Revisiting Graph Coloring Register Allocation: A Study of the Chaitin-Briggs and Callahan-Koblenz Algorithms

Keith Cooper, Anshuman Dasgupta, Jason Eckhardt

Department of Computer Science, Rice University.
{keith, anshuman, jle}@cs.rice.edu

Abstract. Techniques for global register allocation via graph coloring have been extensively studied and widely implemented in compiler frameworks. This paper examines a particular variant – the Callahan Koblenz allocator – and compares it to the Chaitin-Briggs graph coloring register allocator. Both algorithms were published in the 1990's, yet the academic literature does not contain an assessment of the Callahan-Koblenz allocator. This paper evaluates and contrasts the allocation decisions made by both algorithms. In particular, we focus on two key differences between the allocators:

Spill code: The Callahan-Koblenz allocator attempts to minimize the effect of spill code by using program structure to guide allocation and spill code placement. We evaluate the impact of this strategy on allocated code.

Copy elimination: Effective register-to-register copy removal is important for producing good code. The allocators use different techniques to eliminate these copies. We compare the mechanisms and provide insights into the relative performance of the contrasting techniques.

The Callahan-Koblenz allocator may potentially insert extra branches as part of the allocation process. We also measure the performance overhead due to these branches.

1 Introduction

While processor speed has increased dramatically in the last 20 years, main memory speeds have struggled to keep up. To address this disparity, current computer architectures contain several levels of smaller but faster storage in between main memory and the processor. Consequently, modern compilers must ensure that frequently used values in a program are stored in the higher echelons of this memory hierarchy. In particular, registers are the fastest storage locations and compilers run a register allocation phase to map values in the program to registers available on the target architecture. This phase is critical in producing a speedy program. However, it is prohibitively expensive to optimally conduct global register allocation since the problem is NP-complete [21]. As a result, allocation is usually performed by a heuristic driven algorithm. Our paper will focus on two such algorithms – the Chaitin-Briggs allocator [6] and the Callahan-Koblenz hierarchical allocator [7] – that map the register allocation problem to a graph coloring problem. Both algorithms construct and color an interference graph that represents correctness constraints. As can be expected, optimal coloring of the interference graph is also NP-complete and the allocators resort to heuristics to color the graph.

The major difference in the two allocators lies in their consideration of program structure. After constructing the interference graph, Chaitin-Briggs does not consider the control flow of the program. In contrast, the Callahan-Koblenz algorithm constructs a hierarchy of tiles to capture loops and conditional control flow in the program. This tile representation of the program is then used to guide allocation and spill decisions. We shall analyze the impact of these locality-based decisions on the quality of generated code. Another key difference in the two allocators lies in their register-to-register copy removal techniques. The removal of unnecessary register copies is an integral part of both algorithms. While the Chaitin-Briggs algorithm conducts copy coalescing to eliminate redundant copies, Callahan-Koblenz uses a *preferencing* technique which is a mechanism that influences the way certain nodes are colored. We shall compare the effectiveness of the two techniques on various benchmarks.

The Chaitin-Briggs allocator has been investigated extensively, and is implemented in practically every industrial and research compiler. In contrast, while the original Callahan-Koblenz article presents a fascinating approach and makes compelling arguments about its functionality, the authors did not present an experimental evaluation. In particular, they described a relatively high-level description of the algorithm and did not provide a comparison to a high-quality baseline allocator. If the Citeseer literature database is any indication, there has been wide interest in the Callahan-Koblenz article – it has been cited almost as frequently as the well-known Briggs paper [6]. However, even after more than a decade since its publication, there still has been no evaluation published in the literature. This is unfortunate since industrial practitioners, in particular, are necessarily conservative about implementing unproven or poorly-understood algorithms in their compilers. This is especially true in the case of the Callahan-Koblenz algorithm, which, as will be seen in the following sections, is significantly more complicated than the proven, easy to implement Chaitin-Briggs allocator. This paper intends to address this gap in the literature and to provide researchers and practitioners with empirical data about the performance of this intriguing algorithm. Because Callahan-Koblenz is considered an extension to graph-coloring techniques, we used Chaitin-Briggs – a well-understood graph coloring algorithm – as the baseline of comparison.

2 Graph Coloring Register Allocation

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 theoretically unlimited virtual registers into a finite number of machine (or physical) registers. Moreover, while conducting this mapping, it needs to maintain the semantics of the program. Graph coloring register allocators construct an interference graph that represents these safety constraints. Program values are represented by nodes in the interference graph and edges between nodes imply that those values cannot share a physical register. Values that cannot share a physical register are said to *interfere* with each other. Both the Chaitin-Briggs and Callahan-Koblenz allocators construct such an interference graph for each procedure in the program and then attempt to color it. However, the two graph coloring algorithms use significantly different techniques to construct and color their interference graphs and to spill registers. To understand and highlight the impact of these differences in allocation decisions, we present a summary of the algorithms in the next two sections.

2.1 Chaitin-Briggs Allocator

As the name suggests, the Chaitin-Briggs allocator ("CB") is based on Chaitin's classical graph coloring allocator. In describing their algorithm, Briggs et al. identify several major phases in their allocator. Our implementation faithfully follows the implementation described in the paper except we do not need to discover and number live ranges (Briggs et. al call this the "Renumber" phase) since this information is already available in the static single assignment form (SSA) based representation we use. The major phases, as depicted in Figure 1 and described in [6] are:

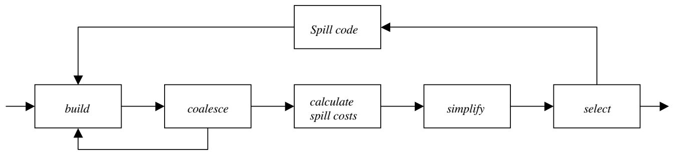

Fig. 1. Overview of the Chaitin-Briggs allocator

Build the Interference Graph: Identify interferences by constructing live ranges and marking interferences between these ranges.
Coalesce: Remove register-to-register copies if the source and the destination registers do not interfere. The build and coalesce phases are repeated until no more coalescing can be conducted. We will provide a detailed analysis of the effects of coalescing in Section 4.2.
Calculate Spill costs and Simplify: These phases calculate spill costs for every node in the interference graph and then order the nodes by pushing them on a stack after removing these nodes from the graph. The Simplify phase first removes all trivially colorable nodes – i.e. nodes that have fewer neighbors than than the number of available physical registers. If it reaches the point where no such node remains in the graph, then this phase consults the spill heuristic, chooses the node with the lowest spill cost, and pushes that node onto the stack. The process is repeated until the graph is empty and all nodes have been placed on the stack.
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.
Spill code insertion: If any nodes were marked for spilling by the previous phase, then the graph was not successfully colored. As a result, spill code is inserted for those nodes and the allocator is restarted on the modified program. The Briggs allocator marks nodes to be spilled at a later stage than Chaitin's algorithm. The authors call this procedure optimistic coloring since the algorithm defers the spilling of a node in the hope that the node will become colorable.

2.2 Callahan-Koblenz Allocator

The Callahan-Koblenz allocator ("CK") extends Chaitin's allocator by directly incorporating program structure into the allocation process. By doing so, the allocator can decide *which* variables to spill, as well as determine *where* to place the spill code. In contrast to the "spill everywhere" approach of Chaitin, Callahan-Koblenz has the potential to place spills in less frequently executed portions of the program.

Callahan-Koblenz represents the hierarchical program structure with a *tile tree*. Roughly, each tile in the tree represents a region of code such as a loop or conditional and each pair of tiles in the tree must either be disjoint or properly nested, one within the other. Such a tree structure isolates the high- and low-frequency code regions and provides a basis for the allocator's overall operation and spill placement decisions.

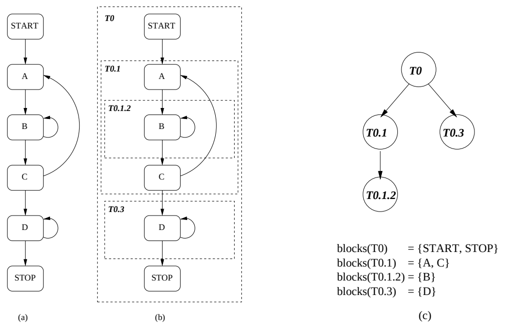

Fig. 2. Example tile tree: (a) CFG; (b) tiles overlaid on CFG; (c) the tile tree.

Figure 2 shows an example control-flow graph and its corresponding tile tree, where the set $blocks(T)$ represents all basic blocks which belong to tile T, but not to any subtiles of T. Each tile boundary represents an implicit split-point of all values live at that boundary. One of the strengths of Callahan-Koblenz lies in the ability to allocate each portion of a live range between the tile boundaries independently. These split-points also become the locations where any necessary spill code for global values will be placed. Figure 3 depicts the overall structure of the Callahan-Koblenz allocator. Once a tile tree has been constructed, two major passes are made over the tile tree.

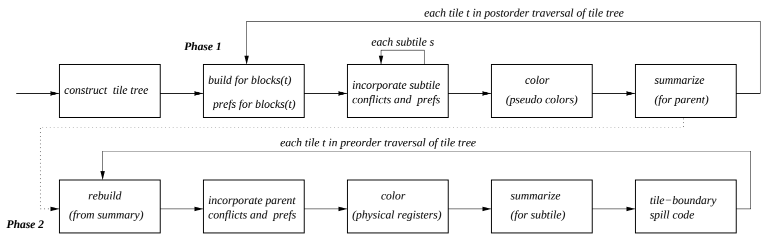

Fig. 3. The Callahan-Koblenz Allocator

Phase 1 (bottom-up): Each tile T is visited in postorder and processed independently with the goal of producing a preliminary allocation. The overall processing of each tile is similar to a Chaitin-Briggs allocator, but includes extra bookkeeping between tiles, and does not perform coalescing.

Build and preferences: Build the interference graph much like Chaitin-Briggs, but restricting attention to $blocks(T)$. Moreover, unlike the standard builder, interferences are not constructed for any variable which is live across, but not referenced in the subtree rooted at T.[1] Preferences (such as for the source and destinations of copy instructions) are also setup at this time.

Incorporate subtile summaries: All subtiles of T will have already been processed, and a compact summary of their allocations stored. This information is incorporated into T's interference graph, as well as certain types of preferences based on the subtiles' allocations.

Color: Coloring operates similarly to the Chaitin-Briggs allocator except that color choice may be influenced by preferences. Further, if a node receives a color, that color

[1] Such live ranges, which we abbreviate "LBNR", are similar to the "delayed bindings" of [18], or the "inactive" live ranges of [3].

may potentially be propagated to other nodes. Except for nodes which must receive a particular physical register, colors assigned in this phase are "pseudo colors" in the sense that they will be re-colored with a physical register in the second phase (and there are k pseudo-colors, corresponding to the k physical registers).
Summarize: After T is processed, a compressed representation of its' interference graph and allocation is constructed and passed up to the parent tile. Included in the summary are all tile-global variables allocated to registers, all tile-globals allocated to memory, and *tile summary variables*. Each TSV corresponds to a set of tile-local variables that were allocated the same color, so that the local allocation is represented in a very compact form. Conflicts involving tile-locals are stored in terms of their associated TSVs.
Phase 2 (top-down): Each tile T is visited in preorder with the goal of providing the final assignment of physical registers. Spill code is introduced at tile boundaries to reconcile differences in each tile's allocation.
Rebuild: Reconstruct the interference graph for T from its summary information.
Incorporate parent summaries: Conflicts for LBNRs that were excluded in the first phase are now added to the graph for consideration, if they received a register in the parent. Certain preferences are also setup based on the parent's allocation.
Color: A final coloring is performed, binding pseudo-colors to physical registers. As before, coloring decisions are influenced by any preferences.
Summarize: Save T's allocation and preference information to be passed down to its subtiles.
Spill code: Spill code is introduced at the tile boundaries, which may not be the same tile where a particular spill decision was made. Spill instructions could be loads, stores, or register-register copies, depending on the location of a global in T and its parent.

3 Experimental Setup

For our experimental setup, we used the LLVM compiler infrastructure [17]. LLVM allowed us to design and implement the register allocators in a machine-independent manner. We ran the allocators on an Intel Pentium 4 machine with 1 GB of main memory running Redhat Linux 9.0. The Pentium 4 processor has 7 allocatable integer registers and 8 allocatable floating point registers. We selected benchmarks that performed mostly integer computations, since the current LLVM x86 backend has limited support for global floating-point register allocation. That is, LLVM is generally unable to allocate floating-point values across basic blocks due to complications in handling the stack-based FP register file of x86. As a result, the allocators were evaluated on programs from the SPEC 2000 integer benchmarks and the Mediabench suite. The tables and graphs in our experiments depict results from 8 SPEC benchmarks: `gzip`, `vpr`, `crafty`, `parser`, `eon`, `gap`, `bzip2`, `twolf`, and 1 program from the Mediabench suite: `epic`.

4 Evaluating the Allocators

In evaluating the allocators, we posed and answered two major questions. Since a critical goal of the CK algorithm is to minimize dynamic memory references generated by spill code, the primary question that needs to be addressed is to what extent it improves on the "spill everywhere" approach of Chaitin. Second, the CK allocator might place extra operations on tile boundaries while stitching subtiles back together. We wish to measure this overhead and determine whether it is tolerable. To this end, our evaluation process consisted of running the Callahan-Koblenz and Chaitin-Briggs allocators on a number of benchmarks and comparing two key features of the register-allocated output: the spill instructions emitted and the register-to-register copies eliminated. We measured both the number of static spills and copies emitted as well as the number of these instructions executed on test inputs. We also measured the execution time of the allocated code on these inputs.

While evaluating the allocators, it is tempting to focus solely on the runtime of the allocated program. However, this might prove to be misleading on certain environments due to three issues. First, some architectures (the x86 included) use sophisticated techniques to minimize memory latency. Thus, even if the allocation algorithm allocates more virtual registers to physical registers and reduces the amount of spill code in the program, this improvement might not be reflected in a decrease in execution time. Second, the effects of cache hits and misses on spill code is unpredictable and might affect the runtime of the code. In the degenerate case, code with more spill code might benefit from random cache effects and execute faster than code with fewer spill instructions. The allocators we evaluated do not optimize for cache effects while emitting spill code – as a result, the impact of cache on allocated code is purely accidental and we would like to factor these effects out. Lastly, the evaluated allocators might produce starkly different allocations for rarely executed procedures of a benchmark. This difference might not be reflected in the execution time of the entire program. However, it is sometimes instructive to examine the contrasting allocations of these procedures. Keeping these considerations in mind, we decided on spill code and register copies eliminated as our two major evaluation metrics. An analysis of the spills and copies in the code will give us a relatively architecture-independent understanding of both allocators. In our comparisons, we used both the dynamic as well as the static versions of these metrics.

4.1 Comparing the Spill Code Emitted by Both Allocators

A graph coloring allocator typically uses heuristics to color the interference graph using the same number of colors as available physical registers, k. However, the coloring will be unsuccessful if the graph is not k-colorable, or if the heuristics fail to color a k-colorable graph. At this point, most allocators modify the program and repeat the coloring process. After an unsuccessful coloring effort, Chaitin-Briggs and Callahan-Koblenz relegate uncolorable nodes to memory and rebuild the interference graph. This process of placing a live range in memory instead of a register, known as *spilling*, reduces the length of the live range and, in general, makes the modified graph more colorable. Since the spilled range must now be fetched from memory, the allocator tries to reduce the number of these memory accesses (*spills*) executed at runtime. Callahan-Koblenz and Chaitin-Briggs use heuristic techniques to identify candidates for spilling . Though their heuristics share a general goal – to make the graph more colorable and to minimize the amount of spill code – they differ in their formulations.

Spill code insertion strategy in Chaitin-Briggs In the Briggs allocator, the spill heuristic is computed by counting the load and store instructions required if the live range were to be spilled. Specifically, if d_i is the loop depth of instruction i, the spill cost for a node is estimated to be:

$$SpillCost = LoadsCost + StoresCost \text{ where } LoadsCost = LoadWeight * \sum_{l \in SpillLoads} 10^{d_l}$$

StoresCost is calculated in a similar manner. For our experiments, the weights for load and store costs were set to 1. If a spill is required, the node with the lowest ratio of spill cost to the number of interference edges is selected for spilling. Once a live range is spilled in Chaitin-Briggs, it is loaded before a use and stored after a definition throughout the function.

Spill code insertion strategy in Callahan-Koblenz A more fine-grained spill strategy is used by the CK allocator. We give a brief overview here, but consult [7] for a more detailed discussion. Because live ranges can be split at tile-boundaries, the allocator may choose to place a variable v in different locations for each tile that it crosses. For example, v may be allocated to a register within tile t, while being relegated to memory in the parent or a subtile. The following set of equations forms the cornerstone of this strategy:

$$LocalWeight_t(v) = \sum_{b \in blocks(t)} P(b) \cdot Ref_b(v)$$

where t is a tile, $P(x)$ denotes the probability of executing a block or taking a control flow edge and $Ref_b(v)$ is the number of references to v within b. Assuming that allocating a register to variable v in t is profitable (see below) during the bottom-up phase, $LocalWeight_t(v)$ is analogous to Chaitin-Briggs' $SpillCost$ heuristic and is used, along with the degree of the node corresponding to v, in a similar fashion. However, this cost is computed based only on blocks that occur strictly within tile t, as opposed to the whole function. Moreover, the reference count of block b is weighted by the probability of b being executed. Note that for the purposes of this work, we use a static estimate of $P(b)$ rather than actual profile data to ensure a fair comparison of the spill heuristics for both allocators. If b is a block, we set $P(b) = 10^{depth(b)}$. If e is an edge emanating from a block b, $P(e)$ is computed as the reciprocal of the number of outgoing edges of b.

$$Weight_t(v) = \sum_{s \in subtiles(t)} (Reg_s(v) - Mem_s(v)) + LocalWeight_t(v)$$

Overall decisions regarding whether or not a variable should be spilled are based on $Weight_t(v)$. It is computed as a combination of $LocalWeight_t(v)$ and various penalty costs that may arise from making certain allocation decisions with respect to the parent or children of t. It may happen that the penalty outweighs the benefit of allocating v to a register, indicating that the allocator should force v into memory.

$$Transfer_t(v) = \sum_{e \in E(t)} P(e) \cdot Live_e(v), \quad where\ E(t) = EntryEdges(t) \cup ExitEdges(t).$$

$$Reg_t(v) = \begin{cases} 0, & if\ InReg_t(v) = false \\ \min(Transfer_t(v), Weight_t(v)), & if\ InReg_t(v) = true \end{cases}$$

$$Mem_t(v) = \begin{cases} 0, & if\ InReg_t(v) = true \\ Transfer_t(v), & if\ InReg_t(v) = false \end{cases}$$

where $InReg_t(v)$ is a boolean predicate which is true if v received a register in tile t, and false otherwise. $Live_e(v)$ is a predicate that indicates if variable v is live along edge e.

$Transfer_t(v)$, $Reg_t(v)$, and $Mem_t(v)$ represent the various penalty costs. The first corresponds to the cost due to tile-boundary spills, while the remaining two account for any penalties due to a tile and its parent choosing different locations for the same live range. If v is allocated to a register in tile t, $Reg_t(v)$ is the penalty of allocating

v to memory in the parent of t. Likewise, if v is allocated to memory in tile t, then $Mem_t(v)$ is the penalty of allocating v to a register in the parent of t.

To see how $Weight_t(v)$ might be negative, consider a tile t and a single loop subtile s of t. Suppose on the way up the tree, a variable v was spilled in s. If t were to allocate v to a register, then the quantity $Reg_s(v) - Mem_s(v) = 0 - Transfer_s(v) = -Transfer_s(v)$. If $Transfer_s(v)$ is greater than $LocalWeight_t(v)$, then it is generally better to spill v in t. In other words, if t allocates v to a register, it will cost more dynamic spill operations to transfer v to and from memory around s than it would to just spill v in t in the first place. When this situation arises, v will be marked so that it will not compete with other variables for a color during the coloring phase.

Benchmark	CB	CK					% imp.	
		M	M_{TB}	$M + M_{TB}$	C_{TB}	All		(w/C_{TB})
gzip	96.82	51.01	6.09	57.10	0.99	58.09	41.02	40.00
vpr	10.77	8.96	1.12	10.08	0.00	10.08	6.41	6.41
crafty	71.21	55.10	5.07	60.17	0.44	60.61	15.50	14.89
parser	51.54	27.66	1.05	28.71	1.12	29.83	44.30	42.12
eon	36.10	36.30	0.28	36.58	0.00	36.58	-1.33	-1.33
gap	53.02	43.45	4.29	47.74	0.55	48.29	9.96	8.93
bzip2	103.00	72.14	17.80	89.94	2.14	92.08	12.68	10.60
twolf	53.70	31.81	11.96	43.77	1.32	45.09	18.49	16.03
epic	8.78	4.50	6.85	11.35	0.44	11.79	-29.27	-34.23
MEAN IMPROVEMENTS							**20.52**	**19.07**

Table 1. Dynamic spill operations for SPECInt2000 and `epic` (billions)

Analysis of Spill Code Inserted Table 1 shows the dynamic spill behavior of each benchmark for CB and CK. The column marked CB is the number of dynamic memory operations executed by the CB-compiled version of each benchmark. The CK results are broken down into the three types of spill operations that can occur. Column M is the number of dynamic memory operations executed within tile boundaries (e.g., loops). Column M_{TB} and C_{TB} are the number of dynamic memory and register-to-register copy operations executed on tile boundaries, respectively. The two additional CK columns represent the sum of all dynamic memory operations ($M + M_{TB}$) and the sum of all dynamic spill operations (memory operations or copies). It is useful to isolate the different types of spills for CK in order to see the effects of tiling more directly. Finally, the last two columns show the percent improvement of CK over CB. In the first case, only memory operations are considered, whereas memory and copy operations are considered in the second case. This distinction was made to show how prevalent any remaining tile-boundary register-register copies were (indicating success or failure of inter-tile preferencing), and what overall impact they had on the improvements.

Overall, the benchmarks allocated with CK executed significantly fewer dynamic spill operations than those allocated by CB— up to 44% fewer on `parser`. On average, 20.52% fewer spill operations were executed for CK than for CB. On the other hand, there were two losses for CK. One slight loss in `eon`, and one significant 29.27% loss in `epic` (more on this later).

We examined some of the benchmarks in detail at the assembly language level to understand choices made by each allocator, and why CK performed relatively well compared to CB. Consider the code in Figure 4a, which is a typical scenario present in many of the benchmarks. Here there are two live ranges x and t competing for one register, where x is referenced once early, and heavily in some distant part of the program. There are a only few references to t in a small portion of the program, but they occur in a loop, making them frequently executed. Let us assume the total number

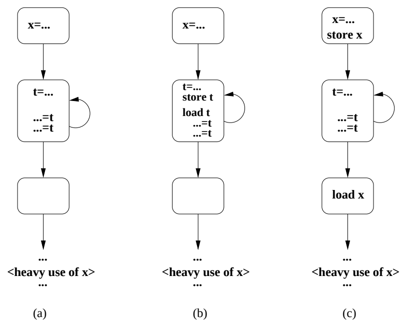

Fig. 4. Example of CK advantage: (a) original code; (b) CB spills t; (c) CK splits x.

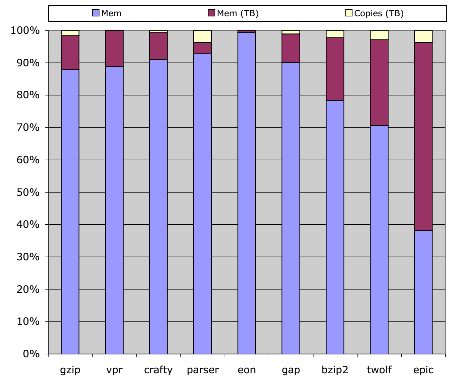

Fig. 5. Dynamic Spill Operation Types by Percentage. `TB` indicates operations on tile boundaries.

of references to x exceeds those of t. In the standard CB scheme, since the spill cost is calculated based on the references throughout the program, then x would get a color and t would be spilled (as in Figure 4b). But from the perspective of t, spilling t is a poor choice, since x is never even referenced in the loop. On the other hand, the opposite choice (giving t the color) is bad too as the many references to x will now be through memory. Because CB must spill a live range entirely, one of two poor choices must be made. As mentioned earlier, CK can consider each live range in fragments, over regions of the program. Here CK splits x before and after the loop, so that the loop portion and non-loop portions are allocated independently. This allows the result seen in Figure 4c, where t gets the register *and* x gets the register (but x is allocated to memory within the loop where it has no references). Notice also that there is a tradeoff in making such a split. A store and a load operation must be placed at loop entry and exit to make the split, which is clearly profitable here.

Returning to the loss in the `epic` benchmark, it is useful to examine the breakdown of spill operation types for CK. The graph in Figure 5 shows the percentage of total spill operations represented by each type. Looking at the `epic` bar, it is evident that something went wrong with CK's heuristics. That is, more than half of all the dynamic spill operations are memory operations on the tile boundaries. Without looking at the code, this would seem indicate that CK did not calculate trade-offs between intra- and inter-tile spilling appropriately.

In fact, on examining the assembly code, we found just that behavior. One routine dominating execution time contains a number of triply-nested loops. In one such nest, there is heavy register pressure in the inner loop, little pressure in the middle loop, and medium pressure in the outermost loop and non-loop code. There are also a number of global values live across the entire loop nest, with references in some loops and not others. Unfortunately, for some of these globals, the constituent fragments within each loop were alternately allocated to registers and memory. That is, the outermost loop allocated g to a register, the next deeper loop allocated g to memory, and the inner loop allocated it to a register. Thus, at every tile boundary there are memory operations to transfer g in and out of memory as appropriate. It turns out that these tile-transfers dominate the spill operation count, as seen in the graph. It would have been better to keep g in the same location across more than one tile boundary.

4.2 Inter-register Copy Elimination and its Impact on Allocation

Prior research has demonstrated that the removal of register-to-register copies improves code quality [12, 13]. Therefore, the efficacy of the copy coalescing phase is critical to the performance of the allocators. An effective copy removal strategy becomes even more imperative for register allocation in a SSA-based intermediate representation such as LLVM. In SSA form, ϕ-functions model merge points in a variable's path through the code. While converting from SSA form to executable code, ϕ-functions are replaced by register-to-register copies [4]. In both implemented allocators, we ran an initial pass that merged the live ranges created by the ϕ-node elimination process. This transformation, specified by Briggs in [3], ensures that the input to the two allocators remained consistent. The two-address nature of x86-instructions poses another challenge to the copy-removal mechanisms in both allocators. Since most instructions on the x86 take two operands, a three operand representation such as LLVM inserts inter-register copies in the code to conform to the specifications of the instruction set. This results in the proliferation of a large number of inter-register copies and eliminating most of these copies is desirable. In addition to copies created due to architectural and intermediate-representation features, wasteful register copies can also be generated in straight-line code. Thus, programs present many opportunities for copy removal. Since the two allocators implement different copy-removal mechanisms, we shall compare this feature in more detail in the next sections.

Coalescing and Biased Coloring The Chaitin-Briggs allocator uses two complementary mechanisms – coalescing and biased coloring – to remove register copies in the code. After building the graph, if the allocator encounters a register copy, it coalesces the source and destination live ranges if they do not interfere. This algorithm is called *aggressive coalescing* because it combines nodes without examining the resulting node's degree. Coalescing two registers changes interference information and may lead to other opportunities for copy removal. Therefore, the algorithm rebuilds the interference graph and repeats the coalesce-rebuild process until no more copies can be eliminated. In Chaitin-Briggs, coalescing is intentionally constrained – to retain flexibility during coloring, it only examines copies between two virtual registers. However, the intermediate representation might contain copies between physical and virtual registers that represent architectural limitations or procedure-calling conventions. The allocator recognizes that these copies can be made redundant by assigning the same color to both registers. To this end, Chaitin-Briggs adds the color associated with the physical register to the list of colors desired by the virtual register and attempts to assign this color to the register during the *biased coloring* phase. Biased coloring is, in spirit, very similar to preferencing in the CK allocator. However, unlike in Callahan-Koblenz, biased coloring plays only a secondary role in Chaitin-Briggs since coalescing is powerful enough to eliminate most copies.

Preferencing Preferencing refers to the notion that it may be attractive to assign the same color to multiple variables By making the coloring algorithm sensitive to such preferences, the likelihood of choosing the desired color for a node is increased. Copy removal in the CK allocator is performed by preferencing the source variable S and destination variable D of a copy together by adding each to the others *preference list.* The preference-guided color assignment algorithm then attempts to give the same color to S and D. If the attempt is successful (the preference was *satisfied*), then the resulting copy is redundant and can be trivially removed. Similarly, if either S or D is a physical register, such as a copy generated to implement subroutine linkage conventions, we setup a *local preference*. This is different than the previous case in that a variable is preferenced to a specific physical register. During color assignment, when a node receives a color, the color is propagated to all the nodes on its preference list as their local preference. If a node has a local preference, then the coloring mechanism will first attempt to assign that register before resorting to using another register. Furthermore, it will try to avoid giving a node a color that is preferred by uncolored neighbors.

In addition to copy removal, preferencing is used to influence the colors that different parts of a global live range receive. Recall that tile boundaries are implicit split-points for variables live at that boundary. Because tiles are processed independently, it is important to pass around information about these variables (in the form of preferences) so that each tile attempts to place the same global into the same register. These preferences, of course, are not generated in response to copy instructions. However, if they are not satisfied, then copy operations will be inserted at the boundary to resolve the differing allocations.

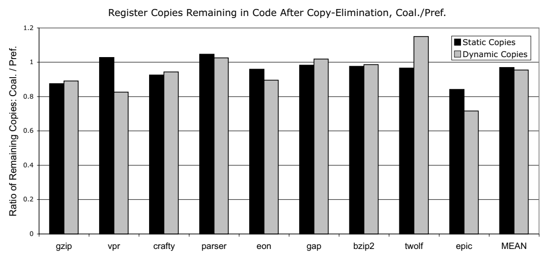

Fig. 6. Aggressive Coalescing & Biased Coloring vs. Preferencing

Experimental Evaluation To obtain meaningful results comparing the contrasting copy-removal techniques, it was important to isolate the copy elimination mechanisms from the general allocation algorithms. We wanted to ensure that our measurements would not be hampered by the inconsistent namespaces created by both allocators. Therefore, we modified CB and CK to operate on the same structure – we constructed a single tile for the entire program and provided this tile as input to the allocators. We were then able to compare the effectiveness of the preferencing algorithm to the aggressive coalescing and biased coloring combination. The results of these experiments are displayed in Figure 6 – it shows the number of copies remaining in the code after copy-removal was conducted. Our experiments show that overall, coalescing used in conjunction with biased coloring performs better and removes 3.6% more copies on average than preferencing. This translates into a 4.5% decrease in copies executed at runtime. We were, however, surprised by how closely the two algorithms performed. In stark contrast to coalescing which is executed each time the interference graph is rebuilt, preferencing can remove copies only while coloring the graph. We conclude that the careful mechanisms built into preferencing allow it to be competitive with a much more aggressive technique.

4.3 Control-flow Overhead of Tiling

A key characteristic of the tile tree construction presented by Callahan and Koblenz is that for any edge (v_1, v_2) that originates from a block outside of a tile T and terminates at a block in T (an *entry* edge), v_1 must be a block in the parent of T. Similarly, tile *exit* edges must terminate at a block in the parent of T. These conditions ensure that empty *spill blocks* can be placed along a particular entry or exit edge of a tile to contain spill code. If all edges in the original control-flow graph do not satisfy these conditions, then the CFG will be modified by adding the appropriate basic blocks. Typically these

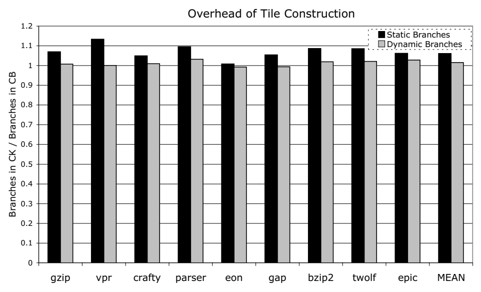

Fig. 7. Static and Dynamic Branches

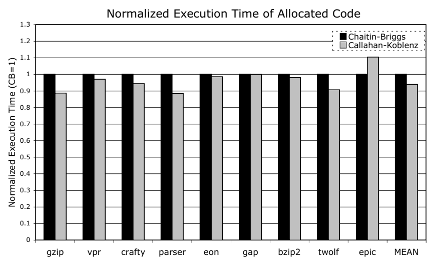

Fig. 8. Runtime of Allocated Code

extra blocks fall through to their successor and thus do not result in any additional branches in the final program. However, there are cases when inserting blocks results in unavoidable branches. Consider a branch in block v_b in the original program that is situated in the innermost loop of a loop nest L, and which targets a block v_t outside of L. Each loop in L will have been placed into its own tile after the tree is constructed, say $t_1, t_2, ..., t_d$, where $v_b \in \mathit{blocks}(t_d)$ and v_t is in the root tile t_0. Since the edge (v_b, v_t) does not terminate at the parent tile of t_d, a spill block will be inserted at every tile boundary between t_d and t_0. If any of these blocks end up with spill code after allocation, and they don't naturally fall-through to the successor, a branch will result. It is possible, then, that the resulting code might execute more branches than the original, potentially degrading performance. In order to quantify the actual impact, we compared the number of branch instructions executed in allocated code produced by both allocators. Figure 7 indicates that the CK allocated code executes more branches than the CB allocated code.

On average, the Callahan-Koblenz allocator inserted around 5.8% more branches in the code. It is interesting, however, to note from Figure 7 that the increase in executed branches was comparatively lower: 1.4% over all benchmarks. This difference between static and dynamic branches indicates that the branches placed at tile boundaries are infrequently executed.

4.4 Execution time differences

We built three versions of each benchmark and compared their execution times – executables were created by running the Chaitin-Briggs allocator, the Callahan-Koblenz allocator, and the default linear-scan allocator that shipped with LLVM. Both CB and CK perform better than the linear-scan allocator, recording improvements on average of 5.4% and 10.6% respectively. The comparison between Callahan-Koblenz and Chaitin-Briggs is summarized in Figure 8. As can be seen from the experimental results, CK outperforms CB on most of the benchmarks – on average, it improved performance by 6.1% over CB. These gains were mainly a result of the substantial reduction in spill instructions executed, as described in Section 4.1. However, on `epic`, as a consequence of the extra spills inserted by Callahan-Koblenz, it performed worse than Chaitin-Briggs, increasing program runtime by 10.4%.

5 Conclusion

We have evaluated the Callahan-Koblenz allocator on three major criteria: the amount of spill code inserted, the register-to-register copies eliminated, and the overhead incurred due to tile construction. As seen in Section 4.1, CK was able to significantly reduce the number of spill instructions when compared to Chaitin-Briggs. This reduction can be attributed, in part, to being able to independently allocate different parts of one live range. Secondly, tile local variables are given precedence over LBNRs in that we prefer to spill a LBNR over a tile local. This strategy is often beneficial, since unreferenced variables are typically long lived and thus conflict with many variables in the same region. The CK results emphasize that the spill-everywhere approach of Chaitin-Briggs can potentially degrade performance.

We were initially concerned that copy coalescing might significantly outperform preferencing because coalescing is conducted repeatedly until all opportunities for copy elimination have been exhausted. In contrast, whether preferencing choices can be satisfied depends on the order of the nodes in the coloring stack. However, as the results in Figure 6 indicate, preferencing is reasonably competitive with coalescing. Our experiments showed that, on average, Callahan-Koblenz emitted fewer spill instructions and produced faster running code than Chaitin-Briggs. However, we reiterate that these experiments were not designed to determine which allocator is better. Rather, our primary goal was to provide an understanding of the CK allocator by using another graph coloring technique as a point of reference. To that end, we did not consider adding improvements in the Chaitin-Briggs spilling strategy as suggested in various research publications. Specifically, modifications proposed by Bergner [1] and Simpson [11] would reduce the number of spills produced by the allocator. Briggs also suggests that aggressively splitting live ranges could help reduce spill code in loops [3]. Rematerialization is a technique that reduces spill instructions by identifying values that are cheaper to recompute than store in memory. [5]. In future research, we intend to devise techniques for improving the quality of spill code in both allocators.

6 Related Work

Though early computer science literature alludes to graph coloring approaches to register allocation, Chaitin et al. presented the first paper comprehensively describing a graph coloring register allocator [9, 8]. Subsequently, a number of improvements have been proposed for Chaitin's Yorktown allocator: Bernstein et. al. augmented the allocator's coloring strategy by choosing the best of three heuristics [2]. They also presented a technique that attempted to reduce the amount spill code inserted by Chaitin's allocator. Bergner and his colleagues noted that spilling can be improved for live ranges that have a small region of overlap [1]. They called their technique interference graph spilling. Our paper focuses on the refinement of Chaitin's allocator by Briggs et. al [6]. By adding deferred spilling, Briggs and his colleagues were able to significantly improve allocation, registering a reduction of spill costs up to 40% in their test suite.

While the Yorktown allocator and its improvements described above focus on reducing spill code, they do not consider program structure while making allocation decisions. Some researchers noticed this deficiency and incorporated program structure into their allocation efforts. Norris et. al. [20] designed an allocator that operates on the program dependence graph and attempted to carefully place spill code. They compared their results to a Chaitin-style allocator which lacked a coalescing phase but was augmented with deferred spilling and reported up to a 3.7% decrease in spill code. Knobe and Zadeck [14] describe a structure-based allocator using the notion of a *control tree*, which is vaguely similar to a tile tree. This allocator is similar to Callahan-Koblenz in that it can split live ranges around control tree nodes, it can spill inside of conditionals, and its *pruning* of *wedges* is not unlike CK's handling of LBNRs; however, no empirical evaluation of the technique is presented. Lueh's "Fusion" allocator also leverages program structure and appears to improve performance over Chaitin-style allocation by an average of 8.4% on the SPEC92 benchmarks [19]. A recent article suggests that with a careful relaxation of the ordering of the coloring stack, more preferences can be satisfied [15]. The hierarchical allocator evaluated in this paper was designed by Callahan and Koblenz and published in 1991 [7]. Since then, we know of one other attempt to implement the CK allocator by Wu [22]. However, the implementation deviates from the published algorithm. The author reserves registers to accommodate machine operands for spilling which significantly cripples the algorithm while the published Callahan and Koblenz paper clearly states that the hierarchical allocator does not reserve registers. There are several other major differences from the published algorithm including ignoring the degree of a node while spilling and not maintaining information during the bottom-up walk of the tree. Many other graph coloring approaches exist that do not derive from Chaitin's algorithm. Notably, Chow and Hennessey constructed basic block level interferences and used splitting to attempt to make uncolorable ranges easier to color [10]. Larus and Hilfiger describe a Chow-style allocator that assumes values reside in registers [16]; their allocator is more directly comparable to the Yorktown allocator.

As described above, several publications propose designs for graph coloring register allocators. However, the papers do not contain detailed and comparative performance results. This is partly due to the difficulty of implementing different allocators on the same intermediate representation and target architecture. In particular, the literature does not contain analyses of competing graph coloring techniques on realistic implementations. By implementing and running the Chaitin-Briggs allocator along with the Callahan-Koblenz allocator on the same environment, we intended to address this deficiency and provide an understanding of a hierarchical allocator that attempts to take advantage of program flow.

7 Acknowledgements

We would like to thank Tim Harvey for patiently answering our numerous queries on the Chaitin-Briggs allocator. We would also like to thank David Callahan, Michael Berg, and Brian Koblenz of Cray for answering our questions about the Callahan-Koblenz allocator. In particular, Brian provided us with a number of very detailed examples and explanations regarding the more subtle aspects of Callahan-Koblenz. Without the enthusiastic help and support provided by the LLVM group at the University of Illinois, this work would have proved much more difficult. Our colleagues in the compiler groups at Rice provided interesting discussions and helpful criticism. To these people go our heartfelt thanks. This work has been supported by the Los Alamos Computer Science Institute, by Texas Instruments, and by the National Science Foundation through grant number 0331654.

References

1. Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill Code Minimization via Interference Region Spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

2. David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–263, 1989.
3. Preston Briggs. Register Allocation via Graph Coloring. Technical Report TR92-183, Rice University, 24, 1992.
4. Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software – Practice and Experience*, 28(8):859–881, 1998.
5. Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 311–321, New York, NY, 1992. ACM Press.
6. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
7. D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. *SIGPLAN*, 26(6):192–203, June 1991.
8. G.J. Chaitin. Register Allocation and Spilling via Graph Coloring. In *SIGPLAN82*, 1982.
9. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:45–57, January 1981.
10. Fred C. Chow and John L. Hennessy. Register Allocation by Priority-Based Coloring. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 222–232, June 1984.
11. K. D. Cooper and L.T. Simpson. Live range Splitting in a Graph Coloring Register Allocator. In *Proceedings of the International Compiler Construction Conference*, March 1998.
12. Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.
13. Suhyun Kim, Soo-Mook Moon, Jinpyo Park, and Kemal Ebciolu. Unroll-based register coalescing. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 296–305, New York, NY, USA, 2000. ACM Press.
14. Kathleen Knobe and Kenneth Zadeck. Register Allocation Using Control Trees. Technical Report CS-92-13, Brown University, Department of Computer Science, March 1992.
15. Akira Koseki, Hideaki Komatsu, and Toshio Nakatani. Preference-directed graph coloring. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 33–44. ACM Press, 2002.
16. James R. Larus and Paul N. Hilfinger. Register Allocation in the SPUR Lisp Compiler. *SIGPLAN Conference on Programming Language Design and Implementation*, 21(7):255–263, July 1986.
17. Chris Lattner and Vikram 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.
18. P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.
19. Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, 22(3):431–470, 2000.
20. Cindy Norris and Lori L. Pollock. Register Allocation over the Program Dependence Graph. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 266–277, 1994.
21. Ravi Sethi. Complete Register Allocation Problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 182–195. ACM, Apr 1973.
22. Q. Wu. Register Allocation via Hierarchical Graph Coloring. Master's thesis, Michigan Technological University, 1996.