Static Array Storage Optimization in MATLAB

Pramod G. Joisha
ECE Department
Northwestern University, Evanston, IL 60208
pjoisha@ece.northwestern.edu

Prithviraj Banerjee
ECE Department
Northwestern University, Evanston, IL 60208
banerjee@ece.northwestern.edu

ABSTRACT

An adaptation of the classic register allocation algorithm to the problem of array storage optimization in MATLAB is presented. The method involves the decomposition of an interference graph's color classes using *inferred* type information. A key trait is the use of symbolic types, along with control flow, in performing the decomposition. On a benchmark suite spanning the published test suites of some recent research MATLAB compilers, our implementation produces savings in the average virtual memory size, with respect to code generated by a commercial MATLAB compiler, of between 51% and 139% in 6 out of 11 programs, and savings between 0.7% and 47% in the remaining. In absolute terms, this ranged from 123KB to over 9MB. Substantial improvements in other categories of memory, such as resident sets and dynamic data (stack plus heap), were also observed and are reported. Speedups in execution times of at least over an order of magnitude in 4 programs, of over 100% in 4 of the remaining, and 10% and over in the rest are also reported.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; D.3.4 [**Programming Languages**]: Processors—*compilers and optimization*

General Terms

Algorithms, Design, Languages, Performance

1. INTRODUCTION

The automatic optimization of program storage is a well-known problem, one that has been extensively investigated and efficiently tackled in the past. Because programs in early computers were constrained by the sizes of their cores, initial work centered on main memory allocation [19, 14, 15]. As large main memories became cheaply available, interest in this area waned with the focus shifting to register allocation [6, 8, 4, 3, 23]. This paper revisits the main memory allocation problem in the context of a typeless array-based programming language called MATLAB and shows how the efficient management of array storage is crucial to its successful *static* compilation. The fact that we have to contend with arrays whose type characteristics are not expressly specified in a program is an important distinction between our work and previous efforts at name reclamation [11, 22] and storage reduction [24, 9]. The issue of storage optimization is particularly relevant to MATLAB because the language is increasingly being used in the prototyping of applications that ultimately get deployed on limited memory platforms such as DSPs, FPGAs and embedded devices, and because it espouses an array-centric programming idiom, one in which the elemental manipulation of data is eschewed for whole array processing.

Closer to our work is Fabri's approach [15] which viewed main memory allocation as a weighted graph coloring problem. Nodes in the interference graph were weighted by the sizes of the arrays in question and a coloring was sought that minimized the total allocated storage. What complicates the problem in MATLAB is that size information may be absent since the language lacks an explicit declaration of type. In fact, even when all array sizes are known, the problem does not naturally map to graph coloring the way the register allocation problem does. Factors such as partial interference (see § 2.1) further ravel the picture.

Surprisingly, though the same problems existed in APL, completely different techniques, like "drag-along" and "beating" [1], delayed evaluation [16], chaining [7] and demand driven execution [5], were used to reduce storage overheads.

Research compilers that translate MATLAB statically [13] and just-in-time [2] have emerged in recent years. However, storage optimization has been an unaddressed area in MATLAB. And although systems such as MaJIC [2] that compile code during execution are better poised to efficiently manage storage, we must keep in mind that such run-time systems will themselves compete with the application for platform storage space. Therefore, we believe that there exists a niche of applications that could greatly benefit from the better static management of MATLAB's array storage.

1.1 Algorithm Overview

The input to our algorithm is a control-flow graph (CFG) that is in the Static Single-Assignment (SSA) [12] form. The algorithm, referred to as GCTD (Graph Coloring with Type-based Decomposition) consists of two phases. The first involves the creation and coloring of an interference graph to determine variables that can share common storage. Because the objective isn't the ultimate assignment of variables

PLDI'03, June 9–11, 2003, San Diego, California, USA.

to a machine's register set, we neither seek an n-coloring, where n is the number of available machine registers, nor deal with related issues such as register spilling, as is the case in [6]. Instead, the goal of the first phase is to determine, at least conservatively, noninterfering *classes* of variables that can be safely assigned to the same area of memory without changing the meaning of the program. To this end, the aim is to find a coloring of the interference graph.[1] The treatment of interference in the first phase is similar to that in [6] except that a new kind of conflict arising from operator semantics needs to be considered and resolved, perhaps through the involvement of inferred type information (see § 2.3). The second phase decomposes the color classes on the basis of expression types. Basically, the idea is to partition each color class into *groups* so that all variables in a group are laid out in memory starting from the same location as a certain distinguished set of variables in the group. Members of that distinguished set have the predominant storage requirement in the group. The cornerstone of the second phase is a partial order that relies on program variable types, and in the symbolic case, on control-flow information too.

2. INTERFERENCE GRAPH (PHASE 1)

The interference graph $G_f^* = (V_f, E_f)$, constructed at the level of a user-defined MATLAB function f, has a node set V_f that represents the variables defined in the CFG and an edge set E_f that represents the interference among the variables. Two variables are considered to interfere if their def-use chains (du-chains) overlap. In particular, we use the Chaitin et al. [6] notion of interference by which two variables interfere if there exists an execution path in f and a point therein at which: (1) the definitions of both variables are visible; (2) both variables are not dead; and (3) the variables have different values. Because determining these criteria is in general undecidable, we use an approximation of interference by considering variables that are both *available* and *live* at each assignment [6, 3]. A variable v is regarded available at a statement s if there is a possible execution path from a definition of v to s. A variable w is treated live at s if there is a possible execution path from s to a use of w along which w is not redefined. "Liveness" and "availability" as defined here are conservative because they indicate a potential, rather than a definitive, use and definition. Our approach to detecting variables that satisfy these two conditions is effectively that described in [3]. A basic block is traversed backwards starting with the set of variables that are live and available at its end. A definition encountered during the traversal is interfered with members of the set. Before moving to the next statement, the set is updated by removing the variable (or variables) defined at that statement and adding variables used at the same statement.

2.1 Partial Interference

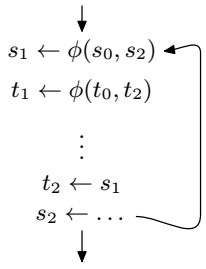
The concept of interference described in [6] was intended for scalars. When directly applied to variables that can also be arrays, which is the case in MATLAB programs, it can be more pessimistic than necessary. This will happen when variables only partially interfere. As an example, the SSA conforming intermediate representation (IR) shown below creates a pair of 2×2 matrices in `a` and `b`,[2] assigns to `c` the first element in `a`,[3] then computes the sum of `b` and `c` in `d`, and finally displays `d`. Because the du-chain of `a` crosses that of `b`, there clearly is an interference between `a` and `b`.

```
a ← rand(2, 2);
b ← rand(2, 2);
c ← subsref(a, 1);
d ← b+c;
disp(d);
```

However, by inspecting the index used in the access, we see that this interference is only in the first element of `a`. Although `a` and `b` are considered to fully interfere in our current implementation, which causes them to be assigned to disjoint areas in memory, note that their storage areas could have been overlapped allowing for the use of a total of five double precision memory locations to perform the above computation in its entirety.

2.2 Handling Copies

A copy statement of the form $X \leftarrow Y$ will by itself not introduce an interference between X and Y. However, if the coloring step ascribes different colors to X and Y, the final code generation stage will have to emit code to copy Y to X. Because variables in MATLAB can be arrays, it is even more important, in relation to a situation that deals only with scalars, to avoid copy operations whenever possible. In [6], this was attacked by coalescing X and Y and then rebuilding the interference graph since the action of coalescing can alter the graph; the process was repeated up to a fixed number of times until no further coalescences were possible. Our strategy is different: we preprocess the CFG to free it of copies by subjecting it to a copy propagation pass followed by dead-code elimination.[4] Obviously, all copies cannot be eliminated this way; one such case is shown in the IR below.



In the above, copy propagating s_1 from $t_2 \leftarrow s_1$ to $t_1 \leftarrow \phi(t_0, t_2)$ will change the meaning of the IR. Of course, t_2 and s_1 in the interference graph will also not be coalescent because they interfere due to overlapping du-chains.

2.2.1 SSA Inversion Copies

Since converting back from the SSA form will reintroduce copies [12], it is imperative that the name defined at a join

[1] Currently, our implementation seeks a coloring that is as close to the minimal coloring as possible. This, however, may not yield the final optimal solution to the problem.

[2] Assignments will be denoted using the ← symbol.

[3] `subsref(a,` i_1`,` i_2`, ..., ` i_m`)` returns the array element `a(`i_1`,` i_2`, ..., ` i_m`)`.

[4] The MATLAB translator has over 20 passes that perform a variety of transformations such as global common-subexpression elimination, constant folding, constant propagation, type determination, code selection and code generation.

node share storage with the names used at that node whenever possible. This will make the reintroduced copies identity assignments and therefore trivially removable. To achieve this for the join node $Z \leftarrow \phi(X, Y)$, the nodes in the interference graph corresponding to Z and X (and then Z and Y) are examined for any interference. If they don't interfere, they are coalesced and the interference graph is appropriately updated. This will force any coloring to assign the same color to Z and X so that the copy $Z \leftarrow X$ reintroduced during the SSA inversion step becomes a trivially removable identity assignment. Of course, such coalescings will affect the chromatic number of the graph and constrain the coloring finally obtained. But we have found the folding of copies to be indispensable to the generation of efficient code, because even a few copies, involving large-sized arrays and nested within loops, can significantly impact the generated code's performance through excessive paging activity.

2.3 Interference due to Operator Semantics

Assignments in our IR have right-hand sides that consist of at most a single MATLAB operation like `*`, or a pseudo operation like the ϕ function. Assignments in MATLAB can be cast into this Single Operator (SO) form through the introduction of temporaries. The code generation pass then directly maps each of the SO form IR assignments to C code.

Consider the IR assignment `c` $\leftarrow$ `a*b`. If `a` and `b` are scalars, storage for `c` and `a`, or `c` and `b`, can be shared, assuming that these pairs don't otherwise interfere. We say that `c` can be computed *in-place* in either operand in the C mapping without violating the semantics of the operation. However, `*` can operate on both scalars and nonscalars in MATLAB. For instance, when `a` and `b` are nonscalar matrices, `*` performs the elementary matrix multiplication operation. In that situation, computing `c` in-place in either `a` or `b` will risk violating the semantics of the operation because elements in `a` or `b` could get overwritten before being fully used. However, when either `a` or `b` is a scalar, and the other operand is an arbitrary array, `c` is produced by the elementwise multiplication of the scalar with the array. In that case, `c` can be computed in-place in the array operand.

Thus, the storage coalescing pass inspects each IR assignment of the form $Y \leftarrow op(X_1, X_2, \ldots, X_m)$, and inserts additional interferences between Y and an X_i $(1 \leq i \leq m)$ on the basis of what *op* is, and the type information inferred for each X_i. Hence, for the case `c` $\leftarrow$ `a*b`, an interference is *not* added between `c` and `a`, and between `c` and `b`, if either `a` or `b` can be determined to be scalars. Since `c` can then be calculated in-place in the larger sized operand, we still have to figure out which of the two operands can accommodate `c`. This is done in the second phase of the GCTD pass.

2.3.1 Array Addition

An example of a different *op* is in the IR statement `c` $\leftarrow$ `a+b`. In MATLAB, `+` is the array addition operation; it always computes the elementwise sum of its operands. Since `c` can be computed in-place in either `a` or `b` provided the operand is sufficiently sized, no additional interferences have to be inserted between `c` and either operand in this case. Figure 1 displays exactly how the C code generated by the MATCH translator performs this in-place computation. (Not shown are preceding shape correctness and resizing checks.)

2.3.2 Right-Hand Side Array Indexing

In the IR statement `c` $\leftarrow$ `subsref(a, 1)` seen in § 2.1, *op* is `subsref`, which corresponds to the right-hand side array indexing operation in MATLAB. We call this the *R*-indexing operation for short. Note that in the case seen in § 2.1, `c` can be computed in-place in `a`. (There are only two possibilities: `subsref(a, 1)` is either illegal, which will happen if `a` is the empty array, or it is legal, in which case it is a scalar.) However, if the statement were

$$\texttt{c} \leftarrow \texttt{subsref(a, } e\texttt{)}$$

where e was unknown, such an in-place computation may not be possible. This is because MATLAB permits e to itself be an array; this allows an arbitrary permutation of the elements of `a` to be returned. For instance, if e were the MATLAB colon expression `4:-1:1`, `subsref(a,` e`)` would give the elements of the 2×2 matrix `a` in reverse. Again, type information could be used to differentiate these situations—in this case, whether the subscript e is a scalar.

2.3.3 Left-Hand Side Array Indexing

MATLAB also offers a left-hand side array indexing operation. In source form, this operation, which we call *L*-indexing for short, is specified by the construction

$$\texttt{a(}l_1\texttt{, } l_2\texttt{, } \ldots\texttt{, } l_m\texttt{)} \leftarrow r$$

This is represented in our IR as `a` $\leftarrow$ `subsasgn(a,` r`,` l_1`,` l_2`, ...,` l_m`)`. When cast to the SSA form, it becomes

$$\texttt{b} \leftarrow \texttt{subsasgn(a, } r\texttt{, } l_1\texttt{, } l_2\texttt{, } \ldots\texttt{, } l_m\texttt{)}$$

The meaning of this operation is as follows: `b` has the same elements as `a`, except for elements located by the m subscripts l_1, l_2 and so on until l_m, which are set to elements from r. The `subsasgn` operator therefore exhibits semantics similar to that of the `Update` operation described in [12] except for two important departures:

- Each subscript l_i $(1 \leq i \leq m)$ can be an arbitrary array. The locations in `b` that are set to elements in r are obtained by taking the Cartesian product of the elements in the subscripts. If p_i is the number of elements in subscript l_i, r is required to have the shape $p_1 \times p_2 \times \cdots \times p_m$. (There are corner cases that need to be handled. See [21] for details.)
- If the maximum elemental value in l_i exceeds the extent of `a` along the ith dimension, `b` has that maximum value for its extent along the ith dimension. Locations created in `b` due to such expansions are set to 0.

In fact, there is one more case that can arise due to an additional "shrinkage" feature that MATLAB provides for the *L*-indexing operation. For example, `a(:, :, 2)` $\leftarrow$ `[]` deletes all elements on the second "page" [21] of an $x \times y \times z$ array, shrinking it to an $x \times y \times (z-1)$ array. This functionality isn't currently supported by the MATCH translator.

2.3.3.1 In-Place *L*-Indexing.

If the requisite amount of storage is available, can `b` be computed in-place in `a` for all possible l_i without risking the

```
if ((___STC(_826s_4, 1) == 1 && ___STC(_826s_4, 2) == 1))
   { /* First operand is a scalar. */
    for (__i = 0; __i < (___STC(_833s_4, 1)*___STC(_833s_4, 2)); __i++)
    _811s_4[__i] = _811s_4[0]+_804s_4[__i];
    }
else if ((___STC(_846s_4, 1) == 1 && ___STC(_846s_4, 2) == 1))
   { /* Second operand is a scalar. */
    for (__i = 0; __i < (___STC(_833s_4, 1)*___STC(_833s_4, 2)); __i++)
    _811s_4[__i] = _811s_4[__i]+_804s_4[0];
    }
else
   { /* Both operands have identical shapes. */
    for (__i = 0; __i < (___STC(_833s_4, 1)*___STC(_833s_4, 2)); __i++)
    _811s_4[__i] = _811s_4[__i]+_804s_4[__i];
    }
```

Figure 1: Partial MATCH C Code for the IR _811s_4 ← _811s_4+_804s_4 from the capr Benchmark

violation of semantics? It turns out that in the absence of the shrinkage feature, this can *always* be done by computing the elements of b from the last to the first. This works because there are only two cases that need consideration:

- *The array* b *doesn't expand and has the same shape as* a. Then, those elements of a that get carried over to b will occupy the same positions in memory.
- *The array* b *expands along one or more dimensions of* a. Then, those elements of a that get carried over to b will occupy the same or higher positions in memory.[5]

Hence, forming b backwards will ensure that the elements in a get carried over to b before they can get overwritten.

2.4 Coloring Heuristic

A simple $O(V_f + E_f)$ greedy heuristic that attempts to use as few colors as possible has been used in our current implementation. The heuristic visits each node in turn, in the lexical order of the corresponding variable definitions, and assigns to it the smallest among the colors used so far that is consistent with its neighbors. If no such color exists, a new color is created and the node is assigned that color.

3. TYPE-BASED ALLOCATION (PHASE 2)

Let $V_f(c)$ be a color class in the interference graph G_f^*. Let $\mathcal{S}(v)$ be the size of the storage allocated for a variable v in $V_f(c)$. Intuitively, if the storage sizes of all variables in $V_f(c)$ were known, they could all be overlaid starting from the same location as the largest sized variable among them. Unfortunately for the compiler writer, MATLAB features both implicit typing (an expression's type may not be explicit from program syntax) and dynamic typing (an expression's type may depend on the control-flow path exercised at run time). Consequently, not only may $\mathcal{S}(v)$ be statically indeterminable, but it may also vary during execution. Our heuristic approach to getting around these problems is to construct a storage-size partial order $\preceq$ that tries to capture a containment relationship while promoting spatial reuse. (The precise definition of this binary relation is given in § 3.2.) The construction is done using *inferred* type information. Once $\preceq$ is formed, its maximal[6] elements are found, which are then used to decompose $V_f(c)$ into groups. If there are q maximal elements under $\preceq$, $V_f(c)$ is partitioned into a collection of q groups, one for each maximal element, such that the storage sizes of variables in a group are bounded by the maximal element corresponding to that group.

3.1 The MAGICA Type Inference Engine

The MATCH translator currently obtains the types of all variables defined in a program using an inference engine called MAGICA (MAthematica system for General-purpose Inferring and Compile-time Analyses) [17]. Given a Mathematica representation of a MATLAB program, MAGICA infers the value range $\upsilon(w)$, intrinsic type $\tau(w)$, shape tuple $\sigma(w)$ and rank (i.e., array dimensionality) $\rho(w)$ of each variable w. If MAGICA cannot explicitly infer the extents or dimensionality of some w, it will return symbolic expressions for $\sigma(w)$ and $\rho(w)$ respectively. The unique aspect of these shape-tuple and rank expressions is that inferences are reused whenever symbolic equivalence can be established [18]. That is, MAGICA will return the shape-tuple expression $\sigma(v)$ for the shape tuple of w if it can establish that $\sigma(v)$ will definitely be live (in the data-flow sense) at the definition of w and that $\sigma(v)$ and $\sigma(w)$ will always be equivalent.

3.2 The Storage-Size Partial Order

The GCTD pass determines the storage size of a variable u either by *statically estimating* it, or by symbolically calculating it to be $|\sigma(u)||\tau(u)|$ where $|\sigma(u)|$ denotes the number of elements in an array whose shape tuple is $\sigma(u)$,[7] and where $|\tau(u)|$ denotes the storage size of a scalar having the intrinsic type $\tau(u)$.[8] The pass uses the following formulation

[5] MATLAB organizes its arrays *à la* FORTRAN. But this observation holds even if the layout was row major.

[6] An element x is maximal under a partial order $\preceq$ if there exists no y such that $x \prec y$.

[7] An instance of a shape tuple would be $\langle 1, 4, 5\rangle$, which denotes the shape of an array having the extents 1, 4 and 5 in the first, second and third dimensions respectively.

[8] Intrinsic types in MAGICA can be any of BOOLEAN, BYTE, INTEGER, REAL, COMPLEX, NONREAL and the abstract "illegal" intrinsic type i that signifies intrinsic type errors.

to relate the storage sizes of two variables u and v:

$$\mathcal{S}(u) \preceq \mathcal{S}(v) \text{ iff } \begin{cases} \mathcal{S}(u) \text{ and } \mathcal{S}(v) \text{ can be} \\ \text{statically estimated,} \\ \tau(u) = \tau(v) \text{ and } \mathcal{S}(u) \leq \mathcal{S}(v), \\ \\ \mathcal{S}(u) \text{ and } \mathcal{S}(v) \text{ cannot be} \\ \text{statically estimated,} \\ u \text{ is available at} \\ \text{the definition of } v, \\ \tau(u) = \tau(v) \text{ and } \mathcal{S}(u) \leq \mathcal{S}(v). \end{cases} \quad (1)$$

Because "available at the definition" is both a reflexive and transitive relationship (see § 2), it is easy to see from Relation 1 that $\preceq$ is indeed a partial order. The motivation behind the formulation is to identify two categories of arrays: those whose layouts can be fixed at compile time and those among the dynamically allocated arrays whose storages can be grown in a regular way. The two disjoint criteria for $\preceq$ require identical intrinsic types; this was intentional so as to avoid both the use of type castings in the generated C code and the issue of alignment restriction in C.

3.2.1 Stack Allocation

Static estimation of storage size is done in two situations:

1. the inferred shape tuple $\sigma(u)$ of a variable u is *explicit*—that is, of the form $\langle p_1, p_2, \ldots, p_k \rangle$ where each extent p_i $(1 \leq i \leq k)$ is an integer; or

2. the variable u is defined at the join node $u \leftarrow \phi(v, w)$ and the sizes of both v and w are statically estimable.

The estimated size in the first case is $|\sigma(u)||\tau(u)|$, while in the second case it is $\max(\mathcal{S}(v), \mathcal{S}(w))$. Arrays whose sizes can be statically estimated get allocated on the stack in the C translation. In particular, scalars in MATLAB can be directly mapped to scalar automatics in C. Besides enabling procedures to be reentrant, a stack allocation discipline automatically materializes and disappears objects as procedure activation records get pushed and popped. Backend C compilers can also take advantage of the fact that the relative displacements of each of the arrays within a stack frame will be compile-time knowns; knowing this could be helpful while gathering data dependency or aliasing information. Furthermore, because all statically estimable sizes of the same intrinsic type within a color class form a single chain under the partial order $\preceq$, the corresponding variables, which will form a single group, could all be allocated within a single array whose size is the maximal element in the chain.

3.2.2 Heap Allocation

Variables that satisfy neither condition in § 3.2.1 are deemed as having statically inestimable sizes. These variables get allocated on the heap. Their storage sizes are expressed symbolically as $|\sigma(u)||\tau(u)|$. The GCTD pass binds all variables within a group to a common storage area. Code generated by our implementation also attempts to alleviate heap memory pressure by resizing storage on the fly to the specific needs of each variable in a group. This is in contrast to the stack allocation case where the storage size of a group remains fixed at the maximal during a procedure activation. By incorporating the "available at the definition" clause, the second criterion of Relation 1 tries to identify stretches in the execution path along which arrays grow in one direction. Specifically, if $\mathcal{S}(u) \preceq \mathcal{S}(v)$ by the second criterion, then u must both be available at the definition of v and dead after it. (Otherwise, u would have interfered with v in Phase 1.) Thus, chained elements in $\preceq$ connected by the second criterion would potentially correspond to definitions and uses that get performed in sequence at run time and where the definitions step through nondecreasingly sized arrays. ("Potentially" because availability as defined in § 2 is conservative.) Moreover, the closer u is to v in the control-flow path, the better would be the spatial reuse characteristics because u would still be in the higher levels of a memory hierarchy when it gets resized to v. In fact, the closest that u could be to v is if it is used in the same statement that defines v. Closeness may thus be fostered by ensuring that the chains included in each group are as long as possible.

Example 1: Nonresized Arrays with Symbolic Types
When presented with the IR shown below on the left, the intrinsic types of t_1, t_2 and t_3 will be inferred by MAGICA to be COMPLEX, assuming that nothing is known about t_0.

```
t1 ← t0-1.345;          s◇ ← s-1.345;
t2 ← 2.788.^t1;   ⟹     s◇ ← 2.788.^s;
t3 ← tan(t2);           s◇ ← tan(s);
```

MAGICA will also return the *symbolic* expression $\sigma(t_0)$ for the shape tuples of t_1, t_2 and t_3 [18]. This reflects the fact that under *all* executions of the IR, the shapes of t_1, t_2 and t_3, which are all results of elementwise operations in MATLAB, will be identical to that of t_0. Assuming that t_0, t_1 and t_2 are dead on exit from the code fragment, we see that no pair of variables interfere, either due to overlapping du-chains or due to operator semantics. Therefore all nodes in the fragment's interference graph can be ascribed the same color. Furthermore, because $\sigma(t_0) = \sigma(t_1) = \sigma(t_2) = \sigma(t_3)$, $\tau(t_0) = \tau(t_1) = \tau(t_2) = \tau(t_3)$, and each t_i is available at the definition of t_{i+1} $(0 \leq i \leq 2)$, we get $\mathcal{S}(t_0) \preceq \mathcal{S}(t_1) \preceq \mathcal{S}(t_2) \preceq \mathcal{S}(t_3)$. Thus, all the variables in the fragment can be bound to a common storage that will be reused during the fragment's execution. In fact, since $|\sigma(t_0)||\tau(t_0)| = |\sigma(t_1)||\tau(t_1)| = |\sigma(t_2)||\tau(t_2)| = |\sigma(t_3)||\tau(t_3)|$, the storage sizes for all four variables can be statically determined to be the same. This means that at each of their definitions, their associated storage needn't be resized at run time. The IR shown above on the right indicates this by using the $\diamond$ superscript to denote a defined array that isn't resized.

Example 2: Expandable Arrays with Symbolic Types
A more interesting example that may involve array expansion is shown in the IR given below. Here, an $x \times y$ identity matrix is created in a, which is then used to create a matrix b through the subsasgn operator.

```
a ← eye(x, y);
b ← subsasgn(a, 1, i1, i2);
```

Assuming that a is dead at the end of the code fragment, we see that the du-chains of a and b don't overlap. And because b can be formed in a provided the latter is large enough (see § 2.3.3), we observe that a and b don't interfere. Hence, a minimal coloring of the interference graph will put both a and b in the same color class. For the intrinsic types of a and b, MAGICA will return BOOLEAN; it will also return expressions for the shape tuples of a and b that in the absence of further information on x, y, i_1 and i_2

will likely be symbolic. However, because of the semantics described in § 2.3.3, we can be assured that $|\sigma(\mathtt{a})| \leq |\sigma(\mathtt{b})|$ will *always* be true. Thus, because $\tau(\mathtt{a}) = \tau(\mathtt{b})$ is also true, and a is available at the definition of b, $\mathcal{S}(\mathtt{a}) \preceq \mathcal{S}(\mathtt{b})$ will hold by Relation 1. Hence, a can share the same storage as b. If the storage sizes of both a and b are statically inestimable, a resizing check would have to be inserted before the code generated for each of the IR statements. If a and b have statically estimable sizes, both will be stack allocated within a maximal sized storage. These two situations are shown in the IR below where the $\pm$ superscript indicates a defined array that may need resizing and the $+$ superscript indicates a defined array that if resized, will be grown. The superscripts are applicable only in the dynamic case.

$s^{\pm} \leftarrow$ `eye(`x`, `y`);`
$s^{+} \leftarrow$ `subsasgn(`s`, 1, `i_1`, `i_2`);`

There exists one situation where a and b won't share the same storage even if they don't interfere; this will happen if the size of only one of them can be statically estimated.

3.3 Decomposing a Color Class into Groups

A directed graph $G = (V, E)$ is used to represent the storage-size partial order $\preceq$ on a color class V. A directed edge $u \rightarrow v$ is introduced between a pair of distinct variables u and v in V if and only if $\mathcal{S}(u) \preceq \mathcal{S}(v)$. The algorithm DECOMPOSE-COLOR-CLASS given below performs the decomposition.

DECOMPOSE-COLOR-CLASS(G)
1 Find the *component graph* G^{SCC} of G.

The component graph $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$ of a directed graph $G = (V, E)$ has a node for each strongly connected component (SCC) in G, and a directed edge $x \rightarrow y$ if there is a directed edge from a node in the SCC in G corresponding to x to a node in the SCC in G corresponding to y [10].

2 Decompose G^{SCC} into a forest of trees by invoking either depth-first search (DFS) or bread-first search (BFS) on nodes with in-degrees of 0.

All variables in an SCC of G have the same storage size. Thus, the root of each tree returned by DECOMPOSE-COLOR-CLASS corresponds to an SCC in G whose variables have a maximal storage size under $\preceq$. This is because the roots have in-degrees of 0 in G^{SCC}. Note that in-degrees of 0 will exist because G^{SCC} has the important property of being acyclic [10]. Therefore, variables in all those SCCs in G that correspond to a returned tree's nodes have storage sizes that are bounded by a maximal element in $\preceq$. Hence, the collection of trees returned by DECOMPOSE-COLOR-CLASS forms a decomposition of a color class into groups.

LEMMA 1. *If every node in a color class belongs to a unique maximal chain*[9] *under* $\preceq$*, then* DECOMPOSE-COLOR-CLASS *assigns all nodes in a maximal chain to the same group.*

PROOF. Suppose there exists a maximal chain in which nodes belong to different groups. Because both BFS and DFS touch all nodes reachable from a root, this means that there exists a u such that $\mathcal{S}(u)$ is bounded by at least two distinct maximal elements, which is a contradiction. □

[9]A chain is maximal if no other chain properly subsets it.

Hence, from Lemma 1, DECOMPOSE-COLOR-CLASS automatically fosters the closeness property mentioned in § 3.2.2 except in one situation. That exceptional situation occurs if a node is common to two or more maximal chains. The implementation currently assigns nodes common to two maximal chains wholly to one of them.

In terms of complexity, the running time of DECOMPOSE-COLOR-CLASS is $O(V+E)$ since Step 1 can be done in $O(V+E)$ time [10], and because Step 2 can also be done in $O(V+E)$ time since $V^{\mathrm{SCC}} = O(V)$ and $E^{\mathrm{SCC}} = O(E)$.

4. PERFORMANCE EVALUATION

The efficacy of the GCTD pass was evaluated by collecting metrics relating to memory footprints and execution times over a set of 11 MATLAB programs obtained from different sources. These benchmarks are listed in Table 1 along with brief descriptions, their origins, and their sizes in terms of the number of files that constitute their source codes (called *M-files* in MATLAB jargon) and the total number of nonempty noncomment lines in them.

4.1 Benchmarks' Organization

Programs in the set were organized along the lines of the FALCON benchmark suite in which the main function in a program is invoked from a driver routine. Typical tasks performed by a driver are the preparation of arguments for an invocation, the display of results from an invocation and the timing of the entire execution. Though MATLAB can handle built-in functions like `disp` and `fprintf` that produce outputs to external files, no support currently exists for loading data from external files. Some of the drivers in their original form did read data from external files; these were modified by the inclusion of the loaded data within the driver. This wasn't found to be a problem because the loaded data was confined to only a few scalars. Nevertheless, the current inability to handle MATLAB's `load` built-in function is a limitation of our implementation. Programs that produce data by alternate means, such as by using the MATLAB random number generator `rand`, can be handled by our system.

4.2 Platform Specifications

All measurements were done on a 440 MHz UltraSPARC-IIi workstation running Solaris 7 and having 128MB of main memory. The version of the MATLAB interpreter used was 6.1 (Release 12), while the version of `mcc` (The MathWorks' MATLAB-to-C compiler) used was 2.2.[10] Version 5.1 of the Sun Workshop C compiler was used for back-end compilation by both `mcc` and `mat2c` (the MATLAB program). Recall from § 3.1 that an external engine is used to infer the program variable types; this is invoked transparently by `mat2c` and was executed on version 4.1 of the Mathematica kernel.

The back-end C compiler in both cases was passed the `-xO4` and `-xlibmil` options that turn on a host of global and local optimizations, and inline certain library routines for faster execution. All optimizations offered by `mcc` were turned on. To minimize memory usage, the MATLAB interpreter was always run with the `-nojvm` option. This suppresses the loading of a Java virtual machine that allows a MATLAB session to draw on Java's capabilities.

[10]In July of this year, latest versions of both—6.5 (Release 13) and 3.0—were announced.

Benchmark	Synopsis	Origin	M-Files	Lines
adpt	Adaptive Quadrature by Simpson's Rule	† FALCON	2	79
capr	Transmission Line Capacitance	Chalmers University of Technology, Sweden	5	68
clos	Transitive Closure	☥ OTTER	2	30
crni	Crank-Nicholson Heat Equation Solver	FALCON	3	48
diff	Young's Two-Slit Diffraction Experiment	The MathWorks' Central File Exchange	2	40
dich	Dirichlet Solution to Laplace's Equation	FALCON	2	49
edit	Edit Distance	The MathWorks' Central File Exchange	2	34
❑ fdtd	Finite Difference Time Domain (FDTD) Technique	Chalmers University of Technology, Sweden	2	47
fiff	Finite-Difference Solution to the Wave Equation	FALCON	2	32
nb1d	One-Dimensional N-Body Simulation	OTTER	2	53
❑ nb3d	Three-Dimensional N-Body Simulation	Modified nb1d	2	46

❑ Benchmarks involve three-dimensional arrays.
† FALCON MATLAB Compiler Test Suite [13].
☥ OTTER Parallel MATLAB Compiler Test Suite [20].

Table 1: Benchmark Suite Description

Benchmark	Static/Dynamic Variable Reduction	Original Variable Count	Storage Reduction (KB)
adpt	127/74	271	0.96
capr	84/75	301	0.68
clos	24/0	46	1216.14
crni	73/0	113	4055.85
diff	48/1	93	12.77
dich	82/0	107	144.90
edit	25/21	108	0.21
fdtd	111/0	168	4374.61
fiff	51/0	77	12712.92
nb1d	66/63	235	0.55
nb3d	58/54	191	0.59

Table 2: Array Storage Coalescing Reductions

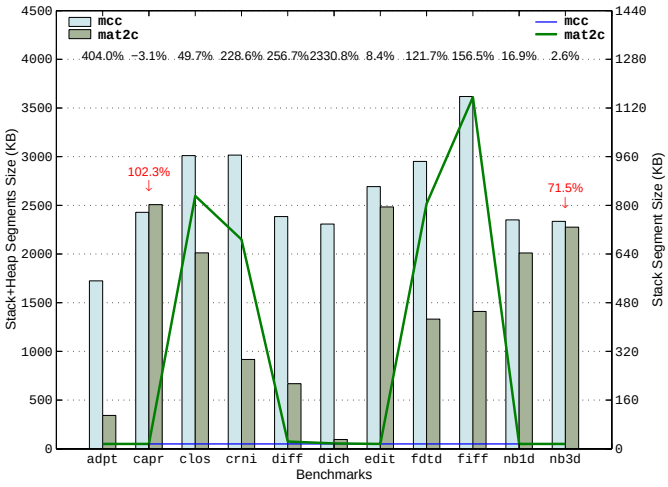


Figure 2: Average Stack, and Stack+Heap Levels

4.3 Storage Reductions

Table 2 shows reductions in variable count and corresponding reductions in storage size due to the GCTD algorithm. Entries in the "Static/Dynamic Variable Reduction" column are of the form s/d where s is the number of variables whose array sizes are statically estimable and that get subsumed in another array by the GCTD pass, and where d is the number of variables whose sizes are statically inestimable (thus requiring dynamic allocation) but which can still be statically subsumed within another dynamically allocated variable because of the storage-size partial order $\preceq$.

The "Original Variable Count" column indicates the total number of variables in the CFG on entry to the GCTD pass. This includes variables in the program source, as well as temporaries introduced in preceding transformations like the SSA and SO form (see § 2.3) conversions. Other kinds of variables that also come under the coalescing regime of the GCTD pass, such as those assigned in symbolic shape tuple and rank expressions, are included in the above count.

Observe that for five benchmarks, d is 0. This is because MAGICA manages to explicitly infer all the shape tuples in them, causing all of their storage to be stack allocated by the GCTD algorithm. The corresponding reduction in bytes due to the coalescing of stack allocated variables is displayed in the "Storage Reduction" column. This figure is thus conservative in the savings in storage that it reflects because reductions due to the coalescing of heap allocated variables aren't included in it. Still, in 4 out of 11 programs, static reductions are seen to be over a megabyte, and close to 13 MB in fiff. This is on account of the large coalescent arrays ($\approx 451 \times 451$) that are operated upon by this benchmark. This is also the reason why with coalescing turned on, code generated by mat2c for fiff runs nearly two orders of magnitude faster than code generated by mcc (see Figure 5), and why without coalescing, code generated by mat2c is nearly six orders of magnitude slower (see Figure 6)! The key contributors to the relatively large reductions in variable count are the temporaries introduced as part of the SO form conversion process. Since these temporaries only serve to break long expressions into smaller ones, they have considerable potential for reuse both within and across expressions.

4.4 How mcc Handles Arrays

The mcc compiler relies on run-time type determination for its generated C code. In essence, the approach is to represent every array by a C struct called mxArray, which besides embedding the contents of the array, has a number of fields that contain meta information such as the array's

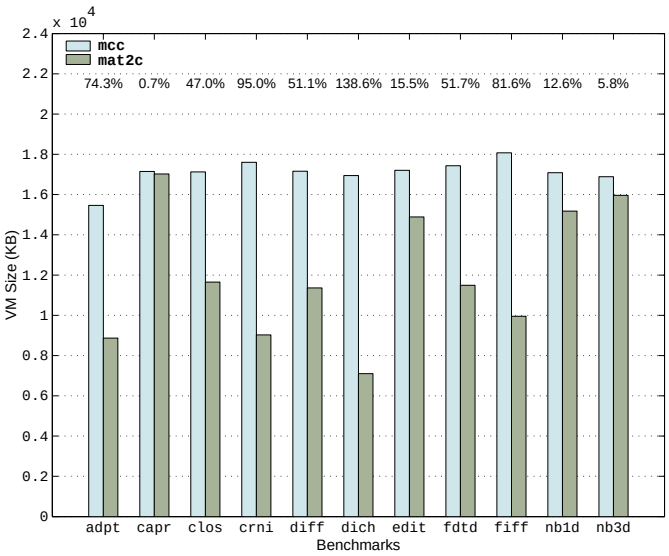


Figure 3: Average Virtual Memory Levels

shape and intrinsic type. These fields are set up at run time when arrays get created, and are examined and modified as conformance checks are performed and arrays evolve. A consequence is that all arrays in the `mcc`-generated C code are allocated on the heap. As a result, memory usage conservation also happens at run time where the library versions of each of the MATLAB operators have the onus of creating and deleting arrays. Array duplication due to copies is also minimized dynamically through sharing and a "copy-on-write" scheme. The way the automated memory management works is that `mxArray` structures created and returned within nested calls of library functions are deallocated immediately after being used. Another consequence of this run-time approach is that an `mxArray` structure, whose size in version 2.2 of `mcc` is 88 bytes, will be allocated for scalars that don't get folded at compile time.

This "library-based" model of compilation isn't unique to MATLAB; it is also used in compilers for other similar typeless array-based languages like APL.

4.5 Run-Time Memory Footprints

Figures 2 to 4 display the average memory levels in the stand-alone C codes automatically generated by `mat2c` with the GCTD pass turned on, and by `mcc`. Reductions in the dynamic program data sizes (stack plus heap space) relative to the `mat2c` C codes are shown as percentages above the bars in Figure 2. Relative reductions in other categories of memory are also shown as percentages in Figures 3 and 4.

4.5.1 Average Stack Trends

When a C program begins execution, there is already one stack frame on its run-time stack that contains the initial process environment such as the `argc` and `argv` parameters and the array of strings representing the process's environment variables. Thus, the stack segment, which grows in units of pages, will initially be at least one page in size, which is 8KB on the Solaris 7 UltraSPARC-IIi platform on which we ran our experiments. Further procedure invocations from `main` can cause the stack segment to grow depending on how local variables are allocated and referenced in the invoked functions. Because the `mcc` C codes bank on the heap for all array allocations and use function interfaces only for the passing and allocation of *handles* to these arrays, the high watermark of their run-time stacks shouldn't be expected to be large. Indeed, the `mcc` C codes for all benchmarks were found to have a stack segment size that grows to 16KB and stays at that. This is why the average stack segment size of the `mcc` C codes in Figure 2 is 16KB.

Figure 2 also shows four prominent peaks for the average stack segment size of the `mat2c` C codes for the `clos`, `crni`, `fdtd`, and `fiff` benchmarks. This is because `mat2c` allocates all arrays in these benchmarks on the stack. In the other benchmarks, significant percentages of the array shapes were symbolic; this led to their storage being primarily allocated on the heap. The exception was `dich` in which though 100% of the array shapes were statically inferred, the overall average stack segment size was only about 17.5KB because most of the arrays in this benchmark were small.

4.5.2 Average Heap Trends

Figure 2 also shows the average stack and heap space sums across all benchmarks. All average memory sizes, be it stack size, virtual memory level or resident set level, were calculated using a weighted time-averaged formula. If m_i was the observed memory size in some small duration Δt_i, then the average memory size M was calculated by the expression

$$M = \frac{\sum_i m_i \Delta t_i}{\sum_i \Delta t_i}. \quad (2)$$

To ensure the interception of rapid fluctuations in memory levels in Equation (2), Δt_i was made as small as possible. In our measurements, Δt_i was between 250 to 350 microseconds, which was usually about a thousandth to a hundredth thousandth of a C code's execution time.

In general, the average dynamic program data sizes of `mat2c` C codes were smaller than that of `mcc` C codes; the relative reductions were over 20% in 7 out of 11 cases, being over 100% in over half of the cases. In the case of `capr`, though the `mcc` C code fared better by a small margin, it still has a far higher kcore-min value as discussed in § 4.5.2.1 below. (For `capr`, the average dynamic program data sizes for the `mcc` and `mat2c` C codes were 2428.02KB and 2506.75KB.)

4.5.2.1 KCore-Min Reductions.

An important point that the average figures don't uncover is the duration of consumption of a memory resource. If processes P_1 and P_2 both consume x kilobytes of memory, P_1 for t seconds and P_2 for $2t$ seconds, then both will have the same time-averaged memory consumption level but clearly P_2 will be the bigger memory hog. To take into consideration the effect of time, UNIX systems use a metric called the *kcore-min* value of a process that is computed as

$$\text{kcore-min} = M \times T$$

where M is the mean memory size in kilobytes and T is the duration of memory usage in minutes.

Thus, even though the average dynamic program data size of the `mat2c` C code was close to that of the `mcc` C code in `capr` and `nb3d`, the `mat2c` versions have lower kcore-min values because of their shorter execution times. The relative kcore-min reductions were 102.3% and 71.5% in these

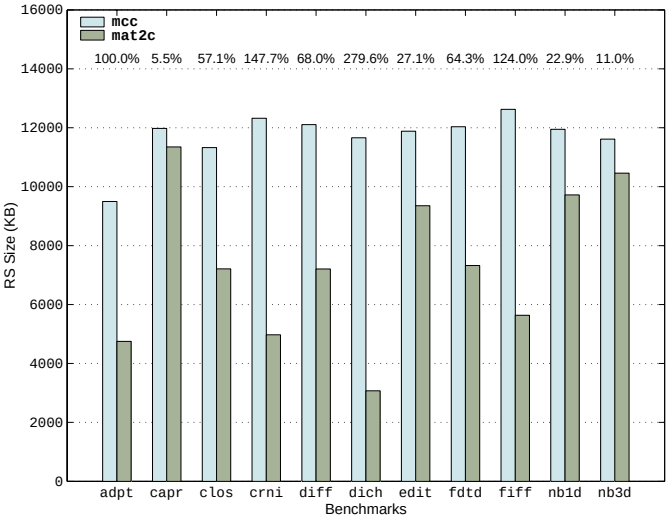


Figure 4: Average Resident Set Levels

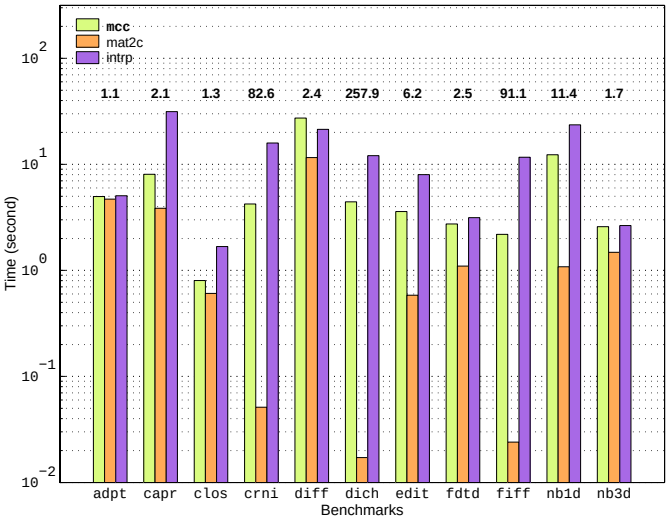


Figure 5: Comparative Execution Times

two cases and are shown using arrow markers in Figure 2. Though not shown in Figure 2, the `mat2c` C codes for all other benchmarks also exhibit kcore-min reductions because besides having lower size averages, they also usually have significantly lower execution times as shown in Figure 5.

4.5.3 *Overall Memory Levels*

To obtain the complete picture, the average virtual memory consumption levels of the `mat2c` and `mcc` C codes, which includes all swapped-out pages, mapped files and devices, is shown in Figure 3. The average resident set sizes (RSS) for all benchmarks are also shown, in Figure 4, which describe the amount of physical memory used by a process. The RSS numbers are significant, especially in an embedded setting, because non-resident pages don't task a RAM. Note that the sizes of the compiled images of the `mat2c` and `mcc` C codes also affect both these levels. However, it should be mentioned that the binary image size of a `mat2c` C code is nearly always larger than that of an `mcc` C code. The reason

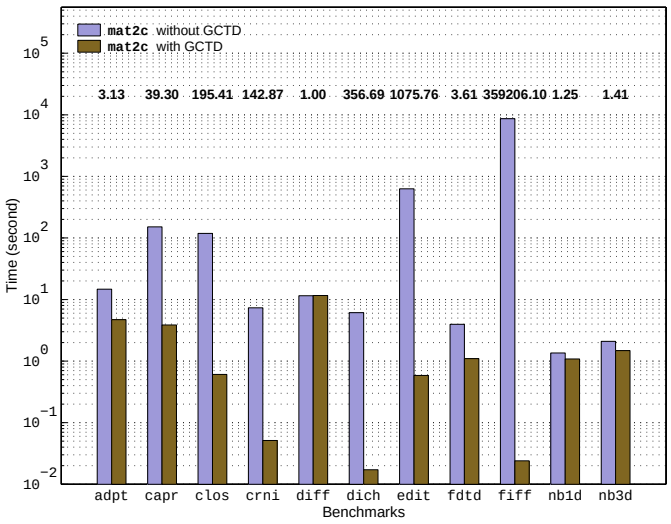


Figure 6: Effect of Coalescing on Execution Times

for this is that `mat2c` inlines out most of the language operations (this can be controlled by a pass-file option) whereas `mcc` always translates them to calls into a library. Hence, at the heart of the resource usage differences are the disparate approaches to compilation used by `mat2c` and `mcc`.

4.6 Execution Time Improvements

Figure 5 compares execution times of the `mat2c` and `mcc` C codes on a log scale. The figure also shows the times taken by the MATLAB interpreter to execute the benchmarks. Indicated above the bars are the performance speedups of the `mat2c` C codes over the `mcc` C codes. In only one case was the speedup marginal—10% for the `adpt` benchmark. The `adpt` benchmark has a parameter called `tol` that specifies the error tolerance in the quadrature values produced. Decreasing this value decreases the fractional overhead due to type checking, making numerical computation the dominant work in the benchmark. Measurements for `adpt` were taken with `tol` set at 10^{-12}, which is the setting in the FALCON benchmark suite. Because increasing `tol` increases the relative overhead due to type checking, the `mat2c` C code performs much better than the `mcc` C code at higher values of `tol`. In all other benchmarks, the speedups ranged from 30% and 74% in two cases, to over 100% in the remaining. In fact, in 4 out of 11 benchmarks, the speedups were dramatic, being over an order of magnitude.

4.6.1 *Impact of the GCTD Algorithm*

Figure 6 exhibits the influence of the GCTD pass on the execution times of the `mat2c` C codes. All other optimizations offered by the MAT2C system, like common subexpression elimination, constant folding, dead-code elimination and loop unrolling, were active in both cases. Numbers shown above the bars indicate the relative speedups with and without the GCTD pass. These relative speedups, in the context of the timings displayed in Figure 5, show that without it, the `mat2c` C codes would have performed poorly with respect to the `mcc` C codes in 8 out of 11 cases. This demonstrates the pivotal role that the GCTD pass plays in improving performance through the better static management of storage.

5. SUMMARY

This paper presented an algorithm for the efficient static management of storage in MATLAB through the coalescing of arrays. The algorithm consists of two phases the first of which uses the classic notion of interference to form classes of variables that don't compete for storage and the second of which decomposes those classes on the basis of program variable types and control flow. Unique aspects of the phases are the consideration of interference due to operator semantics, its resolution using types, and the use of symbolic type information in the decomposition of color classes. Rather than clumping together all arrays having symbolic shapes within a color class, which we have observed to be a poor storage management policy, the second phase uses a partial order to facilitate better spatial reuse characteristics. The algorithm is also applicable to other typeless array-based languages like IDL and APL that present similar issues.

The algorithm is also nonoptimal. The simplest example is an interference graph that consists of three nodes A, B and C representing variables with identical intrinsic types, whose corresponding storage sizes, assuming all are statically estimable, are say 4, 2 and 3 units respectively. If the only edge in the interference graph is between A and B, its chromatic number is 2. However, the aggregate of the coalesced storages will differ depending on which minimal coloring is actually used—if B and C share the same color, the aggregate will be 7 units, whereas if A and C share the same color, the aggregate will be 6 units. Thus, arriving at an optimal solution to the problem even in the simpler case of all array shapes being compile-time knowns would require an exploration of all possible colorings, a point that Fabri had also noted [15]. However, our experiments show that even with a conservative approach to the problem like ours, considerable savings in storage space and improvements in execution performance can be achieved.

6. REFERENCES

[1] P. S. Abrams. *An APL Machine*. Ph.D. dissertation, Stanford University, Feb. 1970. Available as Technical Report SLAC–114 from the Stanford Linear Accelerator Center.

[2] G. Almási and D. A. Padua. MAJIC: Compiling MATLAB for Speed and Responsiveness. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.

[3] P. Briggs. *Register Allocation via Graph Coloring*. Ph.D. dissertation, Rice University, Apr. 1992.

[4] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 275–284, July 1989.

[5] T. Budd. *An APL Compiler*. Springer-Verlag, Inc., New York City, NY 10010, USA, 1988.

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6(1):47–57, Jan. 1981.

[7] W.-M. Ching. Program analysis and code generation in an apl/370 compiler. *IBM Journal of Research and Development*, 30(6):594–602, Nov. 1986.

[8] F. Chow and J. Hennessy. Register Allocation by Priority-Based Coloring. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 222–232, June 1984.

[9] D. Cociorva, G. Baumgartner, C.-C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. E. Bernholdt, and R. Harrison. Space-Time Tradeoff Optimization for a Class of Electronic Structure Calculations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–186, June 2002.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1995.

[11] R. Cytron and J. Ferrante. What's in a Name? The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, Aug. 1987.

[12] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[13] L. A. De Rose and D. A. Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, Mar. 1999.

[14] A. P. Eršov. Reduction of the Problem of Memory Allocation in Programming to the Problem of Coloring the Vertices of Graphs. *Doklady Akademii Nauk SSSR*, 142:785–787, Jan. 1962. English translation in *Soviet Mathematics*, Vol., 3, No., 1, July 1962, pages 163–165.

[15] J. Fabri. Automatic Storage Optimization. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 83–91, Aug. 1979.

[16] L. J. Guibas and D. K. Wyatt. Compilation and Delayed Evaluation in APL. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–8, Jan. 1978.

[17] P. G. Joisha and P. Banerjee. MAGICA: A Software Tool for Inferring Types in MATLAB. Technical Report CPDC–TR–2002–10–004, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Oct. 2002.

[18] P. G. Joisha and P. Banerjee. Implementing an Array Shape Inference System for MATLAB Using Mathematica. Technical Report CPDC–TR–2002–10–003, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Oct. 2002.

[19] S. S. Lavrov. Store Economy in Closed Operator Schemes. *Zhurnal vychislitel'noi matematiki i matematicheskoi fiziki*, 1(4):687–701, 1961. English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics*, Vol., 1, No., 3, 1962, pages 810–828.

[20] A. Malishevsky. Implementing a Run-Time Library for

a Parallel MATLAB Compiler. M.S. report, Oregon State University, Apr. 1998.

[21] The MathWorks, Inc. *MATLAB: The Language of Technical Computing*, Jan. 1997. Using MATLAB (Version 5).

[22] P. Pineo and M. L. Soffa. A Practical Approach to the Symbolic Debugging of Parallelized Code. In *Proceedings of the 5th International Conference on Compiler Construction*, pages 339–356, Apr. 1994.

[23] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.

[24] Y. Zhang and R. Gupta. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 14–28, Apr. 2002.