McGill University
School of Computer Science
Sable Research Group

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

Sable Technical Report No. sable-2010-05

Nurudeen Lameed and Laurie Hendren

July 16, 2010

Contents

List of Figures

List of Tables

Abstract

MATLAB has gained widespread acceptance among engineers and scientists. 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 rely on reference-counting schemes to create copies only when a shared array representation is updated. This reduces array copies, but increases the number of runtime checks. In addition, this sort of reference-counted approach does not work in a garbage-collected system.

In this paper we present a staged static analysis approach that does not require reference counts, thus enabling a garbage-collected virtual machine. Our approach eliminates both unneeded array copies and does not require runtime checks. The first stage combines two simple, yet fast, intraprocedural analyses to eliminate unnecessary copies. The second stage is comprised 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 in the McVM JIT, which is part of a garbage-collected virtual machine for MATLAB. Our results demonstrate that there are significant overheads for both existing reference-counted and naive copy-insertion approaches. We also show that our staged approach is effective. In some cases the first stage is sufficient, but in many cases the second stage is required. Overall, our approach required no dynamic checks and successfully eliminated all unnecessary copies, for our benchmark set.

1 Introduction

MATLAB™[1] 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 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 Mathworks' closed-source implementation does based on the user-level documentation[27, p. 9-2].

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

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. Furthermore, this reference-counting approach does not work in the context of a garbage-collected VM, such as the recently developed McVM, a specializing JIT[10, 11].

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.

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 our static approach finds the minimal number of copies required, without introducing any dynamic checks.

The paper is organized as follows. Section 2 describes the McLab project and where the work presented in this paper fits into that project, and Section 3 gives an overview of the problem and our general approach. Section 4 describes the simple first-stage analyses, and Section 5 and Section 6 describe the second-stage forward and the backward analyses, with examples. In Section 7, we briefly discuss how the forward analysis resolves conflicting names. Section 8 discuses the experimental results of our approach; we give some related work in Section 9 and Section 10 concludes the paper.

2 Background

The work presented in this paper is a key 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 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.

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[6, 7].[2] The front-end and the extensions are built using our group's extensible lexer, Metalexer[9] and JastAdd[14, 2]. There are three backends: McFor, a FORTRAN code generator[20]; a MATLAB generator (to use McLab as

[2]We use ASPECTMATLAB to for some dynamic measurements in Section 8.

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[19] code.

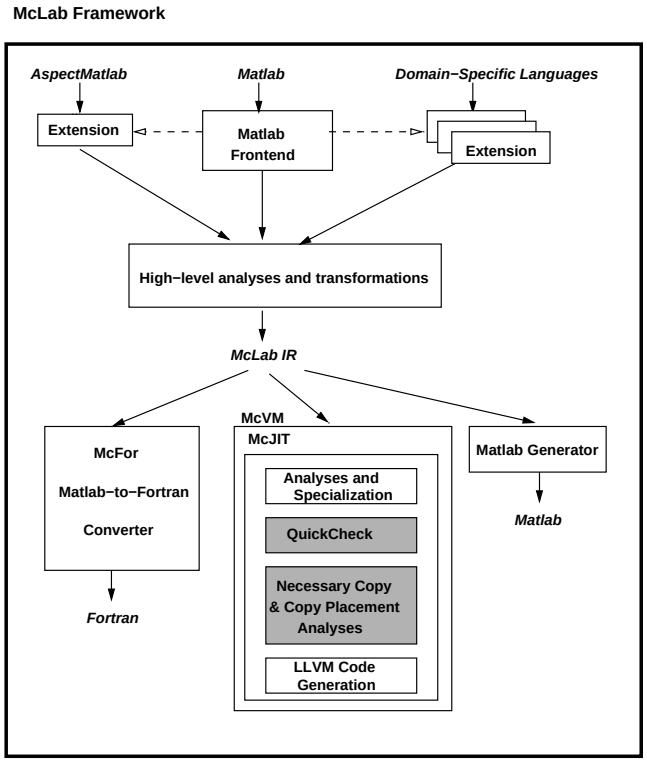

Figure 1: Overview of McLab (shaded boxes correspond to analyses presented in this paper)

The techniques presented in this paper are part of McJIT, the JIT compiler for McVM. McJIT is built upon LLVM, the Boehm garbage collector[8], and several numerical libraries[5, 30]. 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, 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 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.

3 Problem and Overview of Our Approach

To properly understand our analyses, 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 assume copy semantics. Assignment statements in the MATLAB programming language have different forms, for example:

$$a = zeros(10); \quad (1)$$
$$b = a; \quad (2)$$
$$c = myfunc(a, b); \quad (3)$$

A naive implementation of the copy semantics for statements 1 - 3 above would involve making a copy at every assignment statement. Thus, in statement 1, the object (a 10 x 10 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. In statement 3, 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 a copy must be generated when: (1) a variable is defined from an existing object; (2) a parameter is passed from one function to another; and (3) a value is returned from a function. 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' documentation [27] 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 15000 x 15000 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 a staged static analysis.

The first phase of our analysis implements a simple analysis to determine when parameters are never written and a standard copy elimination technique. The second phase is more complex, it 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 does not require the the space and time overheads associated with reference counting, it does not require dynamic checks at each array update, and it enables the use of a garbage-collected VM. Our approach will be successful if the analysis does not insert many spurious copies. As we will see in Section 8, on our benchmarks we inserted the minimal number of copies and avoided the frequent checks required by the reference-counting strategies.

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. 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 Quick Check

The *QuickCheck* phase (*QC*) is a combination of two simple but effective analyses. The first, *written parameters* analysis is a forward analysis and 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 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 assignment 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. Any 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. Otherwise, it skips the statement.

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 performed by *QC* is *copy replacement*, a standard sort of copy propagation/elimination algorithm which is similar to the approach used by an APL compiler [29]. This analysis determines when a copy variable can be replaced by the original variable (copy propagation). If all the uses of a 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). To illustrate this point, consider the following equivalent code snippets. The variable b in statement 3 of Box 1

```
Box 1:
1:          a = rand(15000);
2:          b = a;
3:          c = 2*b
```

```
Box 2:
1:          a = rand(15000);
2:          b = a;
3:          c = 2*a;
```

can be replaced with a as done in Box 2; since b is not referenced after statement 3, statement 2 in Box 2 can be removed by the dead-code optimizer.

The copy replacement analysis computes a set of pairs of variables by examining assignment statements of the form $b = a$. A pair represents the *lhs* and *rhs* of an assignment statement, and indicates that if a successor of the statement *uses* the first member of the pair then the variable used could be replaced with the second member of the pair. For example, if the pair, (b, a) reaches the statement $c = 2*b$ then b could be replaced with a in the statement.

Like the *written parameters* analysis, it is a forward flow analysis. However, in this case the merge function is intersection. The key flow equations for copy replacement analysis are:

b = a if both the *lhs* and the *rhs* are variables, a new pair of variables, that is, (b, a) is generated at the statement.

lhs = rhs if *lhs* is a member of a pair that reaches the statement, such pairs are killed at the statement. This is because the statement is redefining *lhs* and its new value may no longer match that of the other member of the pairs.

At the end of the analysis, the analyzed function is transformed using the result of the analysis.

If all copies can be eliminated with the QuickCheck, 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.

5 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 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 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 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.

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:

$$S_i : a = \texttt{alloc} \tag{4}$$
$$S_i : a = b \tag{5}$$
$$S_i : a(j) = x \tag{6}$$
$$S_i : a = f(arg_1, arg_2, ..., arg_n) \tag{7}$$

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 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:

$$Q_i(m) = \{(x, y) | (x, y) \in in(S_i) \wedge y = m\} \tag{8}$$

Now consider statements of type 5 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 6, $P_i(a) \neq \emptyset$ 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.

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

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**
(4)	$\{(a, S_i)\|\texttt{live}(S_i, a)\}$	$Kill_{define} \cup Kill_{dead}$
(5)	$\{(a, s)\|(b, s) \in in(S_i) \wedge \texttt{live}(S_i, a)\}$	$Kill_{define} \cup Kill_{dead}$
(6)	$\{(a, S_i)\|P_i(a) \neq \emptyset\}$	$Kill_{update} \cup Kill_{dead}$
(7)	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 7. 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 the call of f is a recursive call or 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:

```
1 function [u, v] = f(x, y)
2     u = x;
3     d  = y;
4     v = d;
5 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 sets of integers, [{0}, {1}]. The elements of the vector correspond to the output parameters of the function in the order in which they appear in the function definition. Each set of integers for an output parameter represents the input parameters that the output parameter may reference in the function body. 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.

5.1 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.

#	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*

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 copies 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.

5.2 `if-else` Statement

So far we have been considering sequences of statements. Analyzing an `if-else` statement requires that we analyze 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

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

5.3 Loops

Computing the input set entering a loop requires merging 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 outside the loop from those initiated within the loop, and also to distinguish those that are initiated in different iterations of 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];
   i = 1;
   while (i < 15)
3:    a(i) = 5;
4:    b = a;
5:    a(i+1) = 0;
      i = i + 1;
   end
   disp(a);
   disp(b);
end
```

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 statement 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 statement 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 statement S_3!

5.3.1 Context Recognition

We resolved the foregoing problem by identifying the context in which an array sharing has been created. First we recognize two contexts: the main sequence and the loop to distinguish the

flow entries generated before a loop from those generated within the loop. We assign two unique identifiers to these contexts. Furthermore, we recognize an additional context named *cyclic context* to distinguish the entries generated in the current iteration of a loop from those generated in the previous iterations of the same loop. All contexts have unique identifiers.

5.3.2 Context Sensitivity

We introduce a new field named *flow context* to the flow entry object to ensure that an entry bears the context in which it is generated. When a flow entry is generated by an allocator — an array definition statement — the identifier of the current flow context is assigned to the entry. This could either be the context ID of the main sequence or a loop's context ID. Flow entries are generated for the left-hand side of a copy statement from the entries associated with the right-hand side of the statement. Therefore if the flow entries for the right-hand side are generated in the same context as the copy statement, the entries for the left-hand side have the same context ID as those of the right-hand side. However, if the copy statement occurs within a loop and the entries for the right-hand side are generated before the loop, we create a copy of each entry associated with the right-hand side and assign the loop's flow context ID to the copies. We then generate flow entries for the left-hand side from the new flow entries (copies) associated with the right-hand side.

If an array update statement is a copy generator, a new flow entry is generated with its flow context set to the ID of the current context. However, if the input flow set reaching the statement contains a flow entry whose allocator is the same as the current array update statement, a *cycle* is detected, and all such flow entries are replaced with new entries having their flow context ID set to the loop's cyclic context ID. To determine if a statement S_i is a copy generator, we match flow entries based on the pairs comprising of an allocator and a context number. Therefore Equation 8 becomes

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

that is, $in(S_i)$ is now partitioned based on the pairs of an allocator (m in Equation 9) and a context ID (c in Equation 9).

An analysis of a loop begins with the first statement of the loop. At the beginning of every iteration of the loop, the output set at the end of the previous iteration is merged with the input set at the beginning of the same iteration to form the current input flow set. The output flow set from the main sequence is the input flow set at the beginning of the first iteration. This is the same as the output flow set at the end of the *zeroth* iteration of the loop.

	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*

In a loop, an array copy statement that occurs after an array update statement may have an impact on the behaviour of the array update statement in the iteration following the one in which the array copy statement has been executed. For this reason, the minimum number of iterations required

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

Table III: Context sensitive flow sets for the first three iterations of the analysis for *example2*.

before a fixed point can be reached is two. The input set at the beginning of the third iteration of the loop summarizes the result of the analysis on the loop. The analysis generally converges after three iterations.

Assuming that α, β and θ are respectively the context identifiers of the main sequence, the loop and the loop's cyclic context. We reconstruct Table II using the approach described in this section to obtain the results shown in Table III. This scheme ensures that unneeded copies are not generated by eliminating the false dependency since S_3 is not a copy generator (i.e., $P_3(a) = \emptyset$).

6 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 structured-based analysis that is performed on the low-level AST representation used by McJIT.

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, which is ideal.

6.1 Copy Placement Analysis Details

A copy entry is represented as a three-tuple:

$$e =< copy\_loc, var, alloc\_site > \tag{10}$$

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.

6.1.1 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 5.

Rule 1: array updates, $S_i : a(y) = x$: Recall from Section 5 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

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

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.

```
1: b = [2,  4,  8];
2: a = b;
     if (cond)
3:    c = rand(10);
       ...
4:    a = c;
     end
5: a(i) = 10;
6: disp(a);
7: 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.

6.1.2 `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.

6.1.3 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 6.1.1.

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 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 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. With this assumption, there is no need to compute the intersection of C_{loop} and C_{out}. Hence

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

6.2 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 values are the same for all the flow sets and are therefore omitted from the flow entries shown in the table.

```
1  function test3()
2    a = [1:5]
3    b = a
4    i = 1;
5    if (i > 2)
6       a(1) = 100;
7    else
8      a(1) = 700;
9    end
10   a(1) = 200;
11   disp(a);
12   disp(b);
13 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 *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 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 6.1.2, 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 6.1.1) 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 [12]. The forward analysis information is shown in Table VI. The table shows the *gen* and *in* 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.

20: x = d;
21: n = length(x);

    for j = 1:n−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
31:     x(j) = (x(j)−c(j)∗x(j+1))/b(j);
    end
end
```

The analysis continues by generating the *gen*, *in* and *out* sets according to the rules specified in Section 5. 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.

#	**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	$\emptyset$	$\{(a', S_a, 0), (b, S_b, 0), (b', S_b, 0), (c, S_c, 0), (c', S_c, 0), (d', S_d, 0), (b, S_{25}, 3), (x, S_{29}, 0)\}$

Table VI: Forward Analysis Result for *tridisolve*

Table VII shows the copy placement analysis information at each relevant statement of *tridisolve*. 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 *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.

#	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*

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 Table VII. 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*.

It is interesting to note that the number of copies has been reduced and all the copies generated in loop F_2 were successfully moved out of the loop because there were no "loop dependencies". 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 therefore 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.

7 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 5 relies on the McVM type inference analysis [11, 10] 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.

8 Experimental Results

To evaluate the effectiveness of our approach, we set up experiments using benchmarks collected from disparate sources, including those from [24, 12, 23]. Table VIII 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™ 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 (Section 8.1). Knowing the number of updates gives an idea of how many dynamic checks a reference-counting-based scheme for lazy copying, such as used by Octave and Mathworks' MATLAB, need to perform. Remember 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 reference-counting-based approaches. Secondly, we know that dynamic checks generate overheads, and we would like to measure the amount of these overheads in reference-counting-based systems (Section 8.2). Finally, we would like to assess the impact of our static analyses in terms of their ability to minimize the number of copies (Section 8.3).

8.1 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 reference-counting-based 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* [7]. The *amc* compiler accepts a MATLAB program and an aspect written in ASPECTMATLAB language — an extension of the MATLAB programming language. Our aspect[4] 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 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 thus counts all array updates and all copies that would be required by a reference-counting-based system.

In Table VIII 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 (since they perform copies lazily, and only when required) and are therefore grouped under *Lower Bound* in the table.[6]

At a high-level, the results in Table VIII 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.[7]

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

[5]All of these benchmarks are also available at: www.sable.mcgill.ca/mclab/mcvm_mcjit.html.

[6]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.

[7]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.

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
diff	diffraction pattern calculator	0	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

Table VIII: Benchmarks and the results of the copy analysis[5]

With Analyses comprises of three columns, **Naive**, **QC**, and **CA** representing respectively, the number of copies generated in our naive implementation, with the QuickCheck phase, and with the copy analysis phase. We return to these results in Section 8.3.

8.2 The Overheads of Dynamic Checks

With reference-counting-based 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 IX, for three benchmarks with a high number of updates and no required copies (`dich`, `fiff` and `mbrt`).

	McVM						Octave(O)		
	McJIT		McJIT(+RC)		Overheads(%)		Time(s)		Overhea
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

Table IX: Overheads of Dynamic Checks.

We first created a modified 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 dynamic 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 our McVM JIT implementation to include enough reference-counting machinery to measure the overheads of the checks (remember 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 IX shows, in time and code size, the amount of overheads 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 dynamic checks. We measured code size using the number of LLVM instructions (**# LLVM**) and execution time overhead in seconds. For these three benchmarks the code size overhead was 3% to 14% and the execution time overhead ranged from 12% to 47%.

Our conclusions from these experiments is that the dynamic checks for a reference-counting-based 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 reference-counting-based systems.

8.3 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 VIII. 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 unnecessary copies, it does not achieve the lower bound. Thus, the second stage is really required in many cases.

In Table VIII and under McJIT with *CA*, 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. Although the QuickCheck(QC) is capable of reducing the number of arguments copied from a caller to a callee by identifying the parameters that are modified in the callee, it is incapable of determining whether or not an array-return value be copied. This explains why McJIT with QC makes more copies that with CA even in *crni* and *capr* benchmarks.

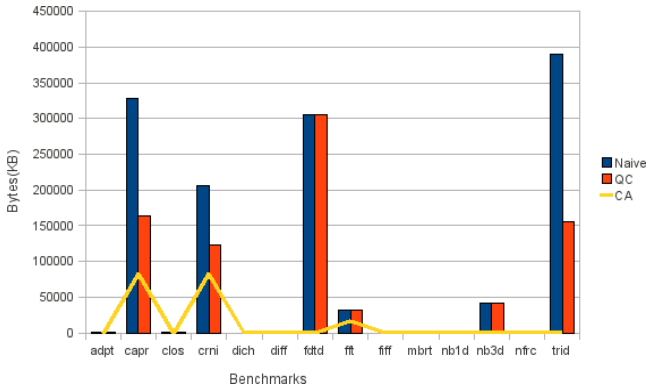

Figure 2: Amount of bytes copied by the benchmarks under the three options.

To show the impact copies have on execution performance, we measure the total bytes of array data copied by each benchmark. This is shown in Figure 2. The results correspond with those in Table VIII for *Naive*, QC and CA. The columns $\frac{Naive}{QC}$ and $\frac{Naive}{CA}$ 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 X 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 overheads of CA is low. It is therefore clear from the results of our experiments that the naive implementation generates significant overheads and is therefore unsuitable for a high-performance system.

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 overheads of dynamic checks. Furthermore, our approach does not require reference counting and thus enables an efficient implementation of array

Bmark	Naive	QC	CA	$\frac{Naive}{QC}$	$\frac{Naive}{CA}$
adpt	1.57	1.57	1.61	1.00	0.98
capr	1.54	0.91	0.58	1.70	2.66
clos	0.49	0.49	0.48	0.99	1.01
crni	135.09	140.35	131.62	0.96	1.03
dich	0.18	0.18	0.18	1.00	1.00
diff	4.26	4.27	4.14	1.00	1.03
fdtd	3.79	3.78	2.80	1.00	1.35
fft	1.50	1.50	1.47	1.00	1.02
fiff	0.39	0.39	0.39	0.99	0.99
mbrt	5.06	5.12	5.04	0.99	1.00
nb1d	0.48	0.48	0.45	1.00	1.07
nb3d	0.48	0.48	0.36	1.00	1.35
nfrc	3.23	3.23	3.25	1.00	0.99
trid	1.57	1.04	1.02	1.51	1.53

Table X: Benchmarks against the total execution times in seconds

copy semantics in garbage-collected systems like McVM.

9 Related Work

Redundant copy elimination is a hard problem and implementations of languages such as Python [4] 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 [15] 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 [17, 22, 25, 26, 28, 16, 13]. 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 [18] is one of the oldest array-based languages. Weigang [29] 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 [17] 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 [21] 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 [28] 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.

10 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.

The advantages of our approach are that it 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.

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] Python. `http://www.python.org`.

[5] 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.

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

[7] 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.

[8] H. Boehm and M. Spertus. N2310: Transparent Programmer-Directed Garbage Collection for C++, June 2007. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2310.pdf`.

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

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

[11] 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.

[12] Cleve Moler. *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics, 2004.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Iverson, Kenneth E. *A Programming Language*. John Wiley and Sons, Inc., 1962.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

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

[29] Weigang, Jim. An Introduction to STSC's APL Compiler. *SIGAPL APL Quote Quad*, 15(4):231–238, 1985.

[30] 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.