Array Recovery and High Level Transformations for DSP Applications

Björn Franke Michael O'Boyle
{bf,mob}@dcs.ed.ac.uk
phone: +44 131 650 5117 fax: +44 131 667 7209
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. As DSP applications typically make extensive use of pointer arithmetic, the alternative use of high level, source to source, transformations has been largely ignored. This paper develops an array recovery technique that automatically converts pointers to arrays, enabling the empirical evaluation of high level 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 pointer to array conversion and 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 [13].

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 [4, 20, 32, 28, 7]. 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 [32], 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, source to source, transformations. These are inherently portable and have been shown to give significant performance improvement for general purpose processors [18], yet there is empirical evaluation of 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 the extensive usage of pointer arithmetic [19, 36, 23] which prevents the application of well developed array based dataflow analysis and transformations. Indeed, in [23] programmers are actively encouraged to use pointer based code in the mistaken belief that the compiler will generate better code. Although at one time, these pointer based programs may have performed well with the contemporary compiler technology, they frequently makes matters *worse* for the current generation of optimising compilers. This is precisely analogous to the development of early scientific codes in Fortran where convoluted code was created to cope with the inadequacies of the then current compilers but have now become "dusty decks", making optimisation much more challenging.

In this paper we develop a technique to transform pointer based programs into an equivalent array based form which opens up the opportunity for application of more extensive high level transformations. Using this array based form, we empirically evaluate the impact of high level transformations on 3 different platforms.

There has been limited work in the evaluation of transformations on embedded systems performance. Most focus on techniques which may be easily applied in the presence of pointers, do not consider the interaction with other transformations and are very restricted in the benchmarks and platforms considered. For instance, source level software pipelining for DSP applications is investigated in [35] and [30]. The evaluation is restricted to a single architecture (Motorola DSP56300) where it achieved good results for a small set of benchmark programs.

In [8], the tradeoff between memory space and execution time with loop unrolling has been investigated and in [16] the impact of

```
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++ ;
  }
}
```

Figure 1: Original pointer-based array traversal

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 [36] 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 of pointer to array conversion. Section 3 describes the program representation used in our conversion and section 4 describes the conversion algorithm. Section 5 describes the DSPstone benchmarks, the three embedded processors and the transformations investigated and is followed in section 6 by a description and analysis of the results. Section 7 discusses related work and the incorporation of high level transformations into a compiler and section 8 concludes the paper.

2. MOTIVATION

Pointer accesses to array data frequently occurs in typical DSP programs. Many DSP architectures have specialised *Address Generation Units (AGUs)* [20] but early compilers were unable to generate efficient code for them, especially in programs containing explicit array references. Programmers, therefore, used pointer-based accesses and pointer arithmetic within their programs in order to give “hints” to the early compiler on how and when to use post/pre-increment/decrement addressing modes available in AGUs. For instance, consider figure 1, a kernel loop of the *DSPstone* benchmark `matrix2.c`. Here the pointer increment accesses “encourage” the compiler to utilise the post-increment address modes of the AGU of a DSP.

If, however, further analysis and 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 correctness, compilers use conservative strategies, limiting the maximal performance of the produced code.

Although general array access and pointer analysis are without further restrictions intractable [21], it is easier to find suitable re-

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

Figure 2: After conversion to explicit array accesses

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

Figure 3: Loop after delinearisation

strictions of the array data dependence problem while keeping the resulting algorithm applicable to real-world programs. Furthermore, as array based analysis is more mature than pointer based analysis within optimising compilers, programs containing arrays rather than pointers are more likely to be efficiently implemented. 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 [4, 20] and enables the application of well known high level transformations.

Figure 2 shows the loop with explicit array indexes that is semantically equivalent to the previous loop in Example 1. Not only it is easier to read and understand for a human reader, but it is amendable to compiler array dataflow analyses, e.g. [11].

Once in an array based form, further program transformations may also be applied. Figure 3 shows the example loop after application of conversion and delinearisation. Delinearisation is the transformation of one-dimensional arrays and their accesses into multi-dimensional arrays [24]. The arrays `A`, `B` and `C` are now 2 dimensional arrays. Such a representation enables more aggressive compiler optimisations such as data layout optimisations [24].

3. PROGRAM REPRESENTATION

This section briefly describes the program representation used in our pointer to array conversion algorithm.

The pointer conversion algorithm is a source-to-source transformation that requires a high-level intermediate representations, preserving C constructs. Therefore, along with the standard Control-Flow Graph(CFG) we use the loop tree [22] where loop structures, array accesses and pointer arithmetic are easily identifiable.

The loop tree represents a partial order on the loops contained in it. The relation of a loop L_2 that is an inner loop of loop L_1 (or any of its inner loops) can be written as $L_2 \subset L_1$. The $\subset$ relation is reflexive, transitive and anti-symmetrical. An example of the loop tree representation is given in figure 4.

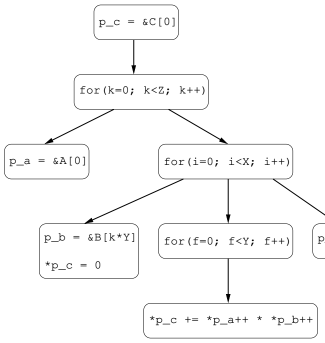

Figure 4: DSPstone `matrix1` C code and the corresponding loop tree

3.1 Other Definitions

For convenience, the innermost embracing loop is denoted by $inner(x)$. If the node x is outside of any loop, then $inner(x) = \perp$.

The functions $lower(L), upper(L)$ return the lower and upper bounds of a loop. $range(L)$ returns the range of a normalised **for** loop L. In cases where the values are not a constant, but an expression, this symbolic expression is returned for further processing.

Any node of the loop tree containing a loop header can be contracted to a special summary node. This type of node is used to summarise the effects of the loop with its possible inner loops on pointers variables. In later analysis steps, summary nodes can be handled in the same way as ordinary nodes i.e. as a single statement.

4. POINTER TO ARRAY CONVERSION

Pointer conversion 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. These checks err on the conservative side ensuring that any program satisfying these constraints may be safely considered by the pointer conversion stage. The second stage gathers information on arrays and pointer initialisation, pointer increments and the properties of loop nests. This information is used to replace pointer accesses by the corresponding explicit array accesses and to remove pointer arithmetic completely. The overall structure of the algorithm is shown in figure 5.

The pointer conversion algorithm simply changes the representation of memory accesses and is largely a syntactic change. Therefore, overlapping arrays and multiple pointers to the same array that often prevent standard program analysis, do not interfere with the conversion algorithm.

Section 4.1 describes the restricted form of programs considered by our scheme and the checks used to guarantee compliance. This is followed in section 4.2 by a description of the actual conversion algorithm based on a dataflow framework.

4.1 Assumptions and Restrictions

In order to facilitate pointer conversion and to guarantee its correctness the overall requirement can be broken into a number of checks in the first step of the algorithm.

Structured loops. Loops are restricted to those with a normalised iteration range going from the lower bound 0 to some upper bound UB with a step of 1. We assume interprocedural constant propagation [9, 27, 14] has taken place and the upper bound may be a constant or a affine expression only containing outer loop variables in the case of a loop nest. Structured loops must have the Single-Entry/Single-Exit property.

It is relatively trivial to determine that all loops conform to this restriction; after loop normalisation, simply check all loops $L_i, i \in 1, \ldots, n$ such that $upper(L_i)$ is an affine expression of syntactically enclosing loops $L_1, \ldots, L_{i-1}$.

1. Check Program
   (a) Ensure structured loops
   (b) Ensure restricted pointer assignment and arithmetic
   (c) Check no pointers to pointers
2. Transform
   (a) IF legal THEN Gather pointer information using dataflow analysis
   (b) IF legal THEN Translate pointers to arrays, remove redundant pointers arithmetic

Figure 5: Overall Algorithm

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. Ensuring the restrictions on pointer assignment and arithmetic can be broken into two stages, the first of which is syntactic. Pointer expressions and assignments are restricted to the following syntactic categories:

1. ptr_expr = (ptr (“++”|“

```
{--}'') $|$  ptr (``+''$|$``-'') constant)

\item ptr\_assign = ptr ``='' (ptr\_expr $| $``\&''
ray ``[''affin\_expr``]''$|$``\&'' var $|$ ar-
ray )

\item   ptr\_assign = ptr ``+='' constant

\item  ptr\_assign =  ptr ``-='' constant
```

Note that dynamic memory allocation and deallocation are implicitly excluded because of the potentially unbounded complexity of dynamic data structures.

In addition any pointer use must be dominated by a node that contains the corresponding pointer assignment. In other words, a pointer cannot be used before it has been correctly initialised. This requirement can be stated more formally:

$\forall p \in P, USE(p) \exists q \in P | q \in DEF(q) \wedge DOM(q,p)$

where P is the set of pointers, and USE, DEF and DOM are the usual dataflow terms referring to the use of a variable, its definition and the dominance of one node over another.

Pointers to other pointers. 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

1. For each procedure P
   (a) Forall statements s in the procedure P
      i. IF a variable v is declared as a pointer, *type* * v, used as a pointer * v,assigned an address of a var, v = & *var* THEN pointer = pointer $\cup v$.
   (b) Forall statements s in the procedure P
      i. IF $v, w \in$ pointer and s is of the form * $v = w$ or v = &w THEN my_contam = my_contam $\cup$ { v,w}
      ii. return_contam = my_contam $\cap$ argument_pointers
2. Check (main, $\emptyset$) where Check (P, inherited pointers) =
   (a) Forall statements s in P
      i. If s a function call, child_contam =child_contam $\cup$ check (s, pointers)
      ii. my_contam = my_contam $\cup$ child_contam
   (b) return my_contam $\cap$ arg_pointers
3. Propagate (main, $\emptyset$) where Propagate (P, inherited) =
   (a) my_contam = my_contam $\cup$ inherited
   (b) Forall statements s in P
      i. If s a function call, Propagate (s, my_contamin$\cap$ actual_args)

Figure 6: Pointer to pointer analysis

is difficult to identify and, fortunately, rarely found in DSP programs. We take a conservative approach where any variable which is declared as a pointer, dereferenced or assigned an address is considered a pointer variable. Just as pointer to pointers are prohibited, so are *function calls* which take addresses of pointers as arguments, as the pointer may be changed. Thus taking the address of a pointer is prohibited in our scheme. These restrictions imply that *function pointers* are prohibited which is acceptable for DSP programs as they are rarely, if ever, used.

This pointer classification scheme can be realised as a simple, interprocedural non-iterative algorithm. Any variable identified as a pointer either by its declaration or use, is labelled as a pointer. The set of pointers is passed into any called function in which any variable "contaminated" with an already recognised pointer is also included in the pointer set. If the address of a pointer is taken or a pointed to variable assigned a pointer value is discovered, the program is rejected. This simple pointer classification algorithm, described in figure 6 will determine potential pointers to pointers occurrences. At present we take a conservative approach where pointer conversion is aborted if any potential pointer to pointer cases are found.

Example. Figure 7 shows a program fragment illustrating legal and illegal constructs for pointer to array conversion. The initialisations of pointers `ptr1` and `ptr2` are legal; `ptr1` is statically initialised and `ptr2` is initialised repeatedly within a loop construct to an array with an affine subscript.

Pointers `pta`, `ptb` and `ptc` are examples of illegal initialisation; a block of memory is dynamically allocated and assigned to `ptra`, whereas the initialisation of `ptrb` is data dependent, the subscript `b[i]` is not an affine expression. Finally, the assignment to `ptrc` is illegal as there is no dominating definition of `ptrX`.

```
int array1[100], array2[100];
int *ptr1 = &array1[5];     /* OK */
int *ptr2,ptra,ptrb,ptrc,ptr;

ptra = (int *) malloc(...);    /* Not ok*/
ptrc = ptrX;                   /* Not ok*/

for(i = 0; i < N; i++)
{
   ptrb = &a[b[i]];        /* Not ok*/
   ptr2 = &array2[i];       /* OK */
   for(j = 0; j < M; j++)
      ...
}
ptr1++;        /* OK*/
ptr1 += 4;     /* OK*/

ptr2 += x;     /* Not ok*/
ptr2 -= f(y);  /* Not ok*/

function1(ptr1);     /* OK */
function2(&ptr1);    /* Not ok */
```

Figure 7: Example of legal and illegal pointer usage

Pointer arithmetic is restricted in our scheme; the constant increments to pointer `ptr1` are legal. However the modification of pointer `ptr2` is illegal as they include compile-time unknown values.

Finally, two examples of passing pointers to functions are also shown. `function1` receives a pointer to the array, thus only the content of the array can be changed, but not the pointer `ptr1` itself. However, `function2` may change the pointer `ptr1`, which is not permissible.

4.2 Dataflow Information

Once the program has been checked, the second stage of the algorithm gathers information on pointer usage before pointer conversion.

Information on pointer assignments and modifications is stored at each node of the loop tree. In particular, for each declared pointer, the mapping between this pointer and an array, including the position of the pointer within the array, is recorded . Additionally, the label of the node containing the most recent assignment to each pointer is stored.

The data for a pointer p is stored in a tuple of the form $((a, x, n), o_1, \ldots, o_l)$ where a is the array pointed to, x the index of a specific element p points to, n the node the assignment to p occurred in, o_k the movement of the pointer p in a loop k and l the number of enclosing loops in the function to be analysed.

After initialisation the elements a,x,n and o_k carry the default value $\perp$ indicating that no specific information is available. More precisely, $\perp$ denotes the state that the pointer has not been assigned any array element. Conversely, the value $\top$ is used to express the conflict between different assignments. For example, consider the situation in figure 8. Here, in a node of joining control paths the conflict between p pointing to a along the one incoming path and p pointing to b along another path is resolved by setting the corresponding element to $\top$. From this it becomes clear that p has been assigned at least along one control path, but the specific mapping

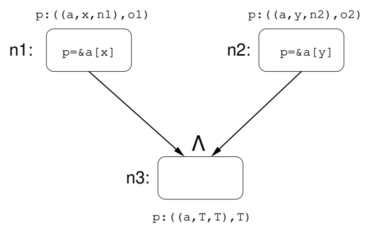

Figure 8: Joining control flow and the meet operator

of p is dependent on the actual program execution path. (denoted by $\top$ in the corresponding field).

The number of increments of a pointer must be *equal* across all control paths reaching a particular statement in order that a pointer reference may be replaced by an array reference. If the number of increments is different on two joining paths, again this is denoted by $\top$ in the appropriate field. Given this requirement, it is now possible to define the meet operator, which combines the outcome of following one of two distinct control-flow paths at the point at which they meet e.g. immediately after an if-statement. The meet operator $\bigwedge$ is formally defined as follows

$$\bigwedge(((a_1,x_1,n_1),o_{1,1},\ldots,o_{1,l}),((a_2,x_2,n_2),o_{2,1},\ldots,o_{2,l}))$$
$$= \begin{cases} ((a_1,x_1,n_1),o_{1,1},\ldots,o_{1,l}) \\ \quad \text{if } ((a_1,x_1,n_1),o_{1,1},\ldots,o_{1,l})=((a_2,x_2,n_2),o_{2,1},\ldots,o_{2,l}) \\ ((a_1,x_1,n_1),o_{1,1},\ldots,\top,\ldots,o_{1,l}) \\ \quad \text{if } a_1=a_2, x_1=x_2, n_1=n_2, o_{1,k}\neq o_{2,k} \\ ((a_1,\top,n_1),\top,\ldots,\top) \\ \quad \text{if } a_1=a_2, n_1=n_2, x_1\neq x_2 \\ ((\top,\top,\top),\top) \\ \quad \text{otherwise} \end{cases}$$

This operator is used in section 4.4. to determine the effects of pointer usage throughout the program.

4.3 The Flow Functions

The flow functions model the effect of executing statement on the pointer-to-array mapping reaching this statement node. The flow functions have the form $f : L^m \rightarrow L^m$ with L the dataflow information $((a,x,n),o_1,\ldots,o_l)$ and m the total number of pointers declared in the C function being processed. For each node n a flow function f_n is defined. This flow function can be split up into individual functions according to the operation contained in node n. The individual functions are either a Generate, Preserve or Exit Functions.

$$f_n(t_1,\ldots,t_m) = (f_n^1(t_1),\ldots,f_n^m(t_m))$$
$$\text{where } t_i = ((a_i,x_i,n_i),o_{i,1},\ldots,o_{i,l})$$

4.3.1 Generate Function

Generate functions cover assignments to pointers. An incoming pointer-to-array mapping is generated or updated for any pointer p. The general form of a Generate function is this:

$$f_n^p(((a_p,x_p,n_p),o_{p,1},\ldots,o_{p,l})) =$$
$$\begin{cases} ((v,0,n),o_{p,1},\ldots,o_{p,l}) & \text{if } \texttt{p = \&v} \text{ for scalar v} \\ ((a,0,n),o_{p,1},\ldots,o_{p,l}) & \text{if } \texttt{p = a} \text{ for an array a} \\ ((a,x,n),o_{p,1},\ldots,o_{p,l}) & \text{if } \texttt{p = \&a[x]} \text{ for an array a} \end{cases}$$

4.3.2 Preserve Function

Preserve functions handle pointer arithmetic, i.e. pointer movements relative to its current position. The pointer-to-array mapping is updated according to the operation of node n.

$$f_n^p(((a_p,x_p,n_p),o_{p,1},\ldots,o_{p,k},\ldots,o_{p,l}))$$
$$= \begin{cases} ((a_p,x_p,n_p),o_{p,1},\ldots,o_{p,k}+1,\ldots,o_{p,l}) \\ \quad \text{if } \texttt{p++} \text{ and } k = \mathit{inner}(n) \\ ((a_p,x_p,n_p),o_{p,1},\ldots,o_{p,k}-1,\ldots,o_{p,l}) \\ \quad \text{if } \texttt{p--} \text{ and } k = \mathit{inner}(n) \\ ((a_p,x_p,n_p),o_{p,1},\ldots,o_{p,k}+c,\ldots,o_{p,l}) \\ \quad \text{if } \texttt{p += c} \text{ and } k = \mathit{inner}(n) \\ ((a_p,x_p,n_p),o_{p,1},\ldots,o_{p,k}-c,\ldots,o_{p,l}) \\ \quad \text{if } \texttt{p -= c} \text{ and } k = \mathit{inner}(n) \\ ((a_q,x_q,n_q),o_{q,1},\ldots,o_{q,k},\ldots,o_{q,l}) \\ \quad \text{if } \texttt{p = q} \text{ and q a pointer} \\ \quad \vdots \\ ((a_p,x_p,n_p),o_{p,1},\ldots,o_{p,k},\ldots,o_{p,l}) \\ \quad \text{otherwise} \end{cases}$$

4.3.3 Exit Function

The Exit functions represent the transition from statement to loop level. Since *Exit* nodes occur at the end of a loop body, Exit functions have the task of summarising the effects of the single statements in the loop body. This is done in such a way that the overall effect of the loop on the pointer mapping is returned as the result.

$$f_n^p(((a_p,x_p,n_p),o_{p,1},\ldots,o_{p,k},\ldots,o_{p,l}))$$
$$= \begin{cases} ((a_p,x_p,n_p),o_{p,1},\ldots,\mathit{range}(\mathit{inner}(n))\times o_{p,l}) \\ \quad \text{if } \mathit{inner}(n)\neq \mathit{inner}(n_p) \text{ and } \mathit{inner}(n)=l \\ ((a_p,x_p,n_p),o_{p,1},\ldots,\sum_{i_k=0}^{f_k(i_0,\ldots,i_k-1)} o_{p,k+1},\ldots,o_{p,l}) \\ \quad \text{if } n=\perp \text{ or } (k=\mathit{inner}(n))\subseteq \mathit{inner}(n_p) \\ ((a_p,x_p,l_p),o_{p,1},\ldots,o_{p,k+1},\ldots,o_{p,l}) \\ \quad \text{if } (k=\mathit{inner}(n))\not\subseteq \mathit{inner}(n_p) \end{cases}$$

To compute the total effect of a statement performing pointer arithmetic in a loop nest, the total number of iterations of of each individual loop must be known. Given a generic loop nest in figure 9, the total number of iterations of the innermost loop is

$$\sum_{i_0=0}^{N}\sum_{i_1=0}^{f_1(i_0)}\sum_{i_2=0}^{f_2(i_0,i_1)}\cdots\sum_{i_n=0}^{f_n(i_0,\ldots,i_{n-1})} 1 \qquad (1)$$

where $f_k(i_0,\ldots i_{k-1})$ is the number of iterations of the kth nested loop, $1 \leq k \leq n$. In the same way, the total number of iterations of an inner loop in level k can be expressed as

$$X(k) = \sum_{i_0=0}^{N}\sum_{i_1=0}^{f_1(i_0)}\sum_{i_2=0}^{f_2(i_0,i_1)}\cdots\sum_{i_k=0}^{f_k(i_0,\ldots,i_{k-1})} 1 \qquad (2)$$

```
for(i_0 = 0;  i_0 < N;  i_0 ++)
{
   \* pointer incremented by m_{0,pre} *\
   for(i_1 = 0;  i_1 < f_1(i_0);  i_1 ++)
   {
      \* pointer incremented by m_{1,pre} *\
      ⋮
         for(i_n = 0;  i_n < f_n(i_0,...,i_{n-1};  i_n ++)
         {
            \* pointer incremented by m_n *\
         }
      \* pointer incremented by m_{1,post} *\
   }
   \* pointer incremented by m_{0,post} *\
}
```

Figure 9: Loop nest with pointer increments

Such terms are frequently encountered in analysing loop nests and can be enumerated using the Omega Calculator [26] or methods based on Ehrhart polynomials [10].

Given equations (1) and (2), the total effect of the pointer increments of a pointer p in the entire loop nest can be derived as follows:

$$\begin{gathered} X(n) \times m_n + \\ X(n-1) \times (m_{m-1,pre} + m_{n-1,post}) + \\ \cdots \\ X(0) \times (m_{0,pre} + m_{0,post}) = \\ \sum_{k=0}^{n-1} (X(k) \times m_{k,pre} + m_{k,post}) + X(n) \times m_n \end{gathered}$$

where $m_{k,pre}$,$m_{k,post}$ is the size of the pointer increment in loop k before entering/after exiting the $k+1$th loop as shown in figure 9.

For rectangular iteration spaces and constant numbers of pointer increments on each loop level, the total number of pointer increments evaluates to affine expressions dependent on the iteration variables. Otherwise, if all upper loop bounds are affine expressions in the outer index variables, the resulting expressions are polynomial functions of the index variables.

Summaries of the effects of inner loops are constructed starting with the innermost loop level. Once the total effect of an inner loop has been determined, the algorithm progresses with the statements after this loop. Each time an end of a loop is seen, its summary is computed. In a non-perfect loop nest, other statements than ends of loops can occur. These are treated as usual until the next end of a loop terminates the next higher loop level which in turn initiates the summary computation for this loop.

4.4 Equation System

The flow functions specify the effects of a statement locally, i.e. on a per node or statement basis. All flow function together form an equation system whose solution is the global dataflow solution. This solution has its representation in the $IN[n]$ and $OUT[n]$ sets associated with each node.

Solving the dataflow equation starts with all pointers being not initialised, i.e. $\forall p : ((a_p, x_p, n_p), o_{p,1}, \ldots, o_{p,l}) = ((\bot,\bot,\bot),\bot)$, resulting in $IN[n] = (((\bot,\bot,\bot),\bot), \ldots, ((\bot,\bot,\bot),\bot))$.

The equation system is solved by visiting all nodes in reverse postorder beginning at the start node n_o and computing the $IN[n]$ and $OUT[n]$ sets as follows:

$$\begin{aligned} OUT[n] &= f_n(IN[n]) \\ IN[n] &= \bigwedge_{m \in pred(n)} OUT[m] \end{aligned}$$

Only a single iteration is required to propagate the pointer mapping values. Thus, its runtime complexity is $O(N)$ with N the number of nodes in the CFG. Unlike iterative dataflow analysis this algorithm does not compute any fix point. The transition from statement to loop level is done at the end of the loop body when the flow function of the loop node is computed.

4.5 Pointer Conversion

In a second pass over the CFG, step 2b in the algorithm, pointer accesses and arithmetic are replaced and removed, respectively. Replacement is based on the dataflow information gathered during the analysis stage. From this information, the index expressions of the array accesses are constructed.

A pointer reference can only be replaced if its array a and offset o components of the tuple $((a, x, l), o_1, \ldots, o_l)$ are unambiguous. This is the case when $a, o \notin \{\bot, \top\}$.

The information required for the conversion of pointer access into an array access is spread throughout the computed dataflow values. The necessary components for the construction of the array access are:

1. array name (available at the node containing the pointer access),
2. initial offset (available at the node containing the pointer access),
3. local offset (available at the node containing the pointer access),
4. coefficients of the index variables (available at the loop exit nodes),
5. global offset (available at the exit node).

The *a*rray name is the name of the array the pointer to be replaced is referring to. The *i*nitial offset is a constant term and specifying the start position of the pointer in an array as given in an initial assignment. *L*ocal and *g*lobal offsets specify the distances a pointer has been moved in the current and outer loop bodies, respectively, since the start of the iteration. Finally, the *c*oefficients of the index variables are multiplicative factors specifying the effect of all previous iterations on the current pointer position.

More precisely, the conversion of a pointer-based memory access in a node n via pointer p requires the information kept in the IN sets of node n and all the summarising exit nodes of its embracing outer loops. Let the IN set for node n and pointer p be $IN_n^p = ((a, x, l), o_1, \ldots, o_l))$. The explicit array access replacing the pointer expression of the original program will have a form $a[exp]$ where a is the name of an array and exp is an expression for the selection of a single array element. The index expression exp is the sum of several components which are only partly available from node n. In general, the expression exp has following form $exp = exp_{initial} + exp_{local} + exp_{coeff} + exp_{global}$.

Both the initial offset $exp_{initial}$ and the local offset exp_{local} can be determined from IN_n^p as $exp_{initial} = x$ and $exp_{local} = o_k$ where $k = inner(n)$.

The global offset exp_{global} and the expression containing the loop index variables exp_{coeff} are constructed from the data at the

Node	Original program		**p_c:((a,x,n),k,i,f**	p_c → c
	int A[N],B[N],C[N];			
	int *p_a, *p_b, *p_c;	→	$((\perp,\perp,\perp),\perp,\perp,\perp)$	
1	p_c = &C[0]	In	$((\perp,\perp,\perp),\perp,\perp,\perp)$	
		Out	$((C,0,1),\perp,\perp,\perp)$	
2	for(k=0; k<Z; k++)	In	$((C,0,1),\perp,\perp,\perp)$	
		Out	$((C,0,1),0,\perp,\perp)$	
	{			
3	p_a = &A[0];	In	$((C,0,1),0,\perp,\perp)$	
		Out	$((C,0,1),0,\perp,\perp)$	
4	for(i=0; i<X; i++)	In	$((C,0,1),0,\perp,\perp)$	
		Out	$((C,0,1),0,X,\perp)$	
	{			
5	p_b = &B[k*X];	In	$((C,0,1),0,0,\perp)$	
		Out	$((C,0,1),0,0,\perp)$	
6	*p_c = 0;	In	$((C,0,1),0,0,\perp)$	
		Out	$((C,0,1),0,0,\perp)$	
7	for(f=0; f<Y; f++)	In	$((C,0,1),0,0,\perp)$	
		Out	$((C,0,1),0,0,0)$	
	{			
8	*p_c += *p_a++ * *p_b++;	In	$((\underline{C,0},1),0,0,\underline{0})$	C, $+0$ (i.o.), $+0$ (l.o.)
		Out	$((\overline{C,0},1),0,0,0)$	
9	} /* Exit f loop */	In	$((C,0,1),\underline{0,0,0})$	$+0\times f$ (cf.), $+0$ (g.o.)
		Out	$((C,0,1),\overline{0,0,0})$	
10	p_c++	In	$((C,0,1),0,0,0)$	
		Out	$((C,0,1),0,1,0)$	
11	} /* Exit i loop */	In	$((C,0,1),0,1,\underline{0})$	$+1\times i$ (cf.)
		Out	$((C,0,1),0,X,0)$	
12	} /* Exit k loop */	In	$((C,0,1),\underline{0,X},0)$	$+X\times k$ (cf.)
		Out	$((C,0,1),\overline{X\times Z},X,0)$	

Figure 10: Dataflow solution for `matrix1`

exit nodes at the same and all higher loop levels, respectively, as node n. For the global offset exp_{global} the IN_k^p set of the exit node of the loop at level $k = inner(n)$ is inspected. By assumption there are no assignments to pointer p in any node of the loop prior to n. Therefore, exp_{global} can be expressed as the sum of all o_i with $i < k$, i.e. the pointer has been moved by as many steps as the sum of the steps in any of the outer loops seen so far.

Finally, the term exp_{coeff} is a sum of products of loop index variables and their coefficients. The general form of it is $exp_{coeff} = \sum_{j=1}^{inner(n)} coeff_j \times j_i$ where the $coeff_j$ are the coefficients and the j_i are the loop index variables of the loops at level i. During the computation of the coefficients $coeff_j$ two cases must be distinguished: a) Loop nests without pointer assignments in it, and b) loop nests containing pointer assignments.

Case a). If a pointer is not assigned a new value in any level of the loop and the pointer progresses at least by one step in the loop, the expression exp_{coeff} will contain coefficients $\neq 0$ for all levels j, $j \geq inner(n)$. The coefficient for level $k = inner(n)$ is the offset o_k, i.e. the number of positions a pointer has been moved in the loop body, multiplied by the number of iterations of that loop. For all outer loops, the coefficients are computed as the sum of number of pointer movements o_j on the current level and the total effect of the inner loops o_{j+1} and multiplied by the current loop range.

For case b), i.e. loops containing an assignment to a pointer p, the coefficients for all the levels up to the one with the assignment are 0.

For all pointers successfully converted into explicit array accesses throughout an entire loop or loop nest, expressions containing arithmetic can be removed. In this step only the expressions modifying a pointer are discarded, the actual pointer accesses have already been converted in the previous step of the conversion. If there are uses of the pointer following transformed loop which cannot be eliminated by the pointer conversion algorithm, an additional statement with an update of the pointer taking into account the total effect of the loop on the pointer must be inserted immediately after the loop.

An example in the next section illustrates how to interpret the dataflow information and to convert pointer accesses into array accesses.

4.6 Example

Figure 10 shows an example illustrating how the dataflow based array recovery algorithm works. The solution of the dataflow problem for `p_c` in the `matrix1` program is shown.

For each node of the program two dataflow values representing the IN and OUT set are kept. These dataflow values are computed during the analysis stage. The last column of figure 10 shows the subexpressions constructed during the conversion stage of the algorithm. Putting together the subexpressions, described in the previous section, enables the pointer based access `*p_c` to array C in node 8 to be replaced with the explicit memory access $C[k*X+i]$.

From the IN set of node 8 the name of the array (C), the initial offset (0) and the local offset (0) are collected. The global offset (0) can be read off from node 9, the exit node of the innermost loop embracing the node 8 which contains the pointer access being converted. The coefficients to the loop index variables f, i and k can be extracted from the IN sets of the exit nodes (nodes 9, 11 and 12) of the corresponding loops. Since there is no pointer movement in the f loop, the coefficient of f is 0. The single step in the i loop is responsible for the $1 \times i$ contribution. Finally, the coefficient of the index variable k of the outermost loop is X since the pointer `*p_c` is moved by X positions per iteration of the k loop. This information is gathered from node 12. Now the entire explicit array access can be constructed from the array name and index expression that results from the sum of the different contributions of the subexpressions.

In a similar way the assignment `*p_c = 0` can be replaced. After the replacement of all accesses via the pointer `p_c`, the statements containing pointer arithmetic on `p_c` can be deleted. Finally, after ensuring that no further use of p_c remains, the declaration of the pointer can be discarded. p_a and p_b are treated in a similar manner. The final program is shown in figure 2.

5. EXPERIMENTAL FRAMEWORK

This paper is concerned with the use of pointer to array conversion in enabling performance enhancing transformations. In section 6 we focus on the resulting execution time behaviour, while here we briefly describe the salient points of the processors, benchmarks and transformations considered.

5.1 Processors

5.1.1 Analog Devices ADSP-21160

The Analog Devices ADSP-21160 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 is 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 run-time overhead.

5.1.2 Philips TriMedia TM-1

The TriMedia TM-1 is a 32-bit floating point VLIW Multimedia processor [25]. 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 between 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.

5.1.3 Texas Instruments TMS320C6201

In contrast to the other two processors, the TMS320C6201 is a 32-bit fixed-point processor [33]. 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.

5.2 Benchmarks

The benchmarks selected are from the well-known DSPstone [36] suite. 11 of the benchmarks containing iterative program structures were selected. Each benchmark is a kernel found in typical DSP applications and makes extensive use of pointer arithmetic.

5.3 Transformations

Converting a pointer based program to one based on arrays enables a number of well studied program transformations to be considered. 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 [5]. Initially, the impact of *Pointer to array conversion* was evaluated in isolation. Next, *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. This was applied both before and after array recovery. The remaining transformations relied on pointer to array conversion. *Delinearisation, tiling and padding* allow cache to be optimised (TM-1), *vectorisation* allows the exploitation of SIMD parallelism on the SHARC and finally *scalar replacement* reduces the number of accesses to memory (SHARC, TM-1, C6201).

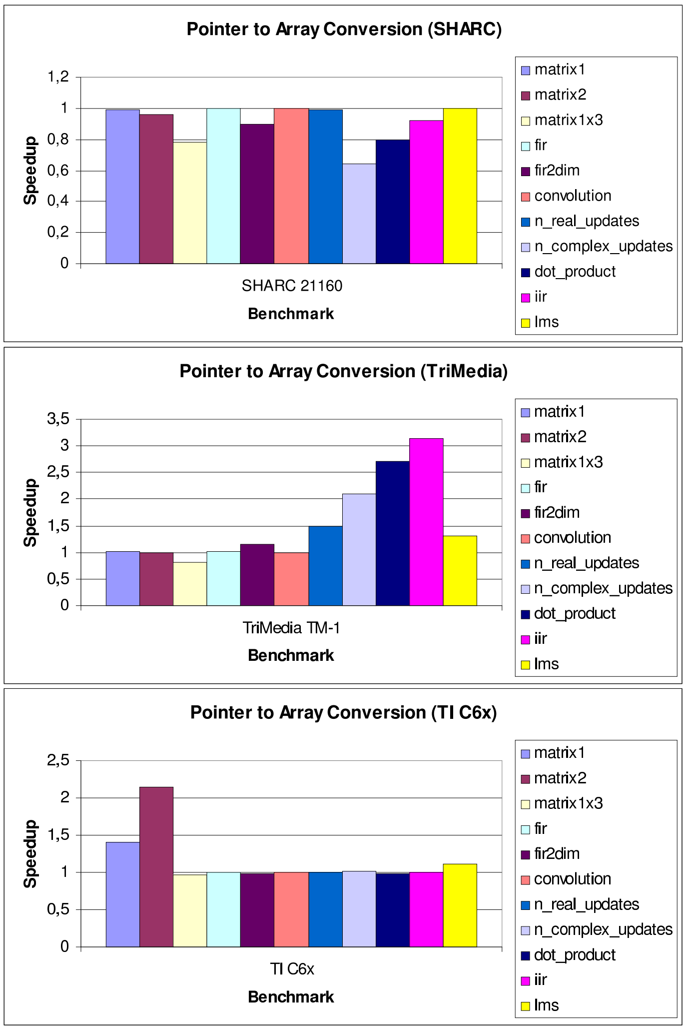

Figure 11: Pointer to Array Conversion

The pointer to array conversion algorithm developed in this paper was implemented in our experimental Octave compiler. All other transformations were applied manually.

After applying transformation on a source-to-source level, the resulting code was 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 the manufacturers' simulators.

6. RESULTS

Since we are interested in high-performance signal processing the emphasis of our benchmarking is execution time speedup given a fixed memory size. Code size and power consumption are impor-

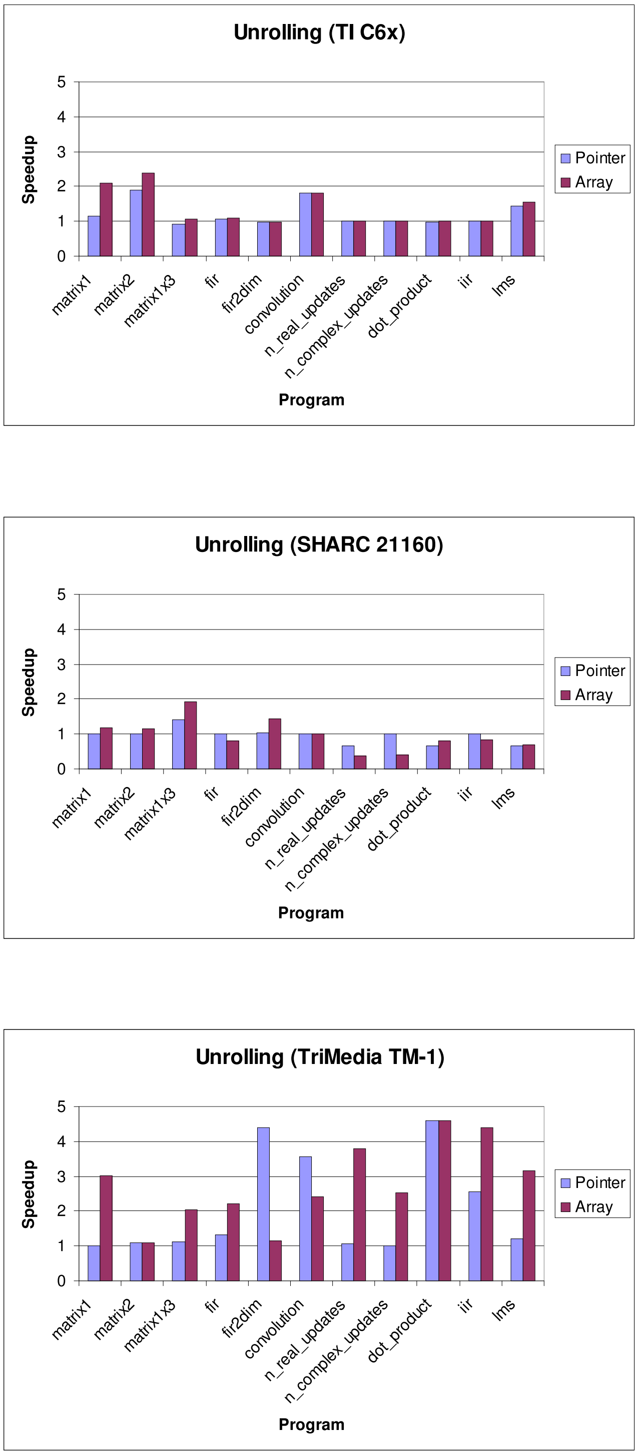

Figure 12: Unroll of pointer and array recovered benchmarks

tant constraints but are beyond the scope of this paper.

6.1 Transformation-oriented Evaluation

In this section we examine the effect of each transformation in isolation on the benchmarks' behaviour shown in figures 11 to 16.

Pointer to array conversion. Converting pointers to arrays based on the algorithm described earlier in this paper enables high level transformations and can often help the native compiler to perform more accurate analysis such as dependence testing. It was successfully applied to all of the benchmarks. and its performance impact varies from program to program and across the three platforms. The TriMedia often benefits from the transformation (see fig. 11), while the SHARC consistently performs worse. It has a variable but usually beneficial impact on the C6201. In the case of `n_complex_updates` on the TriMedia, we have a 2.18 speedup, while there is a slowdown of 0.63 for the same program on the SHARC.

Pointer to array conversion on its own therefore has a mixed impact. On examining the generated assembler, the main benefit for the the TriMedia is in improved data dependence analysis in nested loops. In the case of the C60, however, the generated code is frequently identical, perhaps due to the greater maturity of the the native compiler. In the case of the SHARC, the generated code is similar but the the AGU is not efficiently exploited. Its main benefit, however, is that it enables further transformations discussed below. Apart form unrolling, none of the remaining transformations can be applied without the use of pointer to array conversion. Furthermore, the largely negative impact of pointer to array conversion for the SHARC is offset by SIMD vectorisation which relies on an array based form of the program.

Although our focus is on performance rather than code size, it is worth noting that the effect of pointer to array conversion varies across the benchmarks and platforms giving in some cases upto a 10 % reduction in object code size and a 34% increase in one case.

Unrolling. Unrolling was applied to two versions of each program; with and without pointer to array recovery. Different unroll factors upto a maximum of 20 were evaluated and the best result shown in figure 12. Unrolling can increase ILP for the 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 i.e `mat1x3` but unrolling has 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`.

To further understand the negative impact of unrolling on the SHARC, consider the case of the `n_real_updates` benchmark, the main loop body after array recovery takes just six instructions and utilises the ADSP-21160's capabilities to execute an arithmetic operation in parallel with a memory operation. The unrolled version, however, 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 array recovery only. Thus loop unrolling even with small unroll factors is not always beneficial, especially on the ADSP-21160.

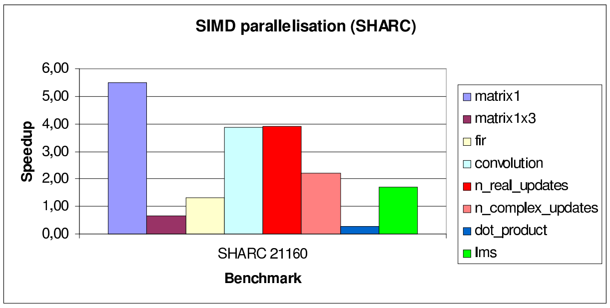

Figure 13: SIMD

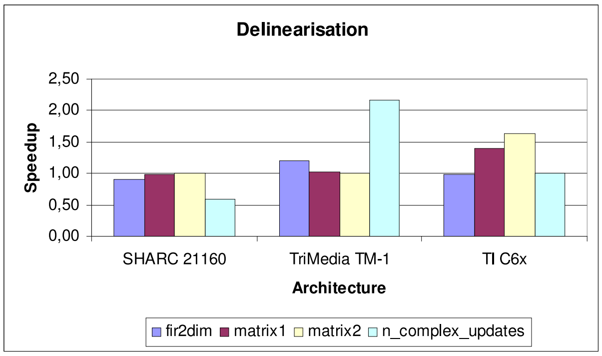

Figure 14: Delinearisation

On the TriMedia and the C6201, however, unrolling 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. Unrolling increases code size and is limited by the amount of instruction memory.

SIMD vectorisation. This transformation 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 greater than the work available. Both `dot_product` and `matrix1x3` have small trip counts and the cost of switching to SIMD mode outweighs the benefit of vectorisation. Array recovery is necessary for this transformation even though, on its own, array recovery decreases performance on the 21160, shown in figure 11. In one case, SIMD vectorisation gives a speedup up of 5.48, due to parallel use of both vector units, use of both buses and improved code generation as a side effect of the pointer to array conversion.

Delinearisation. This is applicable to just four of the benchmarks shown in figure 14, those containing linearised two dimension array traversals. It aids dependence analysis, especially for the Trimedia, and allows further loop and data transformations. Overall it is generally beneficial for the TriMedia and C6201, but costly for the SHARC. The negative impact on the SHARC's performance is due to slightly more complex code generated for the two dimensional array accesses preventing AGU exploitation.

		fir2	mat1	mat2
Pad	TriMedia	1.20	1.11	1.08
Tile	TriMedia	1.21	1.00	1.00
	C6201	0.99	2.11	0.68

Figure 15: Speedup due to Delinearisation, Padding and Tiling

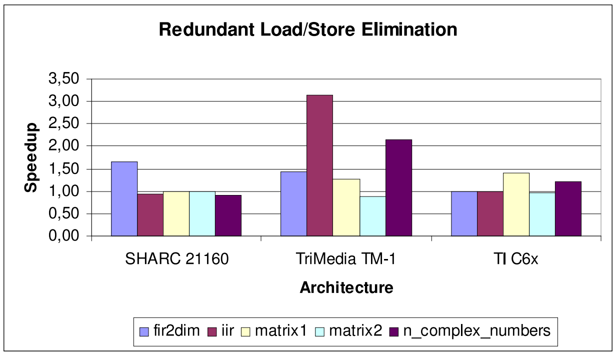

Figure 16: Scalar Replacement

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 (see fig. 15). Although beneficial in all cases, it is applicable to only three programs on one architecture and hence may be of limited general use.

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 (see fig. 15). In the case of `matrix2`, the slowdown is due to the overhead of additional loops which is not offset by increased locality.

Scalar Replacement. In fig. 16, it generally improves the performance on the TriMedia for those benchmarks where it is applicable, 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 in more detail.

6.2 Benchmark-oriented Evaluation

In figure 17, the results for the selected set of benchmarks are summarised. For each benchmark program and each architecture

Benchmark	Architecture SHARC 21160	TriMedia TM-1	TI C6201B
matrix1	Array, Delin.,SIMD	Array, Delin, Scalar, Pad (1), Unroll (9)	Array, Unroll (3)
matrix2	(Original)	Array, Pad (1), Unroll (5), Scalar	Array, Unroll (3)
matrix1x3	Array, Unroll(total)	Array, Unroll (total)	Array Unroll (total)
fir	Array, SIMD (part.)	Array, Unroll (8)	Array, Unroll(15)
fir2dim	Array, Scalar	Array, Delin., Scalar, Unroll	Array, Scalar, Unroll, Pointer
convolution	Array + SIMD	Unroll (10)	Unroll (8)
n_real_updates	Array, SIMD	Array, Unroll (3)	Unroll (9)
n_complex_updates	Array, SIMD (part.)	Array, Delin., Scalar, Unroll (2)	Array, Delin., Scalar
dot_product	(Original)	Array, Unroll (total)	(Original)
iir_biquad_N_sections	Array, Delin., Scalar	Array, Unroll (5)	Original
lms	Array, SIMD (part.)	Array, Unroll (11)	Array, Unroll (7)

Figure 18: Best transformation sequence for the DSPstone benchmark suite

the maximum speedup achieved is shown. Figure 18 lists the transformations needed to obtain such speedups.

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 7.

matrix1 (mat1). The `matrix1` benchmark computes the product of two matrices. After pointer conversion 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 TriMedia benefits most from a delinearised version of the array-based program to which scalarisation, loop unrolling and padding have been applied. 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-1 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 array recovery and loop unrolling together. The execution speed can be more than doubled on the C6201 architecture.

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`. Still, pointer conversion and loop unrolling can be applied and yields a speedup of 1.16.

For VLIW architectures the situation is different. On the TriMedia a speedup of 3.52 can be achieved after array recovery, padding, scalarisation and 5-fold unrolling.

The TI DSP, unlike the SHARC, benefits from the differences of the `matrix2` implementation, after array recovery and 3-fold unrolling a speedup of 2.39 is possible. This speedup mainly results from array recovery with unrolling contributing only a small percentage. The inspection of the assembly code generated by the compiler reveals a more efficient inner loop due to a higher degree of instruction parallelism. After array recovery the number of operations in the loop is not only smaller, but the number of operations executed in parallel is higher. The explicit array accesses increased the efficiency of the data dependence analysis supporting the compiler built-in software pipelining transformation.

matrix1x3 (mat3). This program computes the matrix product of a 3×3 matrix and a 3×1 vector in a simple loop with few iterations. For the ADSP-21160 array recovery and total loop unrolling of the very small loop can speed up the execution by a factor of 1.93.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 an implementation of a Finite Response Filter and contains a single loop. After array recovery the loop is amendable to loop reversal and loop splitting which in turn allowing SIMD parallelisation of one of the resulting loops. Although just a single loop can 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. 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

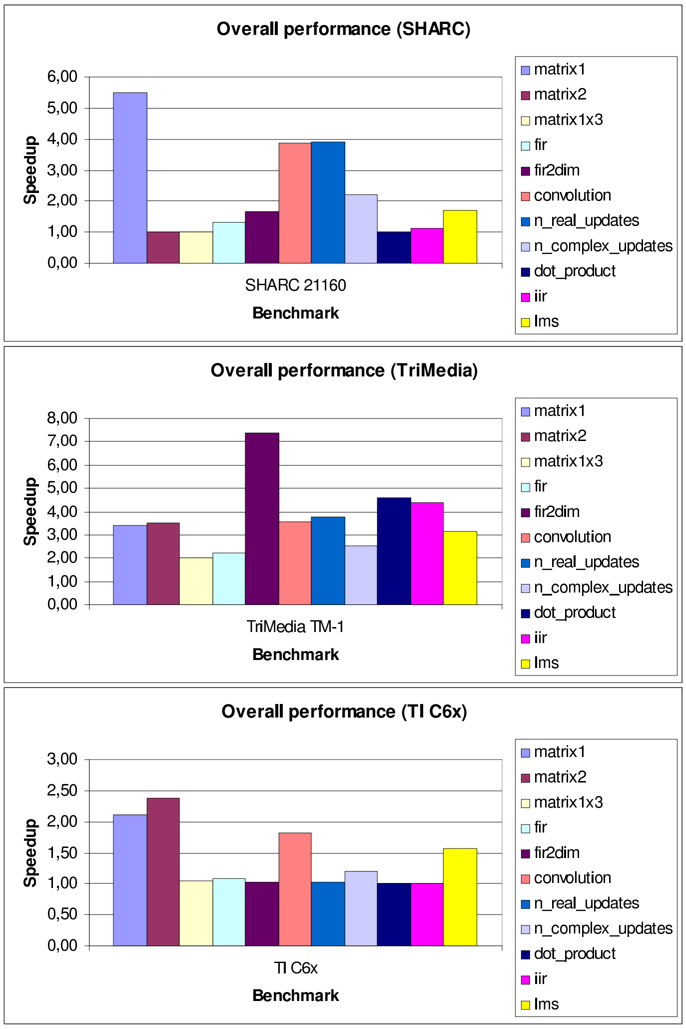

Figure 17: Best overall best performance

the `SIMD_loop` directive. In sequential execution mode, scalarisation can be applied after array recovery of the program and a speedup of 1.66 obtained. Unlike the other programs where scalarisation cannot improve the performance, `fir2dim` benefits from this transformation because it can be applied across the three inner loops where redundant memory accesses are removed. The compiler is not able to remove these accesses without this high-level code transformation. On the TriMedia a speedup of 7.38 is possible after array recovery, delinearisation, scalarisation and total unrolling of the three small inner loops. 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 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 small, this example shows that it is possible to apply 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 loop can easily be parallelised for the ADSP-21160 after array recovery 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 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 loop can be easily parallelised for the ADSP-21160 and a speedup of 3.91 is achieved. The TriMedia benefits most from pointer conversion and unrolling. Array recovery helps the compiler to prove independence of different memory accesses, whereas loop unrolling increases the number of instructions in the loop body. The maximum speedup achievable on the TriMedia is 3.78. However, the maximum speedup on the C6201 is rather small,1.02, and due to 9-fold loop unrolling of the pointer-based code.

n_complex_updates (comp). 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 array recovery, loop splitting and parallelisation of one of the resulting loops. For the TriMedia array recovery once again was proven to be useful supporting other transformations such as delinearisation and scalarisation. Together with 2-fold loop unrolling, a speedup of 2.52 was obtained. The same set of transformations but without unrolling also achieved the largest speedup on the C6201.

dot_product (dot). 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 and can take advantage of the removal of the loop construct during instruction scheduling. The original loop with just two iterations causes many NOP operations and branch penalties which can be eliminated by complete unrolling. 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 array recovery, delinearisation and scalarisation giving a speed up to a factor of 1.14. Array recovery and 5-fold unrolling gives the best performance with a speedup of 4.41 on the TriMedia. An inspection of the compiler generated assembly code shows a much tighter packing of operations into machine instructions, i.e. the number of wasted issue slots filled with NOPs is significantly reduced. 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 due to a good performance of the original code.

lms. This is the kernel of a least mean square filter. The `lms` program contains two loops that can both benefit from SIMD parallelisation. After array recovery, 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 array recovery and loop unrolling and speedups of 3.16 and 1.56, respectively, are possible.

6.3 Architecture-oriented Evaluation

Overall the TriMedia benefits the most from high level transformation with an average speedup of 3.69, the C60 the least with a speedup of 1.39 and the SHARC somewhere in between with an average speedup of 2.31.

TriMedia. In all but one case, the best optimisation for each program required pointer to array conversion. Though as was shown in figure 11, this alone can improve performance on the TriMedia, its major benefit was in enabling further transformations such as delinearisation, padding and scalar replacement. Although unrolling was useful in all cases, it required additional transformations in all bar one case to give the best performance. The TriMedia ,in general, benefitted the most from the application of combined transformation. This is mainly due to its more complex architecture, in particular the combination of a 5 way VLIW processor and an on board cache. Finally, the TriMedia probably benefited most from the transformations in part due to its relatively immature compiler.

TI C6201. The C6201 benefits least of all from the application of high level transformations. Yet even with a mature native compiler, it is possible to get on average a speedup of 1.39. In all but four cases pointer to array conversion was required to get the best performance, two of those cases being when no optimisation gave any improvement. Unrolling was once again useful in exposing ILP especially when combined with pointer to array conversion. Unlike the TriMedia, smaller transformation sequences seemed to perform best. The C60 rarely benefited from scalar replacement as the native compiler was largely capable of detecting redundant memory accesses in all but two cases.

SHARC. Unlike the two previous platforms, the SHARC gained the best speedup when its SIMD capabilities were exploited. Although pointer to array conversion always degraded performance in isolation, when combined with SIMD vectorisation in 6 cases it provided significant performance improvement. Overly restrictive requirements on the pragma prevented fuller exploitation of the SHARC SIMD capabilities.

Summary. Overall, selecting the appropriate high level transformation gave on average a 2.43 speedup across the the three platforms investigated. In 26 out of 33 cases, pointer to array conversion contributed to the increased performance and in only 3 cases was the best performance gained with transformations other then pointer to array conversion.

7. RELATED WORK AND DISCUSSION

There has been very little work in evaluating the impact of transformation sequences on real DSP processors. In [3] a heuristic optimisation method is presented that strongly relies on the information of how a certain compiler for a specific processor exploits high-level program constructs for the code generation. This approach is very restricted in the sense that it cannot easily transferred to a different processor or even a different compiler for the same processor. Although aimed at DSP applications the authors only present few results for a general purpose processor (Pentium).

An address optimisation based on a sequence of source-to-source transformations is shown and evaluated in [15]. This optimisation relies on explicit array accesses and does not work with pointer-based programs. Here our pointer-conversion algorithm can be applied as a preparatory stage that enables the further optimisation. Although aiming at DSP applications the experimental results come from general-purpose CPUs. It is not at all obvious if the transformation extends to DSPs as the authors claim, and a proof of this is still outstanding.

Software pipelining as a source-to-source transformation in the context of DSP applications is investigated in [35] and [30]. The combined effect of different optimisations is neglected apart from two normalisation transformations (renaming and loop distribution) needed by this approach.The evaluation of the effectiveness of the presented transformation is performed on a single architecture (Motorola DSP56300) where it achieved good results, albeit for a small set of benchmark programs.

The effect of Unroll-and-Jam and Scalar Replacement for imperfectly nested loops is evaluated in [29]. A simple heuristic method is used to determine the unroll factor and the results are compared with strip-mining, loop distribution and loop unrolling. The SC140 processor serves as the target architecture for the experimental evaluation. The results are promising, although the number of benchmark programs is very small and only a single architecture has been considered.

One of the main reasons that there has been little evaluation of transformation sequences is due to the pointer based nature of many of the benchmarks. In our previous work [12] we developed a basic pointer conversion algorithm. This however was restricted in that it was unable to fully handle nested loop constructs and functions. Recent work [34] has extended our previous work based on induction variable recognition. It can handle a much greater variety of recurrence relations amongst pointers and covers many difficult program types, fortunately rarely found in practice. It is, however, limited in its applicability to programs without functions. There is also little consideration of conflicting index functions in the presence of control-flow or the potential side effects of pointers to pointers.

This paper has shown that array recovery and 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 [17]. 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.

In the OCEANS project [6] high and low-level optimisations within an iterative compilation framework for embedded VLIW processors were investigated. Experimental results for the TriMedia TM-1000 show that such an approach has promise. Recently, we have further investigated the use of iterative compilation, where different optimisations are selected and evaluated, with the eventual best performing one selected [18]. 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 18 and is the subject of ongoing research.

8. CONCLUSION

The contribution of this paper has been to introduce a new technique for transforming C code with pointer-based array accesses into explicit array accesses to enable high level optimisations on DSP architectures. The approach has been implemented and integrated in our experimental compiler and tested on the DSPstone benchmark suite. This paper has also empirically demonstrated the usefulness of applying high level transformations to DSP applications across three different platforms. 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 to determine how to profitably extend our restricted conversion scheme where the potential scope for high level optimisations is 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.

9. 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] Andreyev, A.L., Balyakhov, D.I., and V.P. Russakov, The Technique of High-Level Optimization of DSP Algorithms Implementation, International Conference on Signal Processing Applications & Technology (ICSPAT), 1996.

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

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

[6] Barreteau, M. et al. OCEANS Optimising Compilers for Embedded Applications, *Proceedings Euro-Par 98*, LNCS 1470, pp. 1123-1130, 1998.

[7] 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.

[8] 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.

[9] Callahan, D., Cooper K.D., Kennedy, K., and L. Torczon, Interprocedural constant propagation, *SIGPLAN Symposium on Compiler Construction*, pp. 152-161, 1986.

[10] Clauss P., Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs, ACM ICS, ACM Press, 1996.

[11] 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.

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

[13] 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.

[14] Grove,D., and L. Torczon, Interprocedural Constant Propagation: A Study of Jump Function Implementations *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 90-99, 1993.

[15] Gupta, S., Miranda, M., Catthoor, F., and R. Gupta, Analysis of High-level Address Code Transformations for Programmable Processor s, *Proceedings of Design and Test in Europe Conference (DATE)*, 2000.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] Morgan, R., Building an Optimizing Compiler, Butterworth-Heinemann, Boston, 1998.

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

[24] 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.

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

[26] Pugh W., Counting Solutions to Presburger Formulas: How and Why, *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation* ACM Press, 1994.

[27] Sagiv, M., Reps, T., and S. Horwitz, Precise Interprocedural Dataflow Analysis with Applications to Constant Propag ation, TAPSOFT'95, Arhus, Denmark, LNCS, Springer-Verlag, 1995.

[28] 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.

[29] Song, Y. and Y. Lin, Unroll-and-Jam for Imperfectly-Nested Loops in DSP Applications, *Proceedings of the ACM International Conference on Compilers, Architectures, Synthesis for Embedded Systems*, November 2000.

[30] Su, B., Wang, J., and A. Esguerra, Source-Level Loop Optimization for DSP Code Generation, 1999 IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP '99), Vol. 4, pp. 2155-2158, Phoenix, USA, 1999.

[31] van Swwaaij, M.F.X.B., Franssen, F.H.M., Catthoor, F.V.M., and H.J. De Man, Automatimg High Level Control Flow Transformations for DSP Memory Management, *Proceedings of the IEEE workshop on VLSI signal processing*, Napa Valley, CA, October 1992. Also in VLSI Signal Processing V, (eds.: K. Yao, R. Jain, W. Przytula), IEEE Press, New York, pp. 397-406, 1992.

[32] 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.

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

[34] van Engelen R. and Gallivan K., An Efficient Algorithm for Pointer-to-Array Access Conversion for Compiling and Optimizing DSP Applications,*in International Workshop on Innovative Architecture (IWIA 2001)*, Maui, 2001.

[35] Wang, J., and B. Su, Software Pipelining of Nested Loops for Real-Time DSP Applications, 1998 IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP '98), Seattle, USA, 1998.

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