Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT compiler

Sable Technical Report No. sable-2010-01

Nurudeen Lameed and Laurie Hendren

March 25, 2010

Contents

List of Figures

List of Tables

Abstract

MATLAB has gained widespread acceptance among engineers and scientists as a platform for programming engineering and scientific applications. Several aspects of the language such as dynamic loading and typing, safe updates, and copy semantics for arrays contribute to its appeal, but at the same time provide many challenges to the compiler and virtual machine. One such problem, minimizing the number of copies and copy checks for MATLAB programs has not received much attention.

Existing MATLAB systems use a reference counting scheme to create copies only when a shared array representation is updated. This reduces array copies, but increases the number of runtime checks. In this paper we present a static analysis approach that eliminates both the unneeded array copies and the runtime checks. Our approach comprises of two analyses that together determine whether a copy should be performed before an array is updated: the first, *necessary copy analysis*, is a forward flow analysis and determines the program points where array copies are required while the second, *copy placement analysis*, is a backward analysis that finds the optimal points to place copies, which also guarantee safe array updates.

We have implemented our approach as part of the McVM JIT and compared our approach to Mathwork's commercial MATLAB implementation and the open-source Octave implementation. The results show that, on our benchmark set, our approach is effective at minimizing the number of copies without requiring run-time checks.

1 Introduction

MATLAB is a popular programming language for scientists and engineers. It was designed for sophisticated matrix and vector operations, which are common in scientific applications. It is also a dynamic language with a simple syntax that is familiar to most engineers and scientists. However, being a dynamic language, MATLAB presents significant compilation challenges. Thus, MATLAB applications are generally slower than those programmed with mainstream imperative programming languages. Improving the performance of MATLAB is an on-going area of research.

The problem addressed in this paper is the efficient compilation of the array copy semantics defined by the MATLAB language. The basic semantics and types in MATLAB are very simple. Every variable is assumed to be an array (scalars are defined as 1x1 arrays) and copy semantics is used for assignments of one array to another array and for parameter passing. Thus a statement of the form `a = b` or a call of the form `foo`(b) semantically means that a copy of `b` is made and that copy is assigned to either the *lhs* of the assignment statement or to the parameter of the function.

In the current implementations of MATLAB the copy semantics is implemented lazily using a reference-count approach. The copies are not made at the time of the assignment, rather an array is shared until an update on an array occurs. At update time (for example a statement of the form `b(i) = x`), if the array being updated (in this case `b`) is shared, a copy is generated, and then the update is performed on that copy. We have verified that this is the approach that Octave open-source system takes (by examining and instrumenting the source code). We have inferred that this approach (or a small variation) is what the Mathwork's closed-source implementation does based on the the user-level documentation[25, p. 9-2].

Although the reference-counting approach reduces unneeded copies, it introduces many redundant checks, requires space for the reference counts, and requires extra code to update the reference counts. In contrast, our approach was to develop static analyses that are part of the JIT compiler and which can statically determine where copies must be inserted without any of the overhead of reference counts.

Our approach has been implemented in McVM a garbage-collecting VM with a specializing JIT[9, 10]. As the system is garbage collected there is no need for reference counts and we felt that the copy problem could be effectively handled by static analyses inside the McVM JIT. Thus, we developed a pair of analyses, a forward analysis to locate all places where an array update requires a copy (*necessary copy analysis)* and then a backward analysis that moves the copies to the best location and which may eliminate redundant copies (*copy placement analysis*).

The paper is organized as follows: Section 2 describes the McLAB project and where the work in paper fits into that project, and Section 3 gives an overview of the problem and our general approach. Section 4 and Section 5 describe the forward and the backward analyses with examples; in Section 6, we briefly discuss how the forward analysis resolves conflicting names; Section 7 discuses the experimental results of our approach; we give some related work in Section 8 and Section 9 concludes the paper.

2 Background

The work presented in this paper is a component of our McLab system[3]. McLab provides an extensible set of compilation, analysis and execution tools built around the core MATLAB language. One goal of the McLab project is to provide an open-source set of tools for the programming languages and compiler community so that researchers (including our group) can develop new domain-specific language extensions and new compiler optimizations. A second goal is to provide these new languages and compilers to scientists and engineers to both provide languages more tailored to their needs and also better performance.

The McLab framework is outlined in Figure 1, with the shaded boxes indicating the components presented in this paper. The framework comprises of an extensible front-end, a high-level analysis and transformation engine and three backends. Currently there is support for the core MATLAB language and also a complete extension supporting ASPECTMATLAB[5, 6]. The front-end and the extensions are built using our group's extensible lexer, Metalexer[8] and JastAdd[12, 2]. There are three backends: McFor, a FORTRAN code generator[17]; a MATLAB generator (to use McLab as a source-to-source compiler); and McVM, a virtual machine that includes a simple interpreter and a sophisticated type-specialization based JIT compiler, which generates LLVM[16] code.

The techniques presented in this paper are part of McJIT, the JIT compiler for McVM. McJIT is built upon LLVM, the Boehm garbage collector[7], and several numerical libraries[4, 27]. For the purposes of this paper, it is important to realize that McJIT specializes code based on the function argument types that occur at run-time. When a function is called the VM checks to see if it already has a compiled version corresponding to the current argument types. If it does not, it applies a sequence of analyses including live variable analysis, type inference and array bounds check elimination. Finally, it generates LLVM code for this version.

When generating code McJIT assumes reference semantics, and not copy semantics for assignments between arrays and parameter passing. That is, arrays are dealt with as pointers and only the pointers are copied. Clearly this does not match the copy semantics specified for MATLAB and thus the need for the two shaded boxes in Figure 1 in order to determine where copies are required and

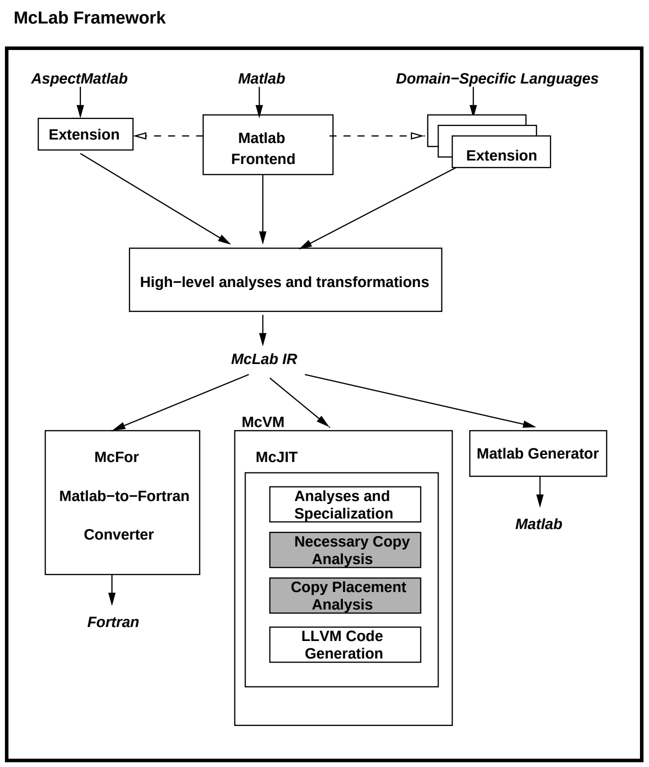

Figure 1: Overview of McLab (shaded boxes correspond to analyses presented in this paper)

the best location for the copies. These two analyses are the core of the techniques presented in this paper.

It is also important to note that the specialization and type inference in McJIT means that variables that certainly have scalar types will be stored in LLVM registers and thus the necessary copy and copy placement analyses only need to consider the remaining variables.

3 Problem and Overview of Our Approach

To properly understand our analysis, we first need to clearly define the problem. As we indicated in the introduction, all variables in MATLAB are assumed to be arrays. Array assignments and array parameter passing assumes copy semantics. Assignment statements in the MATLAB programming language have different forms, for example:

$$\begin{aligned} a &= zeros(10); && (1)\\ b &= a; && (2)\\ a(i) &= 2; && (3)\\ c &= myfunc(a,b); && (4) \end{aligned}$$

A naive implementation of the copy semantics for statements 1 - 4 above would involve making a copy at every assignment statement. Thus, in statement 1, the object (a 10x10 matrix) allocated by the function *zeros* would be copied into the variable *a*. The Matlab language defines a number of memory allocation functions similar to *zeros*. In statement 2, the array *a* would be copied into the variable *b* while in statement 3, the scalar 2, which is also a 1x1 array, would be copied into the element at index *i* of the array *a*. In statement 4, the arguments *a* and *b* in the call to the function *myfunc* would be copied into their corresponding parameters of the function; the return value of *myfunc* would also copied into the variable *c*.

With this naive strategy, every time a variable is defined from an existing object or a parameter is passed from one function to another or a value is returned from a function, a copy must be generated. Obviously, this is inefficient. A more advanced implementation can detect opportunities to convert copy-by-value to copy-by-reference, and similarly, convert call-by-value to call-by-reference.

Copy-by-reference or call-by-reference enables sharing of objects or data blocks. Octave[1] — an open source implementation of the Matlab programming language — uses a copy and call by reference strategy and lazily makes a copy before array writes, where necessary, to guarantee copy-by-value and call-by-value semantics. It implements reference counting to determine when copies should be generated. This involves performing a runtime check before each array update. We believe that Matlab uses a similar strategy. In fact, the MathWorks's documentation [25] is consistent with our position. Using this approach ensures that copies are made when needed. Unfortunately however, this strategy alone does not prevent the runtime system from generating unneeded copies. For example, consider the following code written in the Matlab programming language.

```
1:  function test1()
2:      a = rand(15000);
3:      b = a;
4:
5:      a = [1:10]
6:      disp(a(1:5));
7:      disp(b(1:5));
8:  end
```

```
1:  function test2()
2:      a = rand(15000);
3:      b = a;
4:      b(1) = 10;
5:      a = [1:10];
6:      disp(a(1:5));
7:      disp(b(1:5));
8:  end
```

The difference between *test1* and *test2* is in line 4 where a copy of the 15000x15000 matrix is made before the array element at index 1 is updated. The copying of the array referenced by both *a* and *b* before the update in line 4, is a useless operation since *a* is dead after line 4. This suggests that a liveness analysis is needed to complement reference counting in determining when a copy should be generated.

Our approach differs from both the naive approach and the lazy copy via reference-counting approach. McVM uses garbage collection instead of a reference counter-based memory manager. Thus, we have no need for reference counts, and instead we implement the lazy copying via static analysis. Intuitively this static analysis computes an abstraction of the sharing of arrays and then at every array write it determines if the array being written to is referenced by any other live variable. If so, then that assignment requires a copy. However, in our approach the copy statement is not immediately inserted, as there may be a better placement for the copy. We use a second analysis to determine the best places to insert the copy statements.

Our approach has the advantage of not requiring the the space and time overhead associated with reference counting, and it also does not require dynamic checks at each array update. Our approach will be successful if the analysis does not insert many spurious copies. As we will see in Section 7, on our benchmarks we inserted the minimal number of copies and avoided the frequent checks required the the reference-counting strategies.

In the next two sections we introduce the *necessary copy* and *copy placement* analyses. Remember that because of the type inference and specialization supported by McJIT, these analyses only need to consider the variables that are "real" arrays, and it does not have to consider variables that must be scalars.

4 Necessary Copy Analysis

The *necessary copy analysis* is a forward analysis that collects information that is used to determine whether a copy should be generated before an array is modified. To simplify our description of the analysis, we consider only simple assignment statements or the form $lhs = rhs$. It is straightforward to show that our analysis works for both single assignments (one *lhs* variable) and multiple assignment statements (multiple *lhs* variables). The analysis is implemented as a structured flow analysis on the AST intermediate representation used by McJIT. We describe the analysis by defining the following components.

Domain: the domain of the analysis' flow facts is the set of relations comprising of an array reference variable and the ID of the statement that allocates the memory for the array; henceforth called *allocators*. We write (a, s) if a references the array allocated at the statement s.

Problem Definition: at the program point p, a variable references a shared array if the number of variables that reference the array is greater than one. An array update via an array reference variable requires a copy if the variable *may* reference a shared array at p and at least one of the other variables that reference the same array is *live* after p.

Merge Operator: the merge operator is union; that is, $in(S_i) = \bigcup_{p \in pred(S_i)} out(p)$; where $in(S_i)$ is the input flow set reaching the statement S_i and $out(S_i)$ is the corresponding output set leaving S_i; *pred* is the set of the predecessors of S_i.

Flow Function: $out(S_i) = gen(S_i) \cup (in(S_i) - kill(S_i))$; $gen(S_i)$ and $kill(S_i)$ are respectively the set of flow facts generated and killed by the statement S_i.
Given the assignment statements of the forms:

$$\begin{aligned} S_i : a &= \texttt{alloc} \qquad (5)\\ S_i : a &= b \qquad (6)\\ S_i : a(j) &= x \qquad (7)\\ S_i : a &= f(arg_1, arg_2, ..., arg_n) \qquad (8)\end{aligned}$$

where S_i denotes a statement ID; `alloc` is a new memory allocation performed by statement S_i, a, b are array reference variables; x is a scalar; f is a function, $arg_1, arg_2, ..., arg_n$ denote the arguments passed to the function and the corresponding formal parameters are denoted with $p_1, p_2, ..., p_n$.

We partition $in(S_i)$ using allocators and the partition containing flow entries with the allocator m is:

$$Q_i(m) = \{(x,y)|(x,y) \in in(S_i) \wedge y = m\} \tag{9}$$

Now consider statements of type 6 above; if the variable b has a reaching definition at S_i then there must exist some $(b,m) \in in(S_i)$ and there exists a non-empty $Q_i(m)((b,m) \in Q_i(m))$.

In addition, if b may reference a shared array at S_i then $|Q_i(m)| > 1$. Let us call the set of all such $Q_i(m)$s, P_i. We write $P_i(a)$ if $in(S_i)$ is partitioned based on the allocators of the flow entries with the variable a. Considering statements of the form 7, $P_i(a) \neq \emptyset$ implies that a copy of a must be generated before executing S_i and in that case, S_i is a *copy generator*. This means that after this statement a will point to a fresh copy and no other variable will refer to this copy.

We are now ready to construct a table of *gen* and *kill* sets for the four assignment statement kinds above. To simply the table, we define

$Kill_{define} = \{(a,s)|(a,s) \in in(S_i)\}$
$Kill_{dead} = \{(c,s)|(c,s) \in in(S_i) \wedge \texttt{not live}(S_i,c)\}$
$Kill_{update} = \{(a,s)|(a,s) \in in(S_i) \wedge P_i(a) \neq \emptyset\}$

Stmt	**Gen set**	**Kill set**
(5)	$\{(a,S_i)\|\texttt{live}(S_i,a)\}$	$Kill_{define} \cup Kill_{dead}$
(6)	$\{(a,s)\|(b,s) \in in(S_i) \wedge \texttt{live}(S_i,a)\}$	$Kill_{define} \cup Kill_{dead}$
(7)	$\{(a,S_i)\|P_i(a) \neq \emptyset\}$	$Kill_{update} \cup Kill_{dead}$
(8)	see *gen*(f) below	$Kill_{define} \cup Kill_{dead}$

Computing the *gen* set for a function call is not straightforward. Certain built-in functions allocate memory blocks for arrays; such functions are categorized as *alloc functions*. A question that arises is: does the return value of the called function reference the same shared array as a parameter of the function? If the return value references the same array as a parameter of the function then this sharing must be made explicit in the caller, after the function call statement. Therefore, the *gen* set for a function call is defined as:

$$gen(f) = \begin{cases} \{(a,S_i)|\texttt{live}(S_i,a)\}, \\ \quad \text{if } \texttt{isAllocFunction}(f) \\ \\ \{(a,s)|(arg_j,s) \in in(S_i) \wedge \texttt{live}(S_i,a)\}, \\ \quad \text{if } \texttt{ret}(f) = p_j(f) \\ \\ \{(a,s)|arg \in args(f) \\ \quad \wedge (arg,s) \in in(S_i) \wedge \texttt{live}(S_i,a)\}, \\ \quad \texttt{otherwise} \end{cases}$$

The first alternative generates a flow entry (a,S_i) if the *rhs* is an alloc function and the *lhs*, a is live after the statement S_i; this makes statement S_i an allocator. In the second alternative, the analysis requests for the result of the necessary copy analysis on the function f from an analysis manager. The manager caches the result of the previous analysis on a given function. This is only updated if McJIT triggers a recompilation because the types of the arguments to the function have changed. From the result of the analysis on f, we determine the return variables of f that are aliases to the parameters of f and consequently aliases to the arguments of f. This is explained in detail under initialization. The return variable of f corresponds to the *lhs*, a in statement type 6. Therefore we generate flow entries from the entries of the arguments that the return variable may reference according to the summary information of f and provided that a is also *live* after S_i. The third alternative is conservative: flow entries are generated from all the flow entries of all the arguments to the function f. This is can happen if f cannot be analyzed because it is neither a user-defined function nor an alloc function.

Initialization: The input set for a function is initialized with a flow entry for each parameter and an additional flow entry (a shadow entry) for each parameter is also inserted. This is necessary in order to determine which of the parameters (if any) a return variable references. At the entry to a function, the input set is given as $in(entry) =$
$\{(p, S_p) | p \in Params(f)\} \cup \{(p', S_p) | p \in Params(f)\}$ We illustrate this scheme with an example. Given a function f, defined as:

```
function [u, v] = f(x, y)
   u = x;
   d  = y;
   v = d;
end
```

the *in* set at the entry of f is $\{(x, S_x), (x', S_x), (y, S_y), (y', S_y)\}$ and at the end of the function, the *out* set is
$\{(u, S_x), (x', S_x), (v, S_y), (y', S_y)\}$. We now know that u is an alias for the parameter x and v is an alias for y. We encode this as a vector of vectors, [[0], [1]] for the function f. This is useful during a call of f. For instance, in `[c, d] = f(a, b);` we can determine that c is an alias for the argument a and similarly, d is an alias for b by inspecting the summary information generated for f.

4.1 Simple Example

Let us illustrate how the analysis works with the following simple example.

```
1 function example1()
2    a = rand(15000);
3    b = a;
4    b(1) = 10;
5    a = [1:10];
6    disp(a (1:5));
7    disp(b (1:5);
8 end
```

Table I shows the flow information at each statement of the function, including the *gen*, *kill*, *in* and *out* sets. The statement number is shown in the first column of the table.

The analysis begins by initializing $in(S_2)$ to $\emptyset$ since the function does not have any parameters. The assignment statement S_2 is an allocator because the function *rand* is an alloc function. Table I shows that despite the assignment in line 3, no copy should be generated before the assignment in line 4. This is because the variable a defined in line 2 is no longer *live* after line 3 hence, S_4 is not a copy generator according to our definition and therefore no copy should be generated.

#	**Gen set**	**Kill set**	**In set**	**Out set**
2	$\{(a, S_2)\}$	$\emptyset$	$\emptyset$	$\{(a, S_2)\}$
3	$\{(b, S_2)$	$\{(a, S_2)\}$	$\{(a, S_2)\}$	$\{(b, S_2)\}$
4	$\emptyset$	$\emptyset$	$\{(b, S_2)\}$	$\{(b, S_2)\}$
5	$\{(a, S_5)$	$\emptyset$	$\{(b, S_2)\}$	$\{(b, S_2), (a, S_5)\}$

Table I: Forward Analysis result for *example1*

4.2 `if-else` Statement

So far we have been considering sequences of statements. Analyzing an `if-else` statement requires that we analyse all the alternative blocks and merge the result at the end of the `if-else` statement. We duplicate the *in* set reaching the `if-else` statement and pass a copy to each of the alternative blocks; each block is analyzed as a sequence of statements. We merge the result using the merge operator ($\cup$) after we have analyzed all the blocks of the `if-else` statement.

Let *blocks* denotes the set of all the alternative blocks of an `if-else` statement. The *out* set leaving the `if-else` statement is given as

$out(\texttt{if-else}) = \bigcup_{alternative \in blocks} out(alternative)$

4.3 Loops

Computing the the input and output sets entering and exiting a loop presents some challenges as it requires merging the flow sets coming from two different paths: one from the entry and another from the loop back-edge until a fixed point is reached. This on its own right does not present significant problems. However, when a copy statement occurs in a loop, it becomes necessary to distinguish between the sharing of arrays that are initiated in different iterations of the loop, and also to distinguish between those initiated within a loop from those initiated before the loop otherwise, unneeded copies may be generated. For example consider the following function:

```
function example2()
1: a = [1:2:30];
2: b = [2:2:30];
   while (i < 15)
3:     a(i) = 5;
4:     b = a;
5:     a(i+1) = 0;
   end
end
```

	iteration 1	**iteration 2**	**iteration 3**	**iteration 4**
$in(S_1)$	$\{\}$	$\{\}$	$\{\}$	$\{\}$
$out(S_1)$	$\{(a, S_1)\}$	$\{(a, S_1)\}$	$\{(a, S_1)\}$	$\{(a, S_1)\}$
$in(S_2)$	$\{(a, S_1)\}$	$\{(a, S_1)\}$	$\{(a, S_1)\}$	$\{(a, S_1)\}$
$out(S_2)$	$\{(a, S_1), (b, S_2)\}$	$\{(a, S_1), (b, S_2)\}$	$\{(a, S_1), (b, S_2)\}$	$\{(a, S_1), (b, S_2)\}$
$in(S_3)$	$\{(a, S_1), (b, S_2)\}$	$\{(a, S_5), (a, S_1), (b, S_1), (b, S_2)\}$	$\{(a, S_1), (b, S_2), (a, S_5), (b, S_1), (b, S_3)\}$	$\{(a, S_1), (b, S_2), (a, S_5), (b, S_1), (b, S_3)\}$
$out(S_3)$	$\{(a, S_1), (b, S_2)\}$	$\{(a, S_3), (b, S_1), (b, S_2)\}$	$\{(a, S_3), (b, S_1), (b, S_2), (b, S_3)\}$	$\{(a, S_3), (b, S_1), (b, S_2), (b, S_3)\}$
$in(S_4)$	$\{(a, S_1), (b, S_2)\}$	$\{(a, S_3), (b, S_1), (b, S_2)\}$	$\{(a, S_3), (b, S_1), (b, S_2), (b, S_3)\}$	$\{(a, S_3), (b, S_1), (b, S_2), (b, S_3)\}$
$out(S_4)$	$\{(a, S_1), (b, S_1)\}$	$\{(a, S_3), (b, S_3)\}$	$\{(a, S_3), (b, S_3)\}$	$\{(a, S_3), (b, S_3)\}$
$in(S_5)$	$\{(a, S_1), (b, S_1)\}$	$\{(a, S_3), (b, S_3)\}$	$\{(a, S_3), (b, S_3)\}$	$\{(a, S_3), (b, S_3)\}$
$out(S_5)$	$\{(a, S_5), (b, S_1)\}$	$\{(a, S_5), (b, S_3)\}$	$\{(a, S_5), (b, S_3)\}$	$\{(a, S_5), (b, S_3)\}$

Table II: Flow sets for the first four iterations of the analysis for *example2*

Table II shows the first four iterations of the analysis for *example2* above. A fixed point is reached in the fourth iteration. After the first iteration, the result of the merge of $out(S_5)$ with $out(S_2)$ suggests that a and b may reference the same array (allocated at S_1) at the statement S_3. But at the end of the first iteration and just before the beginning of the second iteration a definitely references the array 'allocated' at S_5. Observe from the table that $P_3(a)$ and $P_5(a)$ are non-empty. This suggests that two copies are needed when actually only one copy, at S_5, is all that is required. The merge has introduced a spurious copy at S_3.

The problem is resolved by viewing a loop as a linear sequence of statements in a manner analogous to a complete unrolling of the loop. With this view, the unique effect of each statement of the loop in different iterations can be determined. We distinguish the flow entries generated before a loop from those generated within the loop by assigning an initial context number to every loop in an increasing order of nesting. The main sequence is assigned a context number zero. When a loop is encountered, we assign the current context number plus 1 as the initial context number of the current. We consider each iteration of the loop as a new context, therefore a loop's current context number is given as the loop's
initial-context number + current iteration count - 1.
For example, if the initial context number of a loop is 1 and the analysis is in the second iteration of the loop, the current context number is 2.

To ensure that a flow entry generated in a context reflects the context in which it has been generated, we introduce a third entity — named *context number* to the flow entry object. Any flow entry generated by an allocator or a copy generator within a loop has its context number set to the current loop context number. This allows us to distinguish entries generated in different iterations of the loop, and to distinguish those entries generated before the loop from those generated within it. If an array copy statement is encountered within a loop, as in line 4 above, we check the context number of the flow entries that correspond to the right-hand-side of the assignment statement, and if any is less than the initial loop context number, we generate a copy of the entry and assign the initial-loop-context number to the context number of the shadow entry. Flow entries for the left-hand-side are then generated from the shadow entries of the right-hand-side only. With this, Equation 9 above becomes

$$Q_i(m,c) = \{(x,y,z) | (x,y,z) \in in(S_i) \wedge (y,z) = (m,c)\} \tag{10}$$

At the beginning of every iteration, we merge the flow sets coming from the main path into the loop with the one from the loop back-edge, which is empty in the first iteration of the loop. To compute the fixed point for the analysis, given any two flow entries
$entry1 = (x,y,z)$ and $entry2 = (u,v,w)$,
$entry1$ equals $entry2$ if:

1. $x = u \wedge y = v \wedge z = w$ **OR**
2. $x = u \wedge y = v \wedge abs(z - w) = 1$

Therefore Table II becomes Table III. A fixed point is reached when flow sets in two consecutive iterations are *equal* according to the definition for equality of flow entries above; in our example, after iteration 3.

	iteration 1	**iteration 2**	**iteration 3**
$in(S_1)$	$\{\}$	$\{\}$	$\{\}$
$out(S_1)$	$\{(a, S_1, 0)\}$	$\{(a, S_1, 0)\}$	$\{(a, S_1, 0)\}$
$in(S_2)$	$\{(a, S_1, 0)\}$	$\{(a, S_1, 0)\}$	$\{(a, S_1, 0)\}$
$out(S_2)$	$\{(a, S_1, 0), (b, S_2, 0)\}$	$\{(a, S_1, 0), (b, S_2, 0)\}$	$\{(a, S_1, 0), (b, S_2, 0)\}$
$in(S_3)$	$\{(a, S_1, 0), (b, S_2, 0)\}$	$\{(a, S_1, 0), (a, S_5, 1), (b, S_2, 0), (b, S_1, 1)\}$	$\{(a, S_1, 0), (a, S_5, 2), (b, S_2, 0), (b, S_1, 1), (b, S_5, 1)\}$
$out(S_3)$	$\{(a, S_1, 0), (b, S_2, 0)\}$	$\{(a, S_1, 0), (a, S_5, 1), (b, S_2, 0), (b, S_1, 1)\}$	$\{(a, S_1, 0), (a, S_5, 2), (b, S_2, 0), (b, S_1, 1), (b, S_5, 1)\}$
$in(S_4)$	$\{(a, S_1, 0), (b, S_2, 0)\}$	$\{(a, S_1, 0), (a, S_5, 1), (b, S_2, 0), (b, S_1, 1)\}$	$\{(a, S_1, 0), (a, S_5, 2), (b, S_2, 0), (b, S_1, 1), (b, S_5, 1)\}$
$out(S_4)$	$\{(a, S_1, 0), (a, S_1, 1), (b, S_1, 1)\}$	$\{(a, S_1, 0), (a, S_1, 1), (a, S_5, 1), (b, S_1, 1), (b, S_5, 1)\}$	$\{(a, S_1, 0), (a, S_1, 1), (a, S_5, 2), (b, S_1, 1), (b, S_5, 2)\}$
$in(S_5)$	$\{(a, S_1, 0), (a, S_1, 1), (b, S_1, 1)\}$	$\{(a, S_1, 0), (a, S_1, 1), (a, S_5, 1), (b, S_1, 1), (b, S_5, 1)\}$	$\{(a, S_1, 0), (a, S_1, 1)(a, S_5, 2), (b, S_1, 1), (b, S_5, 2)\}$
$out(S_5)$	$\{(a, S_5, 1), (b, S_1, 1)\}$	$\{(a, S_5, 2), (b, S_1, 1), (b, S_5, 1)\}$	$\{(a, S_5, 3), (b, S_1, 1), (b, S_5, 2)\}$

Table III: Analysis result with iteration count for the first three iterations.

As shown in Table III, this strategy also ensures that unneeded copies are not generated by eliminating false dependency since S_3 is not a copy generator (i.e., $P_3(a) = \emptyset$).

5 Copy Placement Analysis

In the previous section, we described the forward analysis which determines whether a copy should be generated before an array is updated. One could use this analysis alone to insert the copy statements, but this may not lead to the best placement of the copies and may lead to redundant copies. The backward *copy placement analysis* determines a better placement of the copies, while at the same time ensuring safe updates of a shared array. Examples of moving copies include hoisting copies out of if-then constructs and out of loops.

The *copy placement analysis* uses the information collected in the forward analysis. In particular the analysis uses the input set, generated, and partition sets at an assignment statement. Like the forward analysis, it a structure-based analysis that is performed on the low-level AST representation used by McJIT.

The intution behind this analysis is that often it is better to perform the array copy at, or near, the statement which created the sharing (i.e. statements of the form $a = b$), rather than creating the copy at the array update statement (a statement of the form *a(i)* $= b$). In particular, if the update statement is inside a loop, but the statement that created the sharing is outside the loop, then it is much better to create the copy outside of the loop.

Thus, the *copy placement analysis* is a backwards analysis that pushes the necessary copies upwards as far as possible, often it pushes the copy up to the statement that created the sharing, which is ideal.

5.1 Copy Placement Analysis Details

A copy entry is represented as a three-tuple:

$$e =< copy\_loc, var, alloc\_site > \tag{11}$$

where *copy_loc* denotes the ID of the node that generates the copy, *var* represents variable holding a reference to the array that should be copied and *alloc_site* is the allocation site where the array referenced by *var* was allocated. We refer to the three components of the three-tuple as *e.copy_loc*, *e.var*, and *e.alloc_site*.

Let C denote the set of all copies generated by a function.

Given a function, the analysis begins by traversing the block of statements of the function backward. The domain of the analysis' flow entries is the set of copy objects and the merge operator is intersection.

Define C_{out} as the set of copy objects at the exit of a block and C_{in} as the set of copy objects at the entrance of a block. Since the analysis begins at the end of a function, C_{out} is initialized to $\emptyset$. The rules for generating and placing copies are described here.

5.1.1 Statement Sequence

Given a sequence of statements, we are given a C_{out} for this block and the analysis traverses backwards through the block computing a C_{in} for the block. As each statement is traversed the following rules are applied for the different kinds of the assignment statements in the sequence. When we refer to $in(S_i)$, $Q_i(m)$, $P_i(a)$, we are referring to the entities defined in Section 4.

Rule 1: array updates, $S_i : a(y) = x$ **:** Recall from Section 4 that $P_i(a)$ is the set of different partitions of $in(S_i)$ with shared arrays (based on the different allocators and the context information of the flow entries of the variable a).

Given that the array variable of the *lhs* of the statement S_i is a, when a statement of this form is reached, we add a copy for each partition for a shared array to the current copy set. Thus

$$C_{in} := C_{in} \cup \left\{ \begin{array}{rl} \emptyset & \text{if } P_i(a) = \emptyset \\ \{< S_i, a, m > | (Q_i(m) \in P_i(a)\} & \text{otherwise} \end{array} \right.$$

Rule 2: array assignments, $S_j : a = b$ **:** If in the current block, $\exists e \in C_{in}(e.var = a \texttt{ or } e.var = b)$ we remove e from the current copy flow set C_{in}. This means that the copy has been placed at its current location. Otherwise, we check the copy set, C_{out} at the exit of the current block. If the copy is found in C_{out}, we perform the following:

- if $P_j(a) = \emptyset$, this is usually the case, we move the copy from the statement *e.copy_loc* to S_j and remove e from the flow set. The copy e has now been finally placed.
- if $P_j(a) \neq \emptyset$, $\forall(Q_i(m) \in P_j(a))$, we add a runtime equality test for a against the array reference variable x ($x \neq a$) of each member of $Q_i(m)$ at the statement *e.copy_loc*. This indicates that there is at least a definition of a that dominates this statement and for which a references a shared array. In addition to that, because the copy e was generated after the current block there are two different paths to the statement *e.copy_loc*, the current location of e. We place a copy of e at the current statement S_j and remove e from the flow set. Note that two copies of e have been placed; one at *e.copy_loc* and another at S_j. However, runtime guards have also been placed at *e.copy_loc*, ensuring that only one of these two copies materializes at runtime. In practice however, such checks are rarely generated. The following code snippet illustrates this scenario.

```
S1: b = [2, 4, 8];
S2: a = b;
    if ( ... )
S3:   c = rand(10);
```

```
     ...
S4:    a = c;
     end
S5: a(i) = 10;
S6: disp(a);
S7: disp(b);
```

The statement S_2 dominates the statement S_3; if the `if` block is taken then a references the array allocated at S_3 otherwise, a references the array allocated at S_1. By placing a copy after S_4, it is guaranteed that a is unique if the program takes the path through S_4 and the update at S_5 is therefore safe and no copy will be generated at S_5 because the runtime guard will be false. However, if this path is not taken, then the guard at S_5 will be true and a copy will be generated.

We expect that such guards will not usually be needed, and in fact none of our benchmarks required any guards.

5.1.2 `if-else` Statements

Let C_{if} and C_{else} denote the set of copies generated in an `if` and an `else` block respectively.

First we compute

$$C' := (C_{out} \cap C_{else}) \cup (C_{out} \cap C_{if}) \cup (C_{if} \cap C_{else})$$

Then we compute the differences

$$\begin{aligned} C_{out} &:= C_{out} \setminus C' \\ C_{else} &:= C_{else} \setminus C' \\ C_{if} &:= C_{if} \setminus C' \end{aligned}$$

to separate those copies that do not intersect with those in other blocks but should nevertheless be propagated upward. Since the copies in the intersection will be relocated, they are removed from their current locations.

And finally,

$$\begin{aligned} & C_{in} := C_{out} \cup C_{else} \cup C_{if} \cup \\ & \{< S_{IF}, e.var, e.alloc\_site > | e \in C'\}) \end{aligned}$$

Note that a copy object e with its first component set to S_{IF} is attached to the *if-else* statement S_{IF}. That means if these copies remain at this location, the copies should be generated before the *if-else* statement.

5.1.3 Loops

The main goal here is to identify copies that could be moved out of a loop. To place copies generated in a loop, we apply the rules for statement sequence and the `if-else` statement. The analysis propagates copies upward from the inner-most loop to the outer-most loop and to the main sequence until either loop dependencies exist in the current loop or it is no longer possible to move the copy according to Rule 2 in Section 5.1.1.

An alternative to propagating copies out of a loop is to generate copies in the loop's header so that if the loop does not execute, no copies will be generated. However with this strategy, in a nested loop, copies initiated in an inner loop and that could be generated outside an outer loop would be generated multiple times in different iterations of the outer loop. Furthermore, if there are two or more adjacent loops, and identical copies are generated in the loops, generating copies at loop header will generate the same copies in the loop headers of the adjacent loops, provided that more than one of the loops are executed.

A disadvantage of propogating the copy outside of the loop is that if none of the loops that require copies are executed then we would have generated a useless copy before any of the loop. However, the execution is still correct. For this reason, we assume that a loop will *always* be executed and generate copies outside loops, wherever possible. This is a reasonable assumption because a loop is typically programmed to execute. Although it is possible that none of the loops is executed, this rarely happens in practice. With this assumption, there is no need to compute the intersection of C_{loop} and C_{out}. Hence

$$C_{in} := C_{out} \cup \{< S_{loop}, e.var, e.alloc\_site > |e \in C_{loop}\})$$

5.2 Using the Analyses

This section illustrates how the combination of the forward and the backward analyses is used to determine the optimal copies that should be generated. First consider the following program, *test3*. Again, we begin by computing the flow information for the forward analysis. Table IV shows the result of the forward analysis; the context number is 0 in all the flow set and is therefore omitted from the flow entries shown in the table.

```
function test3()
  a = [1:5]
  b = a
  i = 1;
  if (i > 2)
      a(1) = 100;
  else
     a(1) = 700;
  end
  a(1) = 200;
  disp(a);
  disp(b);
end
```

Table V gives the result of the backward analysis. The

#	Gen set	In	Out
2	$\{(a, S_2)\}$	$\emptyset$	$\{(a, S_2)\}$
3	$\{(b, S_2)$	$\{(a, S_2)\}$	$\{(a, S_2)(b, S_2)\}$
6	$\{a, S_6\}$	$\{(a, S_2), (b, S_2)\}$	$\{(b, S_2)(a, S_6)\}$
8	$\{(a, S_8)$	$\{(a, S_2), (b, S_2)\}$	$\{(b, S_2), (a, S_8)\}$
10	$\emptyset$	$\{(b, S_2), (a, S_6),$ $(a, S_8)\}$	$\{(b, S_2), (a, S_6)$ $(a, S_8)\}$

Table IV: Necessary Copy Analysis Result for *test3*

#	C_{out}	C_{in}	Current Result
10	$\emptyset$	$\emptyset$	$\emptyset$
8	$\emptyset$	$\{< S_8, a, S_2 >\}$	$\{(a, S_8)\}$
6	$\emptyset$	$\{< S_6, a, S_2 >\}$	$\{(a, S_6)\}$
I	$\emptyset$	$\{< S_I, a, S_2 >\}$	$\{(a, S_I)\}$
3	$\{< S_I, a, S_2 >\}$	$\emptyset$	$\{(a, S_I)\}$
2	$\emptyset$	$\emptyset$	$\{(a, S_I)\}$

Table V: Copy Placement Analysis Result for *test3*

I used in Table V stands for the if-else statement in *test3*. The backward analysis begins from line 12 of the function *test3*. The out set C_{out} is empty before this statement. At line 10, C_{out} is still empty. When the `if-else` statement is reached, a copy of C_{out} ($\emptyset$) is passed to the *Else* block and another copy of C_{out} is also passed to the *If* block. The copy $\{< S_8, a, S_2 >$ is generated in the *Else* block because $|Q(S_2) = \{(a, S_2), (b, S_2)\}| = 2$, hence $P_i(a) \neq \emptyset$. Similarly $< S_6, a, S_2 >$ is generated in the *If* block.

By applying the rule for `if-else` statement described in Section 5.1.2, the outputs of the *If* and the *Else* blocks are merged to obtain the result at S_I (the if-else statement). Applying Rule 2 for statement sequence (Section 5.1.1) in S_3, $< S_I, a, S_2 >$ is removed from C_{in} and the analysis terminates at S_2. The final result is that a copy must be generated before the `if-else` statement instead of generating two copies, one in each block of the `if-else` statement. This example illustrates how common copies generated in the alternative blocks of an `if-else` statement could be combined and propagated upward to reduce code size.

The second example, *tridisolve* is a MATLAB function from [24]. The forward analysis information is shown in Table VI. The table shows the *gen*, *in* and *out* sets at each relevant assignment statement of the function *tridisolve*. The results in different loop iterations are shown using a subscript to represent the loop iteration. For example, the row number 25_2 refers to the result at the statement labelled S_{25} in the second iteration of the loop. The Analysis reached a fixed point after the third iteration. At the function's entry, the *in* set is initialized with two flow entries for each parameter of the function — one for the parameter and the other for a shadow entry.

```
function x = tridisolve(a,b,c,d)
% TRIDISOLVE Solve tridiagonal system of
% equations.
% x = TRIDISOLVE(a,b,c,d) solves the system
% of linear equations
%     b(1)*x(1) + c(1)*x(2) = d(1),
%     a(j-1)*x(j-1) + b(j)*x(j) +
%     c(j)*x(j+1) = d(j), j = 2:n-1,
%     a(n-1)*x(n-1) + b(n)*x(n) = d(n).
%
% The algorithm does not use pivoting,
12 % so the results might be inaccurate if
13 % abs(b) is much smaller than abs(a)+abs(c).
14 % More robust, but slower, alternatives with
15 % pivoting are: x = T\d where T = diag(a,-1) +
16 % diag(b,0) + diag(c,1)
17 % x = S\d where S =
18 % spdiags([[a; 0] b [0; c]],[-1 0 1],n,n)
19
20 x = d;
21 n = length(x);
22
23 for j = 1:n-1
24     mu = a(j)/b(j);
25     b(j+1) = b(j+1) - mu*c(j);
26     x(j+1) = x(j+1) - mu*x(j);
27 end
28
29 x(n) = x(n)/b(n);
30 for j = n-1:-1:1
31     x(j) = (x(j)-c(j)*x(j+1))/b(j);
32 end
33 end
```

The analysis continues by generating the *gen*, *in* and *out* sets according to the rules specified in Section 4. Notice that statement S_{25} is an allocator because $P_{25}(b) \neq \emptyset$ since $|Q_{25}(S_b)| = |\{(b, S_b, 0), (b', S_b, 0)\}| > 1$. Similarly, S_{26} and S_{29} are also allocators. This means that generating a copy of the array referenced by the variable b just before executing the statement S_{25} ensures a safe update of the array. The same is true of the array referenced by the variable x in lines 26 and 29. However, are these the best points in the program to generate those copies? Could the number of copies be reduced? We provide the answers to these questions when we examine the results of the backward analysis.

The backward analysis accepts the result of the forward analysis on a function together with the function that is then traversed from the back, beginning with the last statement. Table VII shows the copy placement analysis information at each relevant statement of the function. Recall that the placement analysis is based on blocks. It works by traversing the statements in each block of a function backward. In the case of the function *tridisolve*, the analysis begins in line 31 in the second `for` loop of the function. The set C_{out} is passed to the loop body and is initially empty. The set C_{in} stores all the copies generated in the block of the *for* statement. Line 31 is neither a definition nor an allocator, therefore no changes are recorded at this stage of the analysis.

At the beginning of the loop F_2, the analysis merges with the main path and the result at this point in the function is shown in the line labelled F_2. The statement S_{29} generated a copy as indicated by the forward analysis, therefore C_{in} is updated and the output set is also updated. The analysis then branches off to the first loop and the current C_{in} is passed to the loop body as C_{out}. The copies generated in the loop F_1 are stored in C_{in}, which is then merged with C_{out} at the beginning of the loop to arrive at the result in line F_1. The output set is also updated accordingly; at this stage, the number of copies has been reduced by 1 as shown in the column labelled *Current Result* of the table. The copy flow set that reaches the beginning of the function is non-empty. This suggests that the definition or the allocator of the array variables of the remaining entries could not be reached. Therefore, the array variables of the flow entries *must* be the parameters of the

20	Gen: $\{(x,S_d,0)\}$
	In: $\{(a,S_a,0),(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d,S_d,0),(d',S_d,0)\}$
	Out: $\{(a,S_a,0),(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0)\}$
25_1	Gen: $\{(b,S_{25},1)\}$
	In: $\{(a,S_a,0),(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0)\}$
	Out: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0),(b,S_{25},1)\}$
26_1	Gen: $\{(x,S_{26},1)\}$
	In: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0),(b,S_25,1)\}$
	Out: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(b,S_{25},1),(x,S_{26},1)\}$
25_2	Gen: $\{(b,S_25,2)\}$
	In: $\{(a,S_a,0),(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0),(b,S_{25},1),(x,S_{26},1)\}$
	Out: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0),(b,S_{25},2),(x,S_{26},1)\}$
26_2	Gen: $\{(x,S_{26},2)\}$
	In: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0),(b,S_{25},2),(x,S_{26},1)\}$
	Out: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(b,S_{25},2),(x,S_{26},2)\}$
25_3	Gen: $\{(b,S_{25},3)\}$
	In: $\{(a,S_a,0),(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0),(b,S_{25},2),(x,S_{26},2)\}$
	Out: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0),(b,S_{25},3),(x,S_{26},2)\}$
26_3	Gen: $\{(x,S_{26},3)\}$
	In: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0),(b,S_{25},3),(x,S_{26},2)\}$
	Out: $\{(a,S_a,0),(a',S_a,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(b,S_{25},3),(x,S_{26},3)\}$
29	Gen: $\{(x,S_{29},0)\}$
	In: $\{(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(x,S_d,0)(b,S_{25},3),(x,S_{26},3)\}$
	Out: $\{(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(b,S_{25},3),(x,S_{29},0)\}\}$
31_1	Gen: $\emptyset$
	In: $\{(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(b,S_{25},3),(x,S_{29},0)\}\}$
	Out: $\{(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(b,S_{25},3),(x,S_{29},0)\}\}$
31_2	Gen: $\emptyset$
	In: $\{(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(b,S_{25},3),(x,S_{29},0)\}\}$
	Out: $\{(a',S_a,0),(b,S_b,0),(b',S_b,0),(c,S_c,0),(c',S_c,0),(d',S_d,0),(b,S_{25},3),(x,S_{29},0)\}\}$

Table VI: Forward Analysis Result for *tridisolve*

#	C_{out}	C_{in}	**Current Result**
31	$\emptyset$	$\emptyset$	$\emptyset$
F_2	$\emptyset$	$\emptyset$	$\emptyset$
29	$\emptyset$	$\{(S_{29},a,S_d)\}$	$\{(x,S_{29})\}$
26	$\{(S_{29},x,S_d)\}$	$\{(S_{26},x,S_d)\}$	$\{(x,S_{29}),(x,S_{26})\}$
25	$\{(S_{29},x,S_d)\}$	$\{(S_{25},b,S_b),(S_{26},x,S_d)\}$	$\{(x,S_{29}),(x,S_{26}),(b,S_{25})\}$
F_1	$\{(S_{29},x,S_d)\}$	$\{(S_{F_1},x,S_d),(S_{25},b,S_b)\}$	$\{(x,S_{F_1}),(b,S_{25})\}$
20	$\emptyset$	$\{(S_{25},b,S_b)\}$	$\{(x,S_{F_1}),(b,S_{25})\}$
0	$\emptyset$	$\emptyset$	$\{(x,S_{F_1}),(b,S_0)\}$

Table VII: Backward Analysis Result for *tridisolve*

function and the necessary copy should be generated at the function entry. Hence a copy of the array referenced by b must be generated at the entry of the function *tridisolve*.

It is interesting to note that the number of copies has been reduced and all the copies generated in the loop F_2 were successfully moved out of the loop because there was no "loop dependency". The two copies generated are necessary to ensure that the arguments to the function by the callers are not updated. Even though a, b, c, d are parameters of the function, a and c are read but not written by the function hence no copies were generated for a and c. However, attempt to update the array referenced by d indirectly via x generated a copy. And updating the array referenced by b also generated a copy.

6 Name Resolution

MATLAB views an array as a mapping from the array index type to the array element type and therefore uses the same syntax for both function calls and array accesses. The obvious advantage of doing this is that a data structure initially implemented as an array could be re-implemented as a function without changing the array accesses. The disadvantage of doing this is that it makes efficient compilation difficult. For instance, in the statement below, is b a function or an array?

```
m = b(c, d);
```

Without a suitable analysis, it is hard to tell whether $b(c, d)$ is a function call or an array access. The forward analysis described in Section 4 relies on the McVM type inference analysis [10, 9] to determine the type of a symbol. In the simple assignment statement above, the analysis needs to know whether the variables m, c and d are arrays. And, if b is a function and m, c and d are arrays, the analysis needs to know whether m references the same array as c or d. The forward analysis requests the type information of b and proceeds to analyse b if the result of the look-up indicates that b is a function.

7 Experimental Results

To test our approach, we set up experiments using benchmarks from our benchmarks set which includes benchmarks from disparate sources including those from [21, 24, 20]. The benchmarks implement some algorithms and applications in numerical computing. Table 7 gives a short description of the benchmarks together with the results of our analysis. For all of our experiments we ran the benchmarks on their smallest input size.

The purpose of our experiment was two-fold. First, we wanted to measure the number of array updates performed by the benchmark at run-time (reported in the column labeled *# Array Updates*). This gives an idea of how many dynamic checks a reference-count-based scheme for lazy copying, such as used by Octave and Matlab, would have to perform. Remember that our approach does not usually require any dynamic checks. We determined the array update counts using AspectMatlab, by defining an aspect that matched all array updates and counted them.

The second important measurement is the number of copies generated at run-time. Clearly we would like to verify that our approach did not increase the number of copies as compared to the other approaches. The column *# Copies McVM* gives the the number of copies generated by a benchmark in McVM, that is, using our analysis.

To count the number of copies that are likely to be generated by a benchmark running under Mathwork's implementation of MATLAB, we developed an aspect to profile the benchmarks. We show this result under the column labelled *# Copies Aspect*. The aspect will be described shortly. We also instrumented Octave to count the number of copies made for each benchmark; this is given under the column labelled *# Copies Octave*.

Benchmark		# Array Updates	# Copies McVM	# Copies Aspect	# Copies Octave
adpt	adaptive quadrature using Simpson's rule	19624	0	0	0
capr	capacitance of a transmission line using finite difference and Gauss-Seidel iteration	9790800	10000	10000	10000
clos	transitive closure of a directed graph	2954	0	0	0
crni	Crank-Nicholson solution to the one-dimensional heat equation	21143907	4598	4598	6898
dich	Dirichlet solution to Laplace's equation	6935292	0	0	0
diff	diffraction pattern calculator	0	0	0	0
fdtd	3D FDTD of a hexahedral cavity with conducting walls	803	0	0	0
fft	fast fourier transform	44038144	1	1	1
fiff	finite-difference solution to the wave equation	12243000	0	0	0
mbrt	mandelbrot set	5929	0	0	0
nb1d	N-body problem coded using 1d arrays for the displacement vectors.	55020	0	0	0
nb3d	N-body problem coded using 3d arrays for the displacement vectors.	4878	0	0	0
nfrc	computes a newton fractal in the complex plane -2..2,-2i..2i	12800	0	0	0
trid	Solve tridiagonal system of equations	2998	2	2	2

Table VIII: The benchmarks and the results of the copy elimination analysis.

Mathwork's implementation MATLAB is a proprietary system and thus we were unable to instrument it to make direct measurements. Thus, we developed an alternative approach by instrumenting the benchmark programs themselves via aspects using our ASPECTMATLAB compiler *amc* [6]. The *amc* compiler accepts a MATLAB program and an aspect written in ASPECTMATLAB language — an extension of the MATLAB programming language. Our aspect defines all the patterns or the relevant points in a MATLAB program including all array definitions, array updates, and function calls. It also specifies the actions that should be taken at these points in the source program. In effect, the aspect computes all of the information that a reference-counting-based scheme would have, and thus can determine, at runtime, when an array update triggers a copy because the number of references to the array is greater than one. The aspect also records the total number of such copies for each benchmark.

Since Octave is an open source implementation we were able to add instrumentation code to Octave to obtain the count of all the copies made by the Octave interpreter during the execution of a benchmark.

At a high-level, the results in Table 7 show that our benchmarks often perform a significant number of array updates, but very few updates trigger copies. We observed that no copies were generated in ten out of the fourteen benchmarks. This low rate for array copies is not surprising because MATLAB programmers tend to avoid copying large objects and often only read from function parameters.[1]

If the majority of the MATLAB applications avoid making many copies of arrays, having to perform a runtime check against every array update, as is done in Octave, becomes a redundant operation. The *fft* benchmark performed 44,038,144 array updates. This translates to 44,038,144 runtime checks performed by Octave and in which only one resulted in a copy. We believe that the Mathwork's MATLAB system generates a comparable number of checks, although it may implement some redundant check removals. The *fiff* benchmark made 21,143,907 array updates, and none of the updates triggered a copy. In this particular case, all the runtime checks performed by Octave are redundant. Using our analysis, McVM avoids generating any runtime checks in all these cases.

In all the benchmarks, McVM generated no more copies than the number generated in Octave and the number measured using the aspect. In one benchmark, Octave generated more copies than McVM. This suggests that for all the benchmarks, our analysis effectively determined the optimal number of copies needed to guarantee copy semantics and in all cases avoids generating runtime checks.

Two benchmarks generated a lot of copies: *capr* generated 10,000 copies while *crni* generated 4598 copies. Further examination of the two benchmarks reveals that a copy is generated for each invocation of a function called 10,000 times by the *capr* program. Similarly, two copies were generated in a function called 2299 times by the *crni* program. Except the extra copies that were generated by *crni* running under Octave, all other copies generated were found to be array parameters passed from one function to another and updated in the called function. It was found that Octave generated 6898 copies for *crni* because it generated three copies in the function *tridiagonal*; McVM and the aspect generated two copies in this function.

So, the bottom line is that a very low fraction of array updates result in copies, and frequently no copies are necessary. For our benchmark set our static analysis determined the optimal number of copies, while at the same time avoiding all the overhead of dynamic checks. The secondary benefit of our approach is that it does not require reference counting, so it can be implemented in garbage-collected systems like McVM.

8 Related Work

Redundant copy elimination is a hard problem and implementations of languages such as Python are able to avoid copy elimination optimizations by providing multiple data structures: some with copy semantics and others with reference semantics. Programmers decide when to use mutable data structures. However, efficient implementations of languages like the MATLAB programming language and Sequoia [13] that use copy semantics require copy elimination optimization. The problem is similar to the aggregate update problem in functional languages.

The aggregate update problem has been studied extensively in the context of functional languages [15, 19, 22, 23, 26, 14, 11]. To modify an aggregate in a strict functional language, a copy of the aggregate must be made. This is in contrast to the imperative programming languages where an

[1]You may note that the *diff* benchmark performed no array updates. This benchmark performs a lot of scalar operations and array reads, but does not perform array updates. Thus, McJIT already handles all of the writes by detecting that they are scalars and allocating them to LLVM registers.

aggregate may be modified multiple times. Hudak and Bloss [15] use an approach based on abstract interpretation and conventional flow analysis to detect cases where an aggregate may be modified in place. Their method combines static analysis and dynamic techniques. It involves a rearrangement of the execution order or an optimized version of reference counting, where the static analysis fails. Our approach is based on flow analysis but we do not change the execution order of a program.

Interprocedural aliasing and the side-effect problem [18] is related to the copy elimination problem. By using call by reference semantics, when an argument is passed to a function during a call, the parameter becomes an alias for the argument in the caller and if the argument contains an array reference, the referenced array becomes a shared array; any updates via the parameter in the callee updates the same array referenced by the corresponding argument in the caller. Without performing a separate and expensive flow analysis, our approach easily detects aliasing and side effects in functions. Wand and Clinger present [26] interprocedural flow analyses for aliasing and liveness based on set constraints. They present two operational semantics: the first one permits destructive updates of arrays while the other does not. They also define a transformation from a strict functional language to a language that allows destructive updates. Like Wand and Clinger, our approach combines liveness analysis with flow analysis. However,unlike Wand and Clinger, we have implemented our analysis in a JIT compiler for an imperative language.

9 Conclusions and Future Work

In this paper we have presented an approach for using static analysis to determine where to insert array copies in order to implement the array copy semantics in Matlab. Unlike previous approaches, which used a reference-counting scheme and dynamic checks, our approach is implemented as a pair of static analyses in the McJIT compiler. These are the forwards *necessary copy analysis* that determines which array update statements trigger copies, and the backwards *copy placement analysis* that determines good places to insert the array copies. Both of these analyses have been implemented as structure-based analyses on the McJIT intermediate representation.

The advantages of our approach are that it does not require frequent dynamic checks, nor do we need the space and time overhead to maintain the reference counts. Our approach is particularly appealing in the context of a garbage-collected VM, such as the one we are working with.

Our experimental results validate that, on our benchmark set, we do not introduce any more copies than the reference-counting approach, and we eliminate all dynamic checks.

The work presented in this paper means that McJIT can use efficient call-by-reference and copy-by-reference implementations for arrays most of the time, introducing copies only when necessary to maintain the Matlab call-by-value and copy-by-value semantics.

We are continuing to fine-tune these and our other McJIT analyses and we plan to release the framework under an open-source license for other research groups to build upon.

References

[1] GNU Octave. `http://www.gnu.org/software/octave/index.html`.

[2] JastAdd. `http://jastadd.org/`.

[3] McLab. `http://www.sable.mcgill.ca/mclab/`.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[5] T. Aslam. AspectMatlab: An Aspect-Oriented Scientific Programming Language. Master's thesis, McGill University, 2010.

[6] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren. AspectMatlab: An Aspect-Oriented Scientific Programming Language. In *Proceedings of 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, March 2010.

[7] H. Boehm and M. Spertus. Transparent Programmer-Directed Garbage Collection for C++, 2007.

[8] A. Casey. The MetaLexer Lexical Specification Language. Master's thesis, McGill University, September 2009.

[9] M. Chevalier-Boisvert. McVM: An Optimizing Virtual Machine for the MATLAB Programming Language. Master's thesis, McGill University, August 2009.

[10] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. In *International Conference on Compiler Construction*, pages 46–65, March 2010.

[11] C. Dimoulas and M. Wand. The Higher-Order Aggregate Update Problem. In *VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 44–58, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 1–18, New York, NY, USA, 2007. ACM.

[13] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[14] K. Gopinath and J. L. Hennessy. Copy Elimination in Functional Languages. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 303–314, New York, NY, USA, 1989. ACM.

[15] P. Hudak and A. Bloss. The Aggregate Update Problem in Functional Programming Systems. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 300–314, New York, NY, USA, 1985. ACM.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[17] J. Li. McFor: A MATLAB to FORTRAN 95 Compiler. Master's thesis, McGill University, August 2009.

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

[19] M. Odersky. How to Make Destructive Updates Less Destructive. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–36, New York, NY, USA, 1991. ACM.

[20] Press, H. William and Teukolsky, A. Saul and Vetterling, T. William and Flannery, P. Brian. *Numerical Recipes : the Art of Scientific Computing.* Cambridge University Press, 1986.

[21] L. D. Rose, K. Gallivan, E. Gallopoulos, B. A. Marsolf, and D. A. Padua. FALCON: A MATLAB Interactive Restructuring Compiler. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 269–288, London, UK, 1996. Springer-Verlag.

[22] A. V. S. Sastry. *Efficient Array Update Analysis of Strict Functional Languages.* PhD thesis, Eugene, OR, USA, 1994.

[23] N. Shankar. Static Analysis for Safe Destructive Updates in a Functional Language. In *LOPSTR '01: Selected papers from the 11th International Workshop on Logic Based Program Synthesis and Transformation*, pages 1–24, London, UK, 2001. Springer-Verlag.

[24] The MathWorks. *Numerical Computing with MATLAB.* Society for Industrial and Applied Mathematics, 2004.

[25] The MathWorks. *MATLAB Programming Fundamentals.* The MathWorks, Inc., 2009.

[26] M. Wand and W. D. Clinger. Set Constraints for Destructive Array Update Optimization. *Journal of Functional Programming*, 11(3):319–346, 2001.

[27] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3 – 35, 2001.