Fast Copy Coalescing and Live-Range Identification

Zoran Budimlić, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves
Department of Computer Science
Rice University, Houston, Texas

ABSTRACT
This paper presents a fast new algorithm for modeling and reasoning about interferences for variables in a program without constructing an interference graph. It then describes how to use this information to minimize copy insertion for \( ϕ \)-node instantiation during the conversion of the static single assignment (SSA) form into the control-flow graph (CFG), effectively yielding a new, very fast copy coalescing and live-range identification algorithm.

This paper proves some properties of the SSA form that enable construction of data structures to compute interference information for variables that are considered for folding. The asymptotic complexity of our SSA-to-CFG conversion algorithm is \( O(nα(n)) \), where \( n \) is the number of instructions in the program.

Performing copy folding during the SSA-to-CFG conversion eliminates the need for a separate coalescing phase while simplifying the intermediate code. This may make graph-coloring register allocation more practical in just in time (JIT) and other time-critical compilers. For example, Sun’s Hotspot Server Compiler already employs a graph-coloring register allocator[10].

This paper also presents an improvement to the classical interference-graph-based coalescing optimization that shows a decrease in memory usage of up to three orders of magnitude and a decrease of a factor of two in compilation time, while providing the exact same results.

We present experimental results that demonstrate that our algorithm is almost as precise (within one percent on average) as the original code. In our experience, the number of copies used to instantiate \( ϕ \)-nodes is much higher than the number of copies in the original code.

We present experimental data that shows that, on average, our implementation of this algorithm rivals the effectiveness of the interference-graph based coalescer in removing dynamic copies while using only a fraction of the compilation time required by that algorithm.

1. INTRODUCTION

It has long been known that copies can be folded during the construction of the SSA form [2]. Essentially, each variable that is defined by a copy is replaced in subsequent operations by the source of that copy. The implementation of this strategy has a couple of subtle problems but is otherwise an effective optimization. In effect, copy folding during SSA construction deletes all of the copies in a program, except for those that must be used to instantiate \( ϕ \)-nodes.

This simple observation may reduce the number of copies in a program, but naive \( ϕ \)-node instantiation introduces many unnecessary copies. In our experience, the number of copies used to instantiate \( ϕ \)-nodes is much higher than the number of copies in the original code.

Classically, to solve this problem one would coalesce SSA names into a single name using an interference graph, as described by Chaitin [5, 4]. The interference graph models names as nodes, and edges between the nodes represent an interference — there is at least one point in the code where the two name are simultaneously live. The intuition for register allocation is that if two variables interfere, we will have to use a different register for each variable. The same holds true for copy coalescing: if two names joined by a copy interfere, the copy is necessary and cannot be coalesced.

Unfortunately, this conceptually simple method requires a data structure that is quadratic in the number of names. As Cooper, et al. showed [6], the cost of building an interference graph is significant, and for JIT compilers or other systems where compile time is crucial, this cost may be prohibitive.

This paper presents a new algorithm that performs copy coalescing without building an interference graph. We model interferences using a combination of liveness and dominance information. We present a theoretic background that shows that our optimization is safe. We present experimental data that shows that, on average, our implementation of this algorithm rivals the effectiveness of the interference-graph coalescer in removing dynamic copies while using only a fraction of the compilation time required by that algorithm.

2. SSA PROPERTIES

This section presents a brief description of the SSA properties that we use to construct our algorithm. A full formal description, along with the proofs of all the lemmas and theorems is presented elsewhere [3].

The preliminary step in our algorithm is to ensure that the code is in regular form, which we shall define with the following construction. We start with a standard CFG, \((N, E, b_0)\), made up of a set of \( N \) nodes, \( E \) edges, and a unique start node, \( b_0 \). We impose an added restriction on the code in the incoming CFG, that it be strict.
DEFINITION 2.1. A strict program is one in which, for every variable, \( v \), and for every use of \( v \), every possible path from \( b_0 \) to the use of \( v \) passes through one of the definitions of \( v \).

Strictness is a requirement for languages such as Java, but it can be imposed on languages like C or Fortran as well by initializing every variable at the start of \( b_0 \). The initializations that are unnecessary can then be removed by a dead-code elimination pass, or, alternatively, we can restrict the initializations to only those variables that are in the live-in set of \( b_0 \). Since the live-in set of \( b_0 \) should be empty, variables that make it into this set are those for which there is an upwards-exposed use of the variable, and thus some path from the use backwards up the CFG on which there is no definition of the variable.

Given a strict program, we then convert the code into SSA form. It is a natural result of the SSA algorithm that, after the conversion, not only is every use of a variable dominated by a single definition, but also every definition dominates all of its uses.\(^1\) We call this a regular program.

Given a regular program, we can now define interference:

DEFINITION 2.2. Two variables interfere if and only if they are simultaneously live at some point in a regular program.

With Definition 2.2, we can prove the following theorem:

THEOREM 2.1. If two variables in a regular program interfere, the definition point of one will dominate the definition point of the other.

Informally, the proof for this theorem results from the fact that both variable definitions must dominate the point in the program where they interfere.

We can refine Theorem 2.1 to suit our purposes with the following theorem, which results directly from the computation of live-

THEOREM 2.2. If two variables, \( v_1 \) and \( v_2 \), in a regular program interfere and the definition point of \( v_1 \) dominates the definition point of \( v_2 \), then either \( v_1 \) is in the live-in set of the block in which \( v_2 \) is defined, or both variables are defined in the same block.

Thus, we can quickly check interference between two variables simply by examining the liveness information associated with the blocks in which the variables are defined.

3. COALESCING ALGORITHM

With the code in regular form, we can proceed with the algorithm. Essentially, we use union-find to group together all of the names joined at \( \phi \)-nodes. We then use the previous reasoning about interference to break the union-find sets apart when we discover two members of the same set interfere. To break a set, we insert copies between the member that we want to remove and all of the other members of the set. At the end, all SSA names that are in the same set do not interfere, and, therefore, can share the same name. Note that we build pruned SSA \([8]\) to make the reasoning simpler and because parts of the analysis necessary for pruned SSA, such as liveness analysis, are assumed. The algorithm we present should work for minimal or semi-pruned SSA as well, although the additional inexactness of those forms propagates itself into our analysis, possibly causing the insertion of extra copies that may not otherwise have been added.

If we had not allowed copy folding during the construction of SSA form, the initial union-find sets would contain only values that do not interfere. Indeed, this is precisely the algorithm used by a Chaitin/Briggs register allocator \([5, 4, 1]\) to identify live ranges. The allocator joins all \( \phi \)-node names into a single set and then builds the aforementioned interference graph. It then coalesces live ranges joined at a copy but that do not otherwise interfere. This is a classic pessimistic algorithm: all copies are assumed to be necessary until proven otherwise.

In contrast, the algorithm presented here is optimistic: we assume that every copy is unnecessary, and then we insert those that we cannot prove are unnecessary. The register allocator does not perform copy folding during SSA construction because some of those copies really are necessary — they move a value from one storage location to another, usually to free up the first location for another value. When we fold copies during SSA construction, we, in effect, transfer the information about the exchange into a \( \phi \)-node, where we may later recover the move by inserting a new copy.

The algorithm has four steps. First, we union together all \( \phi \)-node parameters (and the name of the \( \phi \)-node itself). This gives us an initial guess at what are, essentially, live-range names. The second step of the algorithm compares the set members against each other, looking for interferences, which, if found, cause one of the members to be split into a new set, necessitating insertion of copies from that member to members of the first. Then a unique name is given to each set and, finally, the code is rewritten with all necessary copies.

Notice that copies are not actually inserted until the final step, for reasons described in Section 3.6. Instead, we maintain an array, \( Waiting \), indexed by block name, where each entry is a list of pending copy insertions for the end of that block. When one of the earlier stages discovers a copy that needs to be inserted into some block, \( b \), we add the copy to \( Waiting[b] \). Also, because \( \phi \)-node parameters flow along edges, we use the notation \( From(x) \) to specify the block out of which the value in \( x \) flows.

3.1 Building Initial Live Ranges

The first step to coalescing copies and building live ranges is to union together the \( \phi \)-node parameters. As we explained earlier, some of these unions will include names of variables that interfere, and one of each of these pairs will have to be removed from the set.

While we were developing the implementation of our algorithm, we found that some filtering during the building of the unions could save time and give us fewer copies. While unifying names, we detect interferences between just two names (the \( \phi \)-node and the current parameter). Specifically, we use liveness information to detect that the SSA construction erroneously folded a copy. The folding was in error because both variables are live at some point in the code. In general, only a single copy is needed to break the interference. On the other hand, if we wait until later, each of those names may interfere with many of the other names in the union, necessitating many more copies to ensure breaking the interference. Thus, given a \( \phi \)-node, \( p \), defined in block \( b \), with parameters \( a_1 \) through \( a_n \), we apply the following simple tests of interference. These five are not exhaustive, but they handle the simple cases — any interference found will cause a copy to be inserted; otherwise, \( a_i \) is added to the union.

- If \( a_i \) is in the live-in set of \( b \), add a copy from \( a_i \) to \( p \) in \( Waiting[From(a_i)] \). Note that our liveness analysis distinguishes between values that flow into \( b \)'s \( \phi \)-nodes and values that flow directly to some other use in \( b \) or \( b \)'s successors. Only in the latter case will \( a_i \) be in \( b \)'s live-in set.
- If \( p \) is in the live-out set of \( a_i \)'s defining block, add a copy to \( Waiting[From(a_i)] \).
- If \( a_i \) is defined by a \( \phi \)-node and \( p \) is in the live-in set of the block defining \( a_i \), add a copy to \( Waiting[From(a_i)] \).
 inputs: \( IN \) (dominator tree \( DT \))
outputs: \( OUT \) (dominance forest \( DF \))

for depth-first order over dominator tree nodes \( b \)

\[ \text{preorder}(b) = \text{the next preorder name} \]
\[ \text{maxpreorder}(b) = \text{the largest preorder number} \]

of \( b \)'s descendants

Take \( S \) in dominator order

\[ \text{maxpreorder}(	ext{VirtualRoot}) = \text{MAX} \]
\[ \text{CurrentParent} = \text{VirtualRoot} \]

stack.Push(VirtualRoot)

for all the variables \( v \) in \( S \) in sorted order

\[ \text{while} \quad \text{preorder}(v) > \text{maxpreorder}(\text{CurrentParent}) \]

stack.Pop()

\[ \text{CurrentParent} = \text{stack.Top()} \]

make \( v \) a child of the \( \text{CurrentParent} \)
stack.Push(\( v \))

\[ \text{CurrentParent} = v \]

Remove \( \text{VirtualRoot} \) from \( DF \)

Figure 1: Constructing the dominance-forest

- If \( a_i \) has already been added to another set of names because of another \( \phi \)-node in the current block, add a copy to \( \text{Waiting}[\text{From}(a_i)] \).
- If \( a_i \) and \( a_j \) are defined in the same block, then add a copy to either \( \text{Waiting}[\text{From}(a_i)] \) or \( \text{Waiting}[\text{From}(a_j)] \).

3.2 \ The Dominance Forest

At the end of the first step, we have disjoint sets of names that may share the same name. We now need to discover members of the set that interfere with other members of the same set.

A critical part of our algorithm is the engineering required to perform the second step efficiently. It would be prohibitively expensive to do a pairwise comparison of all the members in the set. Instead, we have developed a new data structure, called a dominance forest, which allows us to perform a linear comparison of set members by ordering them according to dominance information.

**Definition 3.1.** Let \( S \) be a set of SSA variables in a regular SSA program such that no two variables in \( S \) are defined in the same block. Let > be a strict dominance relation. Let \( v_i \) be a variable in \( S \), and \( B_j \) be the block in which \( v_i \) is defined. Dominance forest \( DF(S) \) is a graph in which the nodes are the blocks \( B_j \) such that \( v_i \in S \), and there is an edge from \( B_i \) to \( B_j \) if and only if \( B_i > B_j \), and \( \exists \ (v_k \in S) \), \( i \neq k \neq j \), such that \( B_i > B_k > B_j \).

Sucinctly, the dominance forest is a mapping of SSA variables to the basic blocks that contain their definition points, with the edges representing collapsed dominator-path trees. We use Lemma 3.1 to show that we need only check edges in the dominance forest for interferences. We will show in the next section how to use the dominance forest.

Figure 1 shows the pseudo code for dominance-forest construction. This algorithm starts by adding a \( \text{VirtualRoot} \) to the result to simplify the construction process. (We remove the \( \text{VirtualRoot} \) at the end, which may create a forest.) In a depth-first traversal of the dominator tree, we label all nodes in the tree with their preorder sequence of traversal. On the way up in the traversal, we also compute the maximum preorder number of the descendants for each node. This number allows the algorithm to identify the antecedent-descendent information from the dominator tree in constant time and is due to Tarjan[11]. This preorder-numbering process is done only once for the whole SSA.

The algorithm iterates over the variables in order of increasing preorder number, or \( pn \). Within the loop, a variable named \( \text{CurrentParent} \) is maintained, which holds the reference to the root of the subtree currently in construction. There is an edge between \( \text{CurrentParent} \) and the current variable \( v \) if \( pn(v) \) is less than or equal to the the largest preorder number of \( \text{CurrentParent} \), which means that \( \text{CurrentParent} \) dominates \( v \). Traversing the variables in increasing order of \( pn(v) \) ensures that no edges are inserted prematurely (if \( a \) dominates \( b \), which dominates \( c \), the relationship will be: \( pn(a) < pn(b) < pn(c) \), which will ensure that only the edges \((a, b)\) and \((b, c)\) are inserted and not the edge \((a, c)\)).

3.3 \ Walk the Forests

We built the dominance forests to reduce the number of interferences for which we have to check. We can do this using the following Lemma:

**Lemma 3.1.** Given variable \( v_i \) defined in block \( b_i \) and variable \( v_j \) defined in block \( b_j \), if \( b_j \) is a child of \( b_i \) in the dominance forest and \( v_i \) does not interfere with \( v_j \), then \( v_i \) cannot interfere with any of the variables represented by \( b_j \)'s descendants in the dominance forest.

Informally, the proof of this Lemma comes from the fact that if \( v_i \) interferes with any of the \( v_j \)'s descendants — for example \( v_k \), then \( v_i \) must interfere with \( v_j \), since it must be live at the definition point of \( v_k \), thus it must be live at the definition of \( v_j \).

**Lemma 3.1.** In essence, prunes the pairwise search space of names in a set. That is, it shows that when we map the names of some set onto its dominance forest, each variable, \( v \), needs to be compared only to those variables defined in blocks immediately descendant of the block defining \( v \).

Figure 2 contains the algorithm for finding and resolving interferences within a dominance forest. The dominance forest is traversed depth first. If a variable, \( v \), interferes with any other variable in the dominance forest, \( v \) must interfere with one of its children. Because variables are checked for interference only against their children in the dominance forest, any parent variable that interferes with its child forces us to insert all of the copies necessary to break any interferences between the two. This has the effect of separating the variable from any of its more distant descendants it might interfere with.

3.4 \ Local Interferences

Up to this point, only liveness information at block boundaries has been considered. However, there are situations where two variables do not interfere at any block boundaries but are nonetheless live in the same block. This happens when one of the variables, \( v_1 \), is live coming into the block where the second variable, \( v_2 \), is defined. In this case, we need to ensure that \( v_1 \)'s last use occurs before \( v_2 \) is defined, which requires a walk through the block to determine. Our algorithm keeps a list of variable pairs that need to be checked for local interference. After traversing all of the dominance forests and before inserting actual copies, our algorithm traverses each block backwards to find and break any of these local interferences.

3.5 \ Renaming Variables

Once copies have been inserted, those variables that are still connected by \( \phi \)-nodes need to be given a single unique name. Naive iteration over \( \phi \)-nodes and renaming the parameters to a single name is not sufficient. If a variable were a parameter to multiple \( \phi \)-nodes, the variable would be renamed multiple times, resulting in incorrect code.

The solution is to iterate over all those variables that are candidates for renaming (those that were a part of one of the union-find sets and are not on the list of variables to remove from the
for depth-first traversal of $DF$,
   if variable $p$ is $c$'s parent and is in the live-out set of $c$'s defining block
      if $p$ can not interfere with any of its other children and $c$ has fewer copies to insert than $p$
         Insert copies for $c$ and make $c$'s children $p$'s children
   else insert copies for $p$
 else if parent $p$ is in the live-in set of $c$'s defining block or $p$ and $c$ have the same defining block
         Add the variable pair $(p, c)$ to the list to check for local interference later

Figure 2: Finding and resolving interferences

\begin{figure}[h]
\begin{center}
\includegraphics[width=\textwidth]{figure2}
\end{center}
\caption{a. Original Code \hspace{1cm} b. SSA Form With Copies Folded \hspace{1cm} c. \phi-nodes Replaced With Copies}
\end{figure}

Figure 3: The “virtual swap” problem

\begin{figure}[h]
\begin{center}
\includegraphics[width=\textwidth]{figure3}
\end{center}
\caption{a. Inserted Copies for $a_1$ \hspace{1cm} b. $x_2$ Folded Into $b_1$ \hspace{1cm} c. Inserted Copy For $y_2$}
\end{figure}

Figure 4: Inserting copies for the “virtual swap” problem

\begin{figure}[h]
\begin{center}
\includegraphics[width=\textwidth]{figure4}
\end{center}
\caption{a. Inserted Copies for $a_1$ \hspace{1cm} b. $x_2$ Folded Into $b_1$ \hspace{1cm} c. Inserted Copy For $y_2$}
\end{figure}
set by inserting copies) and rename them to a single unique name. The Find function will return a single unique name per set, and that name can be used for renaming all members of that set.

### 3.6 Correct Copy Insertion

We have to be careful when inserting copies for φ-nodes, because naive copy insertion may produce incorrect code. Two such problems, the “lost copy” problem, and the “swap” problem, are described in detail by Briggs, *et al.* [2]. We avoid the “lost copy” problem by splitting critical edges after we have read in the code. The “swap” problem is addressed as in Briggs, *et al.* by careful ordering of copies with temporaries inserted to break any cycles. This is precisely the reason that we use the Waiting array to store pending copies; as Briggs, *et al.* show, the ordering of the full set of copies to be inserted is critical to correctness.

The “virtual swap” problem is a case similar to the “swap” problem, but has to be addressed with special attention when attempting to produce the minimal number of copies. In the “virtual swap” problem, two variables are defined by copies on either side of a conditional, and they take opposite values on opposing sides.

#### 3.6.1 The “Virtual Swap” Problem

Figure 4 illustrates an example where the naive insertion of copies still produces correct code. However, a careful implementation is needed when the copies are inserted into the SSA in our algorithm. The naive algorithm inserts all copies for the φ-nodes, while our algorithm attempts to insert as few copies as possible. When analyzing the φ-nodes, the algorithm determines that the variables α1 and b1 are simultaneously live at the end of the first block and cannot be folded together. The algorithm then picks one of them and inserts copies for it. On the left of the Figure 4 we show the example from Figure 3 with α1 being picked for copy insertion.

After the copies have been inserted, the last pass of the algorithm scans through the SSA and renames the variables as needed. This is the point where some additional interferences can be identified and some additional copies needed. In Figure 4b, all appearances of x2 have been replaced with b1. This exposes an interference between the first and the second φ-nodes, which forces insertion of a copy on Figure 4c.

### 3.7 Algorithmic Complexity

The dominance-forest construction algorithm is linear in the size of the join set. It starts with a depth-first traversal of the dominator tree, which is linear in size of the dominator tree, but it is done only once for the whole SSA. It then uses the radix sort [7] to sort the variables in the set, which is linear as well since the number of variables in the join set cannot be greater than the number of basic blocks in the CFG. Each of the variables in the set is visited exactly once in the loop. So the complexity of the dominance forest construction algorithm is $O(|S|)$.

The copy insertion algorithm begins with constructing join sets for variables in the φ-nodes of the graph. This can be done in $O(n \alpha(n))$ time using the union-find algorithm [7], where $n$ is the number of variables in φ-nodes, and $\alpha$ is the inverse Ackermann’s function. The algorithm then constructs the dominance forests for these sets (which are disjoint), which is linear in the total number of variables in φ-nodes. For each dominance forest, the algorithm visits all the edges, which is linear in the number of nodes in the forest. At the end, all φ-nodes are visited and the variables that are the arguments of the φ-nodes are renamed into a single variable name. The total complexity of this algorithm is $O(n \alpha(n))$, where $n$ is the total number of arguments in all the φ-nodes in the SSA.

Since the inverse of Ackermann’s function is practically a constant, one cannot hope to achieve better algorithmic complexity than what is presented here, since all the φ-nodes and all of their arguments have to be visited at least once in the SSA-to-CFG conversion.

### 4. EXPERIMENTS

In this section, we present experimental evidence to show that this algorithm is both effective and efficient. We compare our algorithm against two versions of the interference-graph coalescer. One is simply a coalescing phase stripped from our implementation of a Chaitin/Briggs register allocator. The second is an improved version, made possible by an insight into building the interference graph, that is significantly faster than the original, but equally precise.

The test suite we used in our experiments is made up of 169 routines that come from Forsythe, *et al.*’s book on numerical methods[9], as well as the Spec and Spec ’95 libraries. We ran these codes on a minimally loaded 300 MHz Sparc Ultra 10 with 256 megabytes of memory. Due to space constraints, we only report on the ten largest results in each experiment. We took the ten programs that took longest to compile using the standard SSA-to-CFG conversion for Table 2 and Table 3 and the ten programs with the most dynamic copies for Table 4 and Table 5. However, we think that these give a reasonable insight into the behavior of the algorithms. We use the following nomenclature to distinguish the algorithms:

- **Briggs** – the Chaitin/Briggs interference-graph coalescer
- **Briggs*** – the improved interference-graph coalescer, below
- **Standard** – the Briggs, *et al.* φ-node-instantiation algorithm that does not attempt to eliminate any copies
- **New** – the algorithm presented herein.

#### 4.1 Engineering the Interference-Graph Coalescer

During the development of the experiments to test this new algorithm, we discovered a simple oversight in building the interference graph as described by Briggs in his dissertation. Briggs’ algorithm requires four steps. First, the code is converted to SSA form. Names are then joined to form live ranges by unioning φ-node parameters. The third step is a loop in which the interference graph is built and then copies whose source and destination do not interfere have those live ranges coalesced. Because the interference graph is not exact, it needs to be rebuilt after all of the coalescible copies have been identified and those live ranges have been unioned together. This can expose additional opportunities for coalescing, so these two steps are iterated until all opportunities have been found. For these experiments, the final step is to rewrite the code to reflect the namespace described by the live ranges.

This algorithm is simple and powerful, but it is very expensive. The interference graph is modeled as a triangular bit matrix, with as many rows or columns as there are live-range names. This data structure requires $n^2/2$ bits that have to be cleared. As Cooper, *et al.* showed [6], this is a considerable part of the overall running time of a graph-coloring register allocator.

The flaw in this algorithm is that it builds an interference graph that includes the full set of live-range names. The reason for this is that if the coalescing phase does not fold any copies, the interference graph is correct, and the remaining phases of the allocator then use it. However, if the coalescing phase unions any live ranges,
the interference graph has to be rebuilt, and in our experiments, we rarely saw input that did not require at least two iterations (and few that required more). While the build/coalesce loop, as Briggs calls it, is iterating, the interference graph should only be built using live-range names that are involved in copies. To maintain a compact namespace, this requires an extra mapping array, but the cost of accessing the array each time a name is examined is offset by the considerable decrease in the size of the interference-graph bit matrix. Table 1 shows the difference in memory used to build the interference graph. Clearly, the savings in memory usage is immense, and the time required to perform coalescing is, on average, less than half the time with our improved algorithm.

To our knowledge, this is the first publication of this insight. Certainly, comparison of our SSA-coalescing algorithm has to be made against the best known coalescer (which, in this case, is the Briggs’ algorithm improved by using the insights described above), which these experiments do.

### 4.2 Running Time

Table 2 shows the running time of the three algorithms, the original SSA $\phi$-node replacement algorithm presented in Briggs, et al., the algorithm presented in this paper, and the improved coalescer from our register allocator. The timer was started immediately before building SSA form, and its value is recorded immediately after the code is rewritten in its various forms by the different algorithms.

Clearly, the additional analysis to restrict the number of copies is more expensive than the universal copy-insertion algorithm. However, our algorithm is considerably faster than the interference-graph coalescer.

Table 3 shows the maximum amount of memory used by the three algorithms during compilation. Over the full test suite, our algorithm uses, on average, 40% more memory than the standard $\phi$-node-replacement algorithm. It uses only 21% more memory than the improved interference-graph coalescer, which itself uses substantially less memory than the previous state of the art. This table and Table 2 show that memory usage alone is not the only determinant of the compiler’s total running time.

### 4.3 Efficacy Measurements

In Table 4, we show the number of copy operations that were executed. Our algorithm produces code that executes about 1% fewer copies, on average, than the interference-graph coalescer. The interference-graph coalescer tries to remove copies out of innermost loops first, on the theory that these are the most profitable to remove. This heuristic sometimes fails, as in the case of intX, but it also sometimes wins; on some of the codes in the test suite, our algorithm results in code that executes up to two-thirds again as many copy operations as the code produced by the interference-graph coalescer.

In Table 5, we show a static measurement of copies left in the code by the three algorithms. On average, our algorithm leaves in approximately three percent more static copies than the interference-graph coalescer, but as with dynamic copies, the results vary significantly. Again, we believe this reflects the heuristic nature of both algorithms.

### 5. CONCLUSIONS AND FUTURE WORK

In this paper, we give a theoretical background that enables fast computation of interference information and present a practical, efficient algorithm to perform copy folding without building an interference graph. The applications of this algorithm are many. It can be used as a standalone pass of an optimizer. It can replace the current copy-insertion phase of an optimizer’s SSA implementation. Finally, it can replace the coalescing phase of a Chaitin/Briggs register allocator.

Our experiments show that this algorithm provides significant improvements in running time over an interference-graph-based algorithm, while maintaining comparable precision. These results make this algorithm very attractive for use in any system, including, perhaps, systems in which compile time is a critical concern, such as JIT compilers.

We have also presented an implementation insight concerning building an interference graph for copy coalescing that can be trivially inserted into existing graph-coloring register allocators to make them run much faster.

Our plan for future research includes design and implementation of a fast register-allocation algorithm that uses the results presented in this paper. We will also consider implementation of several heuristics to improve the precision of this algorithm without sacrificing the compilation time.

### 6. ACKNOWLEDGMENTS

The authors would like to thank the many people who supported this work, including members of the MSCP, especially Linda Torczon, as well as John Mellor-Crummey, both of whom provided many useful suggestions. The authors would like to thank Preston Briggs for his contributions. We are also indebted to Compaq, Los Alamos Computer Science Institute, and the GrADS NSF project (grant #9975020) for their support of this work.
### Table 2: Comparison of compilation times (in seconds)

<table>
<thead>
<tr>
<th>File</th>
<th>Standard</th>
<th>New</th>
<th>Briggs*</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>parmvrX</td>
<td>0.17</td>
<td>0.38</td>
<td>1.26</td>
<td>2.24</td>
</tr>
<tr>
<td>fieldX</td>
<td>0.17</td>
<td>0.48</td>
<td>1.55</td>
<td>2.82</td>
</tr>
<tr>
<td>parmovX</td>
<td>0.15</td>
<td>0.30</td>
<td>1.11</td>
<td>2.00</td>
</tr>
<tr>
<td>radfX</td>
<td>0.10</td>
<td>0.24</td>
<td>0.73</td>
<td>2.40</td>
</tr>
<tr>
<td>radbgX</td>
<td>0.10</td>
<td>0.22</td>
<td>0.67</td>
<td>2.20</td>
</tr>
<tr>
<td>twl drv</td>
<td>0.07</td>
<td>0.16</td>
<td>0.49</td>
<td>2.29</td>
</tr>
<tr>
<td>parmveX</td>
<td>0.06</td>
<td>0.13</td>
<td>0.46</td>
<td>2.17</td>
</tr>
<tr>
<td>initX</td>
<td>0.06</td>
<td>0.11</td>
<td>0.26</td>
<td>1.83</td>
</tr>
<tr>
<td>advbndX</td>
<td>0.04</td>
<td>0.07</td>
<td>0.15</td>
<td>1.75</td>
</tr>
<tr>
<td>dese co</td>
<td>0.03</td>
<td>0.07</td>
<td>0.20</td>
<td>2.33</td>
</tr>
<tr>
<td><strong>AVERAGE</strong></td>
<td>0.10</td>
<td>0.22</td>
<td>0.69</td>
<td>2.20</td>
</tr>
</tbody>
</table>

### Table 3: Comparison of compiler memory usage (in bytes)

<table>
<thead>
<tr>
<th>File</th>
<th>Standard</th>
<th>New</th>
<th>Briggs*</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>parmvrX</td>
<td>4244880</td>
<td>5363964</td>
<td>5126100</td>
<td>1.27</td>
</tr>
<tr>
<td>fieldX</td>
<td>3144244</td>
<td>5062808</td>
<td>3513180</td>
<td>1.61</td>
</tr>
<tr>
<td>parmovX</td>
<td>3740196</td>
<td>4785972</td>
<td>4562856</td>
<td>1.28</td>
</tr>
<tr>
<td>radfX</td>
<td>1594768</td>
<td>2379156</td>
<td>1785964</td>
<td>1.49</td>
</tr>
<tr>
<td>radbgX</td>
<td>1492484</td>
<td>2250356</td>
<td>1695244</td>
<td>1.51</td>
</tr>
<tr>
<td>twl drv</td>
<td>1496208</td>
<td>1805984</td>
<td>1613080</td>
<td>1.21</td>
</tr>
<tr>
<td>parmveX</td>
<td>1734400</td>
<td>2204940</td>
<td>2081044</td>
<td>1.27</td>
</tr>
<tr>
<td>initX</td>
<td>1456780</td>
<td>1814224</td>
<td>1680640</td>
<td>1.25</td>
</tr>
<tr>
<td>advbndX</td>
<td>1009844</td>
<td>1253616</td>
<td>1130072</td>
<td>1.24</td>
</tr>
<tr>
<td>dese co</td>
<td>614496</td>
<td>954356</td>
<td>718856</td>
<td>1.55</td>
</tr>
<tr>
<td><strong>AVERAGE</strong></td>
<td>2049830</td>
<td>2787738</td>
<td>2390704</td>
<td>1.36</td>
</tr>
</tbody>
</table>

### Table 4: Comparison of dynamic copies executed

<table>
<thead>
<tr>
<th>File</th>
<th>Standard</th>
<th>New</th>
<th>Briggs*</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>tomcatv</td>
<td>23572565</td>
<td>64571</td>
<td>64571</td>
<td>0.00</td>
</tr>
<tr>
<td>blts</td>
<td>17222000</td>
<td>3716100</td>
<td>3666100</td>
<td>0.22</td>
</tr>
<tr>
<td>buts</td>
<td>17111000</td>
<td>4055500</td>
<td>4050500</td>
<td>0.24</td>
</tr>
<tr>
<td>getbX</td>
<td>8197640</td>
<td>63520</td>
<td>63520</td>
<td>0.01</td>
</tr>
<tr>
<td>twl drv</td>
<td>5244414</td>
<td>215761</td>
<td>217777</td>
<td>0.04</td>
</tr>
<tr>
<td>smoothX</td>
<td>4213120</td>
<td>599920</td>
<td>467440</td>
<td>0.14</td>
</tr>
<tr>
<td>rhs</td>
<td>3509310</td>
<td>414324</td>
<td>414324</td>
<td>0.12</td>
</tr>
<tr>
<td>parmvrX</td>
<td>2218540</td>
<td>43610</td>
<td>42630</td>
<td>0.02</td>
</tr>
<tr>
<td>saxpy</td>
<td>2000000</td>
<td>20000</td>
<td>20000</td>
<td>0.01</td>
</tr>
<tr>
<td>initX</td>
<td>1755598</td>
<td>92810</td>
<td>390455</td>
<td>0.05</td>
</tr>
<tr>
<td><strong>AVERAGE</strong></td>
<td>7</td>
<td>118</td>
<td>118</td>
<td>0.23</td>
</tr>
</tbody>
</table>

### Table 5: Comparison of static number of copies

<table>
<thead>
<tr>
<th>File</th>
<th>Standard</th>
<th>New</th>
<th>Briggs*</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>tomcatv</td>
<td>112</td>
<td>47</td>
<td>47</td>
<td>0.42</td>
</tr>
<tr>
<td>blts</td>
<td>82</td>
<td>36</td>
<td>31</td>
<td>0.44</td>
</tr>
<tr>
<td>buts</td>
<td>72</td>
<td>33</td>
<td>31</td>
<td>0.46</td>
</tr>
<tr>
<td>getbX</td>
<td>176</td>
<td>28</td>
<td>28</td>
<td>0.16</td>
</tr>
<tr>
<td>twl drv</td>
<td>976</td>
<td>164</td>
<td>169</td>
<td>0.17</td>
</tr>
<tr>
<td>smoothX</td>
<td>668</td>
<td>283</td>
<td>215</td>
<td>0.42</td>
</tr>
<tr>
<td>rhs</td>
<td>678</td>
<td>32</td>
<td>27</td>
<td>0.05</td>
</tr>
<tr>
<td>parmvrX</td>
<td>352</td>
<td>101</td>
<td>101</td>
<td>0.29</td>
</tr>
<tr>
<td>saxpy</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>initX</td>
<td>458</td>
<td>106</td>
<td>118</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Table 5: Comparison of static number of copies
7. REFERENCES


