Tailoring Graph-coloring Register Allocation For Runtime Compilation

Keith D. Cooper and Anshuman Dasgupta
Rice University
Houston, TX., USA
{*keith, anshuman*}*@cs.rice.edu*

Abstract

Just-in-time compilers are invoked during application execution and therefore need to ensure fast compilation times. Consequently, runtime compiler designers are averse to implementing compile-time intensive optimization algorithms. Instead, they tend to select faster but less effective transformations. In this paper, we explore this trade-off for an important optimization – global register allocation. We present a graph-coloring register allocator that has been redesigned for runtime compilation. Compared to Chaitin-Briggs [7], a standard graph-coloring technique, the reformulated algorithm requires considerably less allocation time and produces allocations that are only marginally worse than those of Chaitin-Briggs. Our experimental results indicate that the allocator performs better than the linear-scan and Chaitin-Briggs allocators on most benchmarks in a runtime compilation environment. By increasing allocation efficiency and preserving optimization quality, the presented algorithm increases the suitability and profitability of a graph-coloring register allocation strategy for a runtime compiler.

1 Introduction

In recent years, platform-independent program representations have experienced a drastic growth in popularity. The widely used Java and .Net platforms encode programs in portable representations that are compiled or interpreted at runtime. Running programs on these frameworks is a two-step process: the source-code is first compiled to a machine-independent representation called *bytecode*.[1] The bytecode is then transported to the target processor where it is interpreted by a *virtual machine* (VM). While such portable representations allow programs to be executed on a wide variety of environments, the run-anywhere feature of the code comes with a cost: the interpretation of portable code is generally slower than the execution of a natively compiled version of the program that has been specialized for the target architecture. Two major factors contribute to the relative inefficiency of portable code: the overhead of interpretation, and the genericity of the encoding that, by its very nature, prohibits important machine-specific optimizations. To alleviate these problems, many virtual machines invoke a runtime compiler, also popularly known as a *just-in-time* compiler (JIT).

The JIT translates the bytecode to architecture-specific machine code and, in the process, attempts to optimize the resulting assembly code. A JIT shares many goals with traditional, offline compilers – it strives to tailor its translation to emit the most efficient machine code. However, in contrast to its offline counterpart, a JIT must be acutely cognizant of compilation time. Since the compiler is invoked at runtime, it must optimize for the sum of compile time and runtime. This constraint requires the JIT to strike a fine balance between conducting strong optimizations that tend to be computationally expensive and faster, but less-effective techniques. The decisions taken in choosing these optimizations have a profound impact on program execution efficiency. Our work will examine and address this critically important issue.

1.1 Balancing optimization quality and compilation time

The literature on compilers includes descriptions of numerous optimizations that focus on improving the quality of compiled code. Many of these transformations rely on intricate mathematical analyses. The resulting optimizations can be computationally intensive and, consequently, expensive to conduct. In an offline compiler, compilation time has no direct impact on application efficiency, so a reasonable increase in compile time can be tolerated for an optimization that improves the code. In a JIT environment, where

[1] So named because each opcode occupies a single byte. The term bytecode dates back to the Smalltalk-80 systems [14]; more recently, it has become almost synonymous with Java's bytecode format. In this document, we shall use the term bytecode to refer to a portable representation used for runtime compilation.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)
0-7695-2499-0/06 $20.00 © 2006 IEEE

compilation time is added to application runtime, expensive optimizations are feasible only if they provide consistent and large improvements. Thus, JITs typically exclude most expensive optimization techniques.

This challenge has led JIT authors to focus on the design and implementation of optimizations that use only modest amounts of compile time. Often, these faster algorithms use starkly different strategies than traditional optimizations, thereby sacrificing optimization quality for compilation speed. This paper focuses on an important optimization—register allocation—and its implementation in a JIT environment. Most JITs abandon the traditional, proven techniques that operate by analogy to graph coloring in favor of less expensive algorithms that are also less effective. In this paper, we reformulate graph-coloring register allocation—a computationally expensive technique popular in offline compilers—so that it is fast enough to use in a runtime compiler. Our reformulation sacrifices some optimization quality in favor of efficiency, but it preserves the essential flavor and, consequently, the proficiency of the graph-coloring allocators. Our approach yields the right balance between compilation efficiency and optimization efficacy and makes graph coloring allocation more attractive for use in a runtime compiler.

2 Register Allocation

Register allocation is the process of mapping values in the input program to a limited set of machine registers. Register allocators typically take an intermediate representation of a program as input. This representation does not impose any architectural limitations on the number of registers – values are contained in locations known as virtual registers. It is the allocator's responsibility to map the unlimited set of virtual registers into a finite number of machine (or, *physical*) registers. Moreover, while conducting this mapping, it must maintain the semantics of the program. Register allocation is a critically important, and consequently, well-studied transformation. In general, the allocation problem is NP-complete [20] and several heuristic-based techniques, such as graph coloring, have been designed to conduct allocation on a traditional, offline compiler. Most JITs, however, prefer to implement faster allocation algorithms in an effort to increase allocation efficiency. In particular, linear-scan algorithms are a popular choice for runtime compilers [19, 22, 18]. As the name suggests, the runtime behavior of these algorithms tends to be linear in the size of the input. In practice, the graph coloring approach is more proficient at allocating registers than linear-scan techniques and can improve the performance of the allocated code [19, 22]. But, in a JIT environment, the additional compile time required by the algorithm greatly diminishes the runtime gains achieved by an improved allocation. Thus, the JIT designer is faced with a difficult tradeoff between compilation proficiency and efficiency. In this paper, we address this difficult decision by designing an efficient graph-coloring allocator for a runtime compiler.

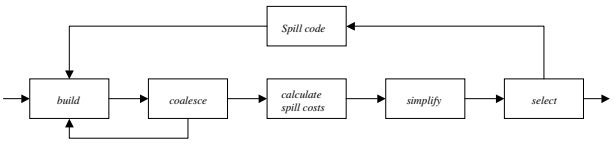


Figure 1. Overview of the Chaitin-Briggs allocator

3 Background: The Chaitin-Briggs Register Allocator

We chose to modify a well-known and widely implemented graph coloring technique—the Chaitin-Briggs algorithm [9, 8, 7]. The algorithm consists of six phases, shown in Figure 1.

1. *Build the Interference Graph*: The initial phase identifies interferences by constructing live ranges and building a graph to represent interferences between these ranges.

2. *Coalesce*: This phase removes register-to-register copies if the source and the destination registers do not interfere. The build and coalesce phases are repeated in tandem until no more coalescing can be conducted.

3. *Calculate Spill costs and Simplify*: To make an informed spill decision, the allocator estimates a spill cost for every node in the interference graph. The Simplify phase consults the spill cost and orders the nodes by placing them on a stack.

4. *Select*: The allocator tries to color the graph by repeatedly popping a node from the stack, inserting it into the graph, and attempting to assign it a color. If all colors have already been exhausted by its neighbors, then the node is marked for spilling and left uncolored.

5. *Spill code insertion*: Spill code is inserted for nodes marked for spilling by the previous phase and the allocator is restarted on the modified program.

Briggs' allocator makes spill decisions later in the process than does Chaitin's. Briggs calls this procedure *optimistic coloring* since the algorithm defers spilling a node in the hope that a color will actually be available for it.

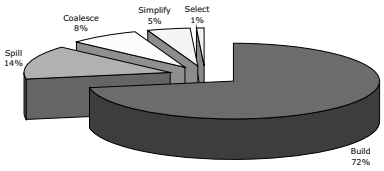
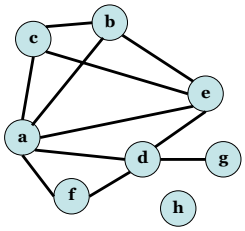


Figure 2. Contribution of phases in Chaitin-Briggs to the total allocation time. The values are geometric means over all benchmarks




```
Procedure P:
  c = ...
  b = ...
  a = 0
  e = load value
  d = b + c
  f = e + 10
  g = f * 50 + a
  h = d + g
  return h
```

Figure 3. An interference graph constructed from a simple procedure

4 Identifying the expensive components of the algorithm

In order to effectively redesign the Chaitin-Briggs algorithm for a JIT, we profiled its execution to measure the cost of each of its different phases. For most programs, the interference graph builder is the most expensive component—its worst-case asymptotic bound is $O(n^2)$, where n is the number of live ranges in the program. However, since experimental results can sometimes differ from those suggested by complexity analyses, we conducted a number of experiments to measure the relative performance of the six phases. Figure 2 presents the results of those experiments. We timed each of the allocator's different components and computed the percentage of time spent in each phase. As displayed in Figure 2, our results show that, as predicted by the complexity analysis, the interference graph builder is the most expensive component. On average, it consumes about 72% of the allocation process. These results show that reducing the time required by the graph builder should increase allocation efficiency. Note that the algorithm can revisit the build phase several times. In particular, if spilling or coalescing occurs, the allocator rebuilds the interference graph. This observation is crucial to understanding the contribution of the build phase towards the overall cost of register allocation. The next few sections describe how we redesigned the interference graph construction algorithm to increase the allocator's efficiency.

```
Build_Live_Sets(Procedure P)
 Use an worklist algorithm to compute live-in
 and live-out sets for each block in P

Build_Interference_Graph(Procedure P)
 call Build_Live_Sets(P);
  for every block B in P
   Current_Live(B) = LiveOut(B)
   for every inst. I in reverse order
    for every definition D in I
     add an interference from D to every
     element in Current_Live - {D} creating
     nodes if necessary
    for every definition D in I
     remove D from Current_Live
    for every use U in I
     add U to Current_Live
```

Figure 4. Algorithm for constructing the interference graph in a Chaitin-Briggs allocator

5 Interference Graph building

The Chaitin-Briggs algorithm models allocation as a graph-coloring problem. It first builds an interference graph that denotes the safety constraints that the allocator must respect. These constraints, essential for maintaining the semantics of the program, are called *interferences*. The interference graph is an undirected graph that consists of nodes and edges. Nodes represent the live ranges in the program. An edge between two nodes indicates that the two corresponding live ranges interfere with each other. Chaitin defines the term interference as: *two names interfere if one of them is live at a definition point of the other*[2] [9]. If two live ranges interfere, then they cannot share the same physical register. Thus, the register allocator must preserve safety by ensuring that interfering live ranges are allocated to different registers. Figure 3 depicts an interference graph constructed from a simple procedure. We shall revisit the structure of the interference graph while describing our modified allocation algorithm in Section 6.

Figure 4 contains the pseudo-code of the Chaitin-Briggs interference graph construction algorithm. Recall that two live ranges interfere if one is live at the other's definition point. First, the algorithm calculates liveness information by using a classic, worklist-driven, data-flow algorithm. This analysis annotates each block in the procedure with the set of live-ranges that are live at the block's beginning and at its end. Next, the graph constructor iterates over every block in reverse. At each instruction, the algorithm computes the set of values that are live at that point in the program incrementally from the previously known set. Live ranges that are defined in that instruction are deleted from the live set and live ranges that are used in the instruction are added to

[2]Chaitin's "names" are derived from connected components of a virtual register's def-use chains and are analogous to live ranges

R_2=R_1 op R_3	load T_1 R_2=T_1 op R_3	T_2=R_1 op R_3 store T_2
Before Spilling	R_1 spilled	R_2 spilled

Figure 5. Spill code insertion

the set. To build the actual graph, the process adds interferences between a value defined in the current operation and all values live at the definition.

After the interference graph is built, the graph-coloring allocator proceeds as shown in Figure 1. Note that during the allocation process, the interference graph information might become outdated. As a result, the allocator must rebuild the interference graph. Rebuilding occurs for two reasons: spill code insertion and coalescing. When spill code is added to the program, new live ranges are created and the spilled live range disappears. The allocator needs interference information for these ranges before it can safely allocate physical registers to them. Therefore, at the end of the spill phase, the algorithm rebuilds the interference graph, re-running the algorithm in Figure 4. Coalescing can also make the interference graph imprecise. Briggs describes this issue comprehensively in his dissertation [6]: when two values are coalesced, the allocator constructs an approximate set of interferences for the combined live range. This approximation, though safe, may be overly conservative. Rebuilding the interference graph after coalescing corrects this problem. Moreover, both coalescing and spilling can invalidate the live sets. Thus, the interference graph builder must rebuild live sets before rebuilding the graph.

6 Redesigning the interference graph builder for a runtime compiler

Our primary goal was to increase the efficiency of the Chaitin-Briggs allocator. As we will show in the next few sections, we modified the interference graph construction component of the algorithm. In doing so, we tried to decrease the graph-building time process while preserving the overall proficiency of the allocator. Our modified allocator introduces some imprecision in the interference graph and significantly reduces graph building time. Borrowing image-compression terminology, we refer to our modified algorithm as the *lossy* allocator.

Before we examine the modified algorithm, consider the spill insertion mechanism in a Chaitin-Briggs allocator. The modifications change the manner in which the interference graph is updated after spilling. Figure 5 summarizes the instructions inserted for spill loads and stores. In the descriptions of the lossy allocator, we shall use the term *temporary register* to refer to the live ranges inserted by the spiller (T_1 and T_2 in our examples). For consistency, we shall use R to refer to a spilled register and T to refer to a temporary register.

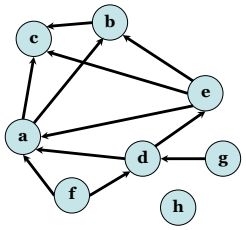


Figure 6. Interference graph for the procedure shown in Figure 3, with edges partitioned into definition and use edges. Here, an arrow from node n_1 to node n_2 indicates that $< n_1, n_2 >$ is a definition edge.

6.1 The lossy allocator

In designing our lossy algorithm, we wished to construct the interference graph once and then use incremental methods to update the graph after spilling and coalescing. To achieve a substantial decrease in allocation time, we were willing to accept some loss in allocation proficiency. To this end, we decided to augment the representation of the interference graph. The unmodified interference graph is represented by two major data structures – a bit matrix and a collection of edge sets. The bit matrix indicates whether two nodes in the graph interfere. Each node in the graph, N, holds an edge-set that lists the nodes with which it interferes.

In the lossy allocator, we added additional information to the edge-sets – each edge originating from a node contains a tag indicating the type of the edge. We classified every edge in the graph as a *definition edge* or *use edge*. To comprehensively define these terms, let us reconsider the algorithm described in Figure 4. The procedure for building an interference graph traverses the program, identifies live ranges, and adds interferences between them. A careful examination of the algorithm shows that there exist three scenarios when an interference edge is added. If the algorithm added an edge between live range L_1 and L_2, then either:

1. The algorithm discovered that L_2 is live at a definition point of L_1, in which case the edge $< L_1, L_2 >$ gets classified as a definition edge, or
2. The algorithm discovered that L_1 is live at a definition point of L_2, in which case the edge $< L_2, L_1 >$ gets classified as a definition edge, or

```
for every block B in the procedure
 iterate through every inst. I in B
  if a load is needed for temporary reg. T
   locate the last def. in B prior to I
    if such a def. D is found
     add the edge (T, D) to the graph
     set D to its name before renaming
     for every def. edge <D, E>
      add the edge (T, E) to the graph
      if D is a copy inst., add an edge
      between the source and T.
    else if no such def. exists
     for every value L in LiveIn(B)
      add edge (T, L) to the graph
  if a store is needed for reg. T
   let D = the name of T before renaming
   for every def. edge <D, E>
    add the edge (T, E) to the graph
  mark all edges added to T as def. edges
  if the load services multiple insts.
   add interferences between T and all
   defs. till the last use of T
  remove spilled nodes from graph
```

Figure 7. Lossy algorithm for reconstructing the interference graph after spilling

3. The algorithm discovered that both cases 1 and 2 occurred – L_1 was live at L_2's definition point and L_2 was live at L_1's definition point. In this case, both edges are classified as definition edges

A use edge is an edge that has not been classified as a definition edge. The algorithms described in this paper focus solely on definition edges. In our initial construction of the interference graph, we used these specifications to categorize the edges. Consider the program that was displayed in Figure 3. Figure 6 shows the interference graph for the program with edges partitioned into the two categories. This modification is similar to transforming the undirected interference graph into a constrained directed graph.[3] This seemingly minor distinction between edges holds the key to a faster algorithm. In the next few paragraphs, we shall use tuple notation to denote edges only when a distinction between use and definition edges is warranted.

6.1.1 Updating the graph after spilling

Once the enhanced interference graph has been constructed, the lossy allocator utilizes the additional information embedded in the graph to guide post-spill incremental updates of the graph. Let a live range R be spilled at an instruction I in block B of the program, and a new temporary register T created in its place. The allocator must, as before, compute the interference edges for T. Prior to inserting the spill instruction, our algorithm iterates backwards through the block and locates the nearest definition point. Let D be the live range defined at that location. The allocator adds all of D's definition edges as interference edges for T. Further, it classifies these edges as definition edges for T. If a definition point is not found before the beginning of the block is encountered, then the algorithm adds all members of the live-in set of B as definition edges for T. This procedure results in generating a safe but conservative estimate for the interference edges of node T. Figure 7 outlines the lossy spill algorithm.

In the next section, we shall prove two lemmas that confirm that our updates are safe for spill stores and loads that service one instruction. Lemma 1 proves that given a safe pre-spill interference graph, if a definition point is found in the block while iterating backwards from P, then the addition of these edges to T is sufficient to ensure safety. Lemma 2 proves that if a definition point is not encountered before the beginning of the block and all edges from the live-in set is added to T, then the interference graph is safe after the update. Since the updates occur before register assignment, the registers referred to in the lemmas are virtual registers.

Preliminary definition: An interference graph for a procedure P is considered *safe* iff the following condition holds: Given any two live ranges L_1 and L_2 in P, L_1 and L_2 interfere $\rightarrow \exists$ edge (L_1, L_2) in the interference graph

Invariant 1: For a node N in the interference graph, the set of definition edges contains the nodes of all values live at N's definition points. Note that due to the classification of edges described in Section 6.1, this proposition is true after the initial construction of the interference graph.

Lemma 1 *Given:*

(i) a spilled register R renamed to T at instruction I in basic block B
(ii) I contains the only uses of T before spill code insertion
(iii) a safe interference graph for which Invariant 1 holds before processing the spill
(iv) D is the first definition of a register in B found by iterating backwards from I

If the edge (T, D) and all of D's definition edges are added to T, the resulting node for T contains all interferences needed to ensure register allocation safety. Further if all added edges are marked as definition edges originating from T, Invariant 1 is preserved by this update.

Proof: *We shall prove this by contradiction. Let us first consider what it means for the update not to be safe. The*

[3] Specifically, the directed graph, DG, must maintain the one-one mapping between itself and an undirected graph – if $< n1, n2 > \in DG \Rightarrow < n2, n1 > \in DG$. This is similar in structure to Cooper and Simpson's containment graph [10] but encodes very different semantics

update is unsafe iff after the edges are added, the graph does not contain an edge between T and a live range L even though L and T interfere. This follows from the definition of safety in an interference graph. Let (T, L) be such an edge that is not added to the graph after the update.

First, note that L cannot be equal to D since we know that (T, D) is added to the graph. Further, if L and T interfere, since the definition point of T for both spill loads and stores is located in the instruction immediately preceding its only use, it implies that L was live at the definition point of T. Recall that if a spill store is being inserted, then the definition point of T is I. If, however, a spill load is inserted, a new instruction defining T will be inserted just before I. In both cases, remember that our algorithm updates interferences before inserting the spill instruction.

Since the algorithm adds all the definition edges of D to T, the absence of (T, L) and Invariant 1 implies the absence of $< D, L >$ in the graph and that L was not live at the definition point of D. Therefore, L must have been killed (i.e. defined) between T's and D's definition points in B. But, we know that D is the first definition of a register in block B while iterating backwards from I and that L is not D. Thus, we reach a contradiction. It also follows that since Invariant 1 holds before the update and that no registers are killed between the definition points of T and D, marking the added edges as definition edges for T preserves the invariant.

Lemma 2 *Given conditions (i) through (iii) from Lemma 1 and:*

(iv) there are no register definitions from the beginning of B till I

If all edges in live-in(B) are added as definition edges originating from T, then the resulting node for T contains all interferences needed to ensure register allocation safety. Further, Invariant 1 is preserved by this update.

Proof: *Again, as in Lemma 1, the update is unsafe iff after the edges are added, the graph does not contain an edge between T and a live range L even though L and T interfere. Now, if L interferes with T and there are no definitions between instruction I and the beginning of the block, then L must be in the live-in set of block B. Thus, adding all members of the live-in set of B will ensure the presence of an edge between T and all ranges that interfere with T. Note that since there are no values killed between the beginning of B and I, marking all values in live-in(B) as definition edges for T will preserve Invariant 1.*

Lemma 1 and Lemma 2 prove that our updates are safe for spills that service one instruction. We, however, need to consider the situation when a load services multiple instructions.

6.1.2 Spill loads servicing multiple instructions

If the allocator decides to spill a live range, it must iterate through the code and insert spill code—loads before a use and stores after a definition. [4] The placement of stores is simple – a store is inserted after every definition of a spilled range. However, load placement is more intricate. By inserting a load, the spiller essentially marks the beginning of a new live range. In certain situations, multiple uses of the same live range can be found in separate instructions that are situated close together without an intervening live-range death[5]. In this case, the spiller tries to schedule one load for those uses. Thus, after spilling, a single load can service multiple uses of a live range. The lossy algorithm needs to account for the entire live range of this load when it updates the interference graph. To achieve this effect, it first updates the graph as described in the preceding subsection. It then iterates forward through the block till the last use of the loaded value and adds all definitions encountered as use interferences for the newly created temporary live range. Since there are no deaths in between the load and the uses, it only needs to track live range definitions while iterating. There are no dead live ranges to be removed.

Lemma 3 *Given condition (i) from Lemma 1 and:*

(ii) T is used (read from) in instruction I and a spill load is placed immediately preceding instruction I

(iii) T is also used in instruction J that appears after I in B

(iv) a safe interference graph for which Invariant 1 holds before processing the spill

We know from Lemmas 1 and 2 that if T was used solely in I then after we update the interferences for T, the interference graph will be safe. We need to show that after the update, if we add all live range definitions between I and J, then we will end up with a safe interference graph and Invariant 1 will be preserved.

Proof: *Consider an interference caused by the occurrence of T in J that is not present after the update. This can only be caused by a live range whose definition point lies in between I and J. Thus, if the algorithm adds all definition points in between I and J, we will capture all interferences caused by the use of T in J. We know from Lemmas 1 and 2 that Invariant 1 is preserved for the updates described in Section 6.2.1. In this case, there are multiple uses of T. However, the presence of additional uses of T does affect the values live at its definition point. Thus Invariant 1 is preserved as shown in Lemmas 1 and 2. In this lemma, we do not specify the lack of live range deaths between the uses*

[4] We use Harvey's suggested modification to Briggs' algorithm [15] but the same updates can be conducted on Briggs' original algorithm.

[5] the last use of a live range

of T as a pre-condition since it affects strictly the precision, and not the safety of the updates.
The three preceding lemmas prove that our modifications of the interference graph do not compromise on the correctness of register allocation and that the graph contains at least all the edges required to preserve safety during register allocation.

6.1.3 Sources of imprecision

Though we have proven that the lossy allocator constructs safe interference graphs, it may add unnecessary edges to the graph. There are three sources of imprecision. Firstly, remember that after spilling the algorithm searches backwards for a definition point. It then adds all interferences for the definition (say D) as interferences for the temporary live range. But, D might interfere with more registers than the temporary live range. Specifically, if the death of a live range occurs between the definition point and the spill instruction then D will interfere with that live range while the temporary live range will not. This situation arises only for spill loads since for stores, the temporary live range is defined in the instruction. D can also be defined in multiple locations which may lead to the addition of extraneous interferences. However, JIT instruction selectors are designed to be fast and typically reserve new virtual registers for each defined value. Therefore, we do not expect multiple definitions to be a major source of imprecision. In our experiments using LLVM, we encounter this imprecision mainly for definitions by copy instructions that are generated to eliminate SSA ϕ-nodes. Second, if the definition point is a copy instruction, then the source and the target register of the copy do not interfere with each other. However, we are conservative and add an edge between the source and T. This edge is superfluous if the source dies between the copy instruction and the spill instruction. Finally, another source of imprecision arises from adding edges between values in the live-in set and T. Again, such an edge is extraneous if the death of the live-in value occurs before the spill instruction in the block.

6.1.4 Updating live set information after spilling and coalescing

When a live range is spilled, the lossy allocator follows the original Chaitin-Briggs algorithm – it replaces the reference to the original range by a newly created, temporary register and inserts the appropriate spill instructions. This invalidates live set information. In Chaitin-Briggs, live set information is reconstructed when the interference graph is rebuilt. Since the lossy allocator does not rebuild the graph after spilling, it must update the live sets to reflect the post-spill changes. Thus, after a live range is spilled in block B, the algorithm erases the live range from the live-in set of every block in the procedure. Notice that the temporary live ranges are local to the block they are created in. Therefore, these values can be ignored by the live-in and live-out sets. In the coalescing phase, the allocator attempts to merge live ranges. Combining two live ranges into one can affect liveness information and we must carefully propagate these changes to the live sets after a coalescing iteration. To this end, the lossy allocator updates live information by replacing all references of the coalesced live range in the live sets with the name of the newly merged live range.

7 Experimental methodology and results

In the next few sections, we shall present the results of our experiments that evaluated the performance of the lossy allocator. For the experiments we used the LLVM compiler infrastructure since it was modular, flexible, and very well documented [16]. LLVM uses a SSA-based intermediate representation [6] and provides two types of compilers: a dynamic, JIT-driven compiler as well as a static compiler. The LLVM runtime system uses a compile-only approach – the JIT compiles a procedure to native code upon the first invocation of the procedure. We implemented both the classic Chaitin-Briggs algorithm and our lossy allocator in LLVM. We compiled and evaluated our benchmarks on an Intel Pentium 4, 3.2GHz processor with 1 GB of main memory running Linux. The Pentium 4 has 7 allocatable integer registers and 8 allocatable floating-point registers. We evaluated the allocators on benchmarks from the SPEC CINT2000 suite. We selected these benchmarks since they perform mostly integer computations. Currently, the LLVM x86 backend has limited support for global floating-point register allocation and is generally unable to allocate floating-point values across basic blocks due to complications in handling the stack-based floating-point register file. The reported times are the sum of the system and user times consumed by a process.

7.1 Performance of the lossy allocator in an offline compiler

Our first goal was to examine how effectively the lossy allocator reduced allocation time in contrast to a standard graph-coloring algorithm for offline compilers. We compared the performance of the new algorithm to the Chaitin-Briggs allocator. In our first experiment, we statically compiled our benchmarks with the two allocators and compared the allocation times. Figure 8 shows the results of the experiment. As the graph indicates, the lossy algorithm

[6] The code presented to the register allocator, however, is not in SSA format due to ϕ-node elimination and architecture-specific transformations.

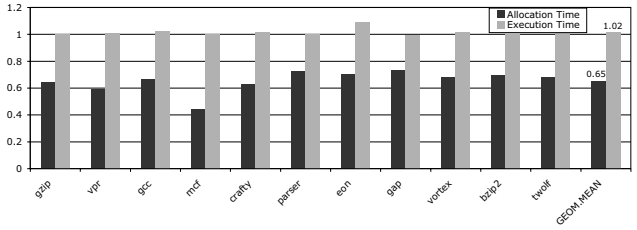


Figure 8. Allocation and execution times using the lossy allocator in an offline compiler. Displayed times are relative to the Chaitin-Briggs allocator

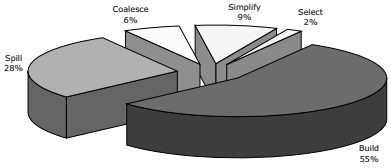


Figure 9. Contribution of phases in the lossy allocator. The values are geometric means over all benchmarks.

performed well and conducted allocation much more efficiently than Chaitin-Briggs. On average, the lossy allocator decreased allocation time by 35%. This is a significant decrease over the original Chaitin-Briggs algorithm. Since the lossy algorithm can add superfluous edges to the interference graph, our next experiment compared the execution times of programs allocated by the lossy and Chaitin-Briggs algorithms. The results, also shown in Figure 8, confirm that the lossy algorithm increases the runtime of the allocated program. However, we were pleased to note that the runtime only increases by around 2% on average. We will provide a more thorough measurement of the imprecision of the lossy allocator that led to this runtime degradation in the next few sections. Note that these two experiments measured the performance of the allocator in an offline compilation environment. Our intention in designing these experiments was to understand the potential of the modified allocator by examining its efficiency and allocation performance in isolation. We were heartened by the sharp reduction of allocation time and the competitive performance of the code generated by lossy allocation. However, there is one major difference between allocation time measurements obtained in a static and runtime compiler. In most runtime compilers, procedures are compiled on-demand. If a procedure is not invoked at runtime, then the JIT does not translate that procedure to native code. The offline compiler, in contrast, compiles all procedures oblivious to their utilization at runtime. Hence, for a particular benchmark,

	SPILLS			EDGES		
	LOSSY	CB	RATIO	LOSSY	CB	RATIO
gzip	1061	1032	1.028	18852	17020	1.108
vpr	6665	6469	1.030	103334	90136	1.147
gcc	61214	58601	1.045	953856	857890	1.111
mcf	378	378	1.000	10098	9056	1.115
crafty	10477	10084	1.039	190942	169614	1.126
parser	4136	3974	1.041	75250	70822	1.063
eon	17559	17368	1.011	218664	202294	1.081
gap	23620	23041	1.025	395012	370348	1.067
vortex	8583	8459	1.015	225194	217736	1.034
bzip2	1264	1204	1.050	16640	15210	1.094
twolf	11627	11204	1.038	183962	166196	1.107
G.MEAN	6636.0	6436.8	1.031	117355.6	106752.7	1.099

Table 1. Extra spills and edges added by the lossy allocator when compared to the Chaitin-Briggs (CB) allocator.

we expect to see the register allocator operating on fewer procedures in a dynamic compiler than suggested by these results. The measurements in Figure 8 thus provide an upper bound on the allocation-time improvements we anticipate in a dynamic environment. Figure 9 depicts the running time of the lossy allocator broken down into its different components. When compared to Figure 2, the diagram demonstrates that the gains in allocation efficiency were due to a significant reduction in interference graph construction time.

7.2 Imprecision in the lossy allocator

The lossy allocator, in its bid to increase allocation efficiency, may add more edges to the interference graph than the Chaitin-Briggs algorithm. Our next set of experiments examine the imprecision of the lossy allocator. We compiled the SPEC integer benchmarks using two allocation algorithms – the Chaitin-Briggs and the lossy allocators – and compared the interference graphs that were produced for every procedure in the benchmark. We tallied the number of edges that were produced by both allocators. Our results, as displayed in Table 1, show that the lossy allocator adds a moderate number of extra edges to the graph. On average, the allocator added around 10% superfluous edges.

The addition of extra edges by the lossy allocator affects the colorability of the interference graph. Since some nodes have more edges than they would if the allocator rebuilt the graph from scratch (e.g., Chaitin-Briggs), they might be more difficult to color. Consequently, the lossy allocator may generate more spill code than Chaitin-Briggs. We measured the amount of spill code generated by both allocators and present the comparison in Table 1. Compared to the Chaitin-Briggs allocator, the lossy algorithm generated around 3% more spills on average. As Table 1 shows, the relative increase in edges is greater than the relative increase

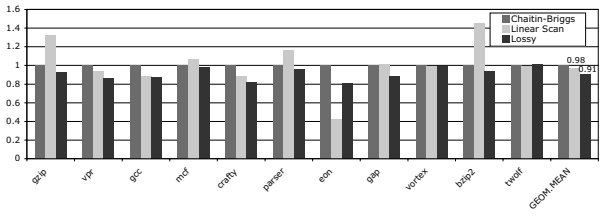


Figure 10. Observed runtime of benchmarks in a dynamically compiled environment using different allocators. Reported times are shown relative to the Chaitin-Briggs allocator.

in spills. This occurs primarily because spilled nodes in the lossy allocator contain a higher number of extraneous edges as compared to unspilled nodes.

7.3 Performance of the lossy allocator in a runtime compiler

Since we designed the lossy algorithm for runtime compilation, our most important experiment measures the performance of the allocator in a dynamically compiled environment. We present a summary of our results in Figure 10. As can be seen, the lossy allocator outperformed the Chaitin-Briggs algorithm on all the benchmarks. The results demonstrate that the significant decrease in compilation time for the lossy allocator more than compensated for the marginal increase in execution time. Note that, as is the case with dynamic compilation, not every procedure in a benchmark was compiled. However, the incremental techniques reduced compilation time considerably. On average, the lossy allocator reduced observed runtime (compile time + execution time) by 9% when compared to the Chaitin-Briggs allocator.

To gauge how effective the lossy allocator was in comparison to a JIT specific algorithm, we compared its performance to the linear-scan algorithm that was bundled with LLVM. The LLVM linear-scan algorithm extends [18] by removing the need to reserve spill registers and adding the ability to propagate spill code into instructions [13]. In our experiments, the lossy algorithm outperformed linear-scan on 9 of the 11 SPEC integer benchmarks. On 2 SPEC benchmarks—eon, and twolf—the allocator performed worse than linear-scan. On these benchmarks, graph-coloring required significantly more compilation time than linear-scan and thus the improved allocation could not compensate for the substantive compile-time handicap. Note that on both benchmarks, the lossy allocator performed better than Chaitin-Briggs. Second, on 4 benchmarks: `vpr`, `gcc`, `crafty`, and `vortex`, the lossy allocator outperformed linear-scan which, in turn, bested the Chaitin-Briggs technique. This result emphasizes the success of redesigning a strong allocation technique for runtime compilation. Further, we note that the input data size plays a major role in the relative performance of the allocators in a runtime environment. These results were obtained by running the SPEC benchmarks on large data sets. In the next section, we will examine the relationship between input data complexity and allocator performance.

7.3.1 Input data size and allocator performance

The results obtained by our experiments highlight a difficult decision that JIT compiler designers must make. Optimizations that are expensive to conduct may improve the dynamically compiled code for current and future invocations. However, the additional time consumed by these optimizations can outweigh the advantages of executing strongly optimized code. In general, the longer a program[7] continues to execute, the greater the advantages of expending additional time to optimize the code. We wished to understand the impact of program running time on the relative performance of our allocators. Therefore, we gradually increased the input complexity of our benchmarks and measured the performance of code allocated by the linear-scan, Chaitin-Briggs, and lossy algorithms on a runtime compilation environment. Figure 11 shows the results of our experiments for 3 benchmarks.

The performance of the allocators exhibits an interesting progression. For smaller input sizes, the linear-scan allocator outperforms all other allocators. For these sizes, the overhead of graph-coloring register allocation swamps the benefits afforded by stronger allocation. As the input size increases, procedures in the program are executed more frequently. Progressively, the more proficiently allocated code begins to recoup the extra time it ceded during optimization. We were pleased to note that, as shown in Figure 11, the runtime compiler that uses the lossy allocator starts outperforming the linear-scan JIT much before its Chaitin-Briggs counterpart. (The crossover points are marked on the graph with dotted vertical lines.) We wish to highlight three key features of these results. First, the performance curve demonstrates that program runtime changes the choice of the best allocation algorithm in a runtime compiler. Second, the lossy allocator considerably reduces the switch-over point between the compile-time efficient linear-scan technique and a graph-coloring algorithm. For instance, as can be seen in the `parser` graph, the crossover point between linear-scan and the lossy allocator is around 2400 lines of input. In contrast, the program allocated with the Chaitin-Briggs graph-coloring allocator sur-

[7] or procedure if different optimization algorithms can be selected for each procedure in the program

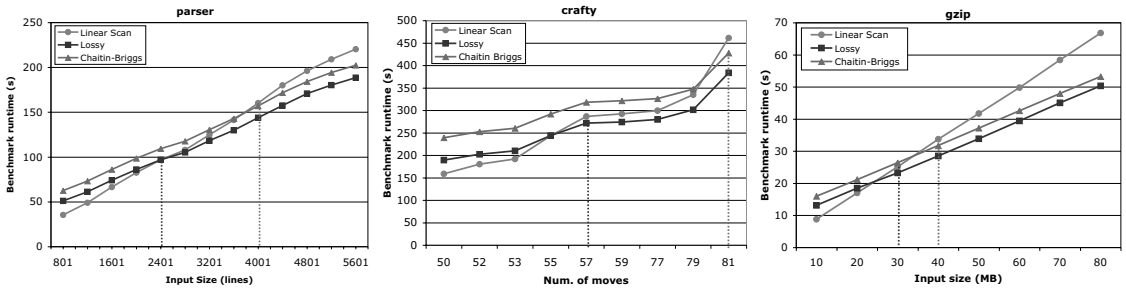


Figure 11. Behavior of allocated code in a runtime compiler for 3 benchmarks as the input complexity increases. Lower y-axis values signify better performance. The dashed vertical lines indicate the cross-over points for the Chaitin-Briggs and lossy allocators with respect to linear-scan.

passes linear-scan at around 4000 lines of input. As a result, the lossy allocator can be profitably invoked even on programs that do not run for an extended period of time. Lastly, note that for all 3 benchmarks, the lossy allocator maintains its dominance over Chaitin-Briggs even for larger input sets. This indicates that the extra spill-code in the lossy allocated program did not occur in frequently executed regions of the benchmark. This is partly fortuitous and further extends the performance benefits of the lossy allocator. However, as we have discussed in Section 7.2, the lossy allocator inserts only a marginal amount of additional spill code. Thus, we have noticed this same progression in our other benchmarks.

8 Related Work

Researchers have long explored the merits of optimizing code at runtime. Early work in this field was motivated by the optimization opportunities inherent in dynamic languages. As narrated in [3], runtime compilers have evolved from their humble beginnings in APL to full-blown and sophisticated just-in-time compilers for Java. Several studies have highlighted the advantages of runtime optimization over pure interpretation of bytecode [11, 12, 2]. Importantly, these studies have argued that the runtime system must be judicious in invoking the dynamic compiler. If the JIT is called indiscriminately, the overhead of the compiler swamps the benefit of producing faster native code. Therefore, many execution systems conduct a cost-benefit analysis before deciding to compile using the JIT. For instance, the IBM Java JIT is invoked when the number of method calls crosses a threshold [21]. Similarly, the HotSpot virtual machine interprets the bytecode until it detects a frequently executed region. The VM then triggers the JIT on that region, thereby compiling it to machine code [17].

Careful JIT-invocation decisions thus impact the runtime of the program. But, on being triggered, the runtime compiler too must contribute to keeping the dynamic compilation costs as low as possible. JITs are generally structured like conventional compilers – they subject the input to a series of pre-determined passes. JITs, however, focus more on reducing compilation time than other compilers. Consequently, JIT developers are more likely to choose optimizations that are efficient, sacrificing some optimization quality in the process. This concern with compilation-time is illustrated by an examination of current-day runtime compilers – literature on the HotSpot, Intel, and the IBM JITs reveal that they implement several local optimizations, preferring them over stronger, global techniques [17, 21, 1]. We were motivated by examining this trade-off and wished to explore whether traditionally expensive algorithms can be effectively tailored for runtime compilers. In this paper, we have focused on a crucial JIT optimization – register allocation – for which many global algorithms are considered too expensive for indiscriminate use on a JIT.

While register allocation is a critical pass in a compiler, optimal register allocation has been proved to be NP-complete [20]. As a result, allocation is usually performed by a heuristic-driven algorithm. Early allocation efforts consisted of simple, local algorithms. Proficient allocation of registers assumed increasing importance with the ever-widening disparity between processor and memory speeds. To tackle the NP-complete nature of the problem, some researchers modeled register allocation as a conflict graph colored using heuristics. Chaitin et al. presented the first paper comprehensively describing a graph-coloring register allocator [9, 8]. Bernstein et al. and Bergner et al. subsequently added improvements to Chaitin's technique [5, 4]. Briggs et al. redesigned the Chaitin allocator to delay spill decisions until later on in the allocation process. This change can potentially improve coloring decisions. We have used the Briggs allocator as our base algorithm. [7].

Most JITs implement simpler register allocation algorithms than those described above. Linear-scan techniques are particularly popular among JIT developers. Poletto and Sarkar presented a linear-scan algorithm that was faster than

a graph-coloring allocator [19]. As the name suggests, the allocation proceeds by making a linear pass over the program, scanning live ranges, and mapping these ranges to registers. Recent research by Traub et al. and Mossenbock and Pfeiffer has refined this strategy [22, 18]. Since linear-scan techniques are generally more efficient than graph coloring, they are attractive to implement on a JIT. However, as is often the case, the improved compilation speed comes with a performance penalty. Graph-coloring allocation tends to outperform linear-scan methods [19, 22]. By redesigning a strong allocation technique, our work captured the best elements from both algorithms – reduced compilation time as well as proficient register allocation.

9 Conclusion

In this paper, we reformulated the Chaitin-Briggs graph-coloring allocator for a runtime compiler. We presented and evaluated the lossy algorithm which attempted to preserve the proficiency of a graph-coloring technique while considerably reducing the required allocation time. We proved that our algorithm is safe and maintains program semantics. Further, our experiments show that the lossy allocator was successful in its goals. While it sacrificed some allocation efficacy when compared to the Chaitin-Briggs algorithm, the improved efficiency led to a significant decrease in application runtime on a JIT. The allocator outperformed both Chaitin-Briggs and linear-scan allocators for most benchmarks in a runtime compilation environment. On average, it improved application performance by 9% over Chaitin-Briggs and 7% over linear-scan. JIT designers are necessarily cautious in invoking strong algorithms lest they adversely affect program runtime. The algorithm we presented addresses this concern by successfully lowering the threshold for invoking a strong allocation technique. Using the lossy algorithm, a runtime compiler can reap most of the benefits of graph-coloring register allocation while avoiding the prohibitive compilation costs incurred by a traditional graph-coloring algorithm.

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1), 2003.

[2] O. Agesen and D. Detlefs. Mixed-mode bytecode execution. Technical report, Sun Microsystems, 2000. SMLI TR-2000-87.

[3] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35:97–113, Jun 2003.

[4] P. Bergner, P. Dahl, D. Engebretsen, and M. T.O'Keefe. Spill Code Minimization via Interference Region Spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[5] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–263, 1989.

[6] P. Briggs. *Register allocation via graph coloring*. PhD thesis, Rice University, Houston, TX, USA, 1992.

[7] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[8] G. Chaitin. Register Allocation and Spilling via Graph Coloring. In *SIGPLAN82*, 1982.

[9] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:45–57, Jan. 1981.

[10] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag.

[11] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[12] D. Detlefs and O. Agesen. The case for multiple compilers. In *OOPSLA'99, Workshop on Peformance Portability, and Simplicity in Virtual Machine Design*, 1999.

[13] A. Evlogimenos. Improvements to linear scan register allocation. University of Illinois, Urbana-Champaign, 2004. Project Report.

[14] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[15] T. J. Harvey. Reducing the impact of spill code. Master's thesis, Rice University, Houston, TX, USA, 1998.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[17] S. Microsystems. The Java hotspot virtual machine. Technical report, 2002. Technical White Paper v1.4.1, d2.

[18] H. Mossenbock and M. Pfeiffer. Linear scan register allocation in the context of ssa form and register constraints. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 229–246. Springer-Verlag, 2002.

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

[20] R. Sethi. Complete Register Allocation Problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 182–195. ACM, Apr 1973.

[21] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, 2000.

[22] O. Traub, G. H. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.