Reducing Computational Complexity with Array Predicates *

Robert Bernecky
Snake Island Research Inc
18 Fifth Street, Ward's Island
Toronto, Ontario M5J 2B9
Canada
+1 416 203 0854
bernecky@acm.org

Abstract

This article describes how *array predicates* were used to reduce the computational complexity of four APL primitive functions when one of their arguments is a permutation vector. The search primitives, *indexof* and *set membership*, and the sorting primitives, *upgrade* and *downgrade*, execute in linear time on such arguments. Our contribution, a method for static determination of array properties, lets us generate code that is optimized for special cases of primitives. Our approach eliminates runtime checks which would otherwise slow down the execution of all cases of the effected primitives. We use the same analysis technique to reduce the type complexity of certain array primitives.

1 Introduction

The search for terse, elegant algorithms is part of the APL mythos. This passionate quest arises, in part, from APL's history as a language for teaching mathematics and from the fact that array-based languages, such as APL and J, naturally lend themselves to the description of non-iterative algorithms. However, another cause of the quest is that APL programmers *need*, rather than desire, non-iterative solutions to such problems, because of the poor performance of interpreter-based commercial APL systems on highly iterative programs. This performance problem arises from the significant setup time and storage management overhead associated with the execution of each primitive in a program. Typically, this means that the execution time of an application is dominated by the number of APL primitives executed, rather than by the size of their array arguments.[Ber97a, Wil91, Jor79]

1.1 Computational Complexity

As we shall see, non-iterative algorithms may have higher computational complexity than iterative ones. The *computational complexity* of an algorithm is a function that tells us the number of steps a serial computer requires to apply the algorithm on an argument of given size.[GJ79] These functions are generally expressed as polynomials or exponential func-

*This paper originally appeared in the APL98 Conference Proceedings. [Ber98]

tions of the argument size. For example, the computational complexity of matrix product on square matrices of order `N` might be given as `k×N*3`, where `k` is a scaling constant having to do with the efficiency of an implementation. More often than not, such constants are omitted, because we are more interested in the order of complexity and finding ways to reduce it.

2 Non-iterative Algorithms

APL programmers seek non-iterative algorithms for reasons beyond performance, because non-iterative algorithms are usually terse, enlightening, and easier to prove correct than iterative ones. Unfortunately, when non-iterative algorithms have higher computational complexity than iterative ones, the APL programmer faces a conundrum. Interpretive overheads make iterative algorithms too slow, yet non-iterative algorithms may be even slower. Their computational complexity makes them impractical for many realistic applications: execution time and memory requirements are unacceptably high. A simple example of this can be found in the *string shuffle* problem, which has exponential time (2^N) and space complexity when solved using a brute-force, non-iterative APL algorithm, but which can be solved in quadratic time (N^2) with an iterative dynamic programming algorithm. These computational complexity problems can sometimes be alleviated, without recourse to explicit iteration or recursion, by designing subtle algorithms that exploit such classical methods as sorting, divide-and-conquer, and parallel-prefix operations, including the APL *scan* adverb.

Unfortunately, these subtle methods generally produce unsatisfying programs for reasons that go beyond their pooer performance. Their didactic nature and feeling of mathematical correctness is lost. Furthermore, they may introduce undesired errors, due to edge conditions, unforseen loss of precision, or unexpected overflow.

Designers and implementors of APL systems have made some progress in dealing with these performance problems, but the state of the art remains inadequate to handle them all. Recently, we made one more step along that road, by creating an APL compiler, APEX, that solves some of these problems in a general manner. APEX reduces the computational complexity of certain APL expressions to that of iterative, scalar-oriented algorithms, while preserving the didactic clarity of expression so dear to APLers.[Ber97a]

3 Reduced Complexity in Compiled Code

APEX, a state of the art APL compiler, uses optimization techniques including loop fusion, code motion, common sub-expression elimination, strength reduction, stack allocation of scalars, and reference count elimination to improve the performance of APL programs. These optimizations reduce the scaling constants associated with APL programs by eliminating syntax analysis, avoiding creation of array-valued temporary results, reducing or eliminating setup and dispatch cost, and by reducing or eliminating memory management operations.

These improvements in generated code culminate in compiled APL applications that run up to 1000 times faster than they do under an interpreter, yet those dramatic improvements are achieved, by and large, without reducing the computational complexity of the program. The computational complexity of the underlying APL primitives generally remains the same. If a programmer expresses an algorithm using an *upgrade*, that *upgrade* and its associated execution time will still be visible in the execution of the generated code.

Our task, to attack those complexity problems and reduce them to manageable size, is partly solved by methods we describe here. We have made progress in improving the performance of *upgrade*, *downgrade*, *indexof*, and *set membership* on a certain class of arguments. In all cases, the resulting primitives execute in linear time. We now turn to the history of this effort.

4 Array Predicates

During the development of the APEX APL Compiler, we were studying the performance of a run-length encoding (RLE) benchmark used by a large financial data supplier to compress massive amounts of time-series data, such as historical records of trading information for the world's many stock exchanges. We predicted the benchmark would execute somewhat faster when compiled than when interpreted, because the APL code was well-designed, non-iterative, and short. Yet, the compiled code ran 15 times slower than it did under the interpreter! Clearly, something was very wrong with the compiler-generated code.

Immediately upon examination of the generated code, the cause became clear. The expression `(N⍴2)⊤intvec` was the culprit. A sophisticated interpreter, such as SHARP APL, would have produced a Boolean result for this case, but the compiler was generating code that emitted an integer result. All further computation using this result was in the integer domain, so the performance of the remainder of the application suffered accordingly.

At first, we suspected a simple result-type determination error in the APEX data flow analyzer, but we soon realized that a fundamental design problem was at hand: the compiler had no way to deduce that the left argument to the *represent* verb (`⊤`) was a vector of 2s. In this case, a base-2 (Boolean) type result is called for, yet the compiler was forced to generate code that emitted an integer-type result.

To see how this situation arose, consider how an interpreter with special-case run-time analysis code in the *represent* primitive would work, compared to the static analysis performed by a compiler. The interpreter would examine the left argument to *represent* at run-time, determine that it consisted entirely of 2s, and execute a fast algorithm that produced a Boolean result. The compiler, by contrast, used data flow analysis (DFA) to infer the type of the left argument to *represent*.[Ber97a, Ber93] In the right argument to the *reshape* expression `N⍴2`, the constant `2` is of type integer. Hence, the result type of the *reshape* expression is also integer. Data flow analysis within the compiler makes this determination statically, just as the interpreter does dynamically. However, since we do not know the *value* of `N` until run-time, we do not know the value of the array resulting from the *reshape*. Hence, the compiler is unable to use partial evaluation at compile time, as it would if the expression were derived from a constant expression such as `8⍴2`.

The compiler knows from DFA that the right argument to *represent* is integer. The previous inference on *reshape* tells us that the left argument to *represent* is also integer. Hence, DFA type determination must predict an integer result for *represent*. Because the compiler has no knowledge of the actual values in the left argument, it has no way to predict the more space- and time-efficient Boolean result.

We recognized that a programmer looking at the expression `N⍴2` would immediately deduce that its result will always consist entirely of the integer `2`. Could we embed that sort of knowledge in the compiler in a general way? We thought a bit about the analysis that the programmer did here, and came to the realization that there are two facets to this form of analysis:

- First, arrays have a number of properties, which we dubbed *array predicates*, that may be of interest to a primitive.

- Second, array predicates are *created*, *destroyed*, and *propagated* by each primitive, in a manner that is dependent on the property and on the primitive.

The array predicate of interest in this case was that the *array consists entirely of the integer* `2`. This property was created by analysis of the constant `2`. It was then passed to the data flow analyzer for reshape, which propagated it on the basis that the set of values generated by reshape is always a subset of its right argument. The property was then used to advantage by the code generator for *represent*, which was able to generate a Boolean result. The array and its associated array predicate were no longer needed at this point, but now *represent* created a Boolean result, so the remainder of the application was now able to generate Boolean results.

To confirm our hypothesis, we spent an hour or so modifying the compiler to include support for array predicates. We then re-compiled and re-ran the RLE benchmark. The compiler now properly emitted code that generated Boolean results. The compiled version of RLE executed 24% faster than it did under the interpreter. Not a great win, but a great improvement over its predecessor. Furthermore, this single special case substantially improved APEX performance on other benchmarks.

For example, a cyclic redundancy check (CRC) benchmark that exploited array predicates and other optimizations ran 46–98 times faster than the interpreter.[1] Nearly all of the remaining execution time for the compiled version of CRC took place in initialization code. When we extend partial evaluation in APEX, that initialization code will effectively vanish – it will be performed once during compilation. At that juncture, the compiled version of CRC will run 500–1000 times faster than the interpreted version.

Given the substantial performance boosts we obtained with such minimal effort, we sought other possible candidates for array predicates, coming up with the list shown in Figure 1. Some of these are discussed in detail in related papers.[Ber97a, Ber97b] This obvious list is not exhaustive; it is merely a starting point.

Predicate	Description
PV	permutation vector
PVSubset	subset of permutation vector
NoDups	elements all unique
All2	elements all integer 2
SortedUp	elements in upgrade order
SortedDown	elements in downgrade order
KnowValue	value known at compile time
NonNeg	elements all non-negative
Integer	elements all integer-valued

Figure 1: Array predicates supported by APEX

The opportunity to exploit another predicate fortuitously arose when we were offered the opportunity to study a performance problem being experienced by a large European financial institution. We now turn to that study, which led us to exploit array predicates for improving the performance of APL's searching and sorting primitives.

5 Fast Searching and Sorting

A German bank was interested in improving the performance of a large securities swapping application that consumed several hundred hours of mainframe processor time each month. Initially, we worked

[1] The CRC benchmark was kindly supplied by Leigh Clayton of Soliton Associates Limited.

with the software firm that provided development and maintenance support for the application to identify hot spots in this APL application. Next, we built driver functions to exercise these hot spots, then compared the performance of the compiled code to that of the same application running under the interpreter on the same machine.

5.1 Improved Indexof Performance

We found that the performance of our *interpolation* benchmark, DBT5, was heavily dominated by the execution of two primitives, *upgrade* (⍋) and *indexof* (⍳). One of the two *upgrade*s was used to sort the incoming data; the *indexof* was used to restore the data to its original order at the end of the interpolation operation. The domination of execution time by two well-researched and highly optimized primitives created a dilemma for us. How could a compiler improve the performance of an application that was spending most of its time in well-written, compiled interpreter code that was itself already optimized?

In the course of studying the benchmark, we noted that the *indexof* operation was using the result of the second *upgrade* as its left argument. That is, one part of the benchmark performed an expression of the form `X[si←⍋X;]`; a later one performed `si⍳Y`. Seeing this, we realized that this case offered us a golden opportunity to reduce the computational complexity of the *indexof* primitive.

At this point, we digress slightly to discuss how *indexof* is implemented on typical APL interpreters. In a well-designed APL implementation, the *indexof* primitive (`α⍳ω`), applied to an integer left argument, takes time that is either roughly linear in the size of ω, with an order O(`(×/⍴α)×(×/⍴ω)`) worst case, if *indexof* is implemented using hash tables. If *indexof* sorts its left argument, then uses binary search to find elements of the other argument in that sorted array, the complexity is of order O(`(⌈2⊛×/⍴α)×(×/⍴α)+(×/⍴ω)`).

Back in the dark ages, we proposed hashing algorithms as an effective way to implement APL search primitives.[2][Ber73] In that era, APL algorithms for *indexof* and *membership* were linear searches that took excruciating amounts of time to search arrays that would fit in 48 kilobyte workspaces. Nowadays, the unpredictable, abysmal worst-case performance of hashing algorithms makes them a poor choice, particularly on the multi-megabyte arguments that now fit handily into PC memory.

Furthermore, the poor locality of spatial reference of hashing algorithms wreaks havoc with cache memory. Their relative speed advantage over algorithms based on binary search, never more than a factor of about two on realistic data, lessens day by day as the relative cost of a cache miss increases with newer processors. In summary, we note that any modern interpreter will execute *indexof* at a good clip, so we return to the problem of beating the performance of a well-designed primitive using compiled code.

The *upgrade* primitive, by definition, produces a permutation vector as its result. From this fact, we deduce that the result of *upgrade* has no duplicates and that the elements in an N-element result are the first N integers. With these facts in hand, we realized that we could use a pigeon-hole algorithm to perform *indexof* on permutation vectors in linear time, by building an N-element table, `T`, from the permutation vector left argument, then indexing `T` with elements of the right argument:

```
T←(⍴α)⍴0
T[α]←⍳⍴α
Z←T[ω]
```

The algorithm as implemented had to perform limit checks on elements of the right argument to en-

[2]It took two decades, but all major APL interpreter vendors now all have fast versions of *indexof*.

sure that they were members of ⍳⍴⍺, but the two requisite compare operations did not materially degrade performance. The result was that we were able to generate code that executed in time of order O((×/⍴⍺)+(×/⍴⍵)), significantly faster than the interpreter algorithms. Also, because these algorithms make only one storage reference per element they are much kinder to cache memory than are hash-based algorithms, which may make an large number of them.

The result of using array predicates to detect cases where a simpler *indexof* algorithm can be used was substantial. The compiled DBT5 benchmark now executed 3.1 times faster than the interpreted code, in spite of the fact that an APEX compiler back-end deficiency caused the compiled version of *upgrade* to perform quite poorly. We expect the relative performance edge of APEX on this benchmark to be 6–10 times faster once the *upgrade* problem is repaired.

5.2 Improved Upgrade Performance

We ran into another case of execution time being dominated by a single primitive in the DBT2 *probability normalization* benchmark. The DBT2 benchmark contained two *upgrade*s, which accounted for 92% of its execution time on the mainframe. As with the DBT5 benchmark, we observed that the result of one *upgrade* was later used in another computationally intensive computation, in this case, the second *upgrade*. We applied the same technique of generating an optimized *upgrade* algorithm for permutation vectors. In this case, it was even simpler than the *indexof* algorithm, and did not require any limit checks on the argument:

```
z←(⍴⍵)⍴0
z[⍵]←⍳⍴⍵
```

This simple optimization made DBT2 execute 20% faster under APEX than under the interpreter. As with DBT5, it suffered from poor performance of the remaining *upgrade*, which consumed 75% of the benchmark's CPU time. Again, we expect significantly better performance when the *upgrade* problem is resolved.

6 Complexity Reduction in Interpreters

Can techniques based on array predicates be applied effectively in APL interpreters to reduce the computational complexity of primitives in much the same way that APEX does it? We do not believe so, at least for the naive interpreters in commercial use now.

The main performance problem faced by interpreters today is the inordinate amount of overhead associated with execution of a single primitive function.[Ber97a, Wie86, Jor79] Introduction of array predicates into the underlying array structures of an interpreter would increase these overheads, because every primitive in the interpreter would have to create, propagate, or destroy predicate values, even if there was no chance of the predicate ever being put to effective use. If, for instance, predicates increase the cost of dispatching a primitive by 5%, but they can only be exploited in 1% of the primitives executed, then the net effect of their introduction will be to slow down most applications, because almost all applications have execution time profiles that closely match the number of primitives executed. The tradeoffs here are quite similar to those in CISC versus RISC computer architecture, and the evidence points the same way: adding features to every code path in an interpreter to detect special cases is usually a losing proposition from the standpoint of system-wide performance.

Things may not be as bad as they seem. We think the best of both worlds can be achieved by using just-in-time (JIT) background compilation tech-

niques within interpreters to get most of the performance benefits of compilation for production applications, without losing the interactive facilities of APL. The history of APL interpreter and compiler implementations makes it clear that such tools must use sophisticated, optimizing compiler-based methods, because the largest performance gains arise from their use.

7 Performance of Related Primitives

The algorithms just described apply equally well to the APL *set membership* primitive and SHARP APL *nubsieve* and *less* primitives, because they can be defined on vector arguments in terms of *indexof*: [3]

```
mem: (α∈ω)≡(ω⍳α)≠¯1+⎕io+⍴ω
nubsieve: (⍳⍴ω)=ω⍳ω
less: (~α mem ω)/α
```

Similarly, the array search and string search primitive *find* (⍷) can be executed faster on a permutation vector right argument, because there can only be a single occurrence of the left argument within the right argument. Hence, the generated code can stop looking for more matches after it finds the first one.

Finally, indexing an array with a permutation vector need not perform array bounds checking within the indexing loop. Instead, a single check for *index error* can be made by merely examining the shape of the permutation vector.

Other optimizations of this type are clearly possible. In this paper, we merely point out some of the more obvious ones.

[3] Of course, the *nubsieve* primitive applied to a permutation vector `P` is merely `(⍴P)⍴1`. This results in a special case that may be interesting for sales and marketing benchmarks, but that is rarely, if ever, used in real applications.

8 Summary

Exploiting array predicates in a compiled APL environment allowed us to improve the storage and CPU-time efficiency of a class of primitive functions. Similar uses of array predicates allowed us to reduce the time complexity of the APL searching and sorting primitives to linear time when one of their arguments is a permutation vector. The analysis is performed entirely at compile time, so there is no adverse impact on the execution time performance of the general-case primitives, as there would be if they contained run-time checks to detect these special cases.

The methods we developed for detecting and optimizing searching and sorting primitives on permutation vectors obviously apply just as well to arithmetic progression vectors (APVs), also known as j-vectors.[Abr70, Ive73]. However, as the APEX compiler does not presently include support for APVs, we are unable to report on this aspect of performance.

9 Future Work

Array predicates give us a tool for simplifying the computational complexity of primitive functions, but they are not a complete answer. Some relatively simple problems remain intractable from a formal analysis standpoint. For example, consider the computation of a histogram from a vector of integers. This can be performed in sequential code in linear time:

```
∇ r←nbuck histlp nums;i
[1] r←nbuck⍴0
[2] :for i :in nums
[3]   r[i]←r[i]+1
[4] :endfor
∇
```

Yet, when expressed as an obvious APL reduction

of an outer product or one of its siblings, we end up doing a quadratic amount of work:

```
+/(⍳nbucks)∘.=nums
((⍳nbucks)+/⍤∘.=)⍤∘.1 0 nums
+/(⍳nbucks)=⍤∘.1 0 nums
```

In general, the set of APL expressions that are of most interest are those with super-linear computational complexity in APL, but linear computational complexity in scalar-oriented languages. These expressions usually involve inner or outer products, scans, or reductions, as well as the searching and sorting primitives discussed here.

One form of expression that is of interest arises from a class of APL algorithms that perform outer products, then extract the diagonal from the resulting array. Such methods perform an immense amount of needless computation. Demand-based evaluation can mitigate this waste to some degree, but it is not always an applicable solution.[Bud88]

A compiler or interpreter can use pattern or phrase recognition to recognize some expressions of this class and generate special-case code for them, but this type of solution is labor-intensive for the compiler writer. It also does not provide a general solution to the problem. We seek an approach to reducing computational complexity that is not merely a collection of special cases, but a general solution that covers a multitude of similar problems.

10 Acknowledgements

Some ideas in this paper arose during a discussion with Roger Hui on the way back from a Minnowbrook Arrays Workshop, when array predicate theory was in its infancy.

References

[Abr70] Philip Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970. SLAC Report No. 114.

[Ber73] Robert Bernecky. Speeding up dyadic iota and dyadic epsilon. In *APL Congress 73*, pages 479–482. North-Holland Publishing Company, 1973. Second printing.

[Ber93] Robert Bernecky. Array morphology. In Elena M. Anzalone, editor, *APL93 Conference Proceedings*, volume 24, pages 6–16. ACM SIGAPL Quote Quad, August 1993.

[Ber97a] Robert Bernecky. APEX: The APL parallel executor. Master's thesis, University of Toronto, 1997.

[Ber97b] Robert Bernecky. An overview of the APEX compiler. Technical Report 305/97, Department of Computer Science, University of Toronto, 1997.

[Ber98] Robert Bernecky. Reducing computational complexity with array predicates. In Sergio Picchi and Marco Micocci, editors, *APL98 Conference Proceedings*, pages 46–54. APL Italiana, July 1998.

[Bud88] Timothy Budd. *An APL Compiler*. Springer-Verlag, 1988.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP- Completeness*. W.H. Freeman, 1979.

[Ive73] Eric B. Iverson. APL/4004 implementation. In *APL Congress 73*, pages 231–236. North-Holland Publishing Company, 1973.

[Jor79] Kevin E. Jordan. Is APL really processing arrays? *ACM SIGAPL Quote Quad*, 10(1), September 1979.

[Wie86] Clark Wiedmann. Field results with the APL compiler. In *APL86 Conference Proceedings*, volume 16, pages 187–196. ACM SIGAPL Quote Quad, July 1986.

[Wil91] R.G. Willhoft. Parallel expression in the APL2 language. *IBM Systems Journal*, 30(4), 1991.