A Global Progressive Register Allocator

David Ryan Koes Seth Copen Goldstein

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
{dkoes,seth}@cs.cmu.edu

Abstract

This paper describes a *global progressive register allocator*, a register allocator that uses an expressive model of the register allocation problem to quickly find a good allocation and then progressively find better allocations until a provably optimal solution is found or a preset time limit is reached. The key contributions of this paper are an expressive model of global register allocation based on multi-commodity network flows that explicitly represents spill code optimization, register preferences, copy insertion, and constant rematerialization; two fast, but effective, heuristic allocators based on this model; and a more elaborate progressive allocator that uses Lagrangian relaxation to compute the optimality of its allocations.

Our progressive allocator demonstrates code size improvements as large as 16.75% compared to a traditional graph allocator. On average, we observe an initial improvement of 3.47%, which increases progressively to 6.84% as more time is permitted for compilation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

General Terms Algorithm, Design, Languages, Performance

Keywords Register Allocation, Progressive Solver

1. Introduction

As we reach the limits of processor performance and architectural complexity increases, more principled approaches to compiler optimization are necessary to fully exploit the performance potential of the underlying architectures. Global register allocation is an example of a compiler optimization where a principled approach is needed in order to to extract maximum performance from complex architectures. The register allocation problem has many components, such as spill code optimization, register preferences, coalescing, and rematerialization, that are not explicitly modeled or solved by existing heuristic-driven algorithms. Optimal algorithms for solving this NP-complete problem demonstrate the significant gap between the performance of current heuristics and the theoretical optimum, but do not exhibit practical compile times. A *progressive algorithm* bridges this gap by allowing a programmer to explicitly trade extra compilation time for improved register allocation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006…$5.00.

In order for a progressive algorithm to find an optimal solution, an *expressive model* that *explicitly represents* all the components of the register allocation problem is needed. Traditional graph coloring based heuristics [10, 12, 11] use a simplified graph model that only explicitly represents the interference constraints. Other components of the register allocation problem are implicitly handled by modifying the heuristic solver [8, 9, 40, 6]. The lack of an expressive underlying model means that even if the graph coloring heuristic in a traditional allocator is replaced with an optimal coloring algorithm, the result is not an optimal register allocation. In fact, the result is significantly worse because so many components of the register allocation problem are *implicit* in the coloring heuristic instead of *explicit* in the graph model [25]. If a provably optimal register allocation is desired, the underlying model of the problem must be expressive enough to fully represent all the components of register allocation, not just the interference constraints.

Given an expressive model of the register allocation problem, it may be possible to find an optimal solution. Models based on integer linear programming (ILP) [30, 36, 2, 27, 17], partitioned boolean quadratic programming (PBQP) [20], and multi-commodity network flow (MCNF) [24] have been formulated to represent the register allocation problem. These models are more expressive than the graph coloring model and admit powerful solution techniques. Unfortunately, although these solution techniques provide optimality guarantees, they can require an impractical amount of time to find any solution, much less the optimal solution.

A natural alternative to using an optimal solver is to use a heuristic solver based on an expressive model. This approach has been used successfully with the PBQP and MCNF models, but there remains a significant gap, both in terms of solution quality and compile time, between the heuristic solvers and the optimal solvers. A progressive solver bridges this gap by quickly finding a good solution and then progressively finding better solutions until an optimal solution is found or a preset time limit is reached. The use of progressive solution techniques fundamentally changes how compiler optimizations are enabled. Instead of selecting an optimization level, a programmer *explicitly* trades compilation time for improved optimization.

In this paper we present a *global progressive register allocator* that combines an improved version of the MCNF model with a progressive solution technique based on Lagrangian relaxation. We extend the MCNF model of [24], which is only suitable for local register allocation, to represent the global register allocation problem. Multi-commodity network flow is a natural representation of register allocation; an allocation of a variable is represented by a flow of a commodity through a network. The network is designed so that valid allocations correspond exactly to valid flows and the cost of a flow is precisely the cost of the allocation. In addition to the required interference and capacity constraints, our global MCNF model can explicitly represent the costs of spill code

insertion, register preferences, copy insertion, and constant rematerialization. Unlike some other models of register allocation, our model does not require that a variable be allocated to a single register for its entire lifetime; a variable may move between memory and registers and between registers at any point in the program. The global MCNF model is expressive enough to represent most of the pertinent components of register allocation.

We have developed two heuristic solvers for our model of register allocation. The simultaneous heuristic solver works similarly to second-chance binpacking allocators [42]. It executes a single pass over the control flow graph, maintaining an allocation for all live variables at each program point and evicting variables to memory as necessary to allocate newly defined variables. The iterative heuristic solver works similarly to a single-pass graph coloring heuristic in that it processes variables in a fixed order and does not revisit allocation decisions, but, unlike traditional heuristics, our heuristic solver benefits from the expressiveness of the underlying model. These simple heuristic solvers quickly find solutions competitive with existing heuristic allocators.

Our progressive solver uses the theory of Lagrangian relaxation to compute a lower bound and to guide the search for an optimal solution. The existence of a lower bound not only provides a termination condition, which is when the value of the best solution matches the lower bound, but also allows us to calculate an upper bound on the optimality of an existing solution (e.g., that the current best solution is within 1% of optimal). The Lagrangian relaxation method produces Lagrangian prices, also known as multipliers, which, as the method converges, are used to push our solvers closer to the optimal solution.

We have implemented our global progressive register allocator in the GNU C compiler targeting the Intel x86 architecture. We evaluate the our progressive allocator in terms of code size on a large selection of benchmarks from the SPEC2000, SPEC95, MediaBench, and MiBench benchmark suits. We use code size as a metric because, in addition to being a valuable metric in the embedded community, it has the advantage that it can be accurately evaluated at compile time. This lets us exactly match the predictions of our global MCNF model with the actual output. Compared to a traditional graph allocator, our allocator demonstrates an average initial code size improvement of 3.47% and, with time, progressively improves the code quality to get an average size improvement of 6.84% with six benchmarks demonstrating a code size improvement of more than 10%.

The contributions of this paper are threefold:

- We describe an expressive model for global register allocation based on multi-commodity network flow that explicitly represents important components of register allocation such as spill code insertion, register preferences, copy insertion and constant rematerialization.
- We present a progressive solver that quickly finds a good solution and then progressively improves the solution over time. Our solver can also accurately gauge the optimality of its solutions.
- We present a comprehensive evaluation of our global progressive register allocator detailing its ability to accurately model the register allocation problem, outperform existing allocators, and provide theoretical guarantees on the quality of the solution.

The remainder of this paper is organized as follows. Section 2 describes our model of register allocation in detail, including those extensions necessary to model the global register allocation problem. Section 3 describes our progressive solution procedure. Section 4 details our implementation in the GNU C compiler. Section 5 contains the experimental evaluation of our register allocator. Finally, Section 6 more fully contrasts this work with prior work and Section 7 concludes.

2. MCNF Model of Register Allocation

In this section we describe a model of register allocation based on multi-commodity network flow. We first describe the general MCNF problem and show how to create an expressive model of register allocation for straight-line code using MCNF. We then extend the MCNF model to handle control flow. Finally, we discuss some limitations of the model. Overall, the our global MCNF model explicitly and exactly represents the pertinent components of the register allocation problem.

2.1 Multi-commodity Network Flow

The multi-commodity network flow (MCNF) problem is finding the minimum cost flow of commodities through a constrained network. The network is defined by nodes and edges where each edge has costs and capacities. Without loss of generality, we can also apply costs and capacities to nodes. The costs and capacities can be specific for each commodity, but edges also have *bundle constraints* which constrain the total capacity of the edge. For example, if an edge has a bundle constraint of 2 and commodities are restricted to a single unit of integer flow, at most two commodities can use that edge in any valid solution. Each commodity has a source and sink node such that the flow from the source must equal the flow into the sink. Although finding the minimum cost flow of a single commodity is readily solved in polynomial time, finding a solution to the MCNF problem where all flows are integer is NP-complete [1].

Formally, the MCNF problem is to minimize the costs of the flows through the network:

$$\min \sum_k c^k x^k$$

subject to the constraints:

$$\sum_k x_{ij}^k \leq u_{ij}$$

$$0 \leq x_{ij}^k \leq v_{ij}^k$$

$$\mathcal{N} x^k = b^k$$

where c^k is the cost vector containing the cost of each edge for commodity k, x^k is the flow vector for commodity k where x_{ij}^k is the flow of commodity k along edge (i,j), u_{ij} is the bundle constraint for edge (i,j), v_{ij}^k is an individual constraint on commodity k over edge (i,j), the matrix $\mathcal{N}$ represents the network topology, and the vector b^k contains the inflow and outflow constraints (source and sink information).

2.2 Local Register Allocation Model

Multi-commodity network flow is a natural basis for an expressive model of the register allocation problem. A flow in our MCNF model corresponds to a detailed allocation of that variable. A simplified example of our MCNF model of register allocation is shown in Figure 1. Although simplified, this example demonstrates how our MCNF model explicitly represents spill costs, constant rematerialization, and instruction register usage constraints and preferences.

The commodities of the MCNF model correspond to variables. The design of the network and individual commodity constraints is dictated by how variables are used. The bundle constraints enforce the limited number of registers available and model instruction usage constraints. The edge costs are used to model both the cost of spilling and the costs of register preferences.

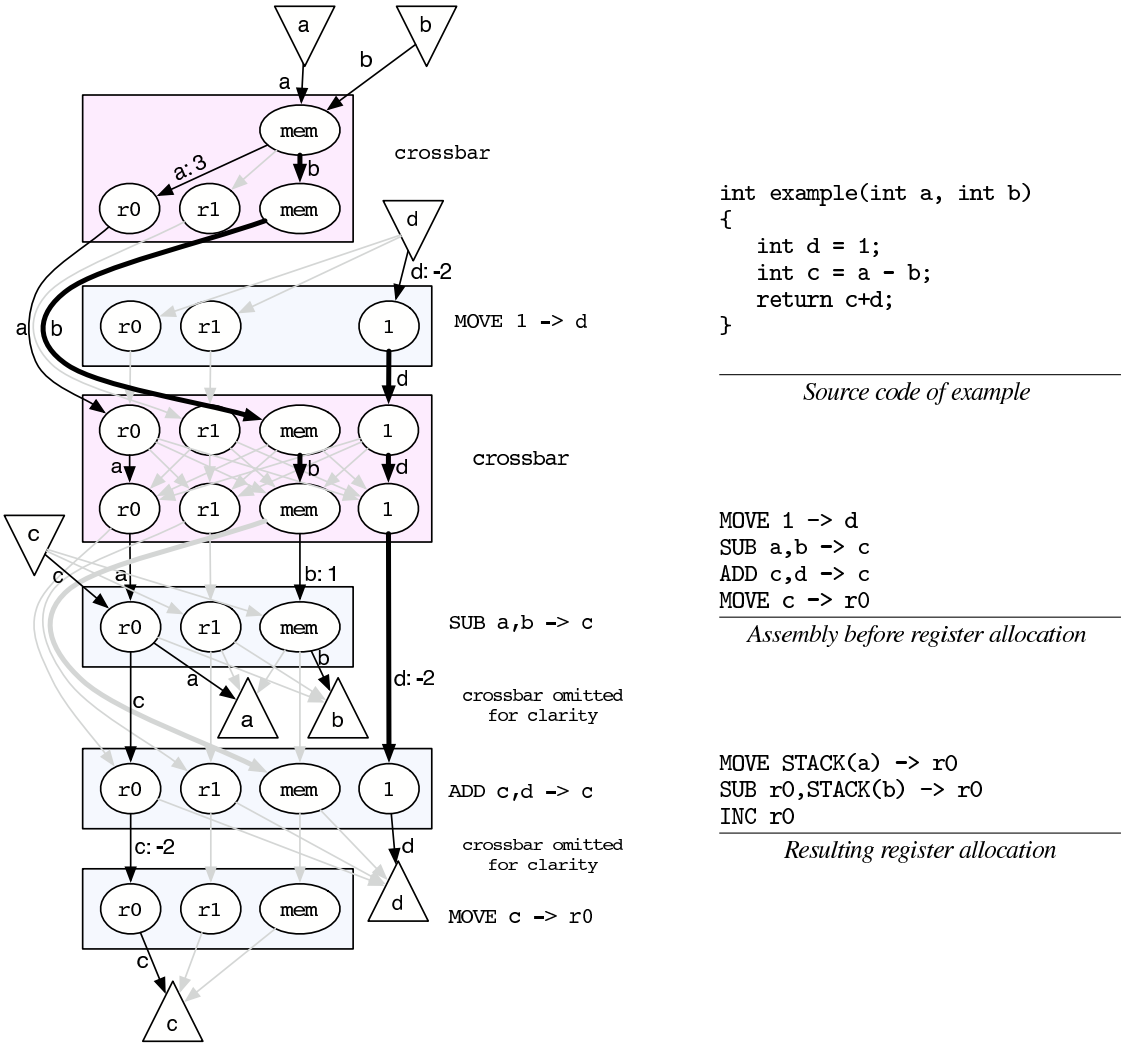

Figure 1. A simplified example of the multi-commodity network flow model of register allocation. Thin edges have a capacity of 1 (as only one variable can be allocated to a register and instructions only support a single memory operand). A thick edge indicates that the edge is uncapacitated. For clarity, edges not used by the displayed solution are in gray and much of the capacity and cost information is omitted. The commodity and cost along each edge used in the solution are shown if the cost is non-zero. In this example the cost of a load is 3, the cost of using a memory operand in the `SUB` instruction is 1, the benefit (negative cost) of allocating c to `r0` in the final `MOVE` instruction is 2 since the move can be deleted in this case. Similarly, allocating d to a constant when it is defined has a benefit of 2. If an operand of the `ADD` instruction is the constant one, then a benefit of 2 is accrued because the more efficient `INC` instruction can be used. The total cost of this solution is -2.

Each node in the network represents an allocation class: a register, constant class, or memory space where a variable's value may be stored. Although a register node represents exactly one register, constant and memory allocation classes do not typically correspond to a single constant or memory location. Instead they refer to a class of constants or memory locations that are all accessed similarly (e.g., constant integers versus symbolic constants).

Nodes are grouped into either instruction or crossbar groups. There is an instruction group for every instruction in the program and a crossbar group for every point between instructions. An instruction group represents a specific instruction in the program and contains a single node for each allocation class that may be used by the instruction. The source node of a variable connects to the network at the defining instruction and the sink node of a variable removes the variable from the network immediately after the last instruction to use the variable. The nodes in an instruction group constrain which allocation classes are legal for the variables used by that instruction. For example, if an instruction does not support memory operands, such as the load of the integer constant one in Figure 1, then no variables are allowed to flow through the memory allocation class node. Similarly, if only a single memory operand is allowed within an instruction, the bundle constraints of the instruction's memory edges are set to 1. This is illustrated in Figure 1 by the thin edges connecting to the memory node of the `SUB` instruction group. Variables used by an instruction must flow through the nodes of the corresponding instruction group. Variables not used by the instruction bypass the instruction into the next crossbar group. This behavior can been seen in the behavior of variables a and b in Figure 1. The flows of these variables bypass the first instruction but are forced to flow through the `SUB` instruction.

Crossbar groups are inserted between every instruction group and allow variables to change allocation classes. For example, the ability to store a variable to memory is represented by an edge

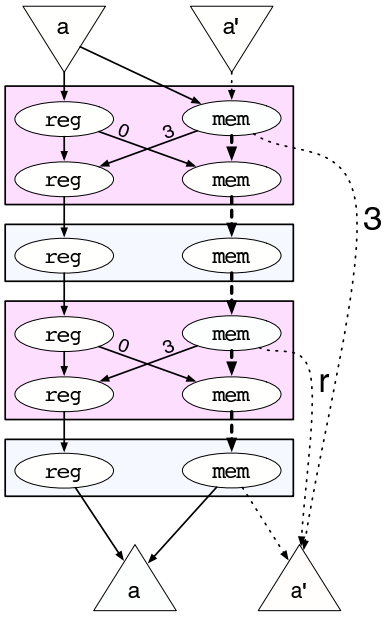

Figure 2. An example of anti-variables. The anti-variable of a, a', is restricted to the memory subnetwork (dashed edges). The edge r is redundant and need not be in the actual network. The cost of the second store can be paid by the first edge. If the r edge is left in the graph, it would have a cost of three, the cost of a store in this example. Multiple anti-variable eviction edges can also be used to model the case where stores have different costs depending on their placement in the instruction stream.

within a crossbar group from a register node to a memory allocation class node. In Figure 1 the variable a, which is assumed to start as a parameter on the stack, flows from the memory node to `r0`, which corresponds to a load. The crossbar groups shown in Figure 1 are full crossbars which means that for some allocations the use of swap instructions, instead of a simple series of move instructions, might be necessary. If swap instructions are not available or are not efficient relative to simple moves, a more elaborate zig-zag crossbar structure can be used.

The cost of an operation, such as a load or move, can usually be represented by a cost on the edge that represents the move between allocation classes. However, this does not accurately reflect the cost of storing to memory. If a variable has already been stored to memory and its value has not changed, it is not necessary to pay the cost of an additional store. That is, values in memory are persistent, unlike those in registers which are assumed to be overwritten.

In order to model the persistence of data in memory, we introduce the notion of anti-variables which are used as shown in Figure 2. An anti-variable is restricted to the memory subnetwork and is constrained such that it cannot coexist with its corresponding variable along any memory edge. An anti-variable can either leave the memory sub-network when the variable itself exits the network or the cost of a store can be paid to leave the memory subnetwork early. There is no cost associated with edges from registers to memory, but for these edges to be usable, the anti-variable must be evicted from memory. The cost of evicting the anti-variable is exactly the cost of a single store. In this way a variable may flow from registers to memory multiple times and yet only pay the cost of a single store (of course, every transition from memory to a register pays the cost of a load). An actual store is only generated for the first move to memory.

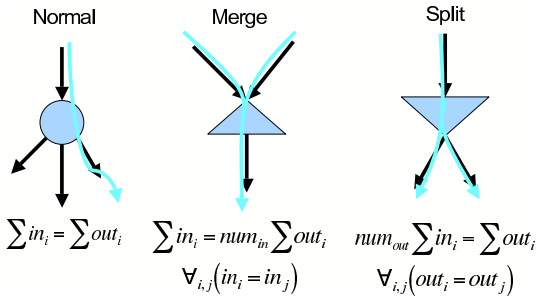

Figure 3. The types of nodes in a global MCNF representation of register allocation. The merge/split nodes not only modify the traditional flow equations with a multiplier, but also require uniformity in the distribution of inputs/outputs.

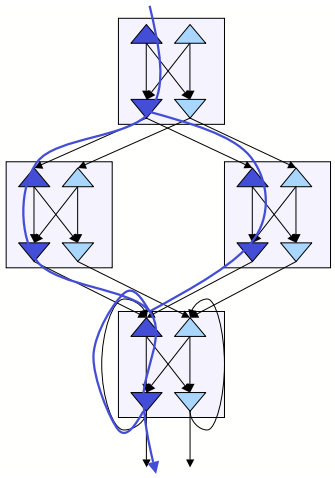

Figure 4. A MCNF based representation of global register allocation with a sample allocation shown with the thicker line. Each block can be thought of as a crossbar where the cost of each edge is the shortest path between a given merge and split node.

2.3 Global Register Allocation Model

Although the described MCNF model is very expressive and able to explicitly model many important components of register allocation, it is unsuitable as a model of global register allocation since it does not model control flow. In order to represent the global register allocation problem, boundary constraints are added to link together the local allocation problems. These constraints are represented by *split* and *merge* nodes as shown in Figure 3.

Similar to normal nodes, split and merge nodes represent a specific allocation class. Merge nodes denote the entry to a basic block. A variable with a flow through a specific merge node is allocated to that allocation class at the entry of the relevant block. The merge property of the merge node, as enforced by the flow equations in Figure 3, requires that a variable be allocated to the same allocation class at the entry of a block as at the exit of all of the predecessors of the block. Similarly, a split node requires that an allocation of a variable at the exit of a block match the allocation at the entry to each of the successors to the block.

More formally, we add the following equality constraint for every commodity k and for every pair $(split, merge)$ of connected split and merge nodes to the definition of the MCNF problem given in Section 2.1:

$$x^k_{in,split} = x^k_{merge,out}$$

Note that split nodes are defined to have exactly one incoming edge and merge nodes to have exactly one outgoing edge. We refer to these constraints as the *boundary constraints*. These constraints replace the normal flow constraint between nodes for split and merge nodes.

A simplified example of a single allocation in the global MCNF model is shown in Figure 4. In this example, the full MCNF representation of each basic block is reduced to a simple crossbar. Unlike the local MCNF model, finding the optimal allocation for a single variable is not a simple shortest path computation. In fact, for general flow graphs the problem is NP-complete (by a reduction from graph coloring).

2.4 Limitations

Our global MCNF model can explicitly model instruction usage constraints and preferences, spill and copy insertion, and constant rematerialization. In addition, our model can model a limited amount of register-allocation driven instruction selection. For example, in Figure 1 the model explicitly encodes the fact that if an operand of the `ADD` instruction is the constant one, a more efficient `INC` instruction can be used. However, the model can not currently represent inter-variable register usage preferences or constraints. That is, the model can not represent a statement like, "if a is allocated to X and b is allocated to Y in this instruction, then a 2 byte smaller instruction can be used." For example, on the x86 a sign extension from a 16-bit variable a to a 32-bit variable b is normally implemented with a 3-byte `movsxw` instruction, but if both a and b are allocated to the register `eax` then a 1-byte `cwde` instruction may be used with the same effect. This saving in code size cannot be exactly represented in our model because edge costs only apply to the flow of a single variable. If the instruction stream was modified so that a move from a to b were performed before the sign extension and the sign extension had b as its only operand, then the model would be capable of exactly representing the cost savings of allocating b to `eax` at the cost of requiring a more constrained instruction stream as input.

Another example where inter-variable register usage preferences are useful is in the modeling of the conversion of a three operand representation of a commutative instruction into a two operand representation. Internally, a compiler might represent addition as $c = a + b$ even though the target architecture requires that one of the source operands be allocated to the same register as the destination operand. Ideally, the model would be able to exactly represent the constraint that one of the source operands, a or b, be allocated identically with c. Converting non-commutative instructions into two operand form does not pose a problem for our model as these instructions can be put into standard form without affecting the quality of register allocation.

On some architectures inter-variable register usage constraints might exist that require a double-width value to be placed into two consecutive registers. The SPARC architecture, for example, requires that 64-bit floating point values be allocated to an even numbered 32-bit floating point register and its immediate successor. Our MCNF model currently is not capable of representing such a constraint.

Unlike traditional allocators, our model does not represent the benefits of move coalescing. Instead, moves are aggressively coalesced before register allocation; the model explicitly represents the benefit of inserting a move so there is no need to leave unnecessary moves in the instruction stream. Put another way, instead of move coalescing, our allocator implements un-coalescing.

An additional limitation of our model is that it assumes that it is never beneficial to allocate the same variable to multiple registers at the same program point. This arises because there is a direct correspondence between the flow of a variable through the network and the allocation of the variable at each program point. The assumption that it will not be beneficial to allocate a variable to multiple registers at the same program point seems reasonable for architectures with few registers. If desired, this limitation can be removed by using a technique similar to how anti-variables are used to model stores.

3. Progressive Solution Procedure

In this section we first explain how we can quickly find a feasible solution to the global MCNF problem using heuristic allocators. We describe two such allocators: an iterative allocator which iterates over all variables, allocating each in its entirety, and a simultaneous allocator which performs a single scan through the control flow graph, allocating all live variables simultaneously. Next we describe how we apply Lagrangian relaxation to the global MCNF problem in order to compute an optimality bound and guide our progressive solver. Finally, we describe how we combine the two heuristic allocators with the output of Lagrangian relaxation in our progressive solver.

3.1 Iterative Heuristic Allocator

The specific form of our global MCNF representation allows us to quickly find a feasible, though possibly low quality, solution using an iterative heuristic allocator. The iterative algorithm allocates variables in some heuristically determined order (such as allocating more frequently referenced variables first). A variable is allocated by traversing the control flow graph in depth first order and computing the shortest path for the variable in each block. Because the blocks are traversed in order, the split nodes at the exit of a processed block will fix the starting point for the shortest path in each successor block. Within each block we will always be able to find a feasible solution because the memory network is uncapacitated. We constrain our shortest-path algorithm to conservatively ignore paths that could potentially make the network infeasible for the variables that still need to be allocated. For example, if an instruction requires a currently unallocated operand to be in a register and there is only one register left that is available for allocation, all other variables would be required to be in memory at that point.

Although we can always find a feasible solution this way, it is unlikely that we will find a good solution since the shortest path computations we perform are independent and are not influenced by the effect that an allocation might have on the global cost of allocation. We can improve upon this heuristic by allocating high priority variables first, using information from the interference graph to prevent variables from initially being allocated to registers that will cause conflicts elsewhere in the partially allocated network, and always charging the cost of a store to compensate for the lack of interference from anti-variables.

The iterative heuristic allocator performs a shortest path computation for every variable v in every block. This shortest path computation is linear in the number of instructions, n, because each block is a topologically ordered directed acyclic graph. Therefore the worst case running time of the algorithm is $O(nv)$.

3.2 Simultaneous Heuristic Allocator

As an alternative to the iterative allocator, we describe a simultaneous allocator which functions similarly to a second-chance bin-packing allocator [42] but uses the global MCNF model to guide eviction decisions. The algorithm traverses the control flow graph in depth first order. For each block, it computes both a forwards and backwards shortest-path computation for every variable. These paths take into account that the entry and exit allocations of a variable may have been fixed by an allocation of a previous block. Having performed this computation, the cost of the best allocation for a

variable at a specific program point and allocation class in a block can be easily determined by simply summing the cost of the shortest paths to the corresponding node from the source and sink of the given variable.

After computing the shortest paths, the algorithm scans through the block, maintaining an allocation for every live variable. The allocations of live-in variables are fixed to their allocations at the exit of the already allocated predecessor blocks. At each level in the network, each variable's allocation is updated to follow the previously computed shortest path to the sink node of the variable (the common case is for a variable to remain in its current location). If two variables' allocations overlap, the conflict is resolved by evicting one of the variables to an alternative allocation.

When a variable is defined, the minimum cost allocation is computed using the shortest path information and a calculation of the cost of evicting any variable already allocated to a desired location. The cost of evicting a variable from its current location is computed by finding the shortest path in the network to a valid eviction edge (an edge from the previous allocation to the new allocation). In computing this shortest path we avoid already allocated nodes in the graph. That is, we do not recursively evict other variables in an attempt to improve the eviction cost. The shortest path is not necessarily a simple store immediately before the eviction location. For example, if the defining instruction of the variable being evicted supports a memory operand, it might be cheaper to define the variable into memory instead of defining it into a register and performing a more costly store later. When a variable is evicted to memory the cost of the corresponding anti-variable eviction is also computed and added to the total eviction cost. When choosing a variable to evict we break ties in the eviction cost by following the standard practice and choosing the variable whose next use is farthest away [4, 30].

Currently, only intra-block evictions are implemented; the earliest a variable can be evicted is at the beginning of the current block. Because of this limitation, this allocator performs more poorly as the amount of control flow increases since poor early allocation decisions can not be undone later in the control flow graph.

The simultaneous heuristic allocator, like the iterative algorithm, must compute shortest paths for every variable v in every block. Unlike the iterative algorithm, the simultaneous allocator does not need to compute each path successively and instead can compute all paths in the same pass. However, although this results in an empirical performance improvement, the worst case asymptotic running time remains $O(nv)$.

3.3 Lagrangian Relaxation

Ideally, we would like to build a solution from simple shortest path computations. Each individual variable's shortest path would need to take into account not only the immediate costs for that variable, but also the marginal cost of that allocation with respect to all other variables. Lagrangian relaxation provides a formal way of computing these marginal costs.

Lagrangian relaxation is a general solution technique [1, 29] that removes one or more constraints from a problem and integrates them into the objective function using Lagrangian multipliers resulting in a more easily solved Lagrangian subproblem. In the case of multi-commodity network flow, the Lagrangian subproblem is to find a price vector w such that $L(w)$ is maximal, where $L(w)$ is defined:

$$L(w) = \min \sum_k c^k x^k + \sum_{(i,j)} w_{ij} \left(\sum_k x_{ij}^k - u_{ij} \right) \tag{1}$$

which can be rewritten as:

$$L(w) = \min \sum_k \sum_{(i,j)} \left(c_{ij}^k + w_{ij} \right) x_{ij}^k - \sum_{(i,j)} w_{ij} u_{ij} \tag{2}$$

subject to

$$x_{ij}^k \geq 0$$

$$\mathcal{N} x^k = b^k$$

$$\sum_i x_{i,split}^k = \sum_j x_{merge,j}^k$$

The bundle constraints have been integrated into the objective function. If an edge x_{ij} is over-allocated then the term $\sum_k x_{ij}^k - u_{ij}$ will increase the value of the objective function, making it less likely that an over-allocated edge will exist in a solution that minimizes the objective function. The w_{ij} terms are the Lagrangian multipliers, called prices in the context of MCNF. The prices, w, are arguments to the subproblem and it is the flow vectors, x^k, that are the free variables in the minimization problem. The Lagrangian subproblem is still subject to the same network and individual flow constraints as in the MCNF problem. As can be seen in (2), the minimum solution to the Lagrangian subproblem decomposes into the minimum solutions of the individual single commodity problems.

Unfortunately, in our global MCNF model the individual single commodity problem remains NP-complete because of the boundary constraints. Fortunately, the boundary constraints can also be brought into the objective function using Lagrangian multipliers:

$$L(w) = \min \sum_k \sum_{(i,j)} \left(c_{ij}^k + w_{ij} \right) x_{ij}^k - \sum_{(i,j)} w_{ij} u_{ij} + \sum_{(split,merge)} w_{split,merge}^k \left(x_{merge,out}^k - x_{in,split}^k \right) \tag{3}$$

subject to

$$x_{ij}^k \geq 0$$

$$\mathcal{N} x^k = b^k$$

Since there are no normal flow constraints between split and merge nodes, the solution to (3) is simply a set of disconnected single commodity flow problems.

The function $L(w)$ has several useful properties [1]. Let $L^* = max_w L(w)$, then L^* provides a lower bound for the optimal solution value. Furthermore, a solution, x, to the relaxed subproblem which is feasible in the original MCNF problem is likely to be optimal. In fact, if the solution obeys the complementary slackness condition, it is provably optimal. The complementary slackness condition simply requires that any edge with a non-zero price be used to its full capacity in the solution. Intuitively, this means that given the choice of two identically priced shortest paths, the path with the lower unpriced cost results in a better solution.

We solve for L^* using an iterative subgradient optimization algorithm. At a step q in the algorithm, we start with a price vector, w^q, and solve $L(w^q)$ for x^k to get an optimal flow vector, y^k, by performing a multiple shortest paths computation in each block. We then update w using the rules:

$$w_{ij}^{q+1} = \max \left(w_{ij}^q + \theta_q \left(\sum_k y_{ij}^k - u_{ij} \right), 0 \right)$$

$$w_{split,merge}^k{}^{q+1} = w_{split,merge}^k{}^q + \theta_q \left(y_{merge,out}^k - y_{in,split}^k \right)$$

where θ_q is the current step size. This algorithm is guaranteed to converge if θ_q satisfies the conditions:

$$\lim_{q \to \infty} \theta_q = 0$$

$$\lim_{q \to \infty} \sum_{i=1}^{q} \theta_i = \infty$$

An example of a method for calculating a step size that satisfies these conditions is the ratio method, $\theta_q = 1/q$. More sophisticated techniques to calculate the step size and update the prices [33, 3] are beyond the scope of this paper.

Although the iterative subgradient algorithm is guaranteed to converge, it is not guaranteed to do so in polynomial time. Furthermore, L^* does not directly lead to an optimal solution of the original, unrelaxed global MCNF problem. However, the Lagrangian prices can be used to effectively guide the allocation algorithms towards better solutions and to provide optimality guarantees.

3.4 Progressive Solver

We combine our allocation heuristics with the Lagrangian relaxation technique to create a progressive solver. The solver first finds an initial solution in the unpriced network. Then, in each iteration of the iterative subgradient algorithm, the current set of prices are used to find another feasible solution. When finding solutions in the priced network, the allocation heuristics compute shortest paths using edge and boundary prices in addition to edge costs. Global information, such as the interference graph, is not used except to break ties between identically priced paths. Instead, the allocators rely exclusively on the influence of the prices in the network to account for the global effect of allocation decisions.

Both allocators attempt to build a feasible solution to the global MCNF problem whose cost in the priced network is as close as possible to the cost of the unconstrained solution found during the update step of the subgradient algorithm. If the algorithm is successful and the found solution obeys the complementary slackness condition, then the solution is provably optimal. When selecting among similarly priced allocation decisions, we can increase the likelihood that the solution will satisfy the complementary slackness condition by favoring allocations with the lowest unpriced cost.

There are several factors that prevent the allocation algorithms from finding the optimal solution given a priced network. Until the iterative subgradient method has fully converged, the prices in the network are only approximations. As a result, we may compute a shortest path for a variable that would not be a shortest path in a network with fully converged prices. The simultaneous allocator is less sensitive to this effect since it can undo bad allocation decisions. However, the values of the boundary prices are critical to the performance of the simultaneous allocator as allocation decisions get fixed at block boundaries.

A potentially more significant impediment to finding an optimal solution is that the lower bound computed using Lagrangian relaxation converges to the value of the optimal solution of the global MCNF problem without integer constraints. If the gap between the value of the solution to the integer problem and the linear problem is nonzero, we will not be able to prove the optimality of a solution. Fortunately, as we show in Section 5.2, this gap is rarely nonzero.

Even given perfectly converged prices and an *a priori* knowledge that the integrality gap is zero, the allocation problem remains difficult. Both allocators must choose among identically priced allocations, not all of which may be valid allocations in an optimal solution. Again, the simultaneous allocator is somewhat insulated from this difficulty since it can undo bad decisions within a block, but it still must rely upon the value of the boundary prices to avoid locally good, globally poor, allocation decisions.

The challenges faced by the allocators in converting a priced network into an optimal register allocation are not unexpected given the NP-completeness of the problem. However, as we shall see, as the iterative subgradient algorithm converges, the quality of solutions found by the allocation heuristics improve and the lower bound on the optimal solution value increases resulting in provably optimal or near-optimal solutions.

4. Implementation

We have implemented our global MCNF allocation framework as a replacement for the register allocator in `gcc` 3.4.3 when targeting the Intel x86 architecture. Before allocation, we execute a preconditioning pass which aggressively coalesces moves and translates instructions that are not in an allocable form. For example, the compiler represents instructions as three operand instructions even though the architecture only supports two operand instructions. If all three operands are live out of the instruction, it is not possible to allocate these three variables to distinct registers and still generate an x86 two operand instruction. The preconditioning pass translates such instructions so that two of the three operands are the same variable.

We next build a global MCNF model for the procedure as described in Section 2. In our model, crossbars are represented as zig-zags since `gcc` does not support the generation of the x86 swap instruction. We simplify the network by only permitting loads and stores of a variable to occur at block boundaries and after a write to the variable (for a store) or before a read of the variable (for a load). This simplification does not change the value of the optimal solution.

We use code size as the cost metric in our model. This metric has the advantage that it can be perfectly evaluated at compile time and exactly represented by our model. We assume a uniform memory cost model. Specifically, we assume that spilled variables will always fit in the 128 bytes below the current frame pointer unless this space is already fully reserved for stack allocated data (such as arrays). As a result, for some large functions that spill more than 32 values the model is inaccurate. We only model constant rematerialization for integer constants. Although it is not required by the architecture, `gcc` requires 64-bit integer values to be allocated to consecutive registers. Since our model currently does not support such constraints, we ignore such values (resulting in all such variables being allocated to memory and fixed up by the reload pass).

We run both the iterative and simultaneous allocators on the initial unpriced network and then for each step of the iterative subgradient algorithm we apply only the simultaneous allocator to the priced network. In addition to being faster, the simultaneous allocator generally performs better than the iterative allocator once the Lagrangian prices start to converge. However, the iterative allocator does better on the unpriced graph because it allocates variables in order of decreasing priority. A sample of the behavior of both allocators is shown in Figure 5.

After running our solver, we insert the appropriate moves, stores, and loads and setup a mapping of variables to registers. The `gcc` reload pass is then run which applies the register map and modifies the instruction stream to contain only references to physical registers. This pass will also fix any illegal allocations that our allocator might make if the model of register preferences and usage constraints is not correct by generating additional fixup code (this is not common).

5. Results

In this section we evaluate our global MCNF model and progressive allocator on a large selection of benchmarks from the SPEC2000, SPEC95, MediaBench, and MiBench benchmark suits. Combined,

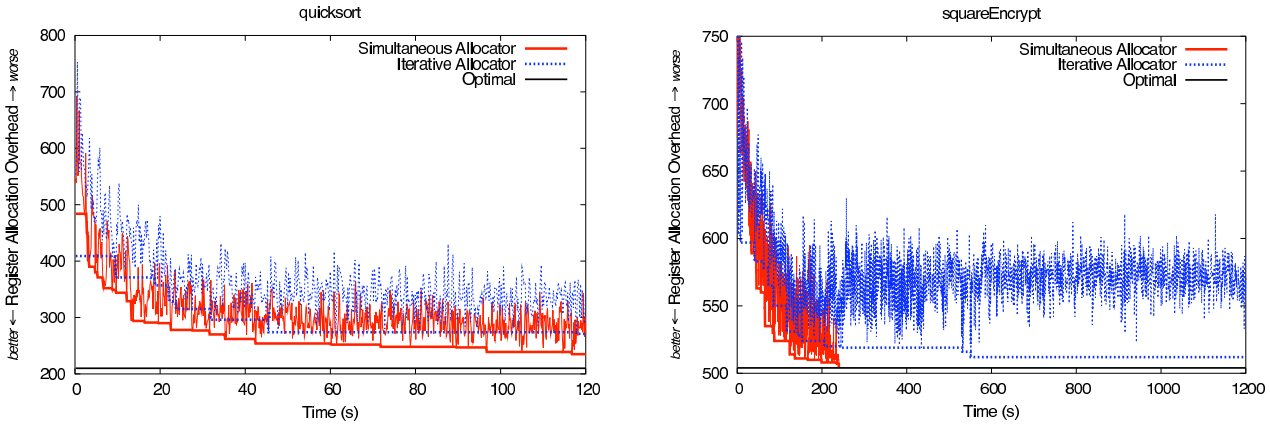

Figure 5. The behavior of the two heuristic allocators as the Lagrangian prices converge executed on a 2.8Ghz Pentium 4 with 1GB of RAM. The `squareEncrypt` function from the pegwit benchmark consists of a single basic block and has 378 instructions, 150 variables, and an average register pressure of 4.99. The `quicksort` function is spread across 57 blocks, has 236 instructions, 58 variables, and an average register pressure of 3.14. Approximately a third of the final size of both functions is due to register allocation overhead. The iterative allocator performs better initially, but as the Lagrangian prices converge the simultaneous allocator performs better. In the case of the `squareEncrypt` function, which has no control flow, the simultaneous allocator finds an optimal solution in less than a quarter of the time it takes the CPLEX solver. Neither allocator succeeded in finding an optimal allocation for `quicksort` before CPLEX found the optimal solution at 112 seconds.

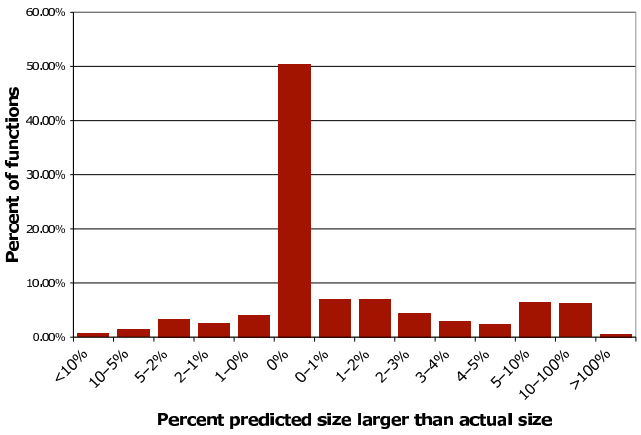

Figure 6. An evaluation of the accuracy of the global MCNF model that compares the predicted size after register allocation to the actual size.

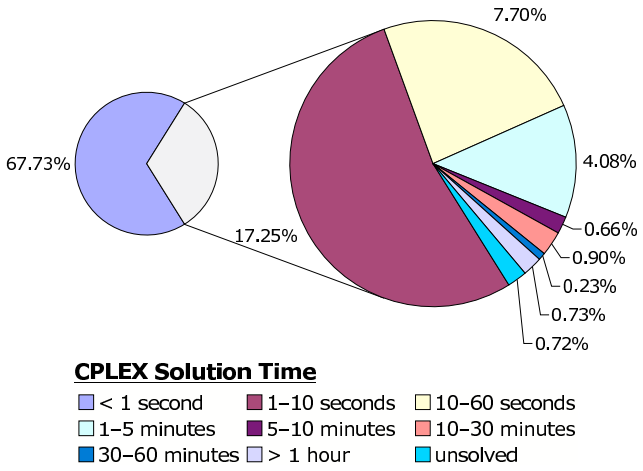

Figure 7. The percentage of functions for which CPLEX could find an optimal solution to the global MCNF representation of register allocation within in a given time limit. A small number of functions (0.72% of the total) could not be solved within 12 hours.

these benchmarks contain more than 10,000 functions. First we validate the accuracy of our model since an accurate model is vital to the performance of our allocator. We then analyze the complexity and difficulty of our global MCNF representation using standard integer linear programming solution techniques. Next we evaluate the quality of our solutions in terms of code size. Because our concern is with evaluating our model and our solver, all size results are taken immediately after the register allocation pass (including `gcc`'s reload pass) to avoid noise from downstream optimizations. Although we explicitly optimize for size, not speed, we also evaluate the performance of code compiled with our progressive register allocator. Finally, we analyze the progressiveness and optimality of our solver as well as its runtime performance.

5.1 Accuracy of the Model

We evaluate the accuracy of the model by comparing the predicted size of a function after ten iterations of our progressive algorithm to the actual size of the function immediately after register allocation. As shown in Figure 6, approximately half of the compiled functions have their size exactly predicted and more than 70% of the functions have their size predicted to within 2% of the actual size. Although these results validate the model to some extent, they also show that the model needs further refinement.

The biggest cause of under-prediction is the uniform memory cost model. Most of the severely under-predicted functions spill more variables than fit in the first 128 bytes of the frame resulting in incorrectly predicted costs in the model for memory operations. The biggest cause of the most severe over-predictions is `gcc`'s instruction sizing function inaccurately reporting the size of certain floating point instructions prior to register allocation.

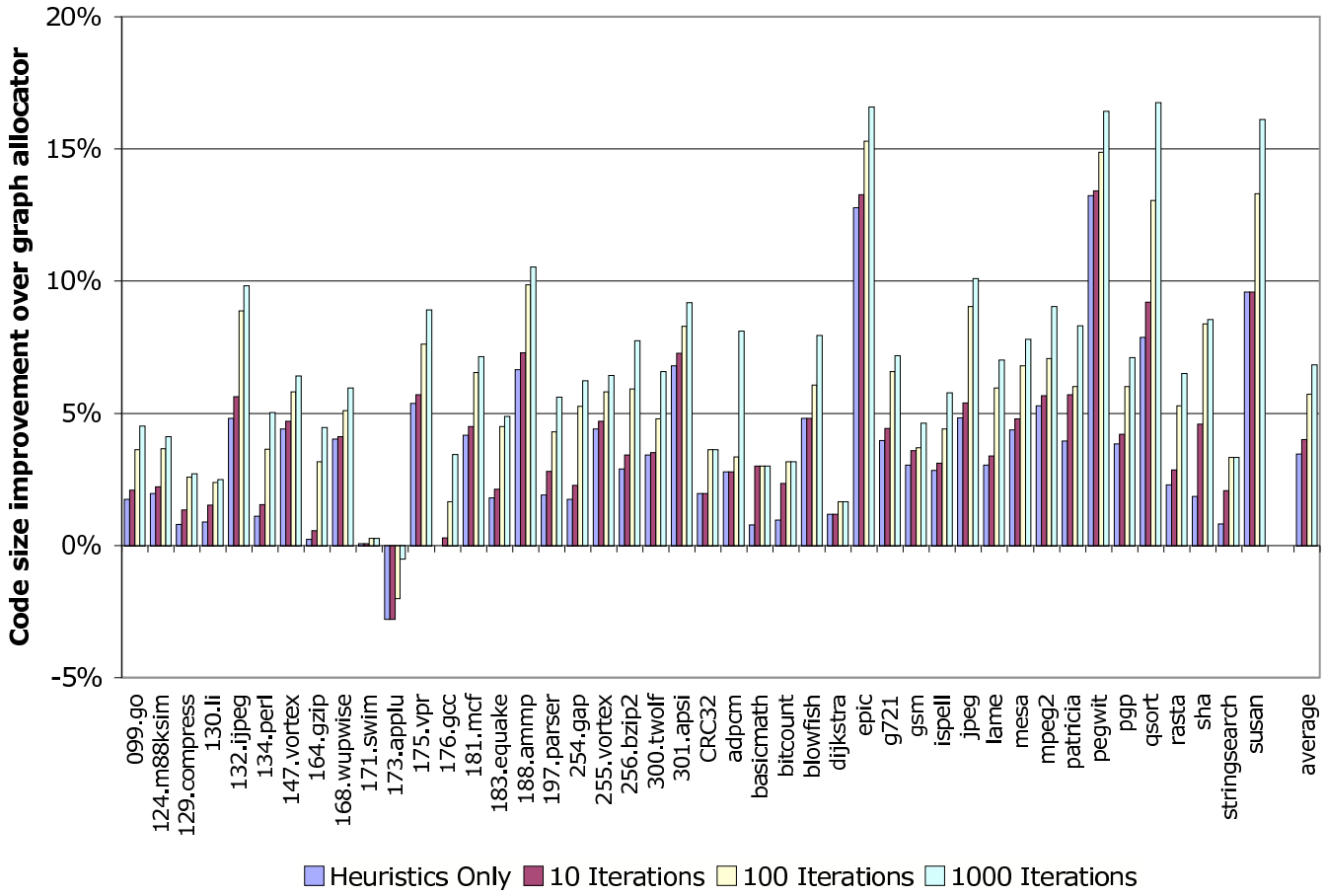

Figure 8. Code size improvement with our allocator compared to a standard iterative graph coloring allocator. All benchmarks were compiled using the `-Os` optimization flag. Note the improvement over time with our allocator. The benchmark `qsort` had the largest improvement with a size improvement of 16.75% after 1000 iterations.

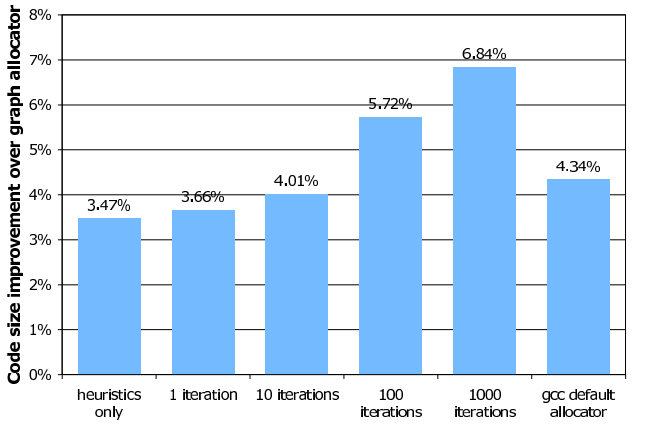

Figure 9. Average code size improvement over all the benchmarks relative to the graph allocator. The highly tuned default allocator outperforms both the graph allocator and our initial allocation.

5.2 Global MCNF Problem Complexity

In order to ascertain the complexity and difficulty of solving our global MCNF problems, we solved the problems corresponding to all the functions in our benchmark suite using version 9.0 of the ILOG CPLEX solver [22]. A text representation of the problem was generated by the compiler and then solved by CPLEX on a 2.8Ghz Pentium 4 with 1GB of RAM. Although 98% of the functions could be solved to optimality in less than ten minutes, it took more than two weeks to process all of the functions. Furthermore, for a handful of functions (0.7%) CPLEX either ran out of memory or failed to find a solution in less than 12 hours. It's worth noting that CPLEX's performance at solving our representation of the register allocation problem roughly corresponds to the performance of CPLEX solving an ILP representation of the register allocation problem [17]. However, it is hard to make a direct comparison since different benchmarks, target architectures, and hardware are used.

The CPLEX solver first solves the linear relaxation of the global MCNF problem and then performs a sophisticated search for an integer solution to the problem during which it may find suboptimal integer solutions. On average, more than half the solve time (54%) is spent solving the linear relaxation. This implies that CPLEX is not appropriate for a progressive solution technique since a significant time commitment is required before any feasible solution is obtained. The dominance of the linear relaxation solver indicates that the resulting bound is likely very useful in pruning the search space during the search for integer solutions. In fact, for more than 99% of the functions for which CPLEX could find a solution, the value of the linear relaxation exactly equalled the value of the integer solution. That is, the integrality gap is zero. This means that for these functions the Lagrangian relaxation converges to an exact lower bound.

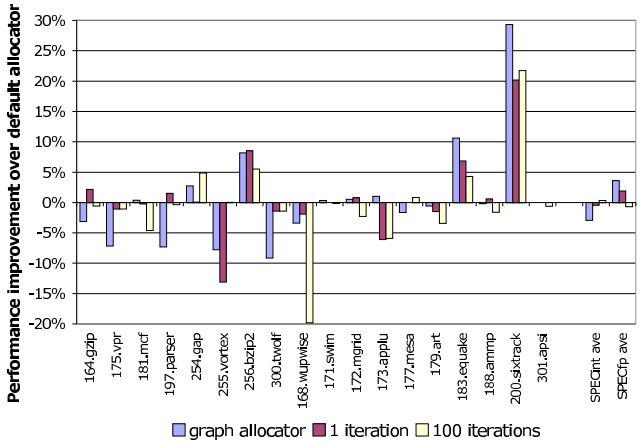

Figure 10. Performance improvement of the graph allocator and our progressive allocator relative to the default allocator. The benchmarks were compiled with the `-Os` optimization flag and executed using a 2.53Ghz Pentium 4 with 2GB of RAM. The Fortran 90 and C++ SPEC benchmarks are omitted due to lack of language support. The graph allocator could not successfully compile 176.gcc, 186.crafty, or 253.perlbmk.

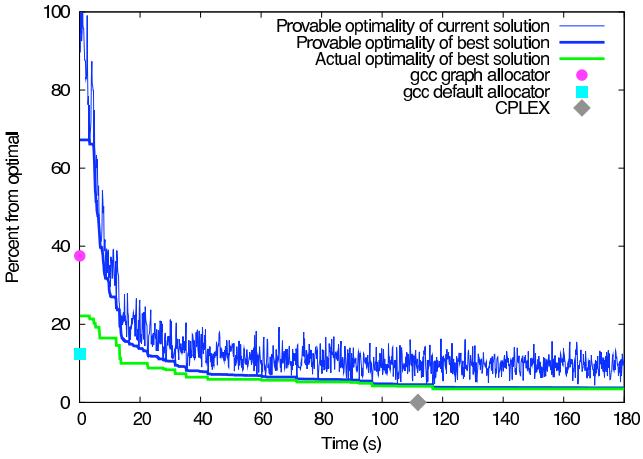

Figure 11. The progressive behavior of our solver when compiling the function `quicksort` on a 2.8Ghz Pentium 4 with 1GB of memory. The convergence of proven optimality and actual optimality indicates that our lower bound is very close to the optimal value.

5.3 Solution Quality

The default `gcc` register allocator divides the register allocation process into local and global passes. In the local pass, only variables that are used in a single basic block are allocated. After local allocation, the remaining variables are allocated using a single-pass graph coloring algorithm. We compare our allocator to the standard iterative graph coloring allocator that can be enabled by passing `-fnew-ra` to `gcc`. This allocator implements standard graph coloring register allocation [10] with iterative coalescing [19] and interference region spilling [5]. The graph allocator generally does not perform as well as the highly-tuned default allocator. Although the default allocator is algorithmically simple, it benefits from decades of development and improvement.

5.3.1 Code Size

As shown in Figure 8 and Figure 9, our heuristic allocator, which runs both the iterative and simultaneous allocators and takes the best result, outperforms the graph allocator on all but one benchmark with an average improvement in code size of 3.47%. As expected, as more time is alloted for compilation, our allocator does progressively better with average code size improvements of 3.66%, 4.01%, 5.72%, and 6.84% for one, ten, 100, and 1000 iterations respectively. Our allocator does not initially do as well as the default allocator; at first we outperform the default allocator on only six benchmarks. However, we outperform even the default allocator on every benchmark when we run our algorithm for 1000 iterations. On average, we outperform the default allocator with only 100 iterations, as shown in Figure 9.

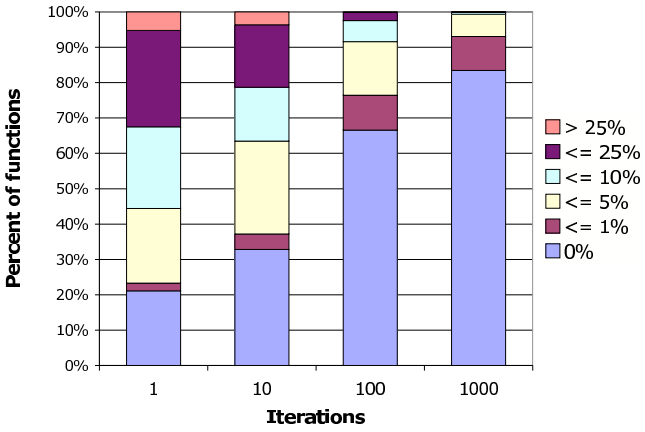

Figure 12. The proven optimality of solutions as the progressive allocator executes more iterations. The found solution is provably within 5% of optimal for 44.4%, 63.5%, 91.6%, and 99.4% of the functions after 1, 10, 100, and 1000 iterations respectively.

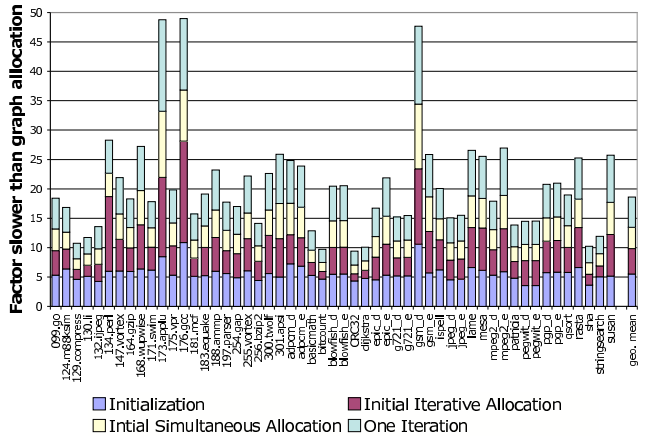

Figure 13. The slowdown of our register allocator relative to the graph allocator. On average, we take about 10 times longer than the graph allocator to find a solution.

5.3.2 Code Performance

We have also evaluated the execution speed of benchmarks from the SPEC2000 benchmark suite when compiled with our allocator and optimized for code size. We compare both our progressive allocator run with a single iteration and with 100 iterations and the graph allocator to the default allocator in Figure 10. Not surprisingly given that we are optimizing for code size, the results are mixed. However, it is important to note that the graph allocator does not do significantly better than our allocator (in fact, in many cases it does worse). This implies that the larger code generated by the

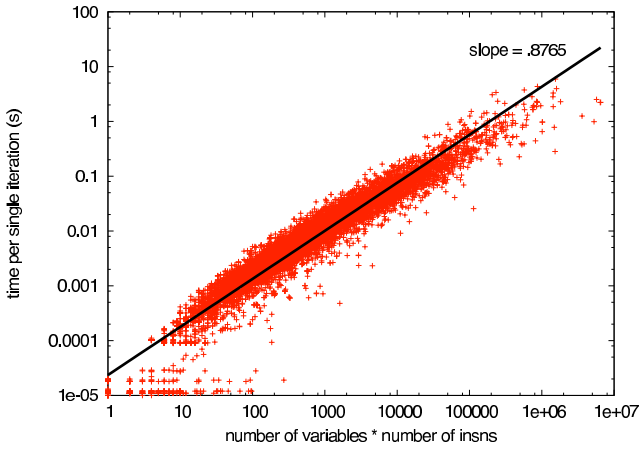

Figure 14. The time spent on a single iteration as a function of the number of variables and instructions in a function. The asymptotic standard error of the fit of the slope is .262%.

graph allocator is not due to the graph allocator optimizing for performance at the expense of code size.

5.4 Progressiveness of Solver

The progressive improvement in quality demonstrated by our allocator can be seen in Figures 8 and 9. After 1000 iterations, our solver improves upon our initial solution by as much as 9.6% (3.5% on average). A more detailed look at the behavior of the solver is shown in Figure 11 which compares the optimality of solutions found by our allocator, `gcc`'s graph and default allocators, and by solving the global MCNF problem using CPLEX. As expected from the benchmark data, our initial heuristic finds a better solution than the graph allocator, but not as good a solution as the default allocator. With time, however, our allocator finds a better solution than the default allocator. CPLEX finds an optimal solution in 112 seconds, at which point our solver has found a solution that is 5.12% from optimal. A key advantage of our solver over CPLEX is that if compilation were limited to 100 seconds, then CPLEX would have no solution while our solver would have a solution that, while not optimal, is better than any initial heuristic solution.

5.5 Optimality of Solutions

Ideally, a progressive solver is guaranteed to eventually find an optimal solution. Although our solver has no such guarantee, the Lagrangian relaxation technique lets us prove an upper bound on the optimality of the solution. As the iterative subgradient algorithm used to solve the Lagrangian relaxation converges, both a better lower bound on the optimal value of the problem is found and the quality of the solutions found by the heuristic solver improves. Consequently, as shown in Figure 12, as more iterations are executed, a larger percentage of compiled function are proven optimal. After 1000 iterations, we have found a provably optimal register allocation for 83.47% of the functions and 99.35% of the functions have a solution that is provably within 5% of optimal.

5.6 Solver Performance

The worst case running time of $O(nv)$ of our heuristic solvers combined with the early developmental stage of our implementation leads us to expect that our allocator will not perform as well as existing allocators in terms of compilation time. Indeed, as shown in Figure 13, allocating with just one heuristic solver is almost ten times slower than the graph allocator, and a single iteration is clearly more expensive than an entire allocation in the graph allocator. These slowdowns are relative to the time spent by the graph allocator which accounts for between 10.5% and 46% of the total compile time (27.5% on average). The graph allocator is, on average, about four times slower than the default allocator. Although it is likely that these results will improve when we optimize our implementation, the $O(nv)$ running time of a single iteration seems to be an accurate characterization of the running time of the algorithm as shown by the log-log plot in Figure 14.

6. Related Work

Traditional graph coloring register allocation was first comprehensively described by Chaitin [12]. Many improvements have been made to the basic Chaitin allocator to improve spill code generation [10, 15, 5, 6], better address features of irregular architectures [8, 9, 40], represent register preferences [13, 28], and exploit program structure [11, 14, 32].

Linear scan register allocation was originally used to solve the local register allocation problem [4, 23, 31, 21, 30]. More recently, variants of linear scan have been used as fast alternatives to graph coloring for dynamic compilation [38, 42, 43].

None of the graph-coloring or linear-scan based algorithms support progressive compilation nor do they compute any sort of optimality bounds. Furthermore, they contain no explicit expressive model of the register allocation problem.

Register allocators that solve the register allocation problem (or some simplification) optimally have been implemented by

- performing a guided exhaustive search through the configuration space, potentially using exponential space [21, 26, 34],
- exploiting the bounded treewidth property of most programs to solve the register sufficiency problem optimally [7, 41, 37]
- formulating register allocation as an integer linear program and then solving this formulation optimally using powerful solution techniques [30, 17, 36, 2], and
- formulating register allocation as a partitioned boolean quadratic optimization problem which can than be solved either approximately or optimally [20].

None of these optimal algorithms support progressive compilation; in some cases the algorithms do not scale beyond unrealistically small programs. However, the techniques based on integer linear programming are more expressive than our global MCNF model and have been extended to additionally model elements of code generation and instruction scheduling [36, 35, 39].

Network flows have been used to model and solve a wide range of problems [1]. Single commodity network flows have been used to allocate registers for improved energy efficiency [18]. A 2-commodity network flow formulation solved using standard ILP techniques has been used to solve the local register allocation problem on a regular architecture [16].

7. Conclusion

This paper describes a *global progressive register allocator*, a register allocator that uses an expressive model of the register allocation problem to quickly find a good allocation and then progressively find better allocations until a provably optimal solution is found or a preset time limit is reached. Our global MCNF model effectively captures the important components of register allocation and our progressive solution technique successfully improves allocation quality as more time is permitted for compilation. The progressive nature of our allocator bridges the gap between fast, but suboptimal, heuristic allocators and optimal, but slow, ILP allocators allowing a programmer to explicitly trade compilation time for improved optimization.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

[2] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *Proc. of the ACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 243–253. ACM Press, 2001.

[3] B. M. Baker and J. Sheasby. Accelerating the convergence of subgradient optimisation. *European Journal of Operational Research*, 117(1):136–144, August 1999.

[4] L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] P. Bergner, P. Dahl, D. Engebretsen, and M. T. O'Keefe. Spill code minimization via interference region spilling. In *ACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 287–295, 1997.

[6] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proc. of the ACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 258–263. ACM Press, 1989.

[7] H. Bodlaender, J. Gustedt, and J. A. Telle. Linear-time register allocation for a fixed number of registers. In *Proc. of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 574–583. Society for Industrial and Applied Mathematics, 1998.

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

[9] P. Briggs, K. D. Cooper, and L. Torczon. Coloring register pairs. *ACM Lett. Program. Lang. Syst.*, 1(1):3–13, 1992.

[10] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[11] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of theACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 192–203. ACM Press, 1991.

[12] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. of the SIGPLAN symposium on Compiler Construction*, pp. 98–101. ACM Press, 1982.

[13] F. Chow and J. Hennessy. Register allocation by priority-based coloring. In *Proc. of the SIGPLAN symposium on Compiler Construction*, pp. 222–232, New York, NY, 1984. ACM Press.

[14] K. Cooper, A. Dasgupta, and J. Eckhardt. Revisiting graph coloring register allocation: A study of the Chaitin-Briggs and Callahan-Koblenz algorithms. In *Proc. of the Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, October 2005.

[15] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *Proc. of the 1998 International Compiler Construction Conference*, 1998.

[16] M. Farach and V. Liberatore. On local register allocation. In *Proc. of the ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[17] C. Fu, K. Wilken, and D. Goodwin. A faster optimal register allocator. *The Journal of Instruction-Level Parallelism*, 7:1–31, January 2005.

[18] C. H. Gebotys. Low energy memory and register allocation using network flow. In *Proc. of the 34th conference on Design Automation*, pp. 435–440. ACM Press, 1997.

[19] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[20] U. Hirnschrott, A. Krall, and B. Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *JMLC*, pp. 202–213, 2003.

[21] W. Hsu, C. N. Fisher, and J. R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Trans. Softw. Eng.*, 15(10):1252–1260, 1989.

[22] ILOG CPLEX. `http://www.ilog.com/products/cplex`.

[23] K. Kennedy. *Design and optimization of compilers*, Index register allocation in straight line code and simple loops, pp. 51–63. Prentice-Hall, 1972.

[24] D. Koes and S. C. Goldstein. A progressive register allocator for irregular architectures. In *Proc. of the International Symposium on Code Generation and Optimization*, pp. 269–280, Washington, DC, 2005. IEEE Computer Society.

[25] D. Koes and S. C. Goldstein. An analysis of graph coloring register allocation. Technical Report CMU-CS-06-111, Carnegie Mellon University, March 2006.

[26] D. J. Kolson, A. Nicolau, N. Dutt, and K. Kennedy. Optimal register assignment to loops for embedded code generation. *ACM Transactions on Design Automation of Electronic Systems.*, 1(2):251–279, 1996.

[27] T. Kong and K. D. Wilken. Precise register allocation for irregular architectures. In *Proc. of the ACM/IEEE International Symposium on Microarchitecture*, pp. 297–307. IEEE Computer Society Press, 1998.

[28] A. Koseki, H. Komatsu, and T. Nakatani. Preference-directed graph coloring. *SIGPLAN Not.*, 37(5):33–44, 2002.

[29] C. Lemaréchal. *Computational Combinatorial Optimization: Optimal or Provably Near-Optimal Solutions*, volume 2241 of *Lecture Notes in Computer Science*, chapter Lagrangian Relaxation, pp. 112–156. Springer-Verlag Heidelberg, 2001.

[30] V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. In *Proc. of the International Compiler Construction Conference* , volume 1575 of *Lecture Notes in Computer Science*. Springer, 1999.

[31] F. Luccio. A comment on index register allocation. *Commun. ACM*, 10(9):572–574, 1967.

[32] G. Lueh, T. Gross, and A. Adl-Tabatabai. Fusion-based register allocation. *ACM Trans. Program. Lang. Syst.*, 22(3):431–470, 2000.

[33] A. R. Madabushi. Lagrangian relaxation / dual approaches for solving large-scale linear programming problems. Master's thesis, Virginia Polytechnic Institute and State University, February 1997.

[34] W. M. Meleis and E. S. Davidson. Optimal local register allocation for a multiple-issue machine. In *Proc. of the 8th international conference on Supercomputing*, pp. 107–116. ACM Press, 1994.

[35] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical report, Stanford University, Stanford, CA, 1995.

[36] M. Naik and J. Palsberg. Compiling with code-size constraints. In *Proc. of the conference on Languages, Compilers and Tools for Embedded Systems*, pp. 120–129. ACM Press, 2002.

[37] M. Ogawa, Z. Hu, and I. Sasano. Iterative-free program analysis. In *Proc. of International Conference on Functional Programming*, pp. 111–123. ACM Press, 2003.

[38] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[39] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proc. of ACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 1–11, New York, NY, 1996. ACM Press.

[40] M. D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. *SIGPLAN Not.*, 39(6):277–288, 2004.

[41] M. Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998.

[42] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proc. of the ACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 142–151, New York, NY, 1998. ACM Press.

[43] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05: Proc. of ACM/USENIX International Conference on Virtual Execution Environments*, pp. 132–141, New York, NY, 2005. ACM Press.