Abstract
ELI is a succinct array-based interactive programming language derived from APL. In this paper we present the overall design and implementation of a bootstrapped ELI-to-C compiler which is implemented in ELI. We provide a brief introduction to the ELI language, a high-level view of the code generation strategy, and a description of our bootstrapping process. We also provide a preliminary performance evaluation. Firstly, we use three existing C benchmarks to demonstrate the performance of the ELI-generated C code as compared with interpreted ELI and native C. Secondly, we use two benchmarks originally from APL to compare the ELI-generated C to interpreted ELI and a naïve hand-generated C version. These preliminary results are encouraging, showing speedups over the interpreter and in many cases performance close to C. The results also show that some future optimizations, such as copy elimination/avoidance, would be beneficial.

CCS Concepts • Software and its engineering → Compilers; Very high level languages

Keywords array programming language, compiler, bootstrapping, performance

1. Introduction
Widely used interpreter-based array languages, such as MATLAB and Python, provide programming productivity, but suffer execution performance in comparison with compiled languages such as C or FORTRAN. In situations where execution performance does matter, one conventional approach is to use a profiling tool to find out where a program spends most of its time and rewrite the computation intensive portion, presumably much smaller than the whole program, in a compiled language and then replace that part of the program with a call to the compiled module. A better approach, which reduces the burden on the programmer, is to provide a compiler for the scripting array language (or a subset of the scripting language). The programmer can then use the compiler to generate the compiled version for all or part of the original scripting program, and then either execute the compiled version of a whole program or replace the portion of a program which can be compiled with a call to the compiled module. This paper presents the design, implementation and evaluation of compiler for an array-based language called ELI.

ELI is an interactive array programming language based on APL [7, 9]. The core part of ELI covers APL1 as defined in the ISO APL standard [15], but it uses ASCII characters instead of the special APL characters. In addition to the features specified in the APL standard, ELI has complex numbers, symbols, date/time as raw data types, lists for non-homogeneous data, dictionaries, tables and basic query capabilities. ELI is available on Windows, Mac OS and Linux.1

Our ELI-to-C compiler is written in ELI, and it translates an ELI program into C. The compiler can handle all core ELI features, i.e. the portion of the ELI language that corresponds to APL1. The compiler follows the same technical approach as a previous APL compiler, COMPC[10]. The ELI-to-C compiler compiles scripts composed of main function declaration (not variable declarations), and the types and ranks of input parameters need to be supplied when invoking the ELI compiler. The choice of targeting C was due to the wide-spread availability of optimizing C compilers, and the fact that C provides a target that is high-level enough to be convenient for code generation, while low-level enough to expose efficient implementations of underlying arrays and primitive ELI operations.

In order to evaluate the quality of the produced C code, we performed two experiments. The first experiment was intended to explore the performance of ELI-to-C generated code versus the ELI interpreter and against existing C implementations of three benchmarks. Ideally the generated code should be significantly faster than the interpreted code, and almost as fast as the existing C version. Our results are encouraging, but there remains further possible improvements. All three benchmarks show good performance improvements over the interpreter, and reasonable performance compared to C. The second experiment was to start with an existing benchmark (originally from APL), written in ELI, and to examine the performance of the generated code with respect to the interpreted ELI, and a hand-generated C code that is a naive translation of the ELI code to C. Both of these cases show good speedups for the ELI-to-C generated C code.

The remainder of the paper is structured as follows. Section 2 provides some background on the ELI language, and it introduces the code for one of the ELI benchmarks. Section 3 describes the structure of the ELI-to-C compiler along with a high-level view of the code generation strategy, and Section 4 provides a detailed discussion on self-compilation. Section 5 reports on the performance measures. Finally, Section 6 outlines future work, Section 7 discusses related work, and Section 8 concludes.
2. Background

APL is an array-oriented programming language invented by Ken Iverson to teach mathematics, for which he received the Turing Award in 1979 [16]. In comparison with other array languages developed later such as MATLAB and Python, APL has two distinct features:

1. **Monadic** (has only right operand) and **dyadic** (has left and right operands) primitive (array) functions are represented by a single character symbol.

2. A line of APL code executes from right to left in a dataflow style, i.e. the output of one operation feeds the next operation as input.

This results in a succinct and very productive language which has been used profitably in a wide range of areas such as finance, actuary work, logistics and computer aided design. ELI [7], at its core, is just an ASCII version of APL1: it replaces each APL character representing primitive functions with one or two ASCII characters, but still maintains the one-character one-symbol feel of APL thus encourages a dataflow style of programming. However, unlike IBM APL2 which introduces nested arrays into APL, ELI provides lists for non-homogeneous data and data nesting.

ELI has four basic data types: **numeric** (boolean, integer, real and complex numbers), **character**, **symbolic** and **temporal** (date, month, time, second, minute, datetime). A single data item is called a **scalar**; a homogeneous and rectangular collection of data is called an **array**.

ELI provides a large number of **primitive** functions (each denoted by one or two characters) which operate on scalars as well as on arrays as a whole; like APL, a primitive function is either monadic or dyadic. There are two kinds of primitive functions: **scalar** and **mixed**. A scalar primitive function (which includes arithmetic, logical and relational functions) has the property that it operates on arrays as an extension of its operation on elements of the arrays. Each array has a shape (dimensions) and the shape of all scalars is an empty vector (vector of length 0). A mixed function may query the shape of a data item or reshape it. In general, a mixed function either transforms an array or takes part of it, including indexing.

In addition, there are four operators: **reduce**, **scan**, **inner product** and **outer product** which apply to one or two scalar primitive functions to produce a derived function. For example, reduce (\(/\)) applied to addition + is a summation (\(+/\)); Inner product (\(\oplus\)) applied to + and * is the matrix multiplication function (\(\oplus\ast\)) for two compatible matrices. A line of code in ELI operates from right to left as a chain of operations in a dataflow style, i.e. the output of one operation feeds as input to the next operation; and all functions are of equal precedence, so 3*2+1 is 9 not 7. For example,

\[ +\{1+2\ast10 \]

results in the partial sums of odd integers from 3 to 21.

A **user-defined** function in ELI can be monadic or dyadic, or niladic (i.e. takes no argument), it can yield results or no result. A user-defined function can have local variables, and the scoping rule for shadowing variables is **dynamic**. ELI provides branching as in APL1, but ELI also provides C-like control structures. The ELI compiler currently covers the APL1 standard features.

The ELI language extends APL1 by providing a general **list** data structure which is a linear collection of data items enclosed by parenthesis (..) and separated by semicolon (;), each of which can be a scalar, an array or another list. There is an **each** operator **\(\ast\)** which for a function \(f\), \(f\ast\) applies \(f\) to each item in its argument list. A dictionary in ELI is a special list which maps a set of keys, i.e. the domain, to a set of lists, called the range. A **dictionary** whose range is a group of lists of equal length is called a **table**. ELI provides **esql** which contains a set of statements similar to standard SQL to query and process data in tables of a database. These extended features are not currently covered by the compiler (see the discussion of future work in Section 6).

To illustrate a non-trivial example (taken from one of our benchmarks), consider the definition of the **morgan** function as given in Figure 1(b). A description of the primitive functions and operators used in **morgan** are given in Figure 1(a). First, note that there are two function declarations, the helper function called **msum** and the main function called **morgan**. The function **msum** is a dyadic function with left parameter \(n\), right parameter \(a\), and return parameter \(r\). The body of **msum** is one line, which should be read from right-to-left. It first defines a new variable \(t\) and then uses \(t\) to compute the final result. The **morgan** function also has two input parameters, \(n\) and \(a\). In addition it declares 7 local variables, \(x;y;as;sx2;sy;sy2;axy\). The body of the computation first extracts two slices from \(a\), then computes five intermediate results using **msum** and finally performs the main computation in the last line.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>x&lt;-value</td>
<td>assign value to (x)</td>
</tr>
<tr>
<td>abs</td>
<td></td>
<td>absolute value</td>
</tr>
<tr>
<td>plus</td>
<td>L+R</td>
<td>elementwise plus</td>
</tr>
<tr>
<td>minus</td>
<td>L-R</td>
<td>elementwise minus</td>
</tr>
<tr>
<td>multiply</td>
<td>L*R</td>
<td>elementwise multiplication</td>
</tr>
<tr>
<td>divide</td>
<td>L/R</td>
<td>elementwise division</td>
</tr>
<tr>
<td>power</td>
<td>L*R</td>
<td>elementwise power</td>
</tr>
<tr>
<td>drop</td>
<td>L!R</td>
<td>remove the first (last) (L) items of array (R), if (L&gt;0) (&lt;0)</td>
</tr>
<tr>
<td>concatenate</td>
<td>L,R</td>
<td>append (R) to (L)</td>
</tr>
</tbody>
</table>

(a) **Primitive operations used in morgan**

\[0\] \(0.\ r<-n\ msum\ a\)
\[1\] \(r<-(0,0,\ldots,\!t\!-\!0,\!0,\ldots\!t++\!a\)
\[2\] \(0.\)

\[0\] \(0.\ r<-n\ morgan\ a;x;y;sx;sx2;sy;sy2;axy\)
\[1\] \(x<-a[1];\)
\[2\] \(y<-a[2];\)
\[3\] \(sx<-n\ msum\ x\)
\[4\] \(sy<-n\ msum\ y\)
\[5\] \(sx2<-n\ msum\ x\ast\ 2\)
\[6\] \(sy2<-n\ msum\ y\ast\ 2\)
\[7\] \(axy<-n\ msum\ x\ast\ y\)
\[8\] \(r<-(axy\%n)-(ax\ast sy\%n\ast\!2\%)
 (\(sx2\%n\ast sy\%n\ast\!2\%\ast\!0.5\)\)
 (\(axy\%n\ast sy\%n\ast\!2\%\ast\!0.5\)\)
\[9\] \(0.\)

(b) **ELI source code for morgan**

**Figure 1.** Example ELI program

3. Writing the ELI Compiler in ELI

ELI, like APL and other interactive and interpreted array languages, can be much less performant than programs written in compiled languages with optimizing compilers like C or FORTRAN. To improve performance, we have built the ELI-to-C compiler, which translates ELI source code into C. This allows programmers to
quickly prototype using the ELI interactive interpreter, and then apply the compiler to automatically generate efficient and stand-alone executables.

The ELI-to-C compiler follows the line of research started with the APL/370 compiler [8] and COMPC, an APL-to-C translator [10]. As COMPC evolved from the APL/370 compiler, our ELI compiler evolved from COMPC which was written in IBM VS APL system covering APL1. The ELI compiler is written completely in ELI - we describe out bootstrapping process in the next section.

Just as COMPC only covers most features in APL1, the ELI compiler can only compile ELI programs where only flat arrays are involved (arrays in ELI are by definition of homogeneous type, or flat). ELI also restricts function names not to be used as variable names (which is allowed in APL1).

An ELI compilation unit consists of a main function and functions in the workspace called directly or indirectly by that main function which takes up to two parameters. The right parameter can be lists, but then the first line in the main function must assign the list(s) to a group of arrays or scalars. No place in any function in the compilation unit can invoke the primitive function execute (.), which dynamically interprets its argument (character) string. Other than taking (two) input parameters, functions in a compilation unit cannot access any global variables in a workspace.

3.1 Compiling ELI to C

Initially, the ELI compiler recognizes two specific variables as the entry point of the compilation from ELI to C. They are the left parameter LPARM and the right parameter RPARM. LPARM’s initial part is the name of the main function of the unit to be compiled, followed by a blank and two characters indicating the types of the left and right arguments (’C’ for character, ’T’ for integer and ’E’ for floating point), and RPARM is an integer vector starting with [1]O (i.e. *dio, array index starting with 0 or 1) and followed by shape elements of the left and right arguments (for a scalar, it is 0, for an array it is the rank followed by dimensions with 1 indicating an unknown length). The COMPILE function is invoked with the two parameters as follows: LPARM COMPILE RPARM.

We illustrate the compilation process with the simple example given in Figure 2. First, a simple ELI function called add is defined in an ELI script file called add.esf, and then appropriate LPARM and RPARM variables are defined. For this case, the LPARM specifies a string which indicates the entry function is called add, and that it has two parameters with base type of floating point. RPARM specifies that 0 indexing should be used (the first value), that the first parameter is a vector of unknown length (the 2nd and 3rd values), and that the second parameter is a scalar (the 4th value). The compiler will infer the shape of the return variable. Since there is only one primitive, *plus, which is an elementwise operator, the shape of z will be a vector with the same length as the input argument a.

```
0. z<~a add b &LPARM C 1 6 &RPARM I 1 4
z<~a+b 'add EE'
0. &
```

Figure 2. A simple function add and the definition of LPARM and RPARM.

After this script is compiled by the ELI compiler, a single file add.c is emitted. It contains about 140 lines of C code, in which the core computation is a loop for plus. In Listing 1, one can see that the compiler identifies that the result is a vector with the same length as the input vector a, that the base types are double, and that the plus operator should be compiled as an itemwise plus of a vector and a scalar.

```c
/* v18 is result vector z, */
/* v19 is input vector a, */
/* v20 is input scalar b */
r0=v19. real[ ]; /* length of result */
INCHEAPP3(v18, incp2 ); /* alloc result */
p2 = (double *) v18.valp;
lo2 = (double *) v19.valp;
for (v1 =0; v1<r0 ; v1++) /* code for + */
p2[v1] = lo2[v1]+v20;
```

The emitted C code is then compiled with the external library elfi31ib to generate a final executable file. We provide library support for frequently used and important primitives. For example, a boolean operation, such as and (’) and or (k), is implemented with bitwise operations.

To run the executable, one also needs to define input data files. In this example, we only need two files: add. LEF and add. RIG, for the left and right parameter separately. The format of a data file has to follow: < type >> n >> dimx > ... < dimn > < data > where n is the total number of dimensions.

3.2 ELI Compiler Front-End

The ELI compiler consists of a front-end and a back-end. The front-end takes the text (as a matrix) of the main function as the entry point. It starts with parsing the main function and for each defined function encountered in a depth first search fashion, it converts and adds the parsed version to a list of parse trees. We note here that our parser does not utilize any external compiler tool, rather it directly implements a top-down parser with a two-symbol look-ahead algorithm. Each line of the input ELI code is scanned and parsed left-to-right to build a parse tree for that line. These parse trees are then grouped together into basic blocks for building a control flow graph for each defined function.

A major technical feature of the front-end is its use of Tarjan’s fast interval finding algorithm [20] to implement the reaching-definition calculation of Allen and Cocke [1] for use-def chaining. Starting with the user-specified base types, and shapes or at least the rank (dimensions) of input parameters, base types and shapes are propagated via the u-d chains. This is made possible by the very specific shape rules for all primitive operations and operators. Thus, after the front-end has completed, a list of control flow graphs, one for each function reachable from the entry point has been created. The basic blocks in the control flow graphs are lists of lines of code, where each line is represented by a parse tree. Furthermore, the parse trees have been annotated with the base type and shape (or rank) information.

3.3 ELI Compiler Back-End

The back-end consists of a main function, treewalk, and a group of code-generating functions each implementing a (group of) primitive or derived function(s) in ELI such as arithmetic functions, logical functions, comparisons, rotate and membership.

The ELI compiler generates C code, starting with declarations of a set of global variables in C for the compiled code to use, and for each function it also declares a set of local variables for each function in the compilation unit as their appropriate data types and dimensions have been found in the front-end.

The back-end applies the treewalk function to each annotated parse tree produced by the front-end for each defined function encountered in the compilation unit. The treewalk function operates on a parse tree in a semi-recursive fashion until it reaches the root of that parse tree. There are four kinds of nodes in a parse tree: constant node, variable node, primitive function node and
defined function node. When *treewalk* encounters a variable node, it accesses the variable table to translate its name. The *treewalk* code generation starts from the right-most leaf node (a leaf node is either a data node indicating a variable or a constant, or a niladic function node) and walks up to its parent node, which is a function node representing a primitive function, a derived function or a defined function, and checks to see whether it has a left child node. If there is no left child or the left child is a leaf node, then it generates a function call to the defined function of that node, or calls a code generating routine to emit C code corresponding to the primitive/derived function of that parent node. In case the left child is not a leaf node, it visits the right-most node of the left sub-tree and calls *treewalk* recursively with the old left child node as its root. There is an exception to this rule: when a parse tree contains a portion called *stream* (marked by the front-end), like $a+b+c$, a fused loop will be generated for that portion.

In addition to generating calls to defined functions, there are three kinds of code generation routines called by the *treewalk* function: (1) routines directly emitting C code which implement the primitive functions involved such as *pfunthm* (for $+$, $-$, etc.) or *pfrotate* (for $\times$) according to the type-rank of its argument node(s); (2) code routines calling a pre-coded C function in eli3lib such as the one for membership which calls a hand-written C function which uses hash table; and (3) a routine for branching, i.e. *goto*, whose target block has been determined in the front-end but its C code target is supplied in the back-end.

3.4 ELI Test Suite

The ELI compiler inherits from COMPC a testing suite of over 100 APL programs with the largest one having 1000 lines of code. The majority of the programs in this test suite come from APL educational samples and applications from a variety of fields and we have translated them to ELI code.

4. Self-Compilation

Following the tradition of exercising a new language by writing a compiler in that language, we have written the ELI-to-C compiler in ELI. In this section we provide an overview of our bootstrapping process and how we dealt with a large input program like a compiler.

We first remark that while ELI provides a workspace for organizing programming tasks as in APL, it is more convenient in ELI to use script files (*.esf) for saving/loading large programs. The ELI compiler has about 15,000 lines of ELI code (saved in ecc.esf) which results in over 330,000 lines of C code when translated to C. Even though for moderate size ELI programs the compilation to one file is convenient, for a large program like ecc.esf, it would be better to generate many smaller files that could be separately compiled and linked together to make the executable. As described in Section 4.2, we provide a utility to provide this functionality. Our final generated file is an executable called ECC.

4.1 Bootstrapping ELI

The initial version of the ELI compiler is denoted by ECCE (ECCE ELI script). The main function of ECCE is *COMPILE* whose left argument LPARM is a character string and whose right argument RPARM is a numeric vector. Hence, after loading ecc.esf, the compiler ECCE is invoked by

```
'mainfn \lcrt' COMPILE qdio shpl shpr
```

(in case there is no left argument, then that part is empty). To compile ECCE itself, the command is:

```
'COMPILE CI' COMPILE 0 1 1 1
```

The source code of the functions in a compilation unit is an implicit input to the compiler. This is supplied in ELI by the system function \[\text{CR}\]:

```
\text{mtxt} \leftarrow \text{[\text{CR}] 'fnam'}
```

which fetches the code of a function named *fnam* in a character matrix.

Initially, ECCE just uses system functions in ELI in an ELI environment, but does not generate their C equivalent. One example is the system function \[\text{CR}\] for which we introduce a new function called *PFQDCR* in ECCE to generate corresponding code in C by calling the equivalent code in elimacsros.cpp. Another example is the system function \[\text{FM}\] which is used to save the value of a variable into a file. Its corresponding C code is the function *writecode*, declared in elimacsros.cpp. We also add the capability to ECCE so that it can handle *.esf* files to enable us to pass in LPARM and RPARM as files.

The tombstone diagram (T-Diagram) in Figure 3 describes the bootstrapping process, and Figure 4 provides a diagram of the workflow.

- In stage T1, the ELI compiler, ECCE, is able to translate ELI source code to C code within the ELI interpreter since it is written in ELI;
- In stage T2, the ECCE compiles itself in the ELI interpreter environment. The procedure of bootstrapping emits about 330K lines of C code, called *ecc.c*;
- In stage T3, the generated C code *ecc.c* is compiled by the available GNU C/C++ compiler into an independent executable file, *ECC*.

We provide an overview of the bootstrapping process in Figure 3. The keyboard (T-Diagram) in Figure 3 describes the bootstrapping process, and Figure 4 provides a diagram of the workflow.

4.2 Utility for Splitting the Generated Code

After ECCE has translated itself into ecc.c, the problem of a large output becomes obvious. The size of the C file ecc.c far exceeds those generated in our test suite. There are 291 functions in ecc.c with more than 330k lines of code which clearly is too large for a C compiler to deal with. Hence, we must partition it into pieces for separate compilation. We decided that one generated function per file was a simple way to get that done, and we wrote a utility program in C to split that single large file into 291 files, utilizing the fact that each generated function starts with an obvious delimiter:

```
char delim[] ="/* CODE SEGMENT FOR";
```

To support the process of compiling functions separately, we create other four files for the whole project:
5. Performance of ECC

The purpose of our preliminary performance evaluation was to: (1) determine if ECC produces compiled code that is significantly faster than the ELI interpreted code; (2) determine if the performance of ECC-compiled code is as fast as hand-written C code; and (3) identify further optimization opportunities for ECC. For this purpose, we chose a total of five benchmarks from two categories: standard C benchmarks and array-based programs. The standard C benchmarks include three benchmarks from standard benchmark sets, the Princeton suite [3] and the Rodinia Benchmarks [5, 6]. The array-based programs contain two programs which are implemented with array programming primitives which have been translated from APL benchmarks.

We used a commodity machine, Sable-Lynx, to measure performance. It is equipped with an Intel i7-3820 CPU, clocked at 3.60GHz and with 8 GB RAM. It runs on Ubuntu 16.04 Xenial, 64-bit version. The version of GCC we used to generate machine code is v5.4.0 for Linux. The compiler flag for C programs is set to -O3. For ELI programs, we use the ELI interpreter version v0.1a for Linux. Each program is executed 10 times, and we report the average of the 10 runs. We maintain all scripts and data in a GitHub repository at https://github.com/Sable/ARRAY17-ECC.

5.1 Standard C Benchmarks

In this section, we measured three standard benchmarks: black-scholes from the Princeton suite [3]; k-means and hotspot are from the Rodinia Benchmark [5, 6]. Since all of them are initially implemented in C, we needed to create equivalent programs in ELI. In each case we implemented the same algorithm, but used an array-based programming style where possible.

5.1.1 black-scholes

This program is used to compute the price of options using black-scholes formula and the cumulative normal distribution function. They are implemented in the BlkSchlsEqEuroNoDiv and CNDF functions respectively. The ELI version is in array form without loops. It uses arithmetical primitives in a dataflow style, i.e. output of one operation feeds as input to the next from right to left. We have used four input sizes, 1M, 2M, 4M and 8M.

![Table 1. Result of black-scholes (Time: ms) and Ratios](https://github.com/Sable/ARRAY17-ECC)

<table>
<thead>
<tr>
<th></th>
<th>1M</th>
<th>2M</th>
<th>4M</th>
<th>8M</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELI (ms)</td>
<td>840</td>
<td>1714</td>
<td>3450</td>
<td>7133</td>
</tr>
<tr>
<td>ECC (ms)</td>
<td>445</td>
<td>902</td>
<td>1986</td>
<td>4240</td>
</tr>
<tr>
<td>C (ms)</td>
<td>364</td>
<td>727</td>
<td>1455</td>
<td>2989</td>
</tr>
<tr>
<td>ELI/ECC</td>
<td>1.89</td>
<td>1.90</td>
<td>1.74</td>
<td>1.68</td>
</tr>
<tr>
<td>C/ECC</td>
<td>0.82</td>
<td>0.8</td>
<td>0.73</td>
<td>0.70</td>
</tr>
</tbody>
</table>

The result of black-scholes is given in Table 1. In this table, and in all subsequent tables, we first give the average execution time for the interpreter (ELI), the ELI-to-C compiled code (ECC) and the hand-written C code (C). We also provide two ratios. The ratio ELI/ECC indicates how many times faster the ECC-compiled code is over the ELI interpreted code. The ratio C/ECC indicates how many times faster the hand-coded C code is over the ECC-compiled code. A value less than 1 indicates that the ECC-compiled code is slower than the hand-coded C code, whereas a value greater than 1 indicates that the ECC-generated code is faster.

For black-scholes, our results show that C always gives the best performance for all four input sizes. However, the ECC-generated code is almost twice as fast as the interpreted code, and only slightly slower than the hand-coded C code (speedup over C of 0.70 to 0.82). The modest speedup over the interpreter is due to the relatively low interpreter overhead for this benchmark. The ELI program is quite straightforward: the main function BlkSchlsEqEuroNoDiv only calls CNDF two times and there are no loops in the ELI version of the benchmark, all computations use array-based operators. Therefore, the interpreter overhead is not very high.

5.1.2 k-means

The k-means benchmark implements a clustering algorithm which is used extensively in data-mining. In k-means, a data object is comprised of several values, called features [5]. It is an iterative program, particularly suited for C; k = 3 means there are 3 clusters on various numbers of data points. For each data point, there are 30 features in our input data. We have used four input sizes, 32K, 64K, 128K and 256K.

As shown in Table 2, the ECC compiled code is over 20 times faster than the ELI interpreted code, which is an excellent result. The ECC compiled code is consistently slower than hand-coded C code because array indexing in the ECC compiled code introduces additional computation and memory cost for intermediate results. The large speedup over the interpreter is because this benchmark has a large interpretive overhead due to the loops inside of kmeans which find the best-fit cluster groups. A further advantage of the ELI approach is that the ELI code is very compact, only 13 lines
of code in this case, while the core part of the C benchmark (i.e., `kmeans_clustering.c`) has about 165 lines of code.

### 5.1.3 hotspot

The hotspot benchmark is a thermal simulation program to iteratively solve a series of differential equations on a rectangle which is used for chip-design. Each output cell in the computational grid represents the average value of temperatures of the corresponding area of the chip [5, 6]. For our tests, the number of iterations is set to 20 and the size of data is increased from 256x256 to 2048x2048.

<table>
<thead>
<tr>
<th>N x N</th>
<th>256x256</th>
<th>512x512</th>
<th>1024x1024</th>
<th>2048x2048</th>
<th>ELI (ms)</th>
<th>ECC (ms)</th>
<th>C (ms)</th>
<th>ELI/ECC</th>
<th>ECC/C</th>
<th>C/ECC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELI (ms)</td>
<td>139</td>
<td>657</td>
<td>3129</td>
<td>12806</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECC (ms)</td>
<td>66</td>
<td>192</td>
<td>914</td>
<td>3690</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C (ms)</td>
<td>44</td>
<td>102</td>
<td>397</td>
<td>1585</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELI/ECC</td>
<td>2.11</td>
<td>3.42</td>
<td>3.42</td>
<td>3.47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/ECC</td>
<td>0.67</td>
<td>0.53</td>
<td>0.43</td>
<td>0.43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECC-Opt (ms)</td>
<td>47</td>
<td>159</td>
<td>784</td>
<td>3190</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECC/ECC-Opt</td>
<td>1.40</td>
<td>1.21</td>
<td>1.17</td>
<td>1.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As shown in Table 3, the speedup of ECC-compiled code over interpreted code ranges from about 2x speedup for the smallest input size, to around 3.4x speedup for the larger input sizes. However, the ECC-generated code is substantially slower than the hand-coded C, with speedups of 0.43 to 0.67. This means that there remains some important optimization opportunities, and by inspecting the generated code, we have determined that extraneous array copies is one potential performance bottleneck.

To test this hypothesis, we created a new version by hand called ECC-Opt, in which we combined array operations to reduce array copies for intermediate results. For example, two contiguous operations: the last column of a matrix is chopped and a leading zero is added in each row, occur many times in the core computation part of hotspot. The ECC compiler generates code for the two operations in two steps, so that an intermediate matrix is used to hold the result of the first operation and two copies are required. In our modified version ECC-Opt, we copy the input matrix and add the leading zero in one step. Both one extra copy and some memory are saved.

We have added two extra rows at the bottom of Table 3 to show the effect of the copy elimination. It is clear that it does significantly improve the performance of the ECC-generated code, with speedups ranging from 1.4x faster for the smallest input to 1.16x faster for the largest input.

### 5.2 Testing with Array-Based Programs

The three benchmarks in the previous subsection were originally written in C, later translated to ELI and eventually compiled to C code. Another way to study performance is to test programs which were written in an array language to start with. Hence, we took two existing APL benchmarks and expressed them as ELI benchmarks (a 1-to-1 translation), and then manually translated the ELI benchmarks to C by strictly following the ELI program structure. We attempted to write efficient and clean C code.

#### 5.2.1 morgan

The morgan benchmark comes from a financial application. As introduced in Figure 1(b), it has two functions: msum and morgan, where morgan is the main function. The core computation is the last line of morgan. This benchmark is very suitable for APL-style programming, with the entire algorithm expressed using array operations without any control structures. The APL version of this benchmark was used in previous work on the automatic parallelization of APL[11]. The morgan function is dyadic, taking a left parameter a and a right parameter b. We experimented on four increasing problem sizes, as shown in the header of Table 4.

<table>
<thead>
<tr>
<th>n</th>
<th>a (2×1024×N)</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELI (ms)</td>
<td>132</td>
<td>43</td>
<td>1575</td>
<td>5987</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECC (ms)</td>
<td>24</td>
<td>38</td>
<td>64</td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C (ms)</td>
<td>21</td>
<td>33</td>
<td>55</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELI/ECC</td>
<td>5.5</td>
<td>11.5</td>
<td>24.6</td>
<td>46.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECC/C</td>
<td>0.88</td>
<td>0.87</td>
<td>0.86</td>
<td>0.84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4 shows the ECC-generated code is substantially faster than the ELI interpreted code, and that the speedup increases as the problem size grows bigger. For the largest problem size, the ECC-generated code is about 46 times faster than the ELI interpreter. The ECC-generated code also is reasonably close to the performance of our hand-written C code, with speedups ranging from 0.84 to 0.88. We studied the differences between our hand-written code and the ECC-generated code in order to determine why the ECC-generated code is slightly slower. The main difference seems to be in the number of array copies. In the ECC-generated code, input arguments are duplicated in each time `msum` is invoked due to ELI’s pass-by-value semantics. Moreover, returning value is duplicated as well after a function invocation completes. However, these copies are unnecessary in this simple context. In hand-written code, we avoid copying the value of arrays to/from called functions. Instead, we use pointers to pass the value of arrays. This also confirms that array-copy elimination is an important optimization to add to ECC.

#### 5.2.2 rprime

Given a number n, `rprime` finds all prime numbers up to n. It is implemented in a recursive way, which repeatedly utilizes the result from previous iteration. By using a boolean vector to identify all possible non-prime numbers, we eventually get all prime numbers within n. The core ELI computation is shown below.

```haskell
[0] p<-(rprime n);pl;b;i
[1] p<-2 3 5 7 11 13 17 19 23 29 31 37 41
   43 47 53 59 61 67 71 73 79 83 89 97
[2] ->(n<=100)/L0
[4] ->0
[5] L0: pl<#p<rprime _._n*0.5
[6] b<-n#0
[7] i<-1
[8] L2: b<-b&n#(~p[i])^1
[9] ->(pl>i<i+1)/L2
[10] p<-p,1!.("b")/ln
```

### Table 2. Result of k-means (Time: ms) and Ratios

<table>
<thead>
<tr>
<th>k=3</th>
<th>32K</th>
<th>64K</th>
<th>128K</th>
<th>256K</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELI (ms)</td>
<td>5148</td>
<td>8132</td>
<td>10538</td>
<td>22438</td>
</tr>
<tr>
<td>ECC (ms)</td>
<td>250</td>
<td>379</td>
<td>466</td>
<td>935</td>
</tr>
<tr>
<td>C (ms)</td>
<td>93</td>
<td>123</td>
<td>149</td>
<td>295</td>
</tr>
<tr>
<td>ELI/ECC</td>
<td>20.59</td>
<td>21.46</td>
<td>22.61</td>
<td>24</td>
</tr>
<tr>
<td>C/ECC</td>
<td>0.37</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
</tr>
</tbody>
</table>
We experimented with the input argument \( n \) ranging from 100K to 800K. Table 5 shows that ECC gives good speedups of 6x to 7x over interpreted ELI. Somewhat surprisingly the ECC generated code is also 8x to 14x faster than our hand-coded C. In studying the differences between the ECC-generated C and our hand-coded C, we noted that the ECC-generated C made heavy use of optimized bit vectors. Since ECC inferred that the base types of the vectors was boolean, it generated very efficient bit-vector representations and bit-vector operations (for example, the bitwise operations \& (and), \| (or) and ~ (not) in lines 8 and 10). On the other hand, our hand-coded C used array of type bool, which are not stored as bit-vectors by the C compiler.

| Table 5. Result of rprime (Time: ms) and Ratios |
|----------------|----------------|----------------|----------------|
| \( n \)        | 100K | 200K | 400K | 800K |
| ELI (ms)       | 42   | 96   | 254  | 642  |
| ECC (ms)       | 7    | 13   | 37   | 87   |
| C (ms)         | 82   | 184  | 314  | 697  |
| ELI/ECC        | 6    | 7.4  | 6.9  | 7.4  |
| C/ECC          | 11.7 | 14.2 | 8.5  | 8    |
| C-Opt/(ms)     | 2    | 3    | 7    | 14   |
| C/C-Opt        | 41   | 61   | 49   | 50   |
| C-Opt/ECC      | 0.29 | 0.23 | 0.19 | 0.16 |

After we profiled the ECC-generated C code, we found that bitwise operations contributed about 79% execution time. We created an optimized version of our hand-coded C program that used bit vectors with efficient bit-vector operators. This code was much more complex and more difficult to debug than the initial code. However, as shown by the last three rows of Table 5, it speeds up the original hand-coded C code by over 40x, and is now faster than the ECC-generated code. However, it is important to note that the fast version of the C code took extra programmer effort, whereas the ECC-generated code automatically used efficient bit-vector operators.

6. Future Work

We plan to do further work on the ECC compiler in four directions: expand compiler coverage, implement automatic parallelization, inective compilation, and copy elimination optimizations.

For the compiler to be useful for all ELI users, the most urgent thing is to remove restrictions on ECC as much as possible. Some restrictions such as allowing new data types of complex numbers, symbols and temporal data can be done easily. Other restrictions such as the exclusion of the function execution is inherently difficult to overcome. A more important restriction to remove is the flat array only rule, i.e. the exclusion of lists since lists are crucial in handling non-regular data. To remove this restriction in general would involve introducing pointers at each list item, and may incur large run-time overheads. We may be able to extend ECC to handle homogeneous lists for non-rectangular data first as well as to implement the each operator on these lists. In short, we will explore the extent we can accommodate lists of various properties.

The second direction of future compiler work is in a sense the main motivation for undertaking the ELI project, that is to implement automatic parallelization for compiled ELI programs to run on multi-core machines similar to what has been done in APL [11].

The compiled code of an ELI program operates in a self-contained unit: it has its own stack and heap. The third direction of future compiler work is to compile a small piece of program to be callable from a large existing environment. We call this inective compilation. For example, the library file standard. est, which is distributed with the ELI interpreter, contains many frequently used functions written in ELI. It would be nice for an ELI program to call such a library function in compiled C instead of an ELI function which needs to be further interpreted. The difficulty of course is that library function operates in the environment of the interpreter. This also will provide a convenient way for an ELI program which on the whole is not compilable because it uses lists or the execute function to call a compiled sub-portion.

As we demonstrated through our experiments, the copies required for the ELI value passing semantics can cause overhead. We would like to add optimizations to the compiler that can remove or avoid copies that are not needed, while at the same time supporting the value semantics.

7. Related Work

The design of the ELI language is to simplify APL language from unicode to ASCII code while preserving the semantics of APL. The step of bootstrapping the ELI compiler which is written in ELI is to free the running environment from ELI interpreter. Our ECC compiler chooses C as the the target for the back-end code generation. There has been other related work on compiling array-based languages, which we discuss in two groups: (1) compilers for APL-like languages, and (2) compilers for other array-based languages.

7.1 Compilers for APL-like Languages

Budd [4] presents an approach for an APL-to-C compiler using typical compiler technologies. His compiler was designed and implemented for the purpose of teaching and its functionality did not cover the complete set of APL1 primitives. Ching [8] presented an APL compiler that generated System/370 assembly code with the support of a majority of APL primitive functions. Like the previous work by Ching, our ELI-to-C compiler handles the APL1 standard, including the full set of primitives. Further, our approach was to bootstrap the compiler, which is completely implemented in ELI, and does not rely on external compiler tools.

Hsu [13, 14] initiated an experimental APL compiler, Co-dfns, for generating efficient low-level code to boost performance. To exploit data parallelism within APL code, it proposed a way by looking at inter-node relationships. Some features this compiler supports are beyond the APL1 standard, so it compiles more language features than our compiler. This compiler is written in Dyalog APL (www.dyalog.com) that constrains its usability within the Dyalog interpreter.

Ching and Zheng [11] developed a back-end which generates efficient parallel C code from APL code. It was derived from their COMPC compiler and improves with optimizations to find out good work-sharing candidates.

Bernecky [2] described how to compile APL-style programs in a newly designed language to C code with parallel directives. However, there is not much discussion on possible optimization strategies in the code generation phase.

7.2 Compilers for Other Array-Based Languages

Rose and Padua introduced FALCON [18], a compiler which was designed to generate efficient FORTRAN code from MATLAB. They have a set of SSA-based intermediate representations for static analysis, including type and shape analyses. Moreover, they deal with different semantics between MATLAB and FORTRAN by generating dynamic code in a dynamic phase. For example, dynamic code to handle dynamic size inference when the index assignment in MATLAB may grow the size of an array if the index exceeds the current bounds of the array.
Mc2FOR \[17\] was a research project which was also examining how to generate efficient FORTRAN code from MATLAB. In their work, they highlighted the challenges caused by the different semantics between MATLAB and FORTRAN. These challenges are similar to our work when converting ELI to C. Moreover, their compiler supports several interprocedural data flow analyses such as is\_complex, is\_scalar, shape analysis and range analysis. MATLAB supports more data shapes and types than ELI, as well as having very complicated shape rules for many built-ins. In the Mc2FOR compiler, a new domain-specific language is created that the compiler writer uses to specify the shape rules for groups of primitive functions. Shape information is propagated through the user-defined functions using an interprocedural dataflow analysis. When the interprocedural analysis encounters a call to a primitive function it uses the shape rules to accurately model the shape of the return values. Since ELI inherits from APL which natively supports a set of well-defined primitive functions for array programming, shape analysis on ELI is simpler. However, our compiler lacks of the range analysis which could help reduce the overheads in array bounds and shape checking.

An optimization opportunity in our compiler is reducing the number of copies of variables. An important semantic difference between ELI and C is: ELI is based on pass-by-value, while C passes arrays by reference. Therefore, we need to copy variables in the parameter passing. This may lead to inefficiencies because there are many situations where a copy is not actually required. For example, a simple case is when a parameter is only read (and not written). Foley-Bourgon and Hendren \[12\] proposed a dataflow analysis on MATLAB to generate efficient JavaScript code with fewer data copies. Similar to C, JavaScript is pass-by-reference while MATLAB is pass-by-value. Their analysis computes the points-to information after each copy in assignment, and then uses that information to identify where copies must be inserted to avoid writing to aliased variables. It would be interesting to integrate a similar analysis into our ELI-to-C compiler.

In Cython \[19\], some extensions were added to Python that allows the programmer to compile Python code to C code. This approach mixes the Python and C/C++ code using a calling mechanism which provides the flexibility to call compiled code at any program point. In this way, it provides great control over Python code using C semantics and it becomes friendly for analyzing variable types statically. However, it offloads the burden to users who have to handle which part of their code should be compiled.

8. Conclusion

In this paper we introduced a new compiler for ELI, an array-based language derived from APL. We introduced the ELI language, and provided an overview of our ELI-to-C compiler which is written in ELI. We also described how we bootstrapped the compiler to produce an executable compiler called ECC.

We provided an experimental study of the performance of the ECC by comparing: (1) ECC-generated C code versus the ELI interpreter; and (2) ELI-generated C code versus hand-coded C. We provided two sets of benchmarks, three benchmarks from traditional C benchmark sets, and two benchmarks from previous APL studies. We showed that the ECC-generated C code led to speedups between about 2x to over 40x speedups over the ELI interpreter. We also demonstrated that ECC-generated code often comes close to the hand-coded C code. Through hand-coded experiments we demonstrated that array copy-elimination is an important optimization to consider in the future.

We are quite encouraged by the results to date, and we have identified several important next steps including: (1) extending language coverage, (2) automatic parallel code generation, (3) injective compilation and (4) array copy-elimination.

References


\[16\] K. Iverson. Notation as a Tool of Thought, 1979 Turing Award Lecture, ACM Turing Award Lectures: The First Twenty Years, 1987.


