Compiler Parallelization of C Programs for Multi-Core DSPs with Multiple Address Spaces

Björn Franke
Institute for Computing Systems Architecture (ICSA)
School of Informatics
University of Edinburgh
B.Franke@sms.ed.ac.uk

M.F.P. O'Boyle
Institute for Computing Systems Architecture (ICSA)
School of Informatics
University of Edinburgh
mob@inf.ed.ac.uk

ABSTRACT

This paper develops a new approach to compiling C programs for multiple address space, multi-processor DSPs. It integrates a novel data transformation technique that exposes the processor location of partitioned data into a parallelization strategy. When this is combined with a new address resolution mechanism, it generates efficient programs that run on multiple address spaces without using message passing. This approach is applied to the UTDSP benchmark suite and evaluated on a four processor TigerSHARC board, where it is shown to outperform existing approaches and gives an average speedup of 3.25 on the parallel benchmarks.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Experimentation, Measurement, Performance

Keywords

DSPs, Data Partitioning, Address Resolution, Multiple Address Space Compilation

1. INTRODUCTION.

Although multi-core DSPs offer a cost-effective method of achieving high performance, porting existing uni-processor applications to such parallel architectures is currently complex and time-consuming. There are no commercially available compilers that will take existing sequential DSP programs and map them automatically onto a multi-processor machine [17]. Instead, users are typically required to rewrite their code as a process network or a set of communicating sequential processes [12]. Such an approach is well known to be highly non-trivial and error-prone, possibly introducing deadlock.

Rewriting an application in a parallel manner is a highly specialized skill. What is needed is a tool that takes existing programs and maps them automatically onto the new multiprocessor architecture efficiently. Although there has been over 20 years of research into auto-parallelizing compilation in scientific computing [5, 6, 7, 2], this has not taken place in the embedded domain. This is due to two main reasons: (i) DSP programs are written in C rather than Fortran [7] and make extensive use of pointer arithmetic and (ii) the multiple address spaces of multi-processor DSPs is difficult to compile for. We overcome the first problem by using an array recovery technique developed in our earlier work [3]. The second problem is solved by combining a novel data transformation scheme, that exposes the processor ID of remote data with a new address resolution technique that uses local pointer arrays to track remote data. This low overhead technique greatly simplifies code generation producing code similar to that produced for single-address space parallelization [5] without introducing complex (and potentially deadlocking) message passing code. By embedding these two techniques into an overall parallelization strategy, we have developed an auto-parallelizing C compiler for DSP applications that outperforms existing approaches on multiple address space embedded processors.

The paper is structured as follows. Section 2 provides a motivating example and is followed in section 3 by a detailed description of our auto-parallelization scheme. Section 4 empirically evaluates our approach and is followed by a short review of related work and some concluding remarks.

2. MOTIVATION & EXAMPLE

Our approach exploits the fact that although multi-processor DSP machines typically have multiple address spaces, part of each processor's memory space is visible from other processors, unlike pure message-passing machines. However, unlike single address space machines, such as the SGI Origin 3000, a processor must know both the identity of the remote processor and the location in memory of the required data value. For example, figure 1 (from [1]) shows the global memory map of a multi-processor system comprising the Analog Devices TigerSHARC processor. Each processor typically has its internal address space for accesses to local data. These accesses are purely local and not reflected

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

on the external bus. In addition, the processors' memories form a global address space where each processor is assigned a certain range of addresses. This global address space is used for bus-based accesses to remote data where the global address (or equivalently the remote processor's identity and the data's local address) must be known.

We have developed a novel technique which combines single-address space parallelization approaches with a novel address resolution mechanism, based on a simple descriptor data structure, which, at runtime, determines the processor and memory location of all items.

2.1 Example

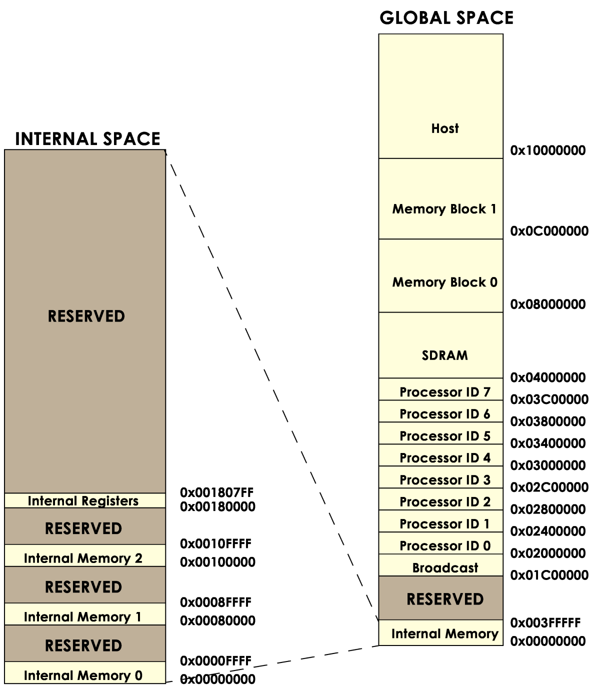

Figure 1: Global Memory Map [1]

The example in figures 2 and 3 illustrates the main points of this paper. The code in figure 2, box (1) is typical of C programs written for DSP processors; it is hybrid of two programs `n_real_updates` and `convolution` from the DSP-stone benchmark suite. The use of post-increment pointer traversal is a well know idiom [19]. This form, however, will prevent many optimizing compilers from performing aggressive optimization and will prevent attempts at parallelization. The second box (2) in figure 2 shows the program after array recovery [3].

The pointers are replaced with array references based on the loop iterator. SPMD owner-computes parallelization by partitioning the data and computation across the processor nodes is straightforward here. There is just one array dimension and one loop, both of which are partitioned by the number of processors. In this example we assume there are four processors. Standard partitioning [5] would generate the program in box (3) and, for single address space machines, this would be sufficient. To access remote data in multi-address space machines, however, the processor location of partitioned data needs to be explicitly available. Our scheme, achieves this by data strip-mining [14] each array to form a two dimensional array whose inner index corresponds to the four processors. This allows the processor location of partitioned data to be explicitly available. For instance, the array `D` is now partitioned such that `D[0][0...7]` resides on processor 0 and `D[1][0...7]` resides on processor 1 etc. Similarly for arrays `C,A` and `B`. The iterator is similarly partitioned to iterate over the work allocated to it, $0 \dots 7$. The resulting code is shown in figure 2, box (4). Only the declarations of `D` and `B` are shown to simplify presentation of the later transformation stages.

We now need to generate a separate program for each processor. The partitioned code for processor 0 (as specified by `z`) is shown in figure 2, box (5). The code for processors 1,2,3 are identical except for `#define z 1,2` or `3`. 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: `D[0][0...7]` becomes `D0[0...7]` and `D[1][0...7]` becomes `D1[0...7]` etc. Similarly for arrays `C, A` and `B`. On processor 0 `D0` is declared as a variable residing on that processor while `D1, D2,D3` are declared `extern` (see the second and third lines of box (6) figure 2) . For processor 1, `D1` is declared local and the remainder are declared `extern` etc.

To access both local and remote data, a local pointer array is set up on each processor. This simple descriptor data structure is an array containing four pointer elements, which are assigned to the start address of the local arrays on the four processors. We use the original name of the array `D[][]` as the pointer array `*D[]` and then initialize the pointer array to point to the four distributed arrays `int *D[4] = {D0,D1,D2,D3}` (see box (6), figure 2) . This is shown diagrammatically in figure 3 for arrays `B` and `D`. The bold lines represent the data accessed by processor 0. Thus, there is local access to array `D0` but remote access to array `B3` by processor 0. Using the original name means that we have exactly the same array access form in all uses of the arrays `D,C,A,B` as in box(4). This has been achieved by using the fact that multi-dimensional arrays in C are arrays of arrays and that higher dimensions arrays are defined as containing an array of pointers to sub-arrays[2]. From a code generation point of view this greatly simplifies implementation and avoids complex and difficult to automate message passing.

3. OVERALL PARALLELIZATION ALGORITHM

Our overall parallelization algorithm is shown in figure 4. Pointer to array conversion is first applied to enable data dependence analysis (for further details see [3]). Once the program is in a pointer free form, standard data dependence analysis is applied to determine if the program is parallel and if so, we check to see if the amount available justifies parallelization[3].

3.1 Partitioning and Mapping

[1]Otherwise they are assumed to be private copies

[2]As defined in section 6.5.2.1 of the ANSI C standard paragraphs 3 and 4

[3]Currently we multiply the parallelized loop trip count by the number of operations and check it is above a certain threshold before continuing

Original code (1)

```
int *p_d,p_c,p_a,p_b;

for (i = 0 ; i <=31 ; i++)
   *p_d++  = *p_c++ + *p_a++ * *p_b-- ;
```

Array Recovery (2)

```
int D[32], C[32], A[32], B[32];

for (i = 0 ; i <=31 ; i++)
   D[i]  = C[i] + A[i] * B[32-i] ;
```

Standard Partitioning (3)

```
int D[32], C[32], A[32], B[32];

lo = (myid *32/numprocs);
hi = ((myid + 1) *32/numprocs) -1;

for (i = lo ; i <=hi ; i++)
   D[i]  = C[i] + A[i] * B[32-i] ;
```

Data transformation exposing processor ID (4)

```
int D[4][8], B[4][8];

for (z = 0; i<=3; i++)
 for (i = 0; i<=7; i++)

 D[z][i]  = C[z][i] + A[z][i] * B[4-z][7-i];
```

One program per processor (5)

```
#define z 0

int D[4][8], B[4][8];

for (i = 0; i<=7; i++)

 D[z][i]  = C[z][i] + A[z][i] * B[4-z][7-i];
```

Address resolution: local pointer arrays(6)

```
#define z 0

int D0[8];                        /* local  */
extern int D1[8], D2[8], D3[8]; /* remote */

int B0[8];                        /* local  */
extern int B1[8], B2[8], B3[8]; /* remote */


int *D[4] ={D0,D1,D2,D3}; /* pointer array */
int *B[4] ={B0,B1,B2,B3}; /* pointer array */

for (i = 0; i<=7; i++)

 D[z][i]  = C[z][i] + A[z][i] * B[4-z][7-i];
```

Figure 2: Example showing data transformation and address resolution scheme.

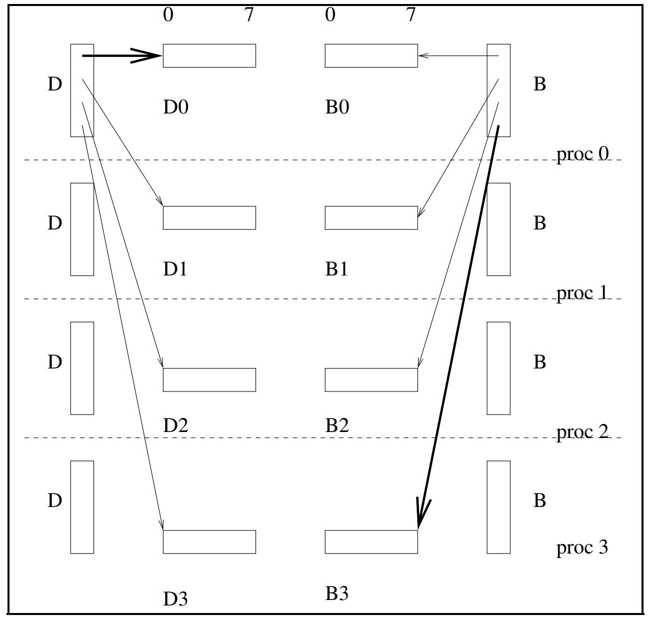

Figure 3: Data layout of arrays `B`, `D` for figure 2 box (5). Each processor has a local pointer array. The contents of the pointer arrays on processor 0 are illustrated. The bold lines represent the data accessed by the program in figure 2 box (6)

1. Convert pointers to arrays
2. IF pointer free THEN perform data dependence analysis
   (a) IF parallelizable and worthwhile
      i. Determine data partition
      ii. Partition + transform data and code
      iii. Perform address resolution

Figure 4: Overall parallelization algorithm

We exploit data parallelism within DSP programs by partitioning the data and computation across the processors. Choosing the best data partition has been studied for many years and is NP-complete [5]. In this paper we use a simple method based on exploiting parallelism and reducing communication [16]; later work will use more sophisticated analysis. The key point is that our partitioning approach using data strip-mining makes the processor identifier explicit, which is exactly what is needed to statically determine whether data is local or remote.

3.1.1 Notation

The loop *iterators* can be represented by a column vector $J = [j_1, j_2, \ldots, j_M]^T$ where M is the number of enclosing loops. Note the loops do not need to be perfectly nested and occur arbitrarily in the program. The loop ranges are described by a system of inequalities defining the *polyhedron* or *iteration space* $BJ \leq b$. The data storage of an array can also be viewed as a polyhedron. We introduce array *indices* $\mathcal{I} = [i_1, i_2, \ldots, i_N]^T$, to describe the *array index space*. This space is given by the polyhedron $A\mathcal{I} \leq a$. 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 is a vector. Thus in figure 2, box(2) the array declaration `D[32]` is represented by

$$\begin{bmatrix} -1 \\ 1 \end{bmatrix} [i_1] \leq \begin{bmatrix} 0 \\ 31 \end{bmatrix} \qquad (1)$$

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 `D[i]` is simply $[1][j_1] + [0]$.

3.1.2 Partitioning

Determining those index dimensions that may be evaluated in parallel, in general, gives a number of options and therefore we use a simple technique to reduce communication based on data alignment. If two array references have every element in a certain dimension corresponding to the same index space points then we say they are aligned. i.e. a[i][j] and b[i][k] are aligned on the first index but not on the second. If two arrays are aligned with respect to a particular index, then no matter how those individual array elements are partitioned, any reference with respect to this index will always be local. Partitioning based on alignment tries to maximize the rows that are equal in a subscript matrix.

Let $\delta(x, y)$ be defined as follows:

$$\delta(x,y) = \begin{cases} 1 & x = y \wedge x \neq 0 \\ 0 & otherwise \end{cases} \qquad (2)$$

This function determines whether two subscripts are non-zero and equal. The function $H(i)$ defined as:

$$H(i) = \sum_t \delta(\mathcal{U}_i^1, \mathcal{U}_i^t) \qquad (3)$$

which measures how well a particular index i of an array read in assignment, $\mathcal{U}^t$, is aligned with the assigned array, $\mathcal{U}^1$. For each index the value of H is calculated, the index with the highest value being the one to partition along.

This technique is applied across all statements and in general there will be conflicting partition requirements. Currently, we consider only those statements in the deepest nested loops as they dominate execution time and calculate the value of H across all these statements for different parallel indices i. The highest value, $MaxH$, determines the index to partition along. We construct a partition matrix $\mathcal{P}$ defined:

$$\mathcal{P}_i = \begin{cases} e_i^T & i = MaxH \\ 0 & otherwize \end{cases} \qquad (4)$$

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

$$\mathcal{P} = \begin{bmatrix} 1 \end{bmatrix} \mathbf{S} = \begin{bmatrix} 0 \end{bmatrix} \qquad (5)$$

3.1.3 Mapping

Once the array indices to partition have been decided, we data strip-mine the indices $\mathcal{I}$ based on the partition matrix $\mathcal{P}$ and strip-mine matrix S to give the new array indices $\mathcal{I}'$. The data strip-mining matrix S is defined:

$$S = \begin{bmatrix} (.)\%p \\ (.)/p \end{bmatrix} \qquad (6)$$

where p is the number of processors. In our example in figure 2, $p = 4$ For further details of the transformation framework see [14]. Here, the transformation is used to expose the processor ID of all array references and is critical to our scheme. To show this, let T be the mapping transformation

$$T = \mathcal{P}S + \mathbf{S} \qquad (7)$$

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

$$T = [1]S + [0] = S \qquad (8)$$

and the new array indices, $\mathcal{I}'$, are given by

$$\mathcal{I}' = T\mathcal{I} \qquad (9)$$

The new array bounds, $A'\mathcal{I}' \leq a'$ are then found using

$$\begin{bmatrix} T & O \\ O & T \end{bmatrix} AT^{-1}\mathcal{I}' \leq \begin{bmatrix} T & O \\ O & T \end{bmatrix} a \qquad (10)$$

which transforms the bounds in equation (1) to:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 3 \\ 7 \end{bmatrix} \qquad (11)$$

i.e `int D[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 `D[i%4,i/4] = C[i%4,i/4] + A[i%4,i/4] * B[(32-i)%4,(32-i)/4]`

However, this framework [14] always generates a suitable recovery loop transformation, in this case the same transformation T. The original loop bounds, $BJ \leq b$ in figure 2, box(2) are given by

$$\begin{bmatrix} -1 \\ 1 \end{bmatrix} [j_1] \leq \begin{bmatrix} 0 \\ 31 \end{bmatrix} \qquad (12)$$

i.e. the iterator `i` ranges over $0 \leq$ `i` ≤ 31. Applying T to the enclosing loop iterators gives the new iterators

$$J' = TJ \qquad (13)$$

The new iterator bounds, $B'J' \leq b'$ are then found using

$$\begin{bmatrix} T & O \\ O & T \end{bmatrix} BT^{-1}J' \leq \begin{bmatrix} T & O \\ O & T \end{bmatrix} b \qquad (14)$$

which transforms the bounds in equation (12) to:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 3 \\ 7 \end{bmatrix} \qquad (15)$$

i.e $0 \leq$ `z` $\leq 3, 0 \leq$ `i` ≤ 7 as shown in box (4) of figure 2. Updating the access matrices we have, for array `D`:

$$\mathcal{U}'' = T\mathcal{U}T^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} \qquad (16)$$

i.e.`D[z][i]`. The resulting code is shown in figure 2 box(4) where we have exposed the processor ID of each reference without any expensive subscript expressions.

1. For each program $q_i \in 1, \ldots p$
   (a) For each arrayName
      i. For $j \in 1, \ldots, p$
         A. IF $(j \neq i)$ THEN insert (`extern`)
         B. insert(TYPE arrayName$j$ `[`N/p`];`)
      ii. insert (TYPE `*arrayName[`p`] = (`)
      iii. For $j \in 1, \ldots, p-1$
         A. insert (arrayName$j$ `,`)
      iv. insert (arrayName$p$ `,);`)

Figure 5: Address Resolution

3.2 Address Resolution

Once the array is partitioned across several processors, each local partition has to be given a new name as there is no single address space supported. Furthermore, the names must be distinct as two variables having the same name on distinct processors are assumed to be private variables. We therefore introduce a new name equal to the old name followed by the processor identify number. Thus, in a four processor system, X will be replaced by four local arrays $X0, X1, X2, X3$. We wish one of these arrays to reside and be allocated on the local processor and the remainder be declared `extern` allowing remote references, the address of which will be resolved by the linker.

In order to minimize the impact on code generation, a pointer array of size p is introduced which points to the start address of each of the p sub-arrays. Unlike the sub-arrays, this pointer array is replicated across the p processors and is initialized by an array initialization statement at the beginning of the program. The complete algorithm is given in figure 5 where the function insert, inserts the declarations. Figure 2, box (6), shows the declarations inserted for two of the arrays, `D` and `B`. The declarations for the remaining arrays are omitted due to lack of space.

The only further change is the type declaration whenever the arrays are passed into function. The type declaration is changed from `int [] []` to `*int[]` and this must be propagated interprocedurely. Once this has been applied, no further transformation or code modification is required.

3.2.1 Synchronization

It is beyond the scope of this paper to describe synchronization placement, though this is essential for correct execution. We mark all cross-processor dependences and use a graph based algorithm to insert the minimal number of barrier synchronizations [15].

4. EXPERIMENTAL FRAMEWORK

The best publicly available, auto-parallelizing, C compiler SUIF produces shared-memory data parallel C codes from sequential C input files and required the porting of the SUIF parallel runtime library to our target architecture. We prototyped our algorithm in the SUIF 1.3 compiler and evaluated it against the well-known DSP benchmark suite, UTDSP [11]. UTDSP contains both DSP kernels and applications. 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++ Compiler (version 6.1.18) with full optimization; all timings are cycle accurate. In addition to the partitioning and address resolution scheme described in the paper, further locality optimisations to exploit DMA transfers were also applied. This is beyond the scope of this paper. For further details see [4].

4.1 Performance

4.1.1 Parallelism and Scalability

While 19 of the 30 UTDSP programs are parallel, only 9 of the 19 are determined profitably parallelizable by our technique. The others are either sequential or exhibit too little work to parallelize and are rejected at stage 2(a) in our overall algorithm (see figure 4). Figure 6 shows how our approach scales with the number of processors for three selected programs, the *Compress, FIR 256/64* and *Mult_10_10* benchmarks. *Mult_10_10* only contains a 10 by 10 array yet after distributing the data across the processors, some speedup is available. In the other cases, with larger data sizes, the full potential of parallel execution can be exploited giving linear speedup with the number of processors.

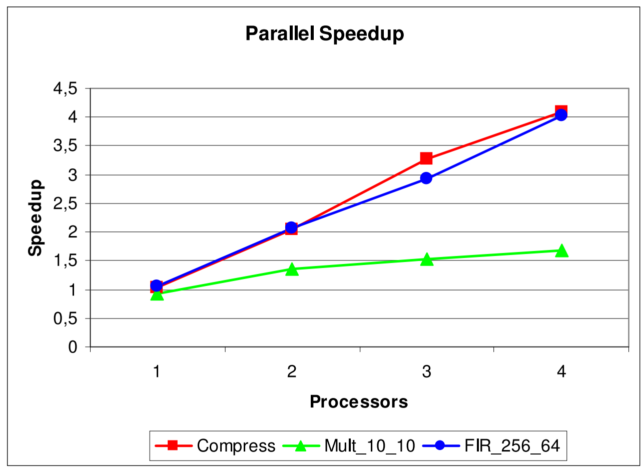

Figure 6: Speedup for different numbers of processors

4.1.2 Speedup due to Parallelization

An overview of the speedup of the selected UTDSP programs over their sequential versions is shown in figure 7. All programs benefit from our approach and show a substantial speedup in the range of 1.54 to 5.69. Linear or close to linear speedup is achieved for four programs (*FIR256/64, IIR4/64, Compress, Edge Detect*), whereas one program (*JPEG filter*) experiences super-linear speedup due to better sequential code generation after parallelization. The remaining three programs do not scale that well with the number of processors, still the speedup due to parallelization is at least 1.5. On average, we have a speedup of 3.25 across the four processors.

5. RELATED WORK

A good overview of existing parallelization techniques is given by Gupta et al. [6]. Cache-coherent multiprocessors with distributed shared memory are the subject of Chandra et al. [5]. Although compilers for such machines must

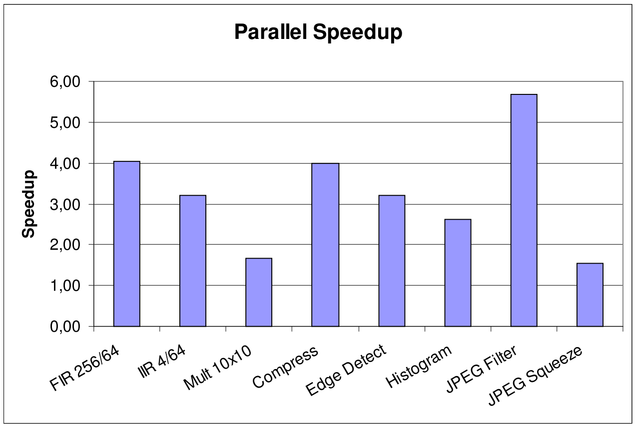

Figure 7: Speedup: UTDSP benchmarks on 4 processors

incorporate data distribution and data locality increasing techniques [2], they are not faced with the problem of multiple, but globally-addressable address spaces. Compiler-Implemented Shared Memory (CISM) as described by Larus [10] and Hiranandani et al. [7] is a method to establish shared memory on message-passing computers. However, these approaches assume separate distributed address spaces and require complex run-time bookkeeping. In [2], data tiling is used to improve spatial locality but the representation does not allow easy integration with other loop and data transformations.

An early paper [18] described how DSP programs may be parallelized but gave no details or experimental results. Similarly, in [8] an interesting overall parallelization framework is described but no mechanism or details of how parallelization might take place is provided. In [13] the impact of different parallelization techniques is considered, however, this was user-directed and no automatic approach provided. In [9] a semi-automatic parallelization method to enable design-space exploration of different multi-processor configurations based on the MOVE architecture is presented. However, no integrated data partitioning strategy was available and a single address space was assumed in the example codes. Furthermore, in the experiments, communication was modelled in their simulator and thus the issue of mapping parallelism combined with distributed address spaces was not addressed.

6. CONCLUSION

In this paper we have developed a new compiler parallelization approach that maps C programs onto multiple address space multi-DSPs. By using a novel data transformation and address resolution mechanism, we have achieved an average 3.25 speedup on four processors for those UTDSP program's containing sufficient parallelism.

The code generated is easy to read and amenable to further sequential optimization. Future work will consider the integration of multi-processor and uni-processor optimizations and consider the integration of task and pipeline parallelism.

7. REFERENCES

[1] Analog Devices, Inc. TigerSHARC DSP Hardware Specification Norwood, Mass., 2001.

[2] J. Anderson, S. Amarasinghe and M. Lam, Data and computation transformations for multiprocessors *ACM PPoPP*, 1995.

[3] B. Franke , M. F.P. O'Boyle, Array recovery and high-level transformations for DSP applications, *ACM Transactions on Embedded Computing Systems (TECS) 2 (2)*,2003.

[4] B. Franke , M. F.P. O'Boyle, Combining Program Recovery, Auto-parallelisation and Locality Analysis for C programs on Multi-processor Embedded Systems (to appear) *IEEE PACT* 2003.

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

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

[7] S. Hiranandani, K. Kennedy, C.-W-. Tseng, Compiling Fortran D for MIMD Distributed-Memory Machines. *CACM,35,(8)*, 1992.

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

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

[10] J.R. Larus. Compiling for Shared-Memory and Message-Passing Computers. ACM LOPLAS 2, (1-4), 1993.

[11] C.G. Lee, UTDSP Benchmark Suite. *http://www.eecg.toronto.edu/ corinna/DSP*

[12] E.A. Lee, Dataflow process networks. *Proceedings of the IEEE*, 83(5), May 1995.

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

[14] O'Boyle M.F.P and Knijnenberg P.M.W., Integrating Loop and Data Transformations for Global Optimisation, *JPDC*, 62, April 2002.

[15] O'Boyle M.F.P. and Stohr E.A., Compile Time Barrier Synchronisation Minimisation, *IEEE TPDS*, 13(6), June 2002.

[16] O'Boyle M.F.P. and Hedayat G.A., Data Alignment: Transformation to Reduce Communication on Distributed Memory Architectures, *IEEE Scalable High Performance Computing Conference*, 1992.

[17] E. Rypkema, E.F. Deprettere, B. Kienhuis, Compilation from Matlab to Process Networks. *CASES*, 1999.

[18] J. Teich, L. Thiele, Uniform Design of Parallel Programs for DSP. *IEEE Int. Symp. Circuits and Systems*, Singapore, June 1991.

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