallel and minimise communication. More sophisticated approaches are available [4] but are beyond the scope of this paper. Partitioning based on alignment[3, 15]. tries to maximise the rows that are equal in a subscript matrix. The most aligned row, $MaxRow$, determines the index to partition along. We construct a partition matrix $\mathcal{P}$ defined:

$$\mathcal{P}_i = \begin{cases} e_i^T & i = MaxRow \\ 0 & otherwise \end{cases} \tag{7}$$

where e_i^T is the ith row of the identity matrix. We also construct a sequential matrix $\mathbf{S}$ containing those indices not partitioned such that $\mathcal{P} + \mathbf{S} = I$. In the example in figure 2 there is only one index to partition along therefore

$$\mathcal{P} = \begin{bmatrix} 1 \end{bmatrix}, \mathbf{S} = \begin{bmatrix} 0 \end{bmatrix} \tag{8}$$

5.2 Mapping

The partitioned indices are to be mapped to the p processors. Our approach, based on rank-modifying transformations [20], explicitly exposes the processor ID, critical for later stages. We achieve this by data strip-mining the indices $\mathcal{I}$ using the strip-mine matrix S:

$$S = \begin{bmatrix} (.)\%p \\ (.)/p \end{bmatrix} \tag{9}$$

to give the new array indices $\mathcal{I}'$, where, in our example in figure 2,$p = 4$. The mapping transformation T is defined as:

$$T = \mathcal{P}S + \mathbf{S} \tag{10}$$

where the partitioned indices are strip-mined and the sequential indices left alone. In our example

$$T = [1]S + [0] = S \tag{11}$$

and when applied to the index space,$\mathcal{I}' = T\mathcal{I}$, we have the new index space or array bounds:

$$A'\mathcal{I}' \leq a' = \begin{bmatrix} T & O \\ O & T \end{bmatrix} AT^{-1}\mathcal{I}' \leq \begin{bmatrix} T & O \\ O & T \end{bmatrix} a \tag{12}$$

which transforms the array bounds in equation (1) to:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 3 \\ 7 \end{bmatrix} \tag{13}$$

i.e `int a[4][8]`. The new array subscripts are also found: $\mathcal{U}' = T\mathcal{U}$. In general, without any further loop transformations, this will introduce `mods` and `divs` into the array accesses. In our example in figure 2 we would have `a[i%4,i/4] += b[i%4,i/4] + c[j%4,j/4]`

However, this framework [20] always generates a suitable recovery loop transformation, in this case the same transformation T. Applying T to the enclosing loop iterators we have $J' = TJ$ and updating the access matrices we have, for array `a`:

$$\mathcal{U}'' = T\mathcal{U}T^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} \tag{14}$$

i.e.`a[z][i]`. The resulting code is shown in figure 2, box(1) where we have exposed the processor ID of each reference without any expensive subscript expressions.

5.3 Address Resolution

Each of the elements of the innermost indices corresponds to a local sub-array on each of the p processors. As there is no single address space, each sub-array is given a unique name. In order to minimise the impact on code generation, a pointer array of size p is introduced which points to the start address of each of the p renamed sub-arrays. Figure 2, box (3), shows the declarations inserted for the arrays `a`. The declarations for the remaining arrays are similar. For further details on address resolution see [8].

6 Locality Analysis

The straightforward code generated by the previous section, however, introduces runtime overhead and does not exploit locality

6.1 Exploiting local accesses

Bandwidth to on-chip SRAM is greater than via the external bus, hence local bandwidth will be underutilised if the node compiler makes conservative assumptions about address location. In general, determining whether an array reference is entirely local throughout the runtime of a program is non-trivial. However, as our partitioning scheme explicitly incorporates the processor ID in the array reference, we simply check to see if it equals the processor ID, z.

This is a simple syntactic transformation. Given an array access $\mathcal{U}J$ and a pointer array name X and the syntactic concatenation operator : we have

$$X[\mathcal{U}J + u] \mapsto X : u_o[\mathcal{U}_{2,\ldots,N}J + u_{2,\ldots,N}] \tag{15}$$

Applying this to the example in figure 2, box(3) we have the code in box(4) where all access to a0, b0 and c0 can be statically determined as local by the native compiler.

6.2 Locality in remote accesses

Repeated reference to a remote data item will incur multiple remote accesses. Our approach is to determine those elements likely to be remote and to transfer the data to a local temporary. The remote data transfer code is transformed to exploit temporal and spatial locality when using the underlying DMA engine.

Index Splitting We first separate local and remote references by splitting the iteration space into regions where data is either local or remote. As the processor ID is explicit in our framework, we do not need array section analysis to perform this.

For each remote reference, the original loop is partitioned into n separate loop nests using index set splitting:

$$Q(AJ \leq b, Q_1) \mapsto Q_i(AJ \leq b \wedge C_i, Q_1), \forall i \in 0, \ldots, n-1 \tag{16}$$

where $n = 2d + 1$ and d is the number of dimensions partitioned. In the case of box(3), figure 2, we partition on just one dimension, hence $n = 3$ and we have the following constraints:

$$C_0 : 0 \leq j_1 \leq z - 1 \tag{17}$$

$$C_1 : j_1 = z \tag{18}$$

$$C_2 : z + 1 \leq j_1 \leq 3 \tag{19}$$

This gives the program in box(4), figure 2.

Remote data buffer size Before any remote access optimisation can take place, there must be sufficient storage. Let s be the storage available for remote accesses. We simply check that the remote data fits i.e.

$$||\mathcal{U}J|| \leq s \tag{20}$$

If this condition is not met, we currently abandon further optimisation.

Load loops Load loops are introduced to refer to exploit locality of remote access. A temporary, with the same subscripts as the remote reference, is introduced which is then followed by loop distribution. The transformation is of the form

$$Q \mapsto (Q_1, \ldots, Q_K) \tag{21}$$

A single loop nest Q is distributed so that there are now K loop nests, $K-1$ of which are load loops. In our example in figure 2, box(5), there is only one remote array, hence $K = 2$

Transform load loops to exploit locality Temporal locality in the load loops corresponds to an invariant access iterator or the null space of the access matrix i.e. $\mathcal{N}(\mathcal{U})$. There always exists a transformation T, found by reducing $\mathcal{U}$ to Smith-normal form that transforms the iteration space such that the invariant iterator(s) is innermost and can be removed by Fourier-Motzkin elimination. The i loops of both the load loops in box(5), figure 2 are invariant and are removed as shown in box (6).

Stride In order to allow large amounts of remote data to be accessed in one go rather than a separate access per array element, it must be accessed in stride-1 order. This can be achieved by a simple loop transformation T, $T = \mathcal{U}$. In our example, box(5), figure 2, T is the identity transformation as the accesses in the load loop is already in stride-1 order.

Linearise Distinct remote references may be declared as having varying dimensions, yet the data storage area we set aside for remotes accesses is fixed and one-dimensional. Therefore the temporary array must be linearised throughout the program and all references updated accordingly:

1. Perform program recovery
2. IF parallel and worthwhile
   (a) Determine data partition
   (b) Partition + transform data and code
   (c) Perform address translation
   (d) Apply locality optimisations

Figure 3. Overall parallelisation algorithm

Benchmark	Parallelisable	Pointer Conversion	Parallelisable after Pointer Conversion	Profitably Parallelisable
biquad		✓		
convolution		✓	✓	✓
dot_product		✓	✓	
fir		✓	✓	✓
fir2dim		✓	✓	✓
lms		✓	✓	✓
mat1x3		✓	✓	
matrix1		✓	✓	✓
matrix2		✓	✓	✓
n_complex_updates		✓	✓	✓
n_real_updates		✓	✓	✓
adpcm				

Figure 4. Exploitable Parallelism in DSPstone

$$\mathcal{U}_t' = L\mathcal{U}_t \tag{22}$$

In our case $L = \begin{bmatrix} 8 & 1 \end{bmatrix}$ and transforms the array accesses from `temp[j1][j2]` to `temp[8*j1+j2]` in box(5), figure 2.

Convert to DMA form The format of a DMA transfer requires the start address of the remote data and the local memory location where it is to be stored plus the amount of data to be stored. This is achieved by effectively vectorising the inner loop by removing it from the loop body and placing it within the DMA call. The start address of the remote element is given by the lower bound of the innermost loop and the size is equal to its range. Thus we transform the remote array access as follows:

$$\mathcal{U}_M = 0, u_M = min(J_m) \tag{23}$$

The temporary array access is similarly updated and we then remove J_m by Fourier-Motzkin elimination. Finally, we replace the assignment statement by a generic DMA transfer call `get(&tempref, &remoteref, size)` to give the final code in box(6), figure 2.

Benchmark	Parallelisable	Modulo Removal	Parallelisable after Modulo Removal	Profitably Parallelisable
FFT				
FIR		(✓)	✓	✓
IIR		✓	✓	✓
LATNRM	✓			
LMSFIR		(✓)	✓	✓
MULT	✓			✓
ADPCM		✓		
Compress	✓			✓
Edge Detect	✓			✓
G.721 (ML)				
G.721 (WF)				
G.722				
Histogram	✓			✓
JPEG Filter	✓			✓
LPC	✓			
Spectral Estimation	✓			
Trellis				
V.32 Modem				

Figure 5. Exploitable Parallelism in UTDSP

7 Empirical Results

Our overall parallelisation algorithm is shown in figure 3. We currently multiply the parallelised loop trip count by the number of operations and check it is above a certain threshold before continuing beyond step 2. We prototyped our algorithm in the SUIF 1.3 compiler.

We evaluated the effectiveness of our parallelisation scheme against two different benchmark sets: DSPstone[2] [23] and UTDSP [16]. The programs were executed on a Transtech TS-P36N board with a cluster of four cross-connected 250MHz TigerSHARC TS-101 DSPs, all sharing the same external bus and 128MB of external SDRAM. The programs were compiled with the Analog Devices VisualDSP++ 2.0 Compiler (version 6.1.18) with full optimisation; all timings are cycle accurate.

Parallelism Detection Figure 4 shows the set of loop-based DSPstone programs. Initially, the compiler fails to parallelise these programs because they make an extensive use of pointer arithmetic for array traversals, as shown in the second column. However, after applying array recovery (column 3) most of the programs become parallelisable (column 4). In fact, the only program that cannot be parallelised after array conversion (`biquad`) contains a cross-iteration data dependence that does not permit parallelisation. `adpcm` is the only program in this benchmark set that cannot be recovered due to its complexity. The fifth column of figure 4 shows whether or not a program can be profitably parallelised. Programs comprising of only

[2] Artificially small data set sets sizes have been selected by its designers to focus on code generation; we have used a scaled version wherever appropriate.

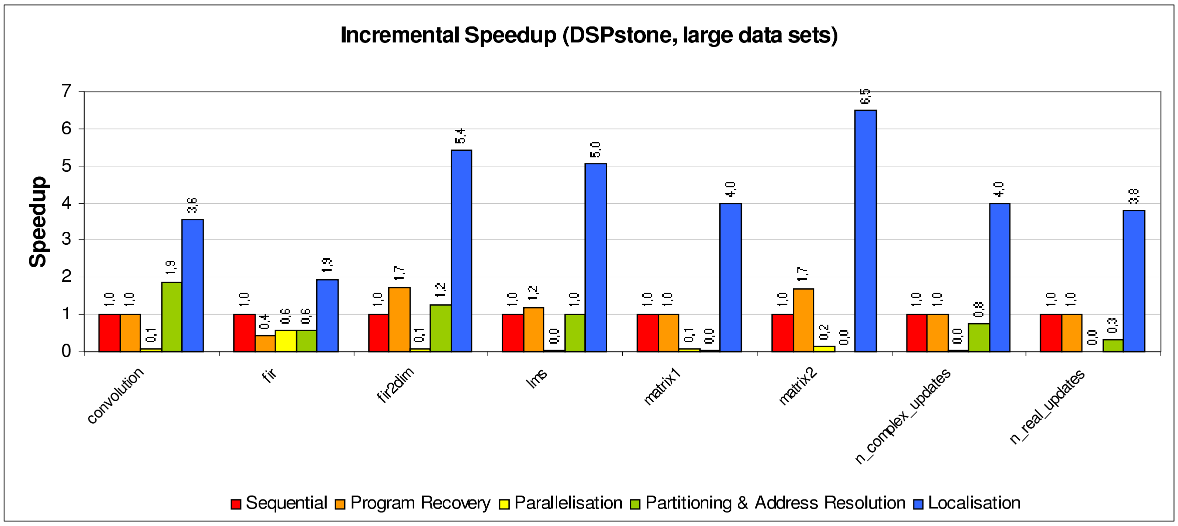

Figure 6. Total Speedup for DSPstone benchmarks

very small loops such as `dot_product` and `matrix1x3` perform better when executed sequentially due to the overhead associated with parallel execution and are filtered out, at stage 2, by our algorithm.

As far as UTDP is concerned, many of the programs are available in their original pointer-based form as well as in an array-based form. Wherever possible, we took the array-based programs as a starting point for our parallelisation[3]. The impact of modulo removal can be seen in figure 5. Four of the UTDSP programs (`iir`,`adpcm`,`fir` and `lmsfir`) can be converted into a modulo-free form by our scheme. Modulo removal has a direct impact on the paralleliser's ability to successfully parallelise those programs – three out of four programs could be parallelised after the application of this transformation. `ADPCM` cannot be parallelised after modulo removal due to data dependences.

Although program recovery is used largely to facilitate parallelisation and multi-processor performance, it can impact sequential performance as well. The first two columns of each set of bars in figures 6 and 7 show the original sequential time and the speedup after program recovery. Three out of the eight DSPstone benchmarks benefit from this transformation, whereas only a single kernel (`fir`) experiences a performance degradation after program recovery. In `fir2dim`,`lms` and `matrix2`, array recovery has enabled better data dependence analysis and allowed a tighter scheduling in each case. `fir` has a very small number of operations such that the slight overhead of enumerating array subscripts has a disproportional effect on its performance. Figure 7 shows the impact of modulo removal on the performance of the UTDSP benchmark. Since a computation of a modulo is a comparatively expensive operation, its removal positively influences the performance of the three programs wherever it is applicable.

[3] Array recovery on the pointer programs gives an equivalent array form

Partitioning and Address Resolution The third column of each set of bars in figures 6 and 7 shows the effect of blindly using a single-address space approach to parallelisation without data distribution on a multiple-address space machine. Not surprisingly, performance is universally poor. The fourth column in each figure shows the performance after applying data partitioning, mapping and address resolution. Although some programs experience a speedup over their sequential version (`convolution` and `fir2dim`), the overall performance is still disappointing. After a closer inspection of the generated assembly codes, it appears that the Analog Devices compiler cannot distinguish between local and remote data. It conservatively assumes all data is remote and generates "slow" accesses, double word instead of quad word, to local data. An increased memory access latency is accounted for in the produced VLIW schedule. In addition all remote memory transactions occur element-wise and do not effectively utilise the DMA engine.

Localisation The final columns of figures 6 and 7 show the performance after the locality optimisations are applied to the partitioned code. Accesses to local

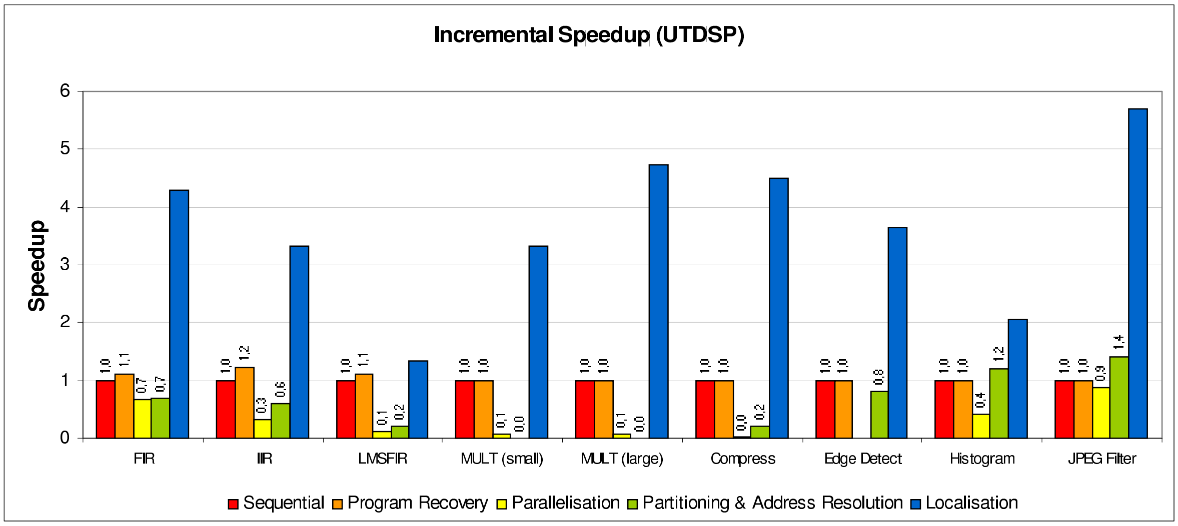

Figure 7. Total Speedup for UTDSP benchmarks

data are made explicit, so the compiler can identify local data and is able to generate tighter and more efficient schedules. In addition, remote memory accesses are grouped to utilise the DMA engine. In the case of DSPstone, linear or superlinear speedups are achieved for all programs bar one (`fir`), where the number of operations is very small. Superlinear speedup occurs in precisely those cases where program recovery has given a sequential improvement over the pointer based code. The overall speedups vary between 1.9 (`fir`) and 6.5 (`matrix2`), their average is 4.28 on four processors. The overall speedup for the UTDSP benchmarks is less dramatic, as the programs are more complex, including full applications, and have a greater communication overhead. These programs show speedups between 1.33 and 5.69, and an average speedup of 3.65. `LMSFIR` and `Histogram` fail to give significant speedup due to the lack of sufficient data parallelism inherent in the programs. Conversely, `FIR`, `MULT(large)`, `Compress` and `JPEG Filter` give superlinear speedup due to improved sequential performance of the programs after parallelisation. As the loops are shorter after parallelisation, it appears that the native loop unrolling algorithm performs better on the reduced trip count.

8 Related Work

There is an extremely large body of work on compiling Fortran dialects for multi-processors. A good overview can be found in [10]. Compiling for message-passing machines had largely focused on the HPF programming language [19]. The main challenge is inserting correctly, efficient message-passing calls into the parallelised program [19, 9] without requiring complex run-time bookkeeping.

Although when compiling for distributed shared memory (DSM), compilers must incorporate data distribution and data locality optimisations [6, 1], they are not faced with the problem of multiple, but globally-addressable address spaces. Both message-passing and DSM platforms have benefitted from the extensive work in automatic data partitioning [4] and alignment [3, 15], potentially removing the need for HPF pragmas for message-passing machines and reducing memory and coherence traffic in the case of DSMs.

The work closest to our approach, [21] examines auto-parallelising techniques for the Cray T3D. To improve communication performance, it introduces private copies of shared data that must be kept consistent using a complex linear memory array access descriptor. In contrast we do not keep copies of shared data, instead we use an access descriptor as a means of having a global name for data. In [21], an analysis is developed for nearest neighbour communication, but not for general communication. As our partitioning scheme exposes the processor ID, it is eliminating the need for any array section analysis and handles general global communication.

In the area of auto-parallelising C compilers, SUIF [11] is the most significant work, though it targets single-address space machines. Modulo recovery for C programs is considered in [2], where a large, highly specialised framework based on Diophantine equations is presented to solve modulo accesses. It, however, introduces floor, div and ceiling functions and its effect on other parts of the program is not considered. There is a large body of work on developing loop and data transformations to improve memory access [13, 5]. In [1], a data transformation, data tiling, is used to improve spatial locality, but the representation does not allow easy integration with other loop and data transformations.

As far as DSP parallelisation is concerned in [12] an interesting overall parallelisation framework is described, but no mechanism or details of how parallelisation might take place is provided. In [18], the impact of different parallelisation techniques is considered, however, this was user-directed and no automatic approach provided. In [14], a semi-automatic parallelisation method to enable design-space exploration of different multi-processor configurations is presented. However, no integrated data partitioning strategy was available and a single address space was assumed in the example codes.

9 Conclusion

Multiple-address space embedded systems have proved a challenge to compiler vendors and researchers due to the complexity of the memory model and idiomatic programming style of DSP applications. This paper has developed an integrated approach that gives an average of 3.78 speedup on four processors when applied to 17 benchmarks from the DSPstone and UTDSP benchmarks. This is a significant finding and suggests that multi-processor DSPs can be by a cost effective solution to high performance embedded applications and that compilers can exploit such architectures automatically. Future work will consider other forms of parallelism found in DSP applications and integrate this with further uni-processor optimisations.

References

[1] **J.M. Anderson, S.P. Amarasinge, M.S. Lam**. Data and Computation Transformations for Multiprocessors. ACM PPoPP, 1995.

[2] **F. Balasa, F.H.M. Franssen, F.V.M. Catthoor, H.J. De Man**. Transformation of Nested Loops with Modulo Indexing to Affine Recurrences Parallel Processing Letters, 4(3), 1994.

[3] **D. Bau, I Kodukla, V. Kotlyar, K. Pingali and P. Stodghill** Solving alignment using elimentary linear algebra, LCPC, LNCS892, 1995.

[4] **B. Bixby, K. Kennedy, U. Kremer**. Automatic data layout using 0-1 integer programming. PACT, 1994.

[5] **S. Carr, K.S. McKinley, C.-W. Tseng**. Compiler Optimizations for Improving Data Locality. ASPLOS, October 1994.

[6] **R. Chandra, D.-K. Chen, R. Cox, D.E. Maydan, N. Nedeljkovic, J.M. Anderson**. Data Distribution Support on Distributed Shared Memory Multiprocessors. ACM SIGPLAN PLDI, 1997

[7] **B. Franke, M. O'Boyle**. Array recovery and high-level transformations for DSP applications ACM TECS 2 (2), 2003.

[8] **B. Franke, M. O'Boyle**. Address Resolution for Multi-Core DSPs with Multiple Address Spaces. *to appear at* CODES-ISSS 2003.

[9] **M. Gupta, E. Schonberg and H. Srinivasan**. A Unified Framework for Optimizing Communication in Data-Parallel Programs. IEEE TPDS, July 1996.

[10] **R. Gupta, S. Pande, K. Psarris, V. Sakar**. Compilation Techniques for Parallel Systems. Parallel Computing, 25(13),1999.

[11] **M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, M.S. Lam**. Maximizing Multiprocessor Performance with the SUIF Compiler. IEEE Computer, December 1996.

[12] **A. Kalavade, J. Othmer, B. Ackland, K.J. Singh**. Software Environment for a Multiprocessor DSP. ACM/IEEE Design Automation Conference, 1999.

[13] **M. Kandemir, J. Ramanujam, A. Choudhary**. Improving cache locality by a combination of loop and data transformations. IEEE TC 48(2), 1999.

[14] **I. Karkowski, H. Corporaal**. Exploiting Fine- and Coarse-grain Parallelism in Embedded Programs. PACT, 1998.

[15] **K. Knobe, J. Lukas, G. Steele Jr.** . Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. JPDC, 8(2),1990.

[16] **C.G. Lee**. UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/ corinna/ DSP/infrastructure/UTDSP.html`.

[17] **A. Leung, K. Palem, A. Pnueli**. Scheduling time-constrained instructions on pipelined processors. ACM TOPLAS 23(1), 2001.

[18] **D.M. Lorts** Combining Parallelization Techniques to Increase Adaptability and Efficiency of Multiprocessing DSP Systems. DSP-2000 Ninth DSP Workshop, Hunt, Texas, 2000.

[19] **J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarria-Miranda, R. Fowler, G. Jin, K. Kennedy, Q. Yi**. Advanced Optimization Strategies in the Rice dHPF Compiler. Concurrency-Practice and Experience, 1, 2001.

[20] **M.F.P. O'Boyle, P.M.W. Knijnenburg**. Integrating loop and Data Transformations for Global Optimisation. PACT 1998.

[21] **Y. Paek, A. G. Navarro, E.L. Zapata and D. A. Padua**. Parallelization of Benchmarks for Scalable Shared-Memory Multiprocessors. PACT 1998.

[22] **S. Scott**. Synchronisation and Communication in the T3E Multiprocessor. ASPLOS, 1996.

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