# Compiler Transformation of Pointers to Explicit Array Accesses in DSP Applications

Björn Franke and Michael O'Boyle

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

**Abstract.** Efficient implementation of DSP applications are critical for embedded systems. However, current applications written in C, make extensive use of pointer arithmetic making compiler analysis and optimisation difficult. This paper presents a method for conversion of a restricted class of pointer-based memory accesses typically found in DSP codes into array accesses with explicit index functions. C programs with pointer accesses to array elements, data independent pointer arithmetic and structured loops can be converted into semantically equivalent representations with explicit array accesses. This technique has been applied to several DSPstone benchmarks on two different processors. where initial results show that this technique can give on average a 11.95 % reduction in execution time.

## 1 Introduction

Embedded processors now account for the vast majority of shipped processors due to the exponential demand in commodity products ranging from cell-phones to power-control systems. Such processors are typically responsible for running digital signal processing (DSP) applications where performance is critical. This demand for performance has led to the development of specialised architectures 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, and optimising compilers. As in other areas of computing, programming in C is much less time consuming than hand-coded assembler but this comes at a price of a less efficient implementation due to the inability for current compiler technology to match hand-coded implementations.

To balance the requirement of absolute performance against program development time, there has been a move to tuning C programs at the source level. Although at one time, these tuned programs may have performed well with the contemporary compiler technology, such program tuning frequently makes matters *worse* for the current generation of optimising compilers. In particular, DSP applications make extensive use of pointer arithmetic as can be seen in the DSPstone Benchmarks [12]. Furthermore in [9] programmers are actively encouraged to use pointer based code in the mistaken belief that the compiler will generate better code. 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.

**Example 2.1** Original pointer-based array traversal

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

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

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

In the next section we provide a motivating example using a typical DSP program and how our technique may transform it into a more efficient form. This is followed in section 3 with a description of the basic algorithm for pointers within simple loop nests. Section 4 describes the general algorithm and is followed in section 5 by an evaluation of our technique on a set of benchmarks from the DSPstone suite across two platforms. Section 6 describes related work in this area and is followed in section 7 by some concluding remarks.

## 2 Motivation

Pointer accesses to array data frequently occurs in typical DSP programs. Many DSP architectures have specialised *Address Generation Units (AGUs)* [5] 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 Example 2.1, 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 semantic correctness compilers use conservative strategies, i.e. many possible array access opti-

**Example 2.2** After conversion to explicit array accesses

```
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];
  }
```

misations are not applied in the presence of pointers. Obviously, this limits the maximal performance of the produced code. It is highly desirable to overcome this drawback, without adversely affecting AGU utilisation.

Although general array access and pointer analysis are without further restrictions intractable [7], it is easier to find suitable restrictions of 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 [2, 5].

Example 2.2 shows the loop with explicit array indexes that is semantically equivalent to the previous loop in Example 2.1. Not only it is easier to read and understand for a human reader, but it is amendable to compiler array data flow analyses, e.g. [3]. The data flow information collected by these analyses can be used for redundant load/store eliminations, software-pipelining and loop parallelisation [13].

A further step towards regaining a high-level representation that can be analysed by existing formal methods is the application of de-linearisation methods. De-linearisation is the transformation of one-dimensional arrays and their accesses into other shapes, in particular, into multi-dimensional arrays [10]. The example 2.3 shows the example loop after application of clean-up conversion and de-linearisation. The arrays `A`, `B` and `C` are no longer linear arrays, but have been transformed into matrices. Such a representation enables more aggressive compiler optimisations such as data layout optimisations [10]. Later phases in the compiler can easily linearise the arrays for the automatic generation of efficient memory accesses.

## 3 Algorithm

Pointer clean-up conversion uses two stages during processing. In the first stage information on arrays and pointer initialisation, pointer increments and decrements as well as loop properties is collected. The second step then uses this information in order to

**Example 2.3** Loop after pointer clean-up conversion and delinearisation

```
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];
  }
```

replace pointer accesses by corresponding explicit array accesses and to remove pointer arithmetic completely.

### 3.1 Assumptions and Restrictions

The general problems of array dependence analysis and pointer analysis are intractable. After simplifying the problem by introducing certain restrictions, analysis might not only be possible but also efficient.

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

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

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

*Structured loops* are loops with a normalised iteration range going from the lower bound 0 to some constant upper bound $N$. The step is normalised to 1. Structured loop have the Single-Entry/Single-Exit property.

*Pointer assignments* apart from initialisations to some start element of the array to be traversed are not permitted. In particular, dynamic memory allocation and deallocation cannot be handled because of the potentially unbounded complexity of dynamic data structures. On the contrary, initialisations of pointers to array elements may be done repeatedly and even in dependence on some induction variable, i.e. within a loop construct. Example 3.1 shows a program fragment with initialisations of the pointers `ptr1` and `ptr2`. Whereas `ptr1` is statically initialised, `ptr2` is initialised repeatedly within a loop construct and with dependence on the outer iteration variable `i`.

**Example 3.1** Legal pointer initialisation / assignment

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

for(i = 0; i < N; i++)
{
   ptr2 = &array2[i];        /* OK */
   for(j = 0; j < M; j++)
      ...
}
```

In example 3.2 some illegal pointer assignments are shown. A block of memory is dynamically allocated and assigned to `ptr1`, whereas the initialisation of `ptr2` might dependent of some input data `b[i]` w which cannot be evaluated at compile time. The assignment to `ptr3` is illegal, if the pointer `ptrX` cannot be statically determined.

**Example 3.2** Illegal pointer initialisations

```
int *ptr1, *ptr2, *ptr3;

ptr1 = (int *) malloc(...);    /* Dynamic memory allocation */
ptr2 = &a[b[i]];               /* b[i] data dependent */
ptr3 = ptrX;                   /* illegal if ptrX unknown */
```

*Data dependent pointer arithmetic* is the change of a pointer itself (i.e. not the value pointed to) in a way that is dependent on the data processed and which might change from one program execution to the other. Because it is not known in advance which data will be processed by future program runs, the compiler cannot know at compile time which final values pointers will eventually have. Although there are some powerful methods available e.g. [11] for this problem of pointer or alias analysis, these kind of program constructs are not considered by the pointer clean-up conversion algorithm and are therefore not permitted. In general, all pointer expressions that can be evaluated statically can be handled.

In a similar way as data dependent pointer arithmetic, *function calls* might change pointers involved in the conversion. If there are functions that take pointers to pointers as arguments, the actual pointers passed to the function itself and not only their content can be changed. Hence, it must be ensured that no function calls of this type occur within the program fragment to be converted. `function1` in example 3.4 receives a pointer to the array, thus only the content of the array can be changed, but not the

**Example 3.3** Data dependent and independent pointer arithmetic

```
ptr++;        /* Data independent */
ptr += 4;     /* Data independent */

ptr += x;     /* Dependent on x */
ptr -= f(y); /* Dependent on f(y) */
```

pointer `ptr` itself. On the contrary, `function2` can change the pointer `ptr` which is not permissible[1].

**Example 3.4** Function calls changing pointer arguments

```
ptr = &array[0];

function1(ptr);     /* OK */

function2(&ptr);    /* not OK */
```

Finally, the number of increments of a pointer must be *equal in all branches* of conditional statements. The compiler cannot determine during compile time which branch will actually be taken during run-time, so no information on the total number of pointer increments after leaving the condition statement is available. Situations with unequal number of pointer increments are extremely rare and typically not found in DSP code. The first if-statement in example 3.5 is legal, because `ptr1` is treated equally is both branches. In the second if-statement the increments are different, therefore this construct cannot be handled.

Note however that overlapping arrays and accesses to single arrays via several different pointers are perfectly acceptable. Because the conversion does not require information on such situations, but only performs a transformation of memory access representation these kind of constructs that often prevent standard program analysis do not interfere with the conversion algorithm.

## 3.2 Overall Algorithm

During data acquisition in the first stage the algorithm traverses the *Control Flow Graph (CFG)* of a function and collects information when a pointer is given its initial reference to an array element and keeps track of all subsequent changes. Note that only changes of the pointer itself and not of the value of the object pointed to are traced. When loops are encountered, simple pointer changes within the loop body have multiplied effects

[1] This case could be handled, if some powerful inter-procedural pointer analysis [6] was used for the static determination of the pointer `ptr`.

**Example 3.5** Pointer increments in different branches

```
if (exp1)         /* Legal */
  x1 = ptr1++;
else
  y1 = ptr1++;

if (exp2)         /* Illegal */
  x2 = ptr2++;
else
  y2 = ptr2 + 2;
```

caused by repeated execution of the loop body. Therefore, information on loop bounds and induction variables is compulsory for the following reconstruction of array index functions. The program analysis step has similarities to abstract program interpretation and creates summary information of pointer-to-array-mappings at each node of the traversed CFG.

The main objective of the second phase is to replace pointer accesses to array elements by explicit array accesses with affine index functions. The mapping between pointers and arrays can be extracted from information gathered from pointer initialisation in the first phase. Array index functions outside loops are constant, whereas inside loops they are dependent on the loop induction variables of the corresponding loops. In order to determine the coefficients of the index functions, information on pointer changes based on pointer arithmetic collected during the first stage is used. Finally pointer-based array references are replaced by semantically equivalent explicit accesses, whereas expressions only serving the purpose of modifying pointers are deleted.

The pointer conversion algorithm can be applied on whole functions and can handle one- and multi-dimensional arrays, general loop nests of structured loops and several consecutive loops with code in between. It is therefore not restricted to handling single loops. Loop bodies can contain conditional control flow.

The presented algorithm is suitable for handling functions with simple loops, one-dimensional arrays and conditional branches. In the interest of simplicity of presentation the algorithm does not cover enhanced features like loops nests.

The algorithm 1 keeps a list of nodes to visit. This list of nodes is obtained by traversing the control flow graph which is supplied as a parameter to the algorithm in pre-order. As long as there are nodes in this list, the next one will be taken and processed. Depending on the type of statement, different actions will be started. Pointer initialisations and assignments result in updates of the pointer-to-array mapping, whereas loops are handled by a separate procedure. Pointer arithmetic expressions are statically evaluated and pointer-based array references are replaced by equivalent explicit array references. The mapping between pointers contains the not only the pointer and the corresponding array, but also the initial offset of the pointer within the array and some local offset for keeping track of pointer increments in loop bodies.

The procedure for handling loops is part of the algorithm 1. Similar to the basic algorithm the loop handling procedure proceeds a pre-order traversal of the nodes of

**Algorithm 1** Pointer clean-up conversion for CFG $G$

**Procedure** clean-up(CFG G)

```
map ← ∅
L ← preorderList(G);
while L not empty do
   stmt ← head(L);
   removeHead(L);
   if stmt is pointer assignment statement then
      if (pointer,array,*,*) ∈ map then
         map ← map - (pointer,array,*,*)
      end if
      map ← map ∪ (pointer,array,offset,0)
   else if stmt contains pointer reference then
      Look up (pointer,array,offset,*) ∈ map
      if stmt contains pointer-based array access then
         replace pointer-based access by array[initial index+offset]
      else if stmt contains pointer arithmetic then
         map ← map - (pointer,array,offset,*)
         calculate new offset
         map ← map ∪ (pointer,array,new offset,0)
      end if
   else if stmt is for loop then
      processLoop(stmt,map)
   end if
end while
```

the loop body. Two passes over all nodes are made: The first pass counts the total offset within one loop iteration of pointers traversing arrays, the second pass then is the actual replacement phase. A final stage adjusts the pointer mapping to the situation after the end of the loop.

The algorithm passes every simple node once, and every node enclosed in a loop construct twice. Hence, the algorithm uses time $O(n)$ with n being the number of nodes of the CFG. The exact time depends on the number and the size of loops. Improvements in handling loops are possible, e.g. by storing the nodes with pointer accesses for further replacement rather than spending a second pass. However, the algorithm will still run in linear time. Space complexity is linearly dependent on the number of different pointers used for accessing array elements, because for every pointer there is a separate entry in the *map* data structure.

*Arrays passed as parameters of functions* If no legal pointer assignment can be found, but the pointer used to traverse an array is a formal parameter of the function to be processed, it can be used as the name of the array [4].

*Loop Nests* Perfect loop nests of structured loops are different from simple loops in the way that the effect of a pointer increment/decrement in the loop body not only

```
Procedure processLoop(statement stmt, mapping map)

  {count pointer increments in loop body and update map}
  L = preorderList(loopBody)
  while L not empty do
    stmt ← head(L);
    removeHead(L);
    if stmt contains array arithmetic then
      Update increment in (pointer,array,offset,increment)
    end if
  end while

  {replace pointer increments according to map}
  L = preorderList(loopBody)
  while L not empty do
    if stmt contains pointer reference then
      Look up (pointer,array,offset,increment) ∈ map
      Look up (pointer,local offset) ∈ offsetMap
      if (pointer,local offset) ∉ offsetMap then
        offsetMap ← offsetMap ∪ (pointer,0)
      end if
      if stmt contains pointer-based array access then
        index ← increment × ind.var. + offset + local offset
        replace pointer-based access by array[index]
      else if stmt contains pointer arithmetic then
        Update local offset in map
      end if
    end if
  end while

  {adjust all mappings to situation after end of loop}
  for all (pointer,*,*,*) ∈ map do
    update offsets in map
  end for
```

multiplies by the iteration range of its immediately enclosing loop construct, but by the ranges of all outer loops. Therefore, all outer loops have to be considered when converting pointers of a perfect loop nest.

Handling perfect loop nests does not require extra passes over the loop. It is sufficient to detect perfect loop nest and descent to the inner loop body while keeping track of the outer loops. Once the actual loop body is reached conversion can be performed as usual, but with incorporating the additional outer loop variables and loop ranges as part of the index functions. Hence, asymptotical run-time complexity of the algorithm is not affected.

General loop nests can similarly be handled with a slightly extended version of the basic algorithm, by tracking which loop a pointer is dependent upon. The introductory example 2.1 illustrates a general loop nest and its conversion.

**Table 3.1** Absolute times for `biquad_N_sections` and the TriMedia

| Benchmark | Level | TriMedia | | Pen tium II | |
|---|---|---|---|---|---|
| | | Array | Pointer | Array | Pointer |
| biquad | -O0 | 11 | 9 | 980 | 1130 |
| | -O1 | 11 | 8 | 360 | 490 |
| | -O2 | 8 | 8 | 360 | 290 |
| | -O3 | 9 | 8 | 350 | 360 |

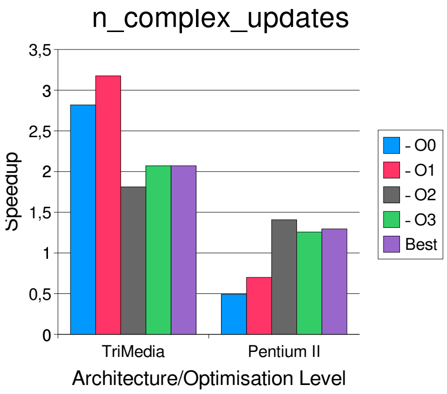


**Fig. 1.** Performance comparison of `n_complex_updates` benchmark

## 4 Experiments

The pointer clean-up conversion algorithm has been implemented as a prototype and integrated into the experimental Octave compiler. After the application of the transformation on source-to-source level the resulting code is then used as an input for the C compilers of the Philips TriMedia TM-1 and the Intel Pentium II (Linux 2.2.16, gcc version 2.95.2). Performance data was collected by executing the programs on a simulator (TM-1) and a real machine (PII).

In order to quantify the benefit of applying the pointer clean-up conversion to DSP applications, program execution times for some programs of the DSPstone benchmark suite were determined. The speedups of the programs with explicit array accesses with respect to the pointer-based versions for varying optimisation level were calculated. As a DSP application developer typically wants the best performance we determined the optimisation level that gave the best execution time for the pointer and array codes separately. Due to the complex interaction between compiler and architecture, the highest optimisation level does not necessarily give the best performance. Thus the ratio between the best pointer-based version and the best explicit array access version is also presented.

Table 3.1 shows an example of the absolute performance figures for the `biquad_N_sections` benchmark on the TriMedia and Pentium II architectures that were the used for the computations of the speedup. As stated above, the minimal execution times are not necessarily achieved with the highest available optimisation level. Thus, it is reasonable to give an additional speedup measure when comparing the best pointer-based and explicit array based versions. For example, the shortest execution time achieved on the Pentium II was 350 for the pointer-based version and 290 for the explicit array access based version. Hence, the best speedup is $\frac{350}{290} = 1.21$. The figures 1 to 6 visualise the speedup measures for all three architectures and all optimisation levels.

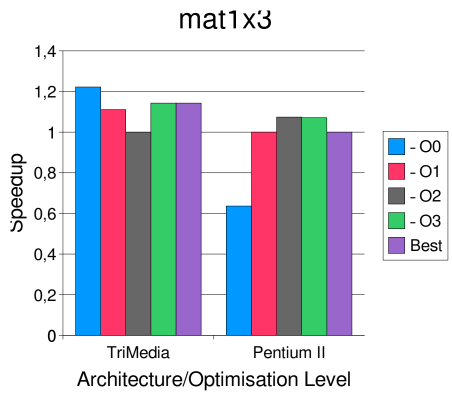


**Fig. 2.** Performance comparison of `mat1x3` benchmark

### 4.1 TriMedia

The performance of the TriMedia compiler/architecture combination usually benefit from using explicit array accesses rather than pointer-based array accesses. When there are no or only a few simple further optimisations applied (optimisation levels O 0 and O1), the program versions with explicit array accesses are significantly faster than the pointer-based programs. The difference can be up to 218% (`n_complex_updates`). For higher optimisation levels the differences become smaller. Although some of the original programs take less time than the transformed versions, in many cases there can be still some performance gain expected from pointer clean-up conversion. The pointer-based programs tend to be superior at optimisations level O2, but the si situation changes again for level O3 and the “best” case. This “best” case is of special interest since it compares the best performance of a pointer-based program to that of an explicit array access based program. The best explicit array based programs perform usually better or at least as good as the best pointer-based programs, only in one case (`dot_product`) a decrease in performance could be observed.

In figure 1 the speedup results for the `n_complex_updates` benchmark are shown. This rather more complex benchmark program shows the largest achieved speedups among all tests. The speedup reaches its maximum 3.18 at the optimisation level O1, and is still at 2.07 when comparing the best versions. This program contains a series of memory accesses that can be successfully analysed and optimised in the explicit array based version, but are not that easily amendable to simple pointer analyses. Although such large speedups are not common for all evaluated programs, it shows clearly the potential benefit of the pointer clean-up conversion.

Figure 2 compares the performances of the `mat1x3` benchmark . All speedups for the TriMedia architecture are $\geq$ 1. The achieved speedups are more typical for the set of test programs. An increase of 14% in performance can be observed for the best explicit array based program version. With only a few other optimisations enabled the performance benefit of the transformed formed program is even higher.

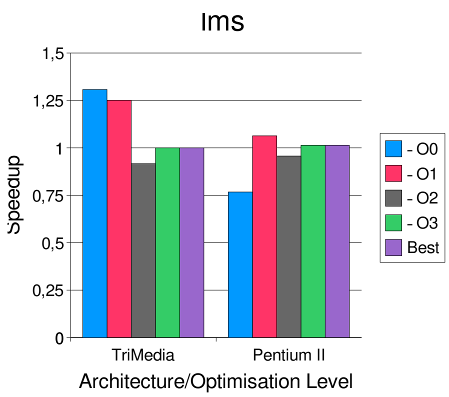


**Fig. 3.** Performance comparison of `lms` benchmark

The figures 3 and 4 represent the results of the `lms` and `matrix1` benchmark, respectively. As before, a significant speedup can be achieved at optimisation levels O0 and O1. The performances of the explicit array accesses based versions are actually below those of the pointer-based versions at level O2, but at the highest level the transformed programs perform better or as good as the original versions. Although there is only a small speed up observable in these examples, it shows that the use of explicit array accesses does not cause any run-time penalty over pointer-based array traversals, but still provide a representation that is better suitable for array data dependence analysis.

In general, with pointer clean-up conversion some substantial benefit can be achieved without the need for further optimisations. On higher optimisation levels the transformed programs still perform better or at least as good as the pointer-based programs. Because all evaluated programs are not too complex, the original programs can often perform as good as the transformed programs at higher levels. But as shown in figure 1, for more complex programs the conversion provides some substantial advantage.

### 4.2 Pentium II

The Pentium II is a general-purpose processor with a super-scalar architecture. Additionally, it supports a CISC instruction set. This makes the Pentium II quite different from the TriMedia. However, many signal processing applications are run on Desktop PCs in which the Intel Processor is commonly found. Therefore, this processor should be included in this evaluation.

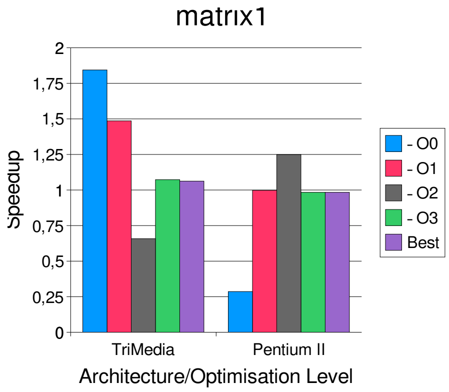


**Fig. 4.** Performance comparison of `matrix1` benchmark

The combination of the Intel Pentium II and the gcc compiler produces results that differ significantly from that of the TriMedia. The performance of the program versions after pointer conversion is quite poor without any further optimisations, but as the optimisation level is increased the transformed programs often outperform the original versions. The maximum speedup of the transformed programs is usually achieved at optimisation levels O2 and O3, respectively.

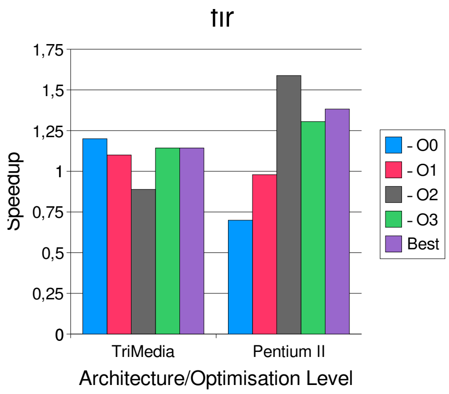


**Fig. 5.** Performance comparison of `fir` benchmark

In figure 5 the results for the `fir` benchmark are charted. Initially, the pointer transformation has negative effect on the performance of this program. A significant speedup can be observed for the higher optimisation levels. The maximum speedup achieved at level O2 is 1.59 and when comparing the two best versions each, the speedup of 1.38 can still be reached. The results of the `fir2dim` benchmark are represented in figure 6. As before the performance of the transformed version is inferior at optimisation levels O0 and O1, but it increases at O2 and O3. The performance can be increased by up to 6% in this case.

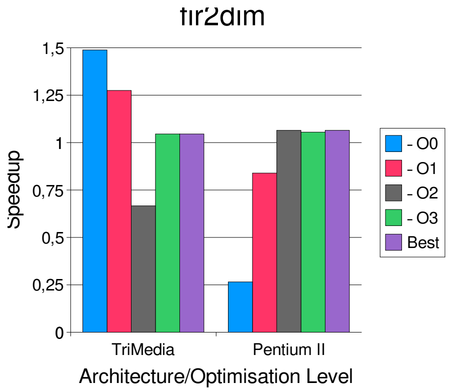

**Fig. 6.** Performance comparison of `fir2dim` benchmark

In general, the achievable speedups on the Pentium II architecture are not that large as they are on the TriMedia. It is not yet clear whether this is a consequence of architectural properties or the influence of the different compilers. However, an increase in the performance can be observed for the Intel processor, in particular at the higher optimisation levels.

## 5 Related Work

Allan and Johnson [1] use their vectorisation and parallelisation framework based on C as an intermediate language for induction variable substitution. Pointer-based array accesses together with pointer arithmetic in loops are regarded as induction variables that can be converted into expressions directly dependent on the loop induction variable. This approach does not regenerate the index expressions, but it supplies equivalent pointer expressions. Their main objective is to produce loop representations that are suitable for vectorisation. Therefore, their method only treats loops individually rather than in a context of a whole function. The approach is based on a heuristic that mostly works efficiently, although backtracking is possible. Hence, in the worst case this solution is inefficient. No information is supplied about treatment of loop nests and multi-dimensional arrays. In his case study of the Intel Reference Compilers for the i386 Architecture Family Muchnick [8] mentions briefly some technique for regenerating array indexes from pointer-based array traversal. No more details including assumptions, restrictions or capabilities are given. The complementary conversion, i.e. from explicit array accesses to pointer-based accesses with simultaneous generation of optimal AGU code, has been studied by Leupers [5] and Araujo [2].

## 6 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 support existing array data flow analyses and optimisations on DSP architectures. The approach has been implemented and integrated into the experimental Octave compiler and tested on examples of the DSPstone benchmark suite. Results show a significant a 11.9 % reduction in execution time. The generation of efficient address generation code is improved since modern compilers are able to analyse array accesses and to generate optimised code for memory accesses that does not rely on the use of pointers at the source level.

We believe the use of explicit array accesses to be the key to automatic parallelisation of DSP applications for Multiprocessor DSP architectures. Future work will focus on distributing computation and data on different co-operating DSPs while making the best use of ILP and coarse-grain parallelism.

## 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. de Araujo, Guido C.S., Code Generation Algorithms for Digital Signal Processors, Dissertation, Princeton University, Department of Electrical Engineering, June 1997.
3. 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.
4. Kernighan, Brian W. and Dennis M. Ritchie, The C Programming Language, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.
5. Leupers, Rainer, Novel Code Optimzation Techniques for DSPs, 2nd European DSP Education and Research Conference, Paris, France, 1998.
6. Lu, J., Interprocedural Pointer Analysis for C, Ph.D. thesis, Department of Computer Science, Rice University, Houston, Texas, 1998. SASIMI, Osaka, 1997.
7. 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.
8. Muchnick, Steven.S., Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, San Francisco, California, 1997.
9. Numerix-DSP Digital Signal Processing Web Site, `http://www.numerix-dsp.com/c_coding.pdf`, 2000.
10. 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.
11. Wilson, R.P., Efficient Context-Sensitive Pointer Analysis for C Programs, Ph.D. thesis, Stanford University, Computer Systems Laboratory, December 1997.
12. Zivojnovic, V., J.M. Velarde, C. Schlager and H. Meyr, DSPstone: A DSP-Oriented Benchmarking Methodology, *Proceedings of Signal Processing Applications & Technology*, Dallas 1994 .
13. H. Zima, Supercompilers for Parallel and Vector Computers, ACM Press, 1991.