Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler*

Nurudeen Lameed and Laurie Hendren

School of Computer Science, McGill University, Montréal, QC, Canada
{nlamee,hendren}@cs.mcgill.ca

Abstract. MATLAB has gained widespread acceptance among scientists. Several dynamic aspects of the language contribute to its appeal, but also provide many challenges. One such problem is caused by the copy semantics of MATLAB. Existing MATLAB systems rely on reference-counting schemes to create copies only when a shared array representation is updated. This reduces array copies, but requires runtime checks.

We present a staged static analysis approach to determine when copies are not required. The first stage uses two simple, intraprocedural analyses, while the second stage combines a forward *necessary copy analysis* with a backward *copy placement analysis*. Our approach eliminates unneeded array copies without requiring reference counting or frequent runtime checks.

We have implemented our approach in the McVM JIT. Our results demonstrate that, for our benchmark set, there are significant overheads for both existing reference-counted and naive copy-insertion approaches, and that our staged approach is effective in avoiding unnecessary copies.

1 Introduction

MATLAB[TM1] 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. 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, parameter passing and for returning values from a function. Thus a statement of the form `a = b` semantically means that a copy of `b` is made and that copy is assigned to `a`. Similarly, for a call of the form `a = foo(c)`, a copy of `c` is made and assigned

* This work was supported, in part, by NSERC and FQRNT.

[1] `http://www.mathworks.com/products/pfo/`

J. Knoop (Ed.): CC 2011, LNCS 6601, pp. 22–41, 2011.

to the parameter of the function `foo`, and the return value of `foo` is copied to `a`. Naive implementations take exactly this approach.

However, 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 to one of the shared arrays 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 [1] takes (by examining and instrumenting the source code). We have inferred that this approach (or a small variation) is what the Mathworks' closed-source implementation does based on the user-level documentation [19, p. 9-2].

Although the reference-counting approach reduces unneeded copies at runtime, it introduces many redundant checks, requires space for the reference counts, and requires extra code to update the reference counts. This is clearly costly in a garbage-collected VM, such as the recently developed McVM, a specializing JIT [8,9]. Furthermore, the reference-counting approach may generate a redundant copy during an update of a variable, if the updated variable shares an array only with dead variables.

Thus, our challenge was to develop a static analysis approach, suitable for a JIT compiler, which could determine which copies were required, without requiring reference counts and without the expense of dynamic checks. Since we are in the context of a JIT compiler, we developed a staged approach. The first phase applies very simple and inexpensive analyses to determine the obvious cases where copies can be avoided. The second phase tackles the harder cases, using a pair of more sophisticated static 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*). We have implemented our analyses in the McJIT compiler as structured flow analyses on the low-level AST intermediate representation used by McJIT.

To demonstrate the applicability of our approach, we have performed several experiments to: (1) demonstrate the behaviour of the reference-counting approaches, (2) to measure the overhead associated with the dynamic checks in the reference-counting approach, and (3) demonstrate the effectiveness of our static analysis approach. Our results show that actual needed copies are infrequent even though the number of dynamic checks can be quite large. We also show that these redundant checks do contribute significant overheads. Finally, we show that for our benchmark set, our static approach finds the needed number of copies, without introducing any dynamic checks.

The paper is organized as follows. Sec. 2 describes the McLab project and how this work fits into the project. Sec. 3 describes the simple first-stage analyses, and Sec. 4 and Sec. 5 describe the second-stage forward and the backward analyses, with examples. Sec. 6 discusses the experimental results of our approach; we review some related work in Sec. 7, and Sec. 8 concludes the paper.

2 Background

The work presented in this paper is a key component of our McLab system [2]. 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 language 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.

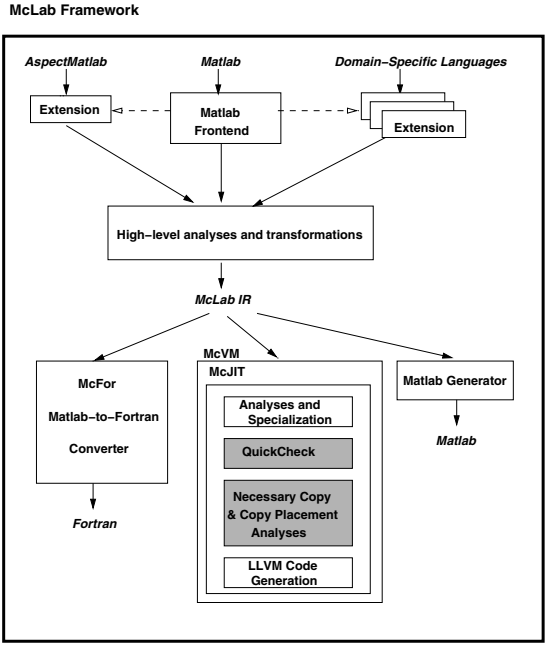

Fig. 1. Overview of McLab (shaded boxes correspond to analyses in this paper)

The McLab framework is outlined in Fig. 1, with the shaded boxes indicating the components presented in this paper. The framework is comprised 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].[2] The front-end and the extensions are built using our group's extensible lexer, Metalexer [7], and JastAdd [11]. There are three backends: McFor, a FORTRAN code generator [18]; a MATLAB generator (to use McLab as a source-to-source compiler); and McVM,

[2] We use ASPECTMATLAB for some dynamic measurements in Sec. 6.

a virtual machine that includes a simple interpreter and a sophisticated type-specialization-based JIT compiler, which generates LLVM [17] 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 [6], and several numerical libraries [4,28]. For the purposes of this paper, it is important to realize that McJIT specializes code based on the function argument types that occur at runtime. 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 and type inference. 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 Fig. 1 in order to determine where copies are required and the best location for the copies. These two analysis stages 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 copy analyses only need to consider the remaining variables. The type-inference analysis is used to disambiguate between function calls and array accesses since MATLAB uses the same syntax for both.

In the next section we introduce the first stage of our approach which is the *QuickCheck*. Following that we introduce the second stage — the *necessary copy* and *copy placement* analyses.

3 Quick Check

The *QuickCheck* phase (*QC*) is a combination of two simple and fast analyses. The first, *written parameters analysis*, is a forward analysis which determines the parameters that *may* be modified by a function. The intuition is that during a call of the function, the arguments passed to it from the caller need to be copied to the corresponding formal parameters of the function only if the function may modify the parameters. Read-only arguments do not need to be copied.

The analysis computes a set of pairs, where each pair represents a parameter and the assignment statement that last defines the parameter. For example, the entry (p_1, d_1) indicates that the last definition point for the parameter p_1 is the statement d_1. The analysis begins with a set of initial definition pairs, one pair for each parameter declaration. The analysis also builds a *copy list*, a list of parameters which must be copied, which is initialized to the empty list. The analysis is a forward flow analysis, using union as the merge operator. The key flow equations are for assignment statements of two forms:

***p* = *rhs*:** If the left-hand side (*lhs*) of the statement is a parameter p, then this statement is redefining p, so all other definitions of p are killed and this

new definition of p is generated. Note that according to the MATLAB copy semantics, such a statement is not creating an alias between p and *rhs*, but rather p is a new copy; subsequent writes to p will write to this new copy.

***p(i) = rhs*:** If the *lhs* is an array index expression (i.e., the assignment statement is writing to an element of p), and the array symbol p is a parameter, it checks if the initial definition of the parameter reaches the current assignment statement and if so, it inserts the parameter into the copy list.

At the end of the analysis, the copy list contains all the parameters that must be copied before executing the body of the function.

The second analysis is *copy replacement*, a standard sort of copy propagation/elimination algorithm that is similar to the approach used by an APL compiler [27]. It determines when a copy variable can be replaced by the original variable (copy propagation). If all the uses of the copy variable can be replaced by the original variable then the copy statement defining the copy can be removed after replacing all the uses of the copy with the original (copy elimination).

If the analysed function does not return an array and all the remaining copy statements have been made redundant by the QC transformation, then there is no need to apply a more sophisticated analysis. However, if copies do remain, then phase 2 is applied, as outlined in the next two sections.

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 of the form *lhs* = *rhs*. It is straightforward to show that our analysis works for both single (one *lhs* variable) and multiple assignment statements (multiple *lhs* variables). We describe the analysis by defining the following components.

Domain: the domain of the analysis' flow facts is the set of pairs 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 may reference the array allocated at statement s.

Problem Definition: at a 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.

Flow Function: $out(S_i) = gen(S_i) \cup (in(S_i) - kill(S_i))$.

Given the assignment statements of the forms:

$$S_i : a = \texttt{alloc} \tag{1}$$
$$S_i : a = b \tag{2}$$
$$S_i : a(j) = x \tag{3}$$
$$S_i : a = f(arg_1, arg_2, ..., arg_n) \tag{4}$$

where S_i denotes a statement ID; `alloc` is a new memory allocation performed by statement S_i[3]; a, b are array reference variables; x is a *rvalue*; 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. The partition, $Q_i(m)$, containing flow entries for allocator m is:

$$Q_i(m) = \{(x, y) | (x, y) \in in(S_i) \wedge y = m\} \tag{5}$$

Now consider statements of type 2 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 $P_i(a)$ for the set of Q_is obtained by partitioning $in(S_i)$ using the allocators of the variable a.

Considering statements of the form 3, $P_i(a) \neq \emptyset$ 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 new 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 simplify the table, we define

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

Stmt	Gen set	Kill set
(1)	$\{(x, s) \mid x = a \wedge s = S_i \wedge \texttt{live}(S_i, x)\}$	$Kill_{define}(a) \cup Kill_{dead}$
(2)	$\{(x, s) \mid x = a \wedge (y, s) \in in(S_i) \wedge y = b \wedge \texttt{live}(S_i, x)\}$	$Kill_{define}(a) \cup Kill_{dead}$
(3)	$\{(x, s) \mid x = a \wedge s = S_i \wedge P_i(x) \neq \emptyset\}$	$Kill_{update}(a) \cup Kill_{dead}$
(4)	see $gen(f)$ below	$Kill_{define}(a) \cup Kill_{dead}$

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:

[3] Functions such as *zeros*, *ones*, *rand* and *magic* are memory allocators in MATLAB.

$$gen(f) = \begin{cases} \{(a, S_i)\}, \text{ if } \texttt{live}(S_i, a) \texttt{ and } \texttt{isAllocFunction}(f) \\ \{(x, s) | x = a \wedge (arg_j, s) \in in(S_i) \wedge \texttt{live}(S_i, x)\}, \\ \quad \text{if } \texttt{ret}(f) \texttt{ aliases } param_j(f),\ 0 < j \leq \texttt{size}(params(f)), \\ \{(a, S_i)\}, \text{ if } \forall (p \in params(f)), \texttt{not } (\texttt{ret}(f) \texttt{ aliases } p) \\ \{(x, s) | x = a \wedge arg \in args(f) \wedge (arg, s) \in in(S_i) \wedge \texttt{live}(S_i, x)\}, \\ \quad \texttt{otherwise} \text{ (e.g., if } f \text{ is recursive)} \end{cases}$$

The first alternative generates a flow entry (a, S_i) if the *rhs* is an *alloc* function and the *lhs* (a) is live after 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 f from an analysis manager.[4] The manager caches the result of the previous analysis on a given function. From the result of the analysis on f, we determine the return variables of f that are aliases to the parameters of f and hence aliases to the arguments of f. This is explained in detail under the next section on Initialization. The return variable of f corresponds to the *lhs* (a) in statement type 4. Therefore we generate new flow entries from those 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 generates $\{(a, S_i)\}$, if the return variable aliases no parameters of f. The fourth alternative is conservative: new flow entries are generated from those of *all* the arguments to f. This can happen if the call of f is recursive or f cannot be analyzed because it is neither a user-defined function nor an *alloc* function.

We chose a simple strategy for recursion because recursive functions occur rarely in MATLAB. In a separate study by our group, we found that out of 15966 functions in 625 projects examined, only 48 functions (0.3%) are directly recursive. None of the programs in our benchmarks had recursive functions.

Therefore, we expect that the conservative option in the definition of $gen(f)$ above will be rarely taken in practice.

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) return variable references. We use a shadow entry to detect when a parameter that has not been assigned to any other variable is updated. At the entry to a function, the input set is given as
$in(entry) = \{(p, s) | p \in params(f) \wedge s = S_p\} \cup \{(p', s) | p \in params(f) \wedge s = S_p\}$.

We illustrate this scheme with an example. Given a function f, defined as:

```
function u = f(x, y)
  u = x;
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 $\{(u, S_x), (x, S_x), (x', S_x), (y, S_y), (y', S_y)\}$.

[4] This uses the same analysis machinery as the type estimation in McJIT.

We now know that u is an alias for x and encode this information as a set of integers. An element of the set is an integer representing the input parameter that the output parameter may reference in the function. In this example, the set is $\{1\}$ since x is the first (1) parameter of f. This is useful during a call of f. For instance, in `c = f(a, b);` we can determine that c is an alias for the argument a by inspecting the summary information generated for f.

4.1 `if-else` Statement

So far we have been considering sequences of statements. As our analysis is done directly on a simplified AST, analyzing an `if-else` statement simply requires that we analyze all the alternative blocks and merge the result at the end of the `if-else` statement using the merge operator ($\cup$).

4.2 Loops

We compute the input set reaching a loop and the output set exiting a loop using standard flow analysis techniques, that is, we merge the input flow set from the loop's entry with the output set from the loop back-edge until a fixed point is reached.

To analyse a loop more precisely, we implemented a context-sensitive loop analysis [16], but found that real MATLAB programs did not require the context-sensitivity to achieve good results. The standard approach is sufficient for typical MATLAB programs.

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 intuition behind this analysis is that often it is better to perform the array copy close to the statement which created the sharing (i.e. statements of the form $a = b$) rather than just before the array update statements (i.e. statements of the form *a(i) = b*) that require the copy. 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 backward analysis that pushes the necessary copies upwards, possibly as far as the statement that created the sharing.

5.1 Copy Placement Analysis Details

A copy entry is a three-tuple:

$$e =< copy\_loc, var, alloc\_site > \tag{6}$$

where *copy_loc* denotes the ID of the node that generates the copy, *var* denotes the variable containing 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 ($\cap$).

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.

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. The sets $in(S_i)$, $Q_i(m)$, $P_i(a)$ are defined in Section 4.

Rule 1: array updates, $S_i : a(y) = x$: Given that the array variable of the *lhs* of 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

$$C_{in} := C_{in} \cup \begin{cases} \emptyset & \text{if } P_i(a) = \emptyset \\ \{< s, x, y > | s = S_i \wedge x = a \wedge Q_i(y) \in P_i(x)\} & \texttt{otherwise} \end{cases}$$

Rule 2: array assignments, $S_j : a = b$: If $\forall e \in C_{in}(e.var \neq a \texttt{ and } e.var \neq b)$, and $\forall e \in C_{out}(e.var \neq a \texttt{ and } e.var \neq b)$, we skip the current statement. However, 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 — the location specified in the copy entry e. Otherwise, if $\exists e \in C_{out}(e.var = a \texttt{ or } e.var = b)$, 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 variable x ($x \neq a$) of each member of $Q_i(m)$ at the statement *e.copy_loc*.

Since $P_j(a) \neq \emptyset$, there is at least a definition of a that reaches this statement and for which a references a shared array. In addition, because the copy e was generated after the current block there are at least 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.

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

`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

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

Then we compute the differences

$$C'_{out} := C_{out} \setminus C'; \quad C'_{else} := C_{else} \setminus C'; \quad C'_{if} := C_{if} \setminus C'$$

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,

$$C_{in} := C'_{out} \cup C'_{else} \cup C'_{if} \cup \{< s, e.var, e.alloc\_site > | s = S_{IF} \wedge e \in C'\}$$

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.

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.

A disadvantage of propagating the copy outside of the loop is that if none of the loops that require copies is executed then we would have generated a useless copy. 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. With this assumption, there is no need to compute the intersection of C_{loop} and C_{out}. Hence

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

5.2 Using the Analyses

This section illustrates how the combination of the forward and the backward analyses is used to determine the actual copies that should be generated. First consider the following program, *test3*. Fig. 2(a) shows the result of the forward analysis.

```
1  function test3()
2     a = [1:5];
3     b = a;
4     i = 1;
5     if (i > 2)    % I
6        a(1) = 100;
7     else
8        a(1) = 700;
9     end
10    a(1) = 200;
11    disp(a); disp(b);
12 end
```

#	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	$\emptyset$	$\{(b, S_2), (a, S_6), (a, S_8)\}$	$\{(b, S_2), (a, S_6), (a, S_8)\}$

(a) Necessary Copy Analysis Result

#	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	$\emptyset$	$\emptyset$	$\{(a, S_I)\}$

(b) Copy Placement Analysis Result

Fig. 2. Introductory example for Copy Placement Analysis

Fig. 2(b) gives the result of the backward analysis. The I used in Fig. 2 stands for the `if-else` statement in *test3*. The analysis begins from line 12 of *test3*. The out set C_{out} is initially empty. 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 is 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, 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) 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 [10]. The forward analysis information is shown in Fig. 3(a). The table shows the *gen* and *in* sets at each relevant assignment statement of *tridisolve*. The results in different loop iterations are shown using a subscript to represent loop iteration. For example, the row number 25_2 refers to the result at the statement labelled S_{25} in the second iteration. 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 as outlined in Sec. 4. 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

```
function x = tridisolve(a,b,c,d)
% TRIDISOLVE Solve tridiagonal system of equations.
20: x = d;
21: n = length(x);
    for j = 1:n−1          %F_1
      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);
    end
29: x(n) = x(n)/b(n);
    for j = n−1:−1:1       %F_2
31:   x(j) = (x(j)−c(j)∗x(j+1))/b(j);
    end
```

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

(a) Necessary Copy Analysis Result

#	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	$\emptyset$	$\emptyset$	$\{(x, S_{F_1}), (b, S_0)\}$

(b) Copy Placement Analysis Result

Fig. 3. Example for *tridisolve*

$|Q_{25}(S_b)| = |\{(b, S_b, 0), (b', S_b, 0)\}| > 1$. 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.

Fig. 3(b) shows the copy placement analysis information at each relevant statement of *tridisolve*. Recall that the placement analysis works by traversing the statements in each block of a function backward. In the case of *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 loop F_2, the analysis merges with the main path and the result at this point is shown in row F_2. Statement S_{29} generated a copy as indicated by the forward analysis, therefore C_{in} is updated and the result set is also updated. The analysis then branches off to the first loop and the current C_{in} is passed to the loop's body as C_{out}. The copies generated in 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 row F_1. The result 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 Fig. 3(b). 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 function and the necessary copy should be generated at the function's entry. Hence, a copy of the array referenced by b must be generated at the entry of *tridisolve*.

6 Experimental Results

To evaluate the effectiveness of our approach, we set up experiments using benchmarks collected from disparate sources, including those from [10,22,23]. Table 1 gives a short description of the benchmarks together with a summary of the results of our analyses, which we discuss in more detail in the following subsections. For all our experiments, we ran the benchmarks with their smallest input size on an AMD Athlon^{TM} 64 X2 Dual Core Processor 3800+, 4GB RAM computer running Linux operating system; GNU Octave, version 3.2.4; MATLAB, version 7.9.0.529 (R2009b) and McVM/McJIT, version 0.5.

The purpose of our experiments was three-fold. First, we wanted to measure the number of array updates and copies performed by the benchmarks at runtime using existing systems (Sec. 6.1). Knowing the number of updates gives an idea of how many dynamic checks a reference-counting-based (RC) scheme for lazy copying, such as used by Octave and Mathworks' MATLAB, need to perform. Recall that our approach does not usually require any dynamic checks. Knowing the number of copies generated by such systems allows us to verify that our approach does not increase the number of copies as compared to the RC approaches. Secondly, we would like to measure the amount of overhead generated in RC systems (Sec. 6.2). Finally, we would like to assess the impact of our static analyses in terms of their ability to minimize the number of copies (Sec. 6.3).

6.1 Dynamic Counts of Array Updates and Copies

Our first measurements were designed to measure the number of array updates and array copies that are required by existing RC systems, Octave and Mathworks' MATLAB. Since we had access to the open-source Octave system we were able to instrument the interpreter and make the measurements directly. However, the Mathworks' implementation of MATLAB is a proprietary system and thus we were unable to instrument it to make direct measurements. Instead, we developed an alternative approach by instrumenting the benchmark programs themselves via aspects using our ASPECTMATLAB compiler *amc* [5]. Our aspect[5] defines all the patterns for 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 RC 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 thus counts all array updates and all copies that would be required by a RC system.

Table 1. Benchmarks and the results of the copy analysis[6]

Benchmark		# Array Updates	# Copies				
			Lower Bound		With Analyses		
			Aspect	Octave	Naive	QC	CA
adpt	adaptive quadrature using Simpson's rule	19624	0	0	12223	12223	0
capr	capacitance of a transmission line using finite difference and Gauss-Seidel iteration	9790800	10000	10000	40000	20000	10000
clos	transitive closure of a directed graph	2954	0	0	2	2	0
crni	Crank-Nicholson solution to the one-dimensional heat equation	21143907	4598	6898	11495	6897	4598
dich	Dirichlet solution to Laplace's equation	6935292	0	0	0	0	0
fdtd	3D FDTD of a hexahedral cavity with conducting walls	803	0	0	5400	5400	0
fft	fast fourier transform	44038144	1	1	2	2	1
fiff	finite-difference solution to the wave equation	12243000	0	0	0	0	0
mbrt	mandelbrot set	5929	0	0	0	0	0
nb1d	N-body problem coded using 1d arrays for the displacement vectors.	55020	0	0	10984	10980	0
nb3d	N-body problem coded using 3d arrays for the displacement vectors.	4878	0	0	5860	5858	0
nfrc	computes a newton fractal in the complex plane -2..2,-2i..2i	12800	0	0	6400	6400	0
trid	Solve tridiagonal system of equations	2998	2	2	5	2	2

In Table 1 the column labelled **# Array Updates** gives the total number of array updates executed. The column **# Copies** shows the number of copies generated by the benchmarks under Octave (reported as **Octave** in the table) and MATLAB (column labelled **Aspect**). The column *# Copies* is split into two: **Lower Bound** and **With Analyses**. The number of copies generated by Octave and MATLAB (Aspect) are considered the expected lower bounds

[5] This aspect is available at: `www.sable.mcgill.ca/mclab/mcvm_mcjit.html`

[6] The benchmarks are also available at: `www.sable.mcgill.ca/mclab/mcvm_mcjit.html`

(since they perform copies lazily, and only when required) and are therefore grouped under *Lower Bound* in the table.[7]

At a high-level, the results in Table 1 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 thirteen 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. *With Analyses* comprises of three columns, **Naive**, **QC**, and **CA** representing respectively, the number of copies generated in our naive system, with the QC phase, and with the copy analysis phase. We return to these results in Sec. 6.3.

6.2 The Overhead of Dynamic Checks

With RC approaches a dynamic check is needed for each array update, in order to test if a copy is needed. Our counts indicated that several of our benchmarks had a high number of updates, but no copies were required. We wanted to measure the overhead for all of these redundant dynamic checks. The ideal measurement would have been to time the redundant checks in a JIT-based system that used reference-counting, such as Mathworks' MATLAB. Unfortunately we do not have access to such a system. Instead we performed two similar experiments, as reported in Table 2, for three benchmarks with a high number of updates and no required copies (`dich`, `fiff` and `mbrt`).

Table 2. Overhead of Dynamic Checks

	McVM						Octave(O)		
	McJIT		McJIT(+RC)		Overhead(%)		Time(s)		Overhead
Bmark	time(s)	# LLVM	time(s)	# LLVM	time	size	O(+RC)	O(-RC)	(%)
dich	0.18	546	0.27	625	47.37	14.47	425.05	408.08	4.16
fiff	0.39	388	0.52	415	33.72	6.96	468.64	438.69	6.83
mbrt	5.06	262	5.65	271	11.69	3.44	34.91	31.95	9.29

We first created a version of Octave that does not insert dynamic checks before array update statements. In general this is not safe, but for these three benchmarks we knew no copies were needed, and thus removing the checks allowed us to measure the overhead without breaking the benchmarks. The column labelled **O(+RC)** gives the execution time with dynamic checks and the column labelled **O(-RC)** gives the times when we artificially removed the checks. The difference gives us the overhead, which is between 4% and 9% for these benchmarks. Although this is not a huge percentage, it is not negligible. Furthermore, we felt that the absolute time for the checks was significant and would be even more significant in a JIT system which has many fewer other overheads.

To measure overheads in a JIT context, we modified McJIT to include enough reference-counting machinery to measure the overhead of the checks (remember

[7] Note that for the benchmark `crni` Octave performs 6898 copies, whereas the lower bound according to the Aspect is 4598. We verified that Octave is doing some spurious copies in this case, and that the Aspect number is the true lower bound.

that McVM is garbage-collected and does not normally have reference counts). For the modified McVM we added a field to the array object representation to store reference counts (which is set to zero for the purposes of this experiment) and we generated LLVM code for a runtime check before each array update statement. Table 2 shows, in time and code size, the amount of overhead generated by redundant checks. The column labelled **McJIT** is the original McJIT and the column labelled **McJIT(+RC)** is the modified version with the added checks. We measured code size using the number of LLVM instructions (**# LLVM**) and execution time overhead in seconds. For these benchmarks the code size overhead was 3% to 14% and the running time overhead ranged from 12% to 47%.

Our conclusions is that the dynamic checks for a RC scheme can be quite significant in both execution time and code size, especially in the context of a JIT. Thus, although the original motivation of our work was to enable a garbage-collected VM that did not require reference counts, we think that our analyses could also be useful to eliminate unneeded checks in RC systems.

6.3 Impact of Our Analyses

Let us now return to the number of copies required by our analyses, which are given in the last three columns of Table 1. As a reminder, our goal was to achieve the same number of copies as the lower bound.

The column labelled **Naive** gives the number of copies required with a naive implementation of MATLAB's copy semantics, where a copy is inserted for each parameter, each return value and each copy statement, where the *lhs* is an array. Clearly this approach leads to many more copies than the lower bound.

The column labelled **CA** gives the number of copies when both phases of our static analyses are enabled. We were very pleased to see that for our benchmarks, the static analyses achieved the same number of copies as the lower bound, without requiring any dynamic checks. The column labelled **QC** shows the number of copies when only the QuickCheck phase is enabled. Although the QuickCheck does eliminate many unneeded copies, it does not achieve the lower bound. Thus, the second stage is really required in many cases.

To show the impact copies have on execution performance, we measured the total bytes of array data copied by a benchmark together with its corresponding execution time. These are shown in Fig. 4 and Table 3 for *Naive*, *QC* and *CA*. The columns $\frac{Naive}{QC}$ and $\frac{Naive}{CA}$ of Table 3 show respectively how many times QC and CA perform better than *Naive*. The table shows that *CA* generally outperforms *QC* and *Naive*. Copying large arrays affects execution performance and the results in Table 3 validate this claim. Where a significant number of bytes were copied by the naive implementation, for example, *capr*, *crni* and *fdtd*, *CA* performs better than both *Naive* and *QC*. In the three benchmarks that do not generate copies, the performance of *CA* is comparable to *Naive* and *QC*. This shows that the overhead of *CA* is low. It is therefore clear from the results of our experiments that the naive implementation generates significant overhead and is therefore unsuitable for an high-performance system.

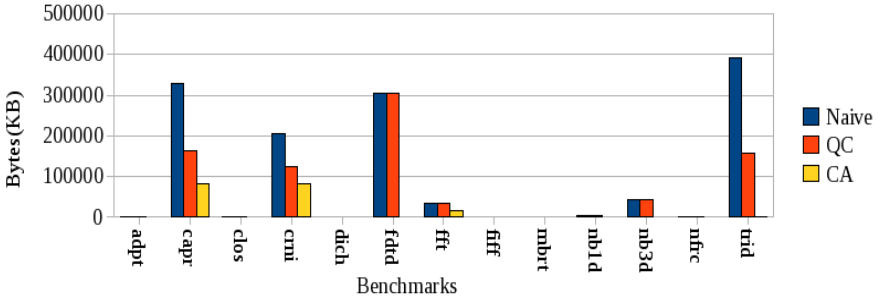

Fig. 4. The total bytes of array data copied by the benchmarks under the three options

Table 3. Benchmarks against the total execution times in seconds

Bmark	Naive	QC	CA	$\frac{Naive}{QC}$	$\frac{Naive}{CA}$	Bmark	Naive	QC	CA	$\frac{Naive}{QC}$	$\frac{Naive}{CA}$
adpt	1.57	1.57	1.61	1.00	0.98	fiff	0.39	0.39	0.39	0.99	0.99
capr	1.54	0.91	0.58	1.70	2.66	mbrt	5.06	5.12	5.04	0.99	1.00
clos	0.49	0.49	0.48	0.99	1.01	nb1d	0.48	0.48	0.45	1.00	1.07
crni	135.09	140.35	131.62	0.96	1.03	nb3d	0.48	0.48	0.36	1.00	1.35
dich	0.18	0.18	0.18	1.00	1.00	nfrc	3.23	3.23	3.25	1.00	0.99
fdtd	3.79	3.78	2.80	1.00	1.35	trid	1.57	1.04	1.02	1.51	1.53
fft	1.50	1.50	1.47	1.00	1.02						

Impact of the First Phase. We measured the number of functions that are completely resolved by the first phase of our approach — in terms of finding all the necessary copies required to guarantee copy semantics. We found that out of the 23 functions in the benchmark set, the first stage (i.e., QuickCheck) was only able to resolve about 17% of the functions. None of the benchmarks was resolved completely by QC. The main reason for this poor performance is that the first phase cannot resolve functions that return arrays to their callers. And like most MATLAB programs, most of the functions in the benchmarks return arrays. This really shows that the second stage is actually required to completely determine the needed copies by typical MATLAB programs.

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 needed number of copies, while at the same time avoiding all the overhead of dynamic checks. Furthermore, our approach does not require reference counting and thus enables an efficient implementation of array copy semantics in garbage-collected systems like McVM.

7 Related Work

Redundant copy elimination is a hard problem and implementations of languages such as Python [3] 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 that use copy semantics require copy elimination optimization. The problem is similar to the aggregate update problem in functional languages [12,14,21,24,26]. To modify an aggregate in a strict functional language, a copy of the aggregate must be made. This is in contrast with the imperative programming languages where an aggregate may be modified multiple times.

APL [15] is one of the oldest array-based languages. Weigang [27] describes a range of optimizations for APL compiler, including a copy optimization that finds uses of a copy of a variable and replaces the copy with the original variable wherever possible. We implemented this optimization as part of our QuickCheck phase. We found the optimization effective at enabling the elimination of redundant copy statements by the dead-code optimizer. However, this optimization is unable to eliminate redundant copies of arguments and return values. Hudak and Bloss [14] 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 [20] 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, our analyses are intraprocedural and have been implemented in a JIT compiler for an imperative language.

The work of Goyal and Paige [13] on copy optimization for SETL [25] is particularly interesting. Their approach combines a RC scheme with static analysis. A combination of must-alias and live-variable analyses is used to identify dead variables and the program points where a statement that redefines a dead variable can be inserted to facilitate destructive updates. Like our approach, this technique is capable of eliminating the redundant copying of a shared location that can occur during an update of the location; however, it is different from our approach. In particular, it generates dynamic checks to detect when to make copies. As mentioned in Sec. 6, our approach rarely generates dynamic checks.

8 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 analysis phases in the McJIT compiler. The first phase implements simple analyses for detecting read-only parameters and standard copy elimination, whereas the second phase consists a forward *necessary copy analysis* that determines which array update statements trigger copies, and a backward *copy placement analysis* that determines good places to insert the array copies. All of these analyses have been implemented as structured-based analyses on the McJIT intermediate representation.

Our approach does not require frequent dynamic checks, nor do we need the space and time overheads 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. However, similar techniques could be used in a reference-counting-based system to remove redundant checks. 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.

References

1. GNU Octave, `http://www.gnu.org/software/octave/index.html`
2. McLab, `http://www.sable.mcgill.ca/mclab/`
3. Python, `http://www.python.org`
4. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. SIAM, Philadelphia (1999)
5. Aslam, T., Doherty, J., Dubrau, A., Hendren, L.: AspectMatlab: An Aspect-Oriented Scientific Programming Language. In: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, pp. 181–192 (March 2010)
6. Boehm, H., Spertus, M.: N2310: Transparent Programmer-Directed Garbage Collection for C++ (June 2007), `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2310.pdf`
7. Casey, A., Hendren, L.: MetaLexer: A Modular Lexical Specification Language. In: Proceedings of the 10th International Conference on Aspect-Oriented Software Development (March 2011)
8. Chevalier-Boisvert, M.: McVM: An Optimizing Virtual Machine for the MATLAB Programming Language. Master's thesis, McGill University (August 2009)
9. Chevalier-Boisvert, M., Hendren, L., Verbrugge, C.: Optimizing MATLAB through Just-In-Time Specialization. In: International Conference on Compiler Construction, pp. 46–65 (March 2010)
10. Moler, C.: Numerical Computing with MATLAB. SIAM, Philadelphia (2004)
11. Ekman, T., Hedin, G.: The Jastadd Extensible Java Compiler. In: OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, pp. 1–18. ACM, New York (2007)

12. Gopinath, K., Hennessy, J.L.: Copy Elimination in Functional Languages. In: POPL 1989: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 303–314. ACM, New York (1989)
13. Goyal, D., Paige, R.: A New Solution to the Hidden Copy Problem. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 327–348. Springer, Heidelberg (1998)
14. Hudak, P., Bloss, A.: The Aggregate Update Problem in Functional Programming Systems. In: POPL 1985: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 300–314. ACM, New York (1985)
15. Iverson, K.E.: A Programming Language. John Wiley and Sons, Inc., Chichester (1962)
16. Lameed, N., Hendren, L.: Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler. Technical Report SABLE-TR-2010-5, School of Computer Science, McGill University, Montréal, Canada (July 2010)
17. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO 2004: Proceedings of the International Symposium on Code Generation and Optimization, p. 75. IEEE Computer Society, Washington, DC, USA (2004)
18. Li, J.: McFor: A MATLAB to FORTRAN 95 Compiler. Master's thesis, McGill University (August 2009)
19. MathWorks. MATLAB Programming Fundamentals. The MathWorks, Inc. (2009)
20. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
21. Odersky, M.: How to Make Destructive Updates Less Destructive. In: POPL 1991: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 25–36. ACM, New York (1991)
22. Press, H.W., Teukolsky, A.S., Vetterling, T.W., Flannery, P.B.: Numerical Recipes: the Art of Scientific Computing. Cambridge University Press, Cambridge (1986)
23. Rose, L.D., Gallivan, K., Gallopoulos, E., Marsolf, B.A., Padua, D.A.: FALCON: A MATLAB Interactive Restructuring Compiler. In: Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1995. LNCS, vol. 1033, pp. 269–288. Springer, Heidelberg (1996)
24. Sastry, A.V.S.: Efficient Array Update Analysis of Strict Functional Languages. PhD thesis, University of Oregon, Eugene, USA (1994)
25. Schwartz, J.T., Dewar, R.B., Schonberg, E., Dubinsky, E.: Programming with Sets; an Introduction to SETL. Springer, New York (1986)
26. Wand, M., Clinger, W.D.: Set Constraints for Destructive Array Update Optimization. Journal of Functional Programming 11(3), 319–346 (2001)
27. Weigang, J.: An Introduction to STSC's APL Compiler. SIGAPL APL Quote Quad 15(4), 231–238 (1985)
28. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS Project. Parallel Computing 27(1-2), 3–35 (2001)