Shape Cliques

Robert Bernecky

October 18, 2007

Abstract

We introduce *shape cliques*, a simple way to organize a subset of the arrays appearing in an array-language-based application into sets of identically shaped arrays - shape cliques - and show how a compiler can analyze an application to infer membership in those cliques.

We describe an algorithm for performing shape clique inference (SCI), and demonstrate that shape cliques can improve the performance of generated code, by permitting extension of an optimization for removal of run-time checks, and by extending the set of arrays to which optimizations, such as Index Vector Elimination (IVE), can be applied.

Implementation of SCI in the APEX APL compiler permitted removal of 25% of run-time checks remaining on 156 benchmarks remaining after other compiler optimizations had eliminated 72% of the 1251 checks present in the original code. In the SAC compiler, IVE using SCI produced typical speedups of 2–14X on benchmarks operating on arrays of non-fixed rank and shape, compared to the operation of IVE in a non-SCI environment.

Shape clique inference data can be exploited to allow certain other optimizations, such as loop fusion and with-loop folding, to be performed on arrays of statically unknown shape and rank, with the potential for significant reductions in execution time.

1 Introduction

Array languages, such as APL[1], J[2], SAC[3, 4] and SISAL[5], are used in such diverse areas as financial modeling, actuarial research, molecular modeling, signal processing, text manipulation and compression, oceanography, economic data bases, and geophysics. Array languages permit high-level algorithmic expression, facilitating terse expression and rapid application development. However, array languages are designed more for the computer between our ears than for the silicon-based hardware of today, so their performance often suffers, due to run-time checking of array bounds and shape conformability, missing optimizations, array-valued intermediate results, and so on.

We have been investigating the relationships among array shapes in applications written in array languages, with an eye toward improving the performance and robustness of applications written in these languages. Specifically, we wish to exploit static knowledge of array shapes to generate compiled code that executes faster or uses less memory. Moreover, we wish to detect, statically (*i.e.*, at compile time), semantic errors in applications, such as the *length error* or *index error* of APL and J.[1] Finally, we want to eliminate, whenever possible, run-time checks for such errors, along the lines of [6].

Section 2 presents the SAC taxonomy of array shape classes [3, 7, 8]; Section 3 introduces array shape analysis; Section 4 introduces the concept of Shape Cliques and how they may be inferred by inspection of an application; Section 5 discusses a Shape Clique Inference algorithm that we implemented in the APEX APL Compiler[9], a model of the algorithm, a run-time-check optimizer that uses shape clique information to improve its performance, and quantitative information on the performance of that optimizer; Section 6 presents an extended version of the SAC Index Vector Elimination optimization, exploiting SCI to improve run-time performance. Section 7 discusses some potential applications of shape cliques in code optimization. Section 8 mentions guarded shape cliques; remaining Sections summarize our findings, related work, and future research directions.

2 Array Shape Classes

Several types of arrays occur in array-language programs. In a perfect world, we know their *rank*, *shape*, and *value*. For instance, the constant vector `[1, 2, 3]` is *rank-1* – it is one-dimensional. It is of *shape* three – it has three elements along its single axis – and we know the *value* of each of those elements. To facilitate discussion of arrays, the SAC developers introduced an array taxonomy, denoting arrays such as the above constant as *AKV*: Arrays of Known Value.

A chessboard at some unknown point in a game has known rank and shape (two and `[8. 8]`, respectively), but unknown value, since we do not know any of the values of the array elements (the pieces on the board). This is an *AKS* array: an Array of Known Shape. These are the arrays of Fortran.

A list of cities whose current temperature is above 10C is a rank-1 array, or vector, of unknown length, because the number of such cities is unknown, except at a particular instant during execution. This is an *AKD* array: an Array of Known Dimension. We know its rank, but its shape along each of those axes is unknown. Other examples are the results of database queries, the size of a document in a text editor, and the on-going trading history of a stock market instrument.

Many array language functions accept array arguments of arbitrary rank or shape. These arrays are of class *AUD*: Arrays of Unknown Dimension.

Thus, we see that array shape classes comprise a hierarchy which orders arrays by decreasing degrees of static knowledge about their structure and value: AKV, AKV, AKS, AKD, and AUD.

[1] A *length error* is signaled when array shape conformability requirements are violated, such as attempting to add a two-element list to a three-element list. An *index error* is signaled when an array index lies outside the bounds of the index set of the array being indexed.

2.1 Array Shape Classes and Performance

Arrays of class AKS are useful in situations where problems are of fixed size. Arrays of class AKD, with statically known rank, but unknown shape, dominate the application space in other areas where problem sizes change frequently. Arrays of class AUD are less common in applications, even though array language semantics encourage writing programs in a general, rank-independent manner that tends to produce AUD arrays. They tend to appear in standard libraries of utility functions, where their added utility simplifies the library holdings and provides substantial benefits to the users of those libraries. For example, a single function might support searches of vectors, matrices, and tensors, rather than requiring the library to contain a whole family of rank-specific functions. Some compilers, specifically `sac2c`, generate rank-specific (AKD) or shape-specific (AKS) versions of AUD functions when they are used in an application, so the AUD code may have vanished by the time compiler optimizations are applied.

As one might expect, the performance of applications written using different classes of arrays generally degrades as the amount of information available about the arrays themselves decreases, because fewer compiler optimizations and simplifications can be used. Optimization of AKS array-based code is fairly well understood; compilers for Fortran, SAC, and SISAL generate highly efficient code on such arrays. Unfortunately, despite the popularity of the AKD style of array programming, AKD-based performance is usually significantly worse than that of AKS, because many optimizations either do not, or can not, operate on AKD arrays. When AUD arrays do manage to survive in generated code, performance is generally even worse than that of applications using AKS or AKD arrays, because array rank and shape information usually must be maintained during execution.

Since many problems do not lend themselves to AKS-based solutions, better optimizations for non-AKS arrays could benefit application programmers. One way to improve those optimizations is to provide them with more semantic knowledge about the shapes of the arrays on which they operate.

3 Array Shape Analysis

Array languages generally perform run-time array shape checking to ensure that language semantics are not violated. For example, element-wise addition of a two-element vector to a three-element vector is forbidden. Compiler-driven removal of such run-time checks, when feasible, may have an impact on run-time performance that is all out of proportion to the number of such checks that are eliminated, due to their enabling effect on optimizations by a compiler. For instance, compositions of scalar functions should always be amenable to loop fusion and array contraction. Yet, the SISAL compiler does not perform loop fusion if array bounds do not match, or if run-time checks are present near those loops.

Similarly, the SAC compiler restricted several powerful optimizations, such as Index Vector Elimination [10] and With Loop Folding [3], to operate only on AKS arrays, even though non-AKS array operations could benefit from them. We found that extending those optimizations to some AKD and AUD arrays was fairly easy, despite the absence of precise knowledge of array shapes.

The key insight was recognition that many optimizations do not need precise knowledge of array shapes. Rather, all they require is knowledge that the array shapes are *identical*: if array shapes match, then the optimizations can be applied. Thus, the problem becomes one of inferring that two arrays that are candidates for optimization do, indeed, have the same shape.

The *shape-preserving* semantics of some array language primitives offer a way to determine that two arrays must be the same shape. In real applications, arrays do not arise out of thin air. Rather, they are usually formed through application of functions to other arrays, the new arrays often taking their shape from the shapes of one or more of the arrays that were arguments to those functions. Typical shape-preserving operations are *assignment* and its cousin, the *PHI* function of Static Single Assignment, monadic scalar functions, such as `signum`, dyadic scalar functions where one argument is scalar, such as `addition`, and structural functions, such as `rotate` and `reverse`.

These shape-preserving functions comprise a subset of the functions that are most amenable to this sort of analysis – the *uniform functions* [11]. whose result shape is a function *only* of their argument shape(s) – argument values play no part in determining the shape of the result.

4 Shape Cliques and Shape Clique Inference

The fact that the shapes of several arrays created by an application are often identical will not come as a surprise to anyone who is familiar with array languages. Set theory provides a simple way to organize those array shapes in a suitable form for use by a compiler: we break the set of arrays created by the application into subsets, each containing arrays that are *known* to be of the same shape, even if we do not know their exact shape. We denote these subsets as *shape cliques*.

We determine these subsets by placing each array in an application into its own set, forming one-element shape cliques.[2] Then, we examine the application, perhaps by walking its abstract syntax tree using standard data-flow-analysis methods, to find shape-preserving functions, assignments, and other places where result array shapes can be determined to be the same shape as the shapes of one or more of the function arguments. For any such functions, we perform a set union of the shape cliques of the appropriate argument(s) and results. The representation chosen for shape cliques is unimportant, as long as it facilitates rapid set membership and set union operations. When this *shape*

[2]By *each array in an application*, we mean each point in the program where an array obtains a value. In a tuple-based Static Single Assignment (SSA) representation [12] of an application, these points comprise all left-hand sides.

clique inference (SCI) process is completed, every array in the application will belong to a shape clique, each of which will contain at least one member.

Shape clique inference is inexact: if two arrays are in the same shape clique, then they are of the same shape. However, arrays in different shape cliques may, nonetheless, be the same shape, despite the fact that the inference process was unable to determine that they have matching shapes.

4.1 Simple SCI Example

We now present a brief example of how SCI works in the abstract, by analysing a fragment of an acoustic signal-processing program, written in a SAC-like language:

```
S = SIG+0.01;
Z = (S+1)*(S-rotate(-1,S));
```

We first transform the program into an Abstract Syntax Tree (AST) in tuple form, with one operation per line. We also initialize each shape clique to indicate which arrays have shapes in common, indicated by array names within parentheses. For simplicity, this examples ignores scalars:

```
S=SIG+0.01;
T1=rotate(-1,S);
T2=S-T1;
T3=S+1;
Z=T3*T2;
```

Cliques: `(SIG), (S), (T1), (T2), (T3), (Z)`

Next, we walk the abstract syntax tree, performing shape inference at each step. Element-wise array addition "`S=SIG+0.01;`" produces a result of the same shape as its non-scalar argument. Hence, we can place `S` and `SIG` in the same shape clique. The same type of inference on "`T3=S+1;`" places `T3` and `S` in the same shape clique:

Cliques: `(SIG, S, T3), (T1), (T2), (Z)`

Turning to the statement "`T1=rotate(-1,S)`", we note that `rotate` produces a result of the same shape as its second argument, so `T1` and `S` must lie in the same shape clique:

Cliques: `(SIG, S, T3, T1), (T2), (Z)`

The statement "`T2=S-T1;`" is a vector-vector operation. Arrays `S` and `T1` are already known to be the same shape, because they are members of the same shape clique, so we can make two inferences: first, we know that the result, `T2`, must lie in the same clique as the function arguments; second, we can elide the run-time conformability check for `subtract` because we have already determined that the array shapes will always match. The resulting shape cliques are now:

Cliques: `(SIG, S, T3, T1, T2), (Z)`

Finally, analysis of the statement "`Z=T3*T2;`" operates as in the previous step: Since the arguments to the multiplication operation are in the same shape clique, the result must be in the same clique, too:

Cliques: `(SIG, S, T3, T1, T2, Z)`

In this simple example, chosen for reasons of space, we ended up with all arrays in the same shape clique, and no iteration over the AST was required. Neither of these situations are likely to occur in practice, of course, due to array shapes really being different, because the semantics of the language and application do not give us adequate information to make stronger inferences, and because the presence of loop-carried variables may require several iterations to reach a fix-point.

5 Shape Clique Inference in APEX

Clearly, SCI is quite straight-forward (design and implementation of the APEX SCI code and using SCI data in an optimization took less than a day). A SAC-like model of the algorithm is shown in Figure 1.

We represent the AST for the program being compiled as an matrix of tuples, each containing a reference to a primitive or non-primitive function, its arguments and results, and metadata about the function. The AST is in SSA form, so each array created by the program has a unique name. Each tuple belongs to a specific shape clique, uniquely identified by the row index into the `AST` of the first member of that clique to appear in the `AST`. This representation permits optimizers to make unit-time determination of two arrays being in the same shape clique, at the cost of more work during SCI set union operations.

The SCI algorithm is integrated into the compiler's existing dataflow analysis (DFA) phase. Dataflow analysis iterates over the AST until types, ranks, and shape cliques reach a fix-point. Other useful data, such as shape, value, and array predicates are gathered at the same time, but they do not enter into the fix-point decision.

The `UnionShapeClique` function computes the union of two shape cliques, and updates the AST appropriately, with the new shape clique identifier, arbitrarily chosen as the minimum clique member index into the AST:

Loops and conditionals are handled by DFA in the SSA PHI functions; user-defined functions are each treated independently, after which shape clique information is exchanged between caller and callee functions by the normal inter-procedural dataflow mechanism in the compiler. Helper variables identify elements of each AST tuple: `astfunction` is the function being applied by this tuple; `astcliqueid` is the identifier for this tuple's shape clique; `astlarg` and `astrarg` are the left and right arguments, respectively, for the tuple's function.

5.1 APEX SCI Termination

Dataflow analysis terminates when the AST ceases to change. For purposes of SCI, this occurs when no more elements are being added to any shape clique.

```
do {
 /* For purposes of exposition, the algorithm
  * is shown as a sequential loop.  The APEX
  * implementation uses FORALLs across each switch type.
  */
for(row=0; row<ASTrows; row++) {
   oldast = AST;
   switch( AST[row; astfunction]) {
    case Assignment:
    case MonadicShapePreservingFunction:
    case DyadicSxAShapePreservingFunction:
     arg = AST[row; astrarg];
     AST = UnionShapeClique( AST, row, arg);
    case DyadicAxSShapePreservingFunction:
     arg = AST[row; astlarg];
     AST = UnionShapeClique( AST, row, arg);
    case SSAPHI:
     /* SSA PHI is limited here to two arguments,
      * but in general, all PHI arguments must be in
      * the same shape clique for the union to occur
      */
    case DyadicAxAShapePreservingFunction:
     if( AST[row; astlarg] == AST[row; astrarg]) {
      arg = AST[row; astlarg];
      AST = UnionShapeClique( AST, row, arg);
     }
    }
   endswitch;
  } until match(AST, oldast);
 }
}

int[.,.]  UnionShapeClique( int[.,.]  AST, int row, int arg)
{
 newid = min(AST[row; astcliqueid], arg);
 if (newid == arg)
    deadcliqueid = AST[row; astcliqueid];
 else
    deadcliqueid = newid;
 AST[row; astcliqueid] = newid;
 /* Renumber soon-to-be-gone clique */
 for(i=0; i<ASTrows; i++) {
  if( AST[i; astcliqueid] == deadcliqueid) {
   AST[i;astcliqueid] = newid;
  }
 return(AST);
}
```

Figure 1: Shape Clique Inference Algorithm

Since the only modification performed on the AST shape cliques are set unions, cliques can not shrink in size; they can only grow (through union with another clique) or disappear into another clique.

The AST comprises a finite number of tuples, each of which can belong to only one shape clique, so the algorithm must terminate within a number of set union operations that is smaller than the number of tuples in the AST.

5.2 Inter-procedural SCI in APEX

Initially, APEX SCI performed intra-procedural shape clique inference, but we extended it to inter-procedural inference (an hour or so of effort) when we found that it produced superior results in the presence of APL function composition (in the absence of inlining). To see why this is so, consider a SAC identity function, `ID`, whose result is its argument. The assignment:

```
A = B;
```

would place `A` and `B` in the same shape clique. However, the assignment:

```
A = ID(B);
```

would not, because the `ID` function call hides the shape of the right argument of `=`. We altered the shape clique identifier to handle inter-procedural array identification by the simple expedient of making the shape clique identifier comprise a function identifier and the AST row index.

The SAC SCI implementation, discussed in Section 6, uses intra-procedural analysis, partly due to the architecture of the SAC compiler, and partly because inlining of application code makes the problem, in practice, largely a non-issue, as our performance results, below, will show.

5.3 Run-time Check Removal

In order to measure the quality of shape clique inference, we used shape clique information to extend the APEX compiler's run-time-check optimization. The compiler emits conformability checks for each dyadic scalar function invocation, unless some static knowledge of the specific function invocation permits elision of the check. For example, the presence of certain array predicates [13], scalar extension of one argument, or statically known matching argument array shapes (AKS) enable such elision. The run-time-check optimization removes, when possible, dyadic scalar function conformability (length error) checks from APEX-generated code.

We extended the APEX compiler so that it also elides these checks when both function arguments are members of the same shape clique. In addition, we generated diagnostic comments, to facilitate counting the number of eliminated checks and identifying the cause(s) of their elimination.

5.4 Run-time Check Removal Performance

We evaluated the performance of our SCI tool and extended optimizer by running the APEX compiler against 156 performance benchmarks, then counting the number of run-time dyadic scalar function conformability checks that were eliminated by exploiting shape clique information:

Type of check	# of checks	Percentage
Total# of dyadic scalar calls	1251	100%
Removed by SCI analysis	86	6.9%
Removed by scalar detection & SCI analysis	55	4.4%
Removed by scalar detection	853	68.2%
Remaining run-time checks	257	20.5%

Shape clique analysis permitted removal of 6.9% of the original run-time checks, or about 25% of those checks remaining after other removal tools had done their job. Shape clique analysis also found 55 sites where scalar detection had redundantly removed the same check.

We were unable to measure the performance impact of this optimization directly, because the SISAL back end of APEX was not operational at the time. However, performance gains resulting from loop fusion and array contraction are usually substantial. Such gains would occur wherever run-time conformability checks were removed, if the fused operations were on reasonably large arrays.

As we expected, shape clique analysis missed some scalar-scalar or scalar-non-scalar cases that were detected by APEX's extant data flow analyzer, so shape clique analysis is not a complete replacement for other inference tools.

6 Extended Index Vector Elimination in SAC

The SAC *with-loop* permits rank-independent specification of data-parallel loop nests. Each N-element *index vector* generated by a with-loop is an accessor to an element or sub-array element of an array. Index vectors are useful from the standpoint of algorithmic expression, but their presence in compiled code degrades performance, so the SAC compiler Index Vector Elimination (IVE) optimization [10] attempts to remove them, replacing them by array offsets or by shared index operations.

Originally, the SAC compiler performed IVE only on AKS arrays. Our APEX benchmarks, mostly AKD in nature, produced unexpectedly high execution times under SAC, and we suspected that IVE was a culprit in that poor performance. Encouraged by the results obtained with the APEX implementation, and by a suggestion from Sven-Bodo Scholz, we decided to implement SCI in the `sac2c` compiler, using its inferences to extend IVE to operate on some AKD and AUD arrays. We knew that AKD and AUD performance would be poorer than than obtained on AKS arrays, due to fewer opportunities to perform constant folding and constant propagation, but were pleased by the results of our work. The author and Stephan Herhut performed that implementation in

June 2006, then conducted experiments to measure the performance of compiled SAC code with, and without, the benefit of SCI-assisted IVE.

6.1 Experimental Framework

We conducted our tests on an AMD-based platform (Opteron 165 (1.8GHz)) equipped with 4GB of RAM, running SuSE Linux 10.1 64-bit. We used the current `sac2c` compiler (rev 15076) with the GNU `gcc` compiler version 4.1.0 as the back-end compiler. We enabled standard `sac2c` optimizations, compiling the resulting C code with the `-O3` option of `gcc`. We modified the SAC compiler so that we could operate it with SCI enabled or disabled, to simplify timing tests.

Our tests were chosen from a set of SAC numerical AKS benchmarks, and from a set of APEX-generated AKD benchmarks, modified to give us AKD- and AKS-based versions. The benchmarks, summarized in Figure 2, are vector-based (rank-1), unless specifically noted as 2-D (rank-2 matrix) or 3-D (rank-3 tensor).

We ran each benchmark five times, discarding the highest CPU time of those runs, to mitigate the effects of dynamic linking overhead. We used the Linux `/usr/bin/time` command to measure CPU (USER) time.

6.2 Experimental Results

Figure 3 and Figure 4 show the speedup in CPU time for each benchmark with SCI enabled and disabled. All execution times are scaled so that the IVE-disabled times correspond to a Y-axis value of 1. The black (lowest) bars indicate the speedup obtained, relative to IVE disabled, with IVE enabled and SCI disabled; the yellow bars stacked on top of those show the additional speedup obtained when both SCI and IVE are enabled.
The performance of IVE with and without SCI enabled is quite variable. In about a third of the AKS cases (*e.g.*, `ipape`, `lltop`), SCI is unable to improve performance. In all but a few of the AKD benchmarks (*e.g.*, `pde1`), enabling SCI improves performance, typically between a factor of 2-4X, but occasionally well in excess of that, as in the case of `relax_fixAKD`, which ran about 14X faster. We suspect that the unrelated changes in `sac2c` impeded IVE operation onf `pde1`, but have not had the luxury of time to investigate this problem. Surprisingly, in a few AKS cases (*e.g.*, `scs`, `rle`, `sdyn4`, `APLtomcatv`), enabling SCI produces a good improvement in performance.

The range of speedup is heavily influenced by the particular coding style used in the benchmark. For example, although the `APLtomcatv` benchmark has AKS arguments, the style of the SAC code results in a number of AKD arrays being created. SCI allowed IVE to eliminate all but three of the 114 `sel` (array selection operations using index vectors as an argument) operations left by the SCI-disabled IVE, contributing to its improved performance.

Benchmark name	**Description**
APLlogd1	hand-coded SAC acoustic signal processing)
APLtomcatv	2-D SPEC tomcatv, APEX-generated, hand-tuned
downgradePV	downgrade of permutation vector
dtb2	drop trailing blanks from 2-D character matrix
histlp	histogram written as loop
histop	histogram written as outer product
ipape	inner product on 2-D character matrices
ipbb	inner product on 2-D Boolean matrices
ipbd	inner product on 2-D Boolean-Double matrices
ipdd	inner product on 2-D Double-Double matrices
lltop	linked-list to permutation vector
logd2	acoustic signal processing
loopis	simple integer scalar loop
matmul	inner product on 2-D Double-Double matrices
mconv	one-dimensional convolution
nmo	2-D geophysics normal move-out calculation
nthone	find n^{th} one in Boolean vector
pde1	3-D Red-Black Poisson solver
primes	prime number finder
relax_fix_rotate	2-D relaxation model
relax_fix	2-D relaxation model
rle	run-length encoding
schedr	J.L. Ryan solution to Roger Hui scheduler problem
scs	2-D string shuffle problem
sdyn4	2-D dynamic programming
tjkc	2-D financial market Julian calendar
unirand	random-number generator
upgradeBool	upgrade Boolean vector
waver	3-D water wave motion model

Figure 2: Benchmark characteristics

7 Other Applications of Shape Clique Inference

We have presented some of the optimizations that can be performed with shape clique inference information, but it may be useful to note a few other potential applications for SCI inferences

In SAC, shape clique inference could be used to extend with-loop folding [14] to some AKD and AUD arrays.

Shape clique information can reduce the number of run-time index-error checks that have to be performed in an application. Consider one possible scenario that would be facilitated by use of SCI information, as in this SAC example, where we assign the values in the list `V3` to the first `shape(V2)` elements of a third vector, `V1`.

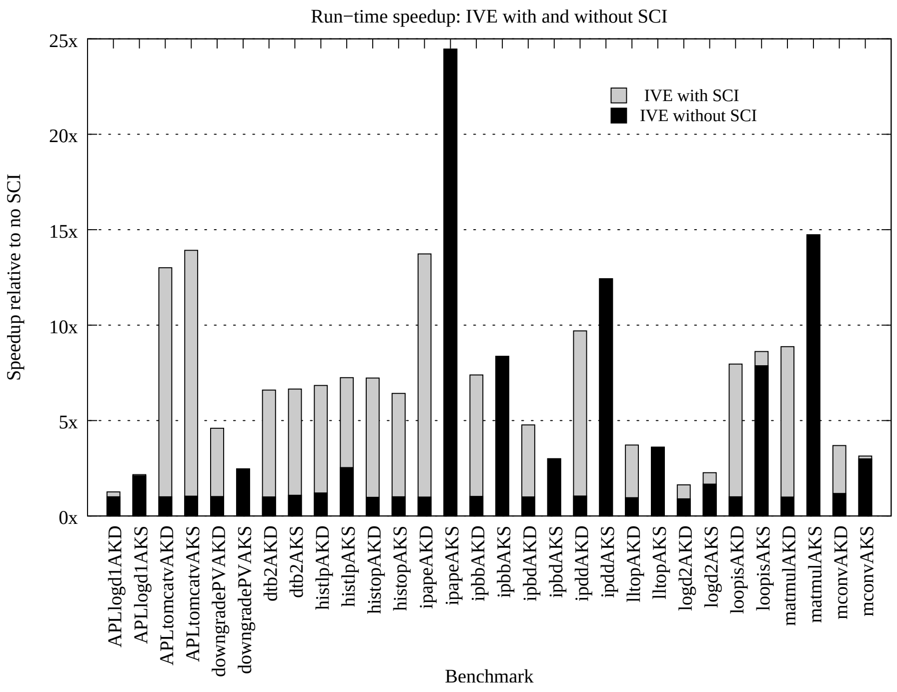

Figure 3: Performance of IVE with and without SCI

```
V1 = iota([10]);
V2 = [2,4,6,8,9];
V3 = [10,20,30,40,50];
Z = with(0*shape(V2) <= iv < shape(V2))
       modarray(V1, V2[iv], V3[iv]);
```

Normally, this operation would require two run-time checks: First, an index error check, to ensure that the appropriate elements of `V2` are all non-negative, but less than `shape(V1)`. Array predicates [13] could be used to determine that `V2` is an Arithmetic Progression Vector (or subset thereof). In that case, this run-time check of all index array elements could be replaced by a single shape check:

```
shape(V2) <= shape(V1)
```

Moreover, if `V1` and `V2` are in the same shape clique, then that error check can also be elided.

The second run-time check is a length-error check, to ensure that the shape of `V3` is no less than the shape of `V2`:

```
shape(V3) >= shape(V2)
```

Similarly, if `V1` and `V3` are in the same shape clique, and the length-error check has been performed or proved nugatory, then this run-time index-error check is also nugatory.

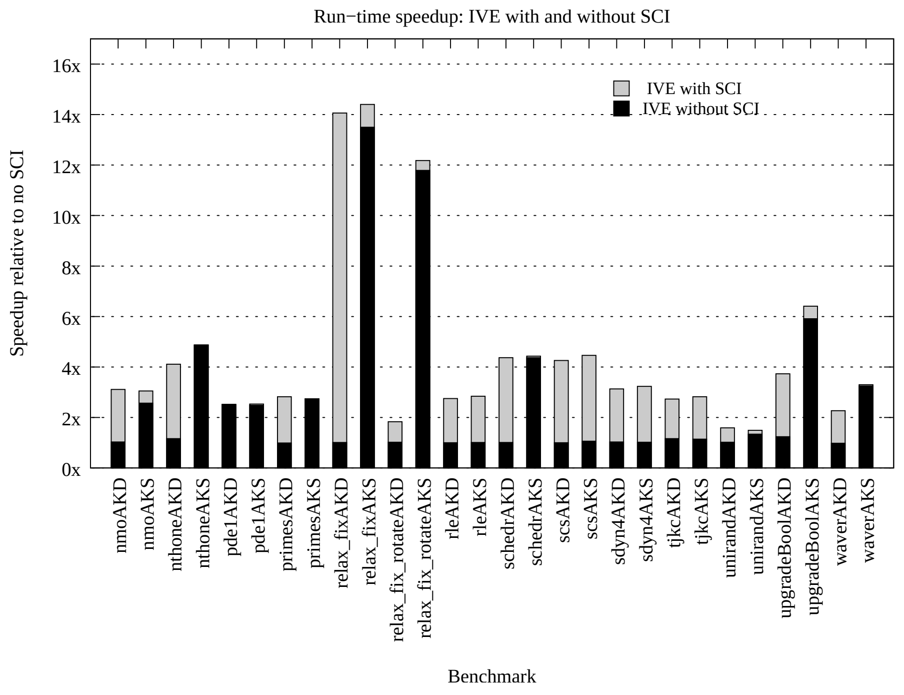

Figure 4: Performance of IVE with and without SCI

8 Guarded Shape Cliques

The semantics of some array languages require that all non-scalar arguments to dyadic scalar functions must match in shape. So, it is perhaps surprising that the APEX SCI tool does not infer anything about dyadic scalar function calls with non-scalar arguments, except when the arguments are already known to be members of the same shape clique. It is problematic to do better than this. Consider a SAC statement such as:

```
B = [2, 3];
C = [4, 5, 6];
Z = B + C;
```

One would think that, since non-scalar arrays `B` and `C` must match in shape, and since the shape of `Z` will match the shape of its arguments, all three arrays should belong to the same shape clique. However, that is true only *after* the operation, including a length error check to enforce the semantics of element-wise array addition, successfully completes.

Contemplate what would happen if shape clique inference blindly placed all three arrays into the same shape clique: code generation would generate, erroneously, array addition code that did not include the requisite length error check, based on the claim made earlier by shape clique inference, that `B` and `C` were in the same shape clique. This could cause the program to crash or produce incorrect answers.

Hence, we must not blindly place these arrays in the same shape clique. Is all lost, or can we rescue some information from the semantics of array addition? With some effort, and the power of Static Single Assignment (SSA), we can indeed get useful information from the array-array addition operation. We proceed by rewriting the program slightly, replacing the original text:

```
Z = B + C;
```

by this:

```
A',B' = Guard( A, B);
Z = B' + C';
```

and replacing all references to `A` and `B` that are dominated by `Guard` by references to `A'` and `B'`. The use of SSA makes the latter a simple operation.

This change decouples references to `A` and `B` from operations that are dominated by `Guard`, and it eliminates the need for us to consider operations that are *not* dominated by `Guard`. We are now left with a new function `Guard`, with the following definition, assuming that `A` and `B` are integer arrays:

```
int[+] int[+] Guard( int[+] A, int[+] B)
{ if ( !all(shape( A) == shape( B))) {
    print( tochar("length error on A, B"));
    exit(666);
    }
  else return (A, B);
}
```

That is, if the shapes of `A` and `B` match, `Guard` merely returns its arguments, unchanged. If the shapes do not match, a run-time error is raised.

Although it appears that `Guard` merely separates the length-error check on `A` and `B` from the code that adds those arrays, we have introduced new arrays (`A'`, `B'`, and `Z`), all of which are in the same shape clique, because the `Guard` operation ensures that to be the case: the execution of `Guard` may complete, because the shapes of `A` and `B` match, in which case any later code generated on that assumption will execute problem-free, because it is dominated by `Guard`. If the execution of `Guard` fails, because the array shapes do not match, an error will be raised, so the potentially erroneous code that follows `Guard` is never executed. Meanwhile, SCI can proceed in the knowledge that arrays `A'`, `B'`, and `Z` are in the same shape clique.

At least two problems remain here: first, what about references to `A` and `B` outside the dominator tree of `Guard`? Second, what if a later inference by SCI determines that `A` and `B` are, in fact, in the same shape clique? The first problem may be solved when SCI examines other parts of the application; the second problem is solvable as follows: if a later stage of SCI was to discover that, for instance, `A` and `B'` are, in fact, in the same clique, then the code generator could eliminate `Guard` calls that involve those two arrays, by replacing the names of the arrays in all expressions of the form `shape(X)` with the name of the *typical* member of the shape clique to which `X` belongs. The particular member of the shape clique denoted as *typical* is unimportant, as long as it is always the same member. Thus, if the typical member of the shape clique for `A` and `A'` was `TYP`, the shape comparison in `Guard` would turned into:

```
    if ( !all(shape( TYP) == shape( TYP)))
```

Common subexpression elimination, constant folding, and other standard optimizations would trivially remove this instance of `Guard` from the generated code.

9 Related Work

Shape cliques represent a subset of a more general topic in array languages - array shape analysis. The FISh language [15] deduces array shapes for entire applications, but it only works in an AKS environment, supports only uniform functions, and it segregates values from shapes, which limits its utility to some degree. The IBM pHPF research compiler:

> ...performs symbolic analysis to generate efficient code in the presence of statically unknown parameters like the number of processors and array sizes [16].

Unfortunately, their paper does not offer specifics about array shape analysis. The PARADIGM compiler for MATLAB [17], like FISh, has to be able to deduce statically all array shapes in a program.

Trojahner, *et al.*, recently implemented an elegant method of symbolic shape analysis (SAA) for SAC [18], which should be able to provide superior information to that provided by SCI. Nonetheless, the utter simplicity of SCI and the significant performance improvements it offers may give it a continued role in array language optimizations.

10 Future Work

We intend to continue investigation of some of the optimizations and run-time error removal work mentioned earlier, and are also investigating other methods for improving the quality of shape clique inference, including *Guarded Shape Cliques*. We also plan to investigate optimizations on arrays whose shapes are only partially known (*e.g.*, a matrix with two columns, but an unknown number of rows), as well as on arrays with known relative shape characteristics (*e.g.*, one array has fewer rows than another, but the same number of columns).

Our APEX implementation supported run-time check removal only for the dyadic scalar functions; there are opportunitities to extend run-time check removal to other families of functions.

Although the concept of shape cliques offers substantial performance improvement for non-AKS problems, the cost of SCI is excessive: it does too much work, examining all arrays in an application, even though optimizers are likely only interested in the shape properties of a few of those arrays. In this regard, other shape-analysis techniques should be able to provide equivalent information with less effort.

11 Summary

We have introduced *shape cliques*, a concept for discussing equivalent array shapes, showed a simple method for organizing the arrays in an application into sets of identical array shapes, and demonstrated a shape clique inference algorithm for inferring membership in those shape cliques.

We also showed that SCI allowed removal of 25% of the dyadic scalar function run-time checks remaining in compiled APL after other removal optimizations.

Finally, we showed that SCI can provide information to extend the capability of Index Vector Elimination to operate on AKD and AUD arrays, resulting in performance improvements of up to 14X.

12 Acknowledgments

The structure and content of this paper benefited considerably from comments by Stephan Herhut, Sven-Bodo Scholz, Clemens Grelck, and the anonymous IFL2006 referees. We are grateful for support from Future Perfect, Ltd.

References

[1] International Standards Organization: International Standard for Programming Language APL. ISO N8485 edn. (1984)

[2] Hui, R.K., Iverson, K.E.: J Dictionary. (1998)

[3] Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming **13**(6) (2003) 1005–1059

[4] Grelck, C., Scholz, S.B.: SAC: A Functional Array Language for Efficient Multithreaded Execution. International Journal of Parallel Programming **34**(4) (2006) 383–427 to appear.

[5] Cann, D.C.: The optimizing SISAL compiler: Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory (1992)

[6] Nguyen, T.V., Irigoin, F.: Efficient and effective array bound checking. ACM Transactions on Programming Languages and Systems **27**(3) (2005) 527–570

[7] Kreye, D.: A Compilation Scheme for a Hierarchy of Array Types. In Arts, T., Mohnen, M., eds.: Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL'01), Stockholm, Sweden, Selected Papers. Volume 2312 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (2002) 18–35

[8] Shafarenko, A., Scholz, S.B., Herhut, S., Grelck, C., Trojahner, K.: Implementing a numerical solution for the KPI equation using Single Assignment C: lessons and experience. In Butterfield, A., ed.: Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05. Volume 4015 of LNCS., Springer (2006) to appear.

[9] Bernecky, R.: APEX: The APL Parallel Executor. Master's thesis, University of Toronto (1997)

[10] Bernecky, R., Herhut, S., Scholz, S.B., Trojahner, K., Grelck, C., Shafarenko, A.: Index Vector Elimination – making index vectors affordable. In Horváth, Z., Zsók, V., eds.: Proceedings of the 18th International Symposium on Implementation of Functional Languages (IFL'06), Budapest, Hungary, Eötvös Loránd University (2006) 23–43

[11] Hui, R.K.: Rank and uniformity. ACM SIGAPL Quote Quad **26**(1) (1995) 83–90

[12] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method for computing static single assignment form. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages. (1989) 23–35

[13] Bernecky, R.: Reducing computational complexity with array predicates. In Picchi, S., Micocci, M., eds.: APL98 Conference Proceedings, APL Italiana (1998) 46–54

[14] Scholz, S.: WITH-loop-folding in SAC – condensing consecutive array operations. In: IFL'97, University of St. Andrews, UK, LNCS 1467, Springer-Verlag (1998) 72–91

[15] Jay, C.: comp.lang.functional:11066 (1998)

[16] Gupta, M., Midkiff, S., Schonberg, E., Seshadri, V., Shields, D., Wang, K.Y., Ching, W.M., Ngo, T.: An HPF compiler for the IBM SP2. In: Proceedings of the 1995 ACM/IEEE Supercomputing Conference. (1995)

[17] Ramaswamy, S., Hodges, E., Banerjee, P.: Compiling MATLAB programs to SCALAPACK: Exploiting task and data parallelism. In: Proceedings of the International Parallel Processing Symposium (IPPS). (1996) 613–620

[18] Trojahner, K., Grelck, C., Scholz, S.B.: On Optimising Shape-Generic Array Language Programs using Symbolic Structural Information. In

Horváth, Z., Zsók, V., eds.: Implementation and Application of Functional Languages, 18th International Workshop (IFL'06). Budapest, Hungary, September 4006, 2006, Revised Selected Papers. Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York (2006)