An Efficient Algorithm for Pointer-to-Array Access Conversion for Compiling and Optimizing DSP Applications

Robert A. van Engelen* and Kyle A. Gallivan†
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530
{*engelen,gallivan*}*@cs.fsu.edu*

Abstract

The complexity of Digital Signal Processing (DSP) applications has been steadily increasing due to advances in hardware design for embedded processors. To meet critical power consumption and timing constraints, many DSP applications are hand-coded in assembly. Because the cost of hand-coding is becoming prohibitive for developing an embedded system, there is a trend toward the use of high-level programming languages, particularly C, and the use of optimizing compilers for software development. Consequently, more than ever there is a need for compilers to optimize DSP application to make effective use of the available hardware resources. Existing DSP codes are often riddled with pointer-based data accesses, because DSP programmers have the mistaken belief that a compiler will always generate better target code. The use of extensive pointer arithmetic makes analysis and optimization difficult for compilers for modern DSPs with regular architectures and large homogeneous registers sets. In this paper, we present a novel algorithm for converting pointer-based code to code with explicit array accesses. The conversion enables a compiler to perform data flow analysis and loop optimizations on DSP codes.

1. Introduction

The complexity of digital signal processing (DSP) applications has been steadily increasing. Advances in hardware design for embedded processors have led to a steep increase in the number of architectural features that can be exploited by DSP applications. Embedded processors are the vast majority of shipped processors, due to the high demand for commodity products such as cell phones. The specialized architectures of embedded systems that are, for example, used in cell phones are traditionally hand-coded in assembly to meet critical power consumption and timing constraints. Because the cost of software development is becoming prohibitive for developing an embedded system, there is a trend toward the use of high-level programming languages, particularly C, and the use of optimizing compilers. Consequently, more than ever there is a need for C compilers to optimize these programs to make effective use of the available hardware resources.

The loop-timing constraints of a DSP application are the most critical of the entire application. Therefore, the main task of a compiler is to optimize loops. The application of loop transformations requires pointer analysis, induction variable recognition, and data dependence analysis. The effectiveness of optimizing loop transformations depends solely on the accuracy of these methods. The inability of compilers to effectively perform program analysis may result in considerable performance and/or power losses caused by worst-case assumptions or when program analysis has to be performed at run-time.

Current compiler analysis is hampered by the extensive pointer arithmetic frequently used in DSP applications written in C. DSP programmers are actively encouraged to use pointer-based code in the mistaken belief that the compiler will always generate better target code [13]. Pointer-based accesses and pointer arithmetic are commonly used to inform compilers to use the *Address Generation Unit* (AGU) post-increment and decrement addressing modes [17]. However, the use of pointer arithmetic makes analysis and optimization difficult for compilers for modern DSPs with regular architectures and large homogeneous registers sets.

In this paper, we present a novel algorithm for converting pointer-based code to code with explicit array accesses. The conversion enables a compiler to perform data flow analysis, e.g. [9], loop optimizations [18, 23, 27], loop scheduling

*Supported in part by NSF grant CCR-9904943
†Supported in part by NSF grant EIA-0072043

for power reduction [24], and DSP architecture-specific optimizations that require explicit array references, e.g. [7, 8]. The pointer-based accesses and induction variables are converted to explicit array accesses with index expressions that directly depend on the loop induction variables of the outer loops. The method can handle pointer arithmetic with linear and non-linear pointer variable updates. The complementary conversion, from explicit array accesses to pointer-based accesses with generation of optimal AGU code, has been studied by others, e.g. [17].

The remaining part of this paper is organized as follows. Section 2 discusses related work which is followed in Section 3 by some motivating examples. In Section 4 we present our general algorithm including a detailed description of the mathematical background of the method. Finally, some concluding remarks are given in Section 5.

2. Related Work

Allen and Johnson [3] used their vectorization and parallelization framework as an intermediate language for induction variable substitution to generate pointer expressions that are more amenable to vectorization than the original representation. However, their approach does not fully convert pointers to index expressions. Muchnick [18] mentions the regeneration of array indexes from pointer-based array traversal, but no explicit details are given.

In [21] we introduced a novel method for induction variable recognition as part of an algorithm for induction variable substitution. This earlier work forms the basis of our pointer-based code analysis and conversion method. In general, loop induction variable recognition [1, 2, 18, 23, 27] is essential for most loop restructuring transformations. Many ad-hoc compiler analysis methods exist that are capable of recognizing linear induction variables, see e.g. [1, 2, 18, 23, 27]. These ad-hoc techniques fall short of recognizing *Generalized Induction Variables* (GIVs) with values that form polynomial and geometric progressions through loop iterations [4, 10, 11, 14, 15, 22]. GIV recognition is an important analysis technique for compilers in general [16, 19, 22]. In particular, the demand driven *sequence classification* method by Gerlek et al. [14] and Haghighat's *symbolic differencing* method [15, 16] are powerful GIV recognition methods. However, symbolic differencing is not safe [21] and its application can lead to non-semantics preserving code transformations. The sequence classification method relies on the use of various solvers to detect GIVs. A solver is required for each type of sequence: linear, polynomial, geometric, periodic, and wrap-around. In contrast, our GIV recognition method is safe and simple to implement yet fast and equally powerful to existing methods, except that the method cannot detect periodic sequences [14] also known as cyclic recurrences [16].

The work presented in this paper is most closely related to the work of Franke and O'Boyle [13]. They developed a compiler transformation to convert pointer-based accesses to explicit array accesses. However, their work has several assumptions and restrictions. In particular, their method is restricted to structured loops with a constant upper bound and all pointer arithmetic has to be data independent, i.e. pointer updates with constant increment/decrement values. Furthermore, pointer assignments, apart from initializations to some start element of the array to be traversed, are not permitted. Existing DSP codes, e.g. the GSM EFR speech codec [12], typically use various forms of pointer initializations and data dependent pointer updates. In addition, DSP codes may use non-rectangular loops. Our approach goes beyond existing work. More specifically, our algorithm can handle non-rectangular loops, more general pointer initializations, and the most common types of data dependent and independent pointer updates.

3. Motivation

We illustrate the need for analyzing data dependent and non-linear pointer updates by a compiler for a DSP architecture with an example code segment of the `Lsp_Az` routine of the GSM Enhanced Full Rate (EFR) speech codec [12], see Figure 1 below.

```
S1:  f += 2;
S2:  lsp += 2;
S3:  for (i = 2; i <= 5; i++)
S4:  { *f = f[-2];
S5:    for (j = 1; j < i; j++, f--)
S6:      *f += f[-2]-2*(*lsp)*f[-1];
S7:    *f -= 2*(*lsp);
S8:    f += i;
S9:    lsp += 2; }
```

Figure 1. Example `Lsp_Az` Code Segment

The code shown in Figure 1 is a slightly modified version of the original `Lsp_Az` routine. The original code adopts function calls for performing specialized word-size arithmetic operations on data but not on pointers. We replaced these function calls with conventional arithmetic operations to enhance readability. The modifications do not affect the applicability of the conversion algorithm.

Note that the loop nest is triangular: the upper bound of the inner loop counter variable `j` is the outer loop counter variable `i`. The pointer decrement `f--` in *S5* is used in a simple linear traversal by the inner loop over the data. Because the loop nest is triangular, `f` will have been decremented by `i-1` (the `j`-loop iteration count) at the end of the `j`-loop (*S7*). This means that the `f` pointer update is

potentially non-linear with respect to the `i`-loop, because the increment is a function of `i`. However, statement *S8* increments `f` by `i` and as a result the `f` pointer has a unit increment through the `i`-loop iterations.

Existing compiler analysis algorithms [13] cannot analyze and transform the example code from pointer-based accesses to explicit array accesses, because the loop nest is triangular and statement *S8* involves a data dependent pointer update. Worse, the fact that the example code exhibits pointer updates that are potentially non-linear makes it hard to analyze this code in general. Our pointer analysis algorithm automatically transforms the example code into the code segment depicted in Figure 2 below.

```
for (i = 0; i <= 3; i++)
{ f[i+2] = f[i];
  for (j = 0; j <= i; j++)
    f[i-j+2] += f[i-j]-2*lsp[2*i+2]*f[i-j+1];
  f[1] -= 2*lsp[2*i+2]; }
```

Figure 2. Transformed `Lsp_Az` Code Segment

In the transformed code, all pointer-based array accesses are replaced by explicit array accesses and all pointer update operations are removed. Note that after the conversion, the array index expressions in the code appear to be affine which makes the code amenable to data flow analysis [9], loop optimizations [18, 23, 27], loop scheduling for power reduction [24], and DSP architecture-specific optimizations that require explicit array references, e.g. [7, 8].

Another hard-to-analyze example is the general radix-2 FFT algorithm whose outline is shown in Figure 3 below.

```
S1:  b = N/2;
S2:  n = 2;
S3:  for (s = 0; s < log2N; s++)
S4:  { p = x;
S5:    q = p+b;
S6:    for (i = 0; i < n/2; i++)
S7:    { for (j = 0; j < b; j++)
S8:      { ... *p ...
S9:        ... *q ...
S10:       p++;
S11:       q++; }
S12:     p += b;
S13:     q += b;
S14:     ... }
S15:   n = n*2;
S16:   b = b/2; }
```

Figure 3. Example Radix-2 FFT

A compiler, for example, might need to determine whether pointers `p` and `q` used in statements *S8* and *S9* are aliases which would mean that the pointer-based array accesses are data dependent. Conversion of the pointer accesses to explicit array accesses is required to enable an accurate data dependence analysis of the array references. Note that the loop nest of the FFT algorithm is non-rectangular and pointer updates *S12* and *S13* are dependent on geometric induction variable `b` which is halved through each iteration of the `s`-loop. Our algorithm can handle multi-dimensional loops with generalized induction variables that form polynomial and geometric progressions through loop iterations and pointer updates that depend on coupled induction variables. The resulting transformation by our algorithm is shown in Figure 4 below.

```
for (s = 0; s <= log2N-1; s++)
{ for (i = 0; i <= 1<<s-1; i++)
  { for (j = 0; j <= N>>(s+1)-1; j++)
    { ... x[j+2*i*(N>>(s+1))] ...
      ... x[j+(2*i+1)*(N>>(s+1))] ... }
    ... }
}
```

Figure 4. Transformed Radix-2 FFT

Note that the `n` and `b` variables are recognized as geometric functions which were replaced by bit-shift operators. Because `j` runs from `0` to `N>>(s+1)-1`, the array accesses are independent with respect to the `i` and `j` loop.

4. Algorithm

In this section we present our conversion algorithm that transforms pointer-based array accesses in loops into explicit array accesses. The algorithm exploits the fact that the analysis of pointer arithmetic can be viewed as a form of induction variable recognition. In [21] we developed an algorithm for generalized induction variable recognition. In this paper, we will extend this algorithm with a method to analyze pointer-based array accesses. To this end, we introduce the notion of *pointer access descriptions* which are canonical representations of pointer accesses to memory as functions of the counter variables of the enclosing loop nest.

4.1. Chains of Recurrences

The mathematical basis of our induction variable recognition method is provided by the *Chains of Recurrences* (CR) formalism. The CR formalism was originally developed by Zima [25, 26] and later improved by Bachmann, Zima, and Wang [6]. In their work CRs are used to expedite the evaluation of real- and complex-valued functions on regular grids by an algorithmic transformation that is essentially a form of loop strength reduction. To expedite the

evaluation of a closed-form function in a loop with counter variable i, the function can be rewritten into a mathematical equivalent chain of recurrences, see e.g. [5, 6, 26]:

$$\Phi_i = \{\phi_0, \odot_1, \{\phi_1, \odot_2, \cdots, \{\phi_{k-1}, \odot_k, f_k\}_i\}_i\}_i \quad (1)$$

which is generally written as a single flattened tuple

$$\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, f_k\}_i \quad (2)$$

with $k = L(\Phi_i)$ the length of the CR.

Borrowing partly from [26], we call Φ_i a *polynomial* or *pure-sum* CR if $\odot_j = +$, for all $j = 1, \ldots, k$. A polynomial CR has a closed-form function that is a k-order polynomial in variable i. Φ_i is an *exponential* or *pure-product* CR if $\odot_j = *$, for all $j = 1, \ldots, k$. An exponential CR $\Phi_i = \{\phi_0, *, f_1\}_i$ is *geometric* if f_1 is i-loop invariant. The CR Φ_i is a *GIV* if $\odot_j = +$ for $j = 1, \ldots, k-1$ and $\odot_k = *$. The CR $\Phi_i = \{\phi_0, *, \phi_1, +, f_2\}_i$ is a *factorial* CR if $\phi_1 \geq 1$ and $f_2 = 1$, or $\phi_1 \leq -1$ and $f_2 = -1$.

The construction of a CR for a closed-form expression E proceeds by replacing every occurrence of the loop counter variable i in E by the CR $\{a, +, s\}_i$, where a is i's (symbolic) initial value and s is the stride. Then, $\mathcal{CR}$ rules shown in Figure 6 are exhaustively applied to simplify E. The exhaustive application of $\mathcal{CR}$ results in so-called *CR-expressions*, which are expressions that *contain* CRs as subexpressions. In [20] we proved that $\mathcal{CR}$ is complete (i.e. confluent and terminating) and, hence, CRs are normal forms for polynomials, exponentials, GIVs, and factorials. We found several new rules (3, 7, 10, 11, 13, and 21 in Figure 6) that we added to the original CR algebra for normalization purposes.

A CR can be directly translated into an algorithm that utilizes induction variables to compute the original function on a regular grid much faster than computing the function values for every grid point [6], see Figure 5.

$F[0] := \phi_0$
$cr_1 := \phi_1$
:
$cr_k := f_k$
for $i = 1$ **to** n **do**
  $F[i] := F[i-1] \odot_1 cr_1$
  $cr_1 := cr_1 \odot_2 cr_2$
  :
  $cr_{k-1} := cr_{k-1} \odot_k cr_k$
od

Figure 5. Computation of Function F of a CR $\{\phi_0, \odot_1, \phi_2, \odot_2, \ldots, \odot_k, f_k\}_i$ on Grid $i = 0, \ldots, n$

The algorithm shown in Figure 5 defines the semantics of CRs as a representation for discrete functions. The algorithm tabulates a function F on a grid $0, \ldots, n$ through updates to induction variables cr_j, $j = 1, \ldots, k$, in a loop over the grid points. The difference between the use of CRs in this algorithm and the work presented in this paper is that we utilize CRs as normal forms for solving the complementary problem: the translation of induction variables and pointer arithmetic to closed-form expressions that depend on the counter variables of the enclosing loop nest.

4.2. Pointer Access Descriptions

We introduce the notion of a *Pointer Access Description* (PAD) which is a pointer-typed CR that describes the memory accesses made by a pointer in a loop nest as a function of the counter variables of the loop nest. In its simplest form, a PAD is a CR $\Phi_i = \{\phi_0, +, \ldots, f_k\}_i$ with ϕ_0 a pointer-typed expression or memory address and ϕ_j for $j = 1, \ldots, k-1$ and f_k are integer-typed expressions or CRs that depend on other counter variables.

As an example, consider a loop with counter variable `i` initialized to zero and having stride one. The PADs for several example array and pointer location accesses are shown in Table 1 below[1].

	Access	*PAD*
1	`a[i]`	$\{\mathtt{a}, +, 1\}_i$
2	`a[2*i+1]`	$\{\mathtt{a}{+}1, +, 2\}_i$
3	`a[(i*i-i)/2]`	$\{\mathtt{a}, +, 0, +, 1\}_i$
4	`a[1<<i]`	$\{\mathtt{a}{+}1, +, 1, *, 2\}_i$
5	`p++`	$\{p_0, +, 1\}_i$
6	`q-=2`	$\{q_0, +, -2\}_i$
7	`r+=i`	$\{r_0, +, 0, +, 1\}_i$
8	`s+=2*i`	$\{s_0, +, 0, +, 2\}_i$

Table 1. Example PADs

The coefficients p_0, q_0, r_0, and s_0 denote the initial values of the pointers prior to the execution of the loop. Note that the PADs 1, 2, 5, and 6 are linear CRs, 3, 7, and 8 are second order polynomial CRs, and 4 is a GIV CR which, in general, is the sum of a polynomial and a geometric function.

Pointers and arrays are exchangeable in C, which contributes to the popularity of C (and C++) for programming with pointer arithmetic for array access. In C, an array access `a[`n`]` can be written as a pointer expression `*(a+`n`)` which dereferences the n^{th} element of `a`. That is, the addition of an integer value n to a pointer in a C expression means the address of the n^{th} element beyond the address the pointer currently points to. The value n is scaled according to the size of the elements the pointer points to, which is determined by the type declaration of the pointer.

[1] The `p`, `q`, `r`, and `s` pointer updates shown in this table are assumed to be unique in the loop.

	LHS		RHS	Condition
1	$\{\phi_0, +, 0\}_i$	$\Rightarrow$	ϕ_0	
2	$\{\phi_0, *, 1\}_i$	$\Rightarrow$	ϕ_0	
3	$\{0, *, f_1\}_i$	$\Rightarrow$	0	
4	$-\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\{-\phi_0, +, -f_1\}_i$	
5	$-\{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\{-\phi_0, *, f_1\}_i$	
6	$\{\phi_0, +, f_1\}_i \pm E$	$\Rightarrow$	$\{\phi_0 \pm E, +, f_1\}_i$	when E is i-loop invariant
7	$\{\phi_0, *, f_1\}_i \pm E$	$\Rightarrow$	$\{\phi_0 \pm E, +, \phi_0 * (f_1 - 1), *, f_1\}_i$	when E and f_1 are i-loop invariant
8	$E * \{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, +, E * f_1\}_i$	when E is i-loop invariant
9	$E * \{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, *, f_1\}_i$	when E is i-loop invariant
10	$E/\{\phi_0, +, f_1\}_1$	$\Rightarrow$	$1/\{\phi_0/E, +, f_1/E\}_i$	when $E \neq 1$ is i-loop invariant
11	$E/\{\phi_0, *, f_1\}_1$	$\Rightarrow$	$\{E/\phi_0, *, 1/f_1\}_i$	when E is i-loop invariant
12	$\{\phi_0, +, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 \pm \psi_0, +, f_1 \pm g_1\}_i$	
13	$\{\phi_0, *, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 \pm \psi_0, +, \{\phi_0 * (f_1 - 1), *, f_1\}_i \pm g_1\}_i$	when f_1 is i-loop invariant
14	$\{\phi_0, +, f_1\}_i * \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 * \psi_0, +, \{\phi_0, +, f_1\}_i * g_1 + \{\psi_0, +, g_1\}_i * f_1 + f_1 * g_1\}_i$	
15	$\{\phi_0, *, f_1\}_i * \{\psi_0, *, g_1\}_i$	$\Rightarrow$	$\{\phi_0 * \psi_0, *, f_1 * g_1\}_i$	
16	$\{\phi_0, *, f_1\}_i^E$	$\Rightarrow$	$\{\phi_0^E, *, f_1^E\}_i$	when E is i-loop invariant
17	$\{\phi_0, *, f_1\}_i^{\{\psi_0, +, g_1\}_i}$	$\Rightarrow$	$\{{\phi_0}^{\psi_0}, *, \{\phi_0, *, f_1\}_i^{g_1} * {f_1}^{\{\psi_0, +, g_1\}_i} * f_1^{g_1}\}_i$	
18	$E^{\{\phi_0, +, f_1\}_i}$	$\Rightarrow$	$\{E^{\phi_0}, *, E^{f_1}\}_i$	when E is i-loop invariant
19	$\{\phi_0, +, f_1\}_i^n$	$\Rightarrow$	$\begin{cases} \{\phi_0, +, f_1\}_i * \{\phi_0, +, f_1\}_i^{n-1} & \textbf{if } n \in \mathbb{Z}, n > 1 \\ 1/\{\phi_0, +, f_1\}_i^{-n} & \textbf{if } n \in \mathbb{Z}, n < 0 \end{cases}$	
20	$\{\phi_0, +, f_1\}_i!$	$\Rightarrow$	$\begin{cases} \{\phi_0!, *, \left(\prod_{j=1}^{f_1} \{\phi_0 + j, +, f_1\}_i\right)\}_i & \textbf{if } f_1 \geq 0 \\ \{\phi_0!, *, \left(\prod_{j=1}^{\lvert f_1 \rvert} \{\phi_0 + j, +, f_1\}_i\right)^{-1}\}_i & \textbf{if } f_1 < 0 \end{cases}$	
21	$\{\phi_0, +, \phi_1, *, f_2\}$	$\Rightarrow$	$\{\phi_0, *, f_2\}_i$	when $\frac{\phi_1}{\phi_0} = f_2 - 1$

Note: Bachmann's algorithm [5] can be used for fast polynomial multiplication in $\mathcal{O}(k^2)$ time, where k is the order of the polynomial. The application of rule 14 requires $\mathcal{O}(k^3)$ operations for multiplication. Zima describes an algorithm for polynomial CR division [26]..

Figure 6. $\mathcal{CR}$

Figure 7 shows valid pointer expressions in C which is presented in the figure as a syntactic convention.

```
1  ptr_expr  ::=  ptr_var
2                |  array_var
3                |  & l_value
4                |  ptr_expr + int_expr
5                |  ptr_expr - int_expr
6  int_expr  ::=  ptr_expr - ptr_expr
7                |  other integer-typed expressions
8  rel_expr  ::=  ptr_expr relop ptr_expr
```

Figure 7. Pointer Expressions in C

A pointer expression is a pointer variable (1), array variable (2), or the memory address of a data object (*l_value*) (3). Integer values can be added or subtracted from pointer expressions to form pointer expressions (4 and 5). Pointer expressions of the same pointer type can be subtracted (6). For example, if `p` and `q` point to elements of the same array, then `q-p` is an integer value that is the number of elements from `p` to `q`. Pointers that point to the elements of the same array can be compared (8) with the relations *relop*$\in$`{<,<=,==,>,>=,!=}`.

We let the usual semantics of pointer expressions in C also apply to pointers and addresses in PADs. For example, the coefficient `a`$+1$ in the second PAD in Table 1 denotes the address of the second element of array `a`, i.e. `&a[1]`.

The $\mathcal{CR}$ algebra preserves the semantics of pointer expressions as illustrated in Table 2 below.

	Operation		*Result*	*$\mathcal{CR}$ Rules*
1	*ptr_expr* $\pm$ *CR*	$\Rightarrow$	*PAD*	$6, 7$
2	*PAD* $\pm$ *constant*	$\Rightarrow$	*PAD*	6
3	*PAD* $\pm$ *CR*	$\Rightarrow$	*PAD*	$12, 13$
4	*PAD* $-$ *ptr_expr*	$\Rightarrow$	*CR*	6
5	*ptr_expr* $-$ *PAD*	$\Rightarrow$	*CR*	$6, 4$
6	$PAD_1 - PAD_2$	$\Rightarrow$	*CR or constant*	$12, 1$

Table 2. Pointer Arithmetic With PADs

This means that the PAD of an explicit array access can be obtained directly by CR construction as described in Section 4.1. For example, the address of `a[i]` translates into `a+i` which is represented by `a`$+\{0, +, 1\}_i$ and simplified to the PAD $\{\texttt{a}, +, 1\}_i$ by $\mathcal{CR}$ rule 6.

In the above discussion, it was assumed that pointers and addresses were loop invariant. When pointers are not

	LHS		RHS	Condition
1	$\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\phi_0 + \{0, +, f_1\}_i$	when $\phi_0 \neq 0$
2	$\{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\phi_0 * \{1, *, f_1\}_i$	when $\phi_0 \neq 1$
3	$\{0, +, -f_1\}_i$	$\Rightarrow$	$-\{0, +, f_1\}_i$	
4	$\{0, +, f_1 + g_1\}_i$	$\Rightarrow$	$\{0, +, f_1\}_i + \{0, +, g_1\}_i$	
5	$\{0, +, f_1 * g_1\}_i$	$\Rightarrow$	$f_1 * \{0, +, g_1\}_i$	when i does not occur in f_1
6	$\{0, +, f_1^i\}_i$	$\Rightarrow$	$\frac{f_1^i - 1}{f_1 - 1}$	when i does not occur in f_1 and $f_1 \neq 1$
7	$\{0, +, f_1^{g_1 + h_1}\}_i$	$\Rightarrow$	$\{0, +, f_1^{g_1} * f_1^{h_1}\}_i$	
8	$\{0, +, f_1^{g_1 * h_1}\}_i$	$\Rightarrow$	$\{0, +, (f_1^{g_1})^{h_i}\}_i$	when i does not occur in f_1 and g_1
9	$\{0, +, f_1\}_i$	$\Rightarrow$	$i * f_1$	when i does not occur in f_1
10	$\{0, +, i\}_i$	$\Rightarrow$	$\frac{i^2 - i}{2}$	
11	$\{0, +, i^n\}_i$	$\Rightarrow$	$\sum_{k=0}^{n} \frac{\binom{n+1}{k}}{n+1} B_k \, i^{n-k+1}$	for $n \in \mathbb{N}$, B_k is k^{th} Bernoulli number
12	$\{1, *, -f_1\}_i$	$\Rightarrow$	$(-1)^i \{1, *, f_1\}_i$	
13	$\{1, *, \frac{1}{f_1}\}_i$	$\Rightarrow$	$\{1, *, f_1\}_i^{-1}$	
14	$\{1, *, f_1 * g_1\}_i$	$\Rightarrow$	$\{1, *, f_1\}_i * \{1, *, g_1\}_i$	
15	$\{1, *, f_1^{g_1}\}_i$	$\Rightarrow$	$f_1^{\{1, *, g_1\}_i}$	when i does not occur in f_1
16	$\{1, *, g_1^{f_1}\}_i$	$\Rightarrow$	$\{1, *, g_1\}_i^{f_1}$	when i does not occur in f_1
17	$\{1, *, f_1\}_i$	$\Rightarrow$	f_1^i	when i does not occur in f_1
18	$\{1, *, i\}_i$	$\Rightarrow$	0^i	
19	$\{1, *, i + f_1\}_i$	$\Rightarrow$	$\frac{(i + f_1 - 1)!}{(f_1 - 1)!}$	when i does not occur in f_1 and $f_1 \geq 1$
20	$\{1, *, f_1 - i\}_i$	$\Rightarrow$	$(-1)^i * \frac{(i - f_1 - 1)!}{(-f_1 - 1)!}$	when i does not occur in f_1 and $f_1 \leq -1$

Figure 8. $\mathcal{CR}^{-1}$

loop invariant, e.g. the pointers in Table 1 examples 5–8, a closed-form expression for each pointer access must be derived that solely depends on the counter variables of the loop nest. The essential step required for this translation is the derivation of a closed-form expression for pointer accesses.

4.3. Generalized Induction Variables

The pointer-based array traversal analysis problem is similar to the problem of finding the closed-form functions of induction variables. That is, pointer update operations in a loop can be viewed as a form of induction variable updates. In particular, the recognition of data dependent pointer updates corresponds to the recognition of *Generalized Induction Variables* (GIVs). A GIV V is characterized [16] by its *characteristic function* χ_V defined by

$$\chi_V(n) = \varphi(n) + r\, a^n \tag{3}$$

where n is the loop iteration number, φ is a polynomial of order k, and a and r are loop-invariant expressions.

The objective of *GIV recognition* is to find the closed-form characteristic function of a GIV in a loop nest. The removal of the updates of a GIV in a loop nest and the substitution of the induction variable with its closed-form characteristic function in expressions is known as *Induction Variable Substitution* (IVS). IVS effectively removes all cross-iteration dependencies induced by GIV updates. This, for example, enables loops to be parallelized [16].

The conversion of pointer traversals with pointer arithmetic to closed-form pointer expressions that depend on the counter variables of the enclosing loop nest is similar to induction variable substitution. The IVS algorithm presented in [21] exploits the CR notation for GIV recognition. The closed-form characteristic function of a GIV is obtained by application of the CR inverse rules $\mathcal{CR}^{-1}$ shown in Figure 8 which we specifically developed for this purpose. To quickly obtain the closed-form of a polynomial CR Φ_i Newton's formula for the interpolating polynomial can be used:

$$\chi(i) = \sum_{j=0}^{k} \phi_j \binom{i}{j} \tag{4}$$

with $k = L(\Phi_i)$, instead of the more complicated $\mathcal{CR}^{-1}$ rules 9–11 shown in Figure 8.

A closed-form pointer expression of a PAD can be obtained using $\mathcal{CR}^{-1}$ and Eq. (4). For example, the closed form of the PAD $\{p_0, +, 1\}_i$ shown in Table 1 is $p_0 + i$ and the closed form of $\{r_0, +, 0, +, 1\}_i$ is $r_0 + (i^2 - i)/2$. The translation of PAD to closed-form by $\mathcal{CR}^{-1}$ and Eq. (4) assumes that the loop is normalized, i.e. the initial value of the loop counter variable i must be zero. The loop normalization is performed by the algorithm after GIV recognition.

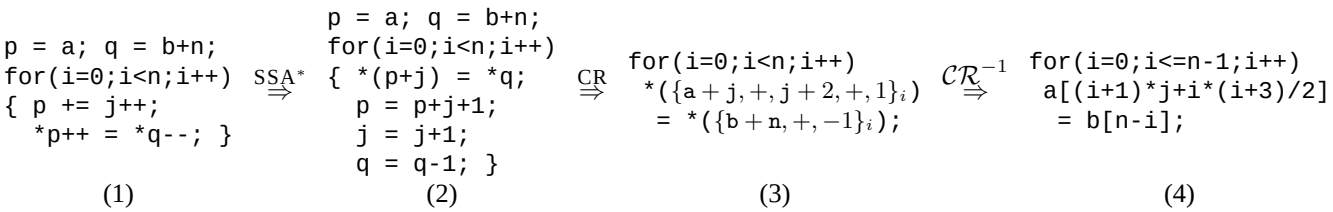

Figure 9. Example Application of IVS$^+$

4.4. Extended IVS Algorithm

The extended induction variable substitution algorithm IVS$^+$ converts pointer accesses to explicit array accesses in loop nests. The worst-case computational complexity of the IVS$^+$ algorithm is $\mathcal{O}(m\,n\,\log(n)\,k^2)$, where m is the maximum loop nesting level, n is the length of the source code fragment, and k is the maximum length of the CRs derived for GIVs in the fragment.

Figure 9 illustrates the four stages of the IVS$^+$ algorithm on an example code fragment. The program input (1) is translated into a special variation of a *Single Static Assignment* (SSA) form (2) in which variable updates are propagated and collected at the end of a loop. After the SSA construction, induction variables and pointers are analyzed. Uses of induction variables are replaced by CRs and pointer accesses are replaced with PADs (3). In the final step of the algorithm, PADs are translated to explicit array accesses (4) by applying the $\mathcal{CR}^{-1}$ rewrite rules and replacing the resulting dereferenced pointer expressions with explicit array accesses.

The complete IVS$^+$ algorithm is listed in Figures 10 and 11. There are three restrictions to the applicability of the algorithm. First, the algorithm does not perform pointer alias analysis. For example, the assignment `p=&i` invalidates the conversion of the code if the code dereferences `p` while `i` is an induction variable. Second, loops are not allowed to have explicit loop exits (`breaks`). All other control flow in a loop can be handled by using if-then-elses in the code. Third, no interprocedural analysis is performed. However, the algorithm works if function calls do not modify any pointers and induction variables. The algorithm converts pointer accesses to one-dimensional array accesses. A method to convert these one-dimensional arrays to multi-dimensional arrays can be found in [13].

The IVS$^+$ algorithm analyzes a loop nest (not necessarily perfectly nested) from the innermost loops, which are the primary candidates for optimization, to the outermost loops. For every loop in the nest, the body is converted to a single-static assignment form with assignments to scalar variables separated from the loop body and stored in set A. Next, algorithm CR converts expressions in A to CR-expressions to detect induction variables. Algorithm HOIST hoists the induction variable update assignments out of the loop and replaces induction expressions in the loop by closed-forms. Polynomial, factorial, GIV, and exponential CRs can always be converted to a closed-form. However, some CRs do not have equivalent closed-forms. To extend our approach beyond traditional GIV recognition, we use algorithm Figure 5 in IVS$^+$.

The SSA* algorithm shown uses the precedence relation on variable-expression pairs defined by

$$(U, Y) \prec (V, X) \qquad \text{if } U \neq V \text{ and } V \text{ occurs in } Y$$

where U and V are variable names, to obtain an ordered set of variable-expression pairs A extracted from a loop body. A directed acyclic graph is constructed for expressions in A and expressions in the loop body such that common-subexpressions share the same node in the graph. Algorithm SSA* fails when $\prec$ is not a partial order on A. This happens in the presence of cyclic recurrences in a loop.

Each variable has only one expression in A. Set A contains the potential induction variables of the loop. Values of conditional induction variables are represented by conditional expressions of the form $C?X{:}Y$, where C is the condition and X and Y are expressions. The conditional expression $C?X{:}X$ is rewritten into X.

The conversion algorithm attempts to remove all pointer assignments. The removable pointer assignments are listed in the Table 3 below (`p` and `q` are pointers, `a` is an array variable).

Initialization	*Removable*	*Initialization*	*Removable*
`p = a`	*yes*	`p = malloc(n)`	*no*
`p = &a[n]`	*yes*	`p = &a[b[i]]`	*no*
`p = q`	*yes*	`p = f(n)`	*no*
`p = q+n`	*yes*	`f(&p)`	*no*

Table 3. Removable Pointer Initializations

When pointer initializations cannot be removed, the conversion may not produce code that a compiler can handle for applying data flow analysis.

Algorithm IVS$^+(S)$
- input: statement list S
- output: induction variable substitution and pointer-to-array conversion applied to S
1. Convert all explicit array references in S to pointer offset accesses by rewriting:
   `a[`X`]` $\Rightarrow$ `*(a+`X`)` for pointers and arrays `a` and index expression X
2. CALL IVSTRANS(S)
3. Use reaching flow information to remove and propagate initial assignments of (pointer) variables to their uses in S (see Table 3)
4. Apply $\mathcal{CR}^{-1}$ rules to every CR and PAD that appears in an expression in S
5. Convert all pointer-offset accesses to explicit array accesses by rewriting:
   `a+`X $\Rightarrow$ `&a[`X`]` for pointers and arrays `a` and index expressions X
   `*(&`X`)` $\Rightarrow X$ for l-value expressions X
6. For every left-over CR Φ_i in S, replace it with an index array $ia[i]$ that is initialized using the algorithm shown in Figure 5,
   where n is the upper bound of the index array which is the maximum value of the induction variable i of the loop in which Φ_i occurs

Algorithm IVSTRANS(S)
- input: statement list S
- output: CR expressions in S **represent the induction expressions in** S
FOR each loop L in statement list S DO
  Let $S(L)$ denote the body of loop L, let i denote the counter variable with initial value a, bound b, and stride s
  CALL IVSTRANS$(S(L))$
  TRY CALL SSA$^*(S(L), A)$
  CATCH FAIL:
    Continue with next do-loop L in S
  CALL CR$(i, a, s, S(L), A)$
  CALL HOIST$(i, a, b, s, S(L), A)$

Algorithm SSA$^*(S, A)$
- input: loop body statement list S
- output: SSA-modified S **and partially ordered set** A**, or** FAIL
A:=$\emptyset$
FOR each statement $S_i \in S$ from the last ($i = |S|$) to the first statement ($i = 1$) DO
  CASE S_i
  OF assignment statement of expression X to V:
    IF V is a numeric scalar or pointer variable, $(V, \perp) \notin A$, and X has no function calls and array accesses THEN
      FOR each statement $S_j \in S$, $j = i+1, \ldots, |S|$ DO
        Substitute in S_j every use of variable V by a reference to X
      FOR each $(U, Y) \in A$ DO
        Substitute in Y every use of variable V by a reference to X
      IF $(V, \_) \notin A$ THEN /* note: _ is a wildcard */
        A:=$A \cup \{(V, X)\}$
      Remove S_i from S
    ELSE /* other kind of assignment */
      Continue with next statement S_i
  OF if-then-else statement with condition C, then-clause $S(T)$, and else-clause $S(E)$:
    CALL SSA$^*(S(T), A_1)$
    CALL SSA$^*(S(E), A_2)$
    CALL MERGE$(C, A_1, A_2, A_\cup)$
    FOR each $(V, X) \in A_\cup$ DO
      FOR each statement S_j, $j = i+1, \ldots, |S|$ DO
        Substitute in S_j every use of variable V by a reference to X
      FOR each $(U, Y) \in A$ DO
        Substitute in Y every use of V by a reference to X
      IF $(V, \_) \notin A$ THEN /* note: _ is a wildcard */
        A:=$A \cup \{(V, X)\}$
  OF loop:
    IF the loop body contains an assignment to a numeric scalar or pointer variable V THEN
      A:=$A \cup (V, \perp)$
Topologically sort A with respect to $\prec$, FAIL if sort not possible (i.e. $\prec$ is not a partial order on A)

Figure 10. Algorithm IVS$^+$

Algorithm MERGE$(C, A_1, A_2, A_\cup)$
- input: Boolean expression C, variable-expression sets A_1 and A_2
- output: merged set $A_\cup$
$A_\cup$:=$\emptyset$
FOR each $(V, X) \in A_1$ DO
  IF $(V, Y) \in A_2$ for some expression Y THEN
    $A_\cup$:=$A_\cup \cup \{(V, C?X:Y)\}$
  ELSE
    $A_\cup$:=$A_\cup \cup \{(V, C?X:V)\}$
FOR each $(V, X) \in A_2$ DO
  IF $(V, \_) \notin A_\cup$ THEN
    $A_\cup$:=$A_\cup \cup \{(V, C?V:X)\}$

Algorithm CR(i, a, s, S, A)
- input: loop counter variable i with initial value a and stride s, statement list S, and topologically ordered set A of variable-expression pairs
- output: expressions in A are converted to CR-expressions
FOR each $(V, X) \in A$ in topological order ($\prec$) DO
  Substitute in X every use of i by $\{a, +, s\}_i$
  Apply $\mathcal{CR}$ rules to simplify expression X
  IF X is of the form $V + C$, where C is an i-loop invariant expression or a CR THEN
    Replace (V, X) in A with $(V, \{V, +, C\}_i)$
    FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
      Substitute every use of V in Y by $\{V, +, C\}_i$
  ELSE IF X is of the form $V * C$, where C is an i-loop invariant expression or a CR THEN
    Replace (V, X) in A with $(V, \{V, *, C\}_i)$
    FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
      Substitute every use of V in Y by $\{V, *, C\}_i$
  ELSE IF V does not occur in X THEN /* wrap-around variable */
    Replace (V, X) in A with $(V, \{V - \mathcal{V}(\mathcal{B}(X)), *, 0\}_i + \mathcal{B}(X))$
    FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
      Substitute every use of V in Y by $\{V - \mathcal{V}(\mathcal{B}(X)), *, 0\}_i + \mathcal{B}(X)$
  ELSE /* other type of assignment */
    Continue with next (V, X) pair in A
FOR each $(V, X) \in A$ in topological order ($\prec$) DO
  FOR each statement $S_j \in S$ (and statements at deeper nesting levels) DO
    Substitute every use of V in S_j by X
    Apply $\mathcal{CR}$ rules to every expression that occurs in S_j

Algorithm HOIST(i, a, b, s, S, A)
- input: loop counter variable i with initial value a, bound b, stride s, loop statement list S, and topologically ordered set A of variable-expression pairs
- output: loop S with induction variable assignments hoisted out of S
T := []
FOR each $(V, X) \in A$ in reversed topological order ($\prec$) DO
  Apply $\mathcal{CR}^{-1}$ to expression X resulting in expression Y
  IF V does not occur in Y THEN
    IF V is live at the end of the loop THEN
      Substitute every use of i in Y by $\lfloor \frac{b-a+s}{s} \rfloor$
      Append $V := Y$ at the end of statement list T
  ELSE
    Append $V := X$ at the end of statement list S
Replace the entire statement list S with a loop with body S and followed by statements T:
S := (`for (`i`=0; `i`<=(`$b-a$`)/`s`; `i`++) { `S` } `T)

Figure 11. Algorithm IVS$^+$ (Cont'd)

5. Concluding Remarks

In this paper we presented a powerful method for the conversion of pointer-based code to explicit array accesses. The conversion enables a compiler to perform data flow analysis and loop optimizations on DSP codes

Two issues need to be addressed to make the approach more practical. First, pointer analysis is required to enable the algorithm in the presence of pointer aliases. Second, interprocedural analysis is required if the functions called in loops modify pointers and/or induction variables.

Acknowledgments

We are grateful to Eugene Zima for numerous discussions on the chains of recurrences formalism. We would also like to thank Carl von Platen of IAR Systems Inc. for his helpful comments on the GSM EFR speech codec.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[2] F. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, pages 79–101, New-Jersey, 1981. Prentice-Hall.

[3] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proc. of the SIGPLAN 1988 Conference of Programming Languages Design and Implementation*, pages 241–249, Atlanta, GA, June 1988.

[4] Z. Ammerguallat and W. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pages 283–295, White Plains, NY, 1990.

[5] O. Bachmann. *Chains of Recurrences*. PhD thesis, Kent State University of Arts and Sciences, 1996.

[6] O. Bachmann, P. Wang, and E. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computing*, pages 242–249, Oxford, 1994. ACM.

[7] A. Basu, R. Leupers, and P. Marwedel. Array index allocation under register constraints in DSP programs. In *12th Int. Conf. on VLSI Design*, Goa, India, 1999.

[8] M. Cintra and G. Araujo. Array reference allocation using SSA-form and live range growth. In *Proc. of the ACM SIGPLAN 2000 LCTES*, pages 26–33, Vancouver, June 2000.

[9] E. Duesterwald, R. Gupta, and M. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Proc. of the SIGPLAN Conference on Programming Languages Design and Implementation*, pages 67–77, Albuquerque, New Mexico, 1993.

[10] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua. Restructuring Fortran programs for Cedar. In *Proc. of ICPP'91*, volume 1, pages 57–66, St. Charles, Illinois, 1991.

[11] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In 4^{th} *Annual Workshop on Languages and Compilers for Parallel Computing, LNCS 589*, pages 65–83, Santa Clara, CA, 1991. Springer Verlag.

[12] European Telecommunication Standard (ETSI). Digital cellular telecommunications system: ANSI-C code for the GSM enhanced full rate (EFR) speech codec. Available from `http://www.etsi.org`.

[13] B. Franke and M. O'Boyle. Compiler transformation of pointers to explicit array accesses in DSP applications. In *Proc. of the ETAPS Conf. on Compiler Construction 2001, LNCS 2027*, pages 69–85, 2001.

[14] M. Gerlek, E. Stolz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(1):85–122, Jan. 1995.

[15] M. Haghighat and C. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In 5^{th} *Annual Workshop on Languages and Compilers for Parallel Computing, LNCS 757*, pages 538–562, New Haven, Connecticut, 1992. Springer Verlag.

[16] M. R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic Publishers, 1995.

[17] R. Leupers. Novel code optimization techniques for DSPs. In *Proc. of the* 2^{nd} *European DSP Education and Research Conference*, Paris, France, 1998.

[18] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Fransisco, CA, 1997.

[19] J. Singh and J. Hennessy. An emperical investigation of the effectiviness and limitations of automatic parallelization. In N. Suzuki, editor, *Shared Memory Multiprocessing*, pages 203–207. MIT press, Cambridge MA, 1992.

[20] R. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical report, TR-000102, Computer Science Dept., Florida State University, 2000.

[21] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proc. of the ETAPS Conference on Compiler Construction 2001, LNCS 2027*, pages 118–132, 2001.

[22] M. Wolfe. Beyond induction variables. In *ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation*, pages 162–174, San Fransisco, CA, 1992.

[23] M. Wolfe. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, CA, 1996.

[24] T. Z. Yu, F. Chen, and E. H.-M. Sha. Loop scheduling algorithms for power reduction. In *Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Seattle, Washington, May 1998.

[25] E. Zima. Recurrent relations and speed-up of computations using computer algebra systems. In *Proc. of DISCO'92*, pages 152–161. LNCS 721, 1992.

[26] E. Zima. Simplification and optimization transformations of chains of recurrences. In *Proc. of the International Symposium on Symbolic and Algebraic Computing*, Montreal, Canada, 1995. ACM.

[27] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.