# A Progressive Register Allocator for Irregular Architectures

David Koes and Seth Copen Goldstein
Computer Science Department
Carnegie Mellon University
{dkoes,seth}@cs.cmu.edu

## Abstract

*Register allocation is one of the most important optimizations a compiler performs. Conventional graph-coloring based register allocators are fast and do well on regular, RISC-like, architectures, but perform poorly on irregular, CISC-like, architectures with few registers and non-orthogonal instruction sets. At the other extreme, optimal register allocators based on integer linear programming are capable of fully modeling and exploiting the peculiarities of irregular architectures but do not scale well. We introduce the idea of a* progressive allocator. *A progressive allocator finds an initial allocation of quality comparable to a conventional allocator, but as more time is allowed for computation the quality of the allocation approaches optimal. This paper presents a progressive register allocator which uses a multi-commodity network flow model to elegantly represent the intricacies of irregular architectures. We evaluate our allocator as a substitute for* `gcc`*'s local register allocation pass.*

## 1. Introduction

Register allocation is one of the most important optimizations a compiler performs. Traditional register allocators were designed for regular, RISC-like architectures with large uniform register sets. Embedded architectures, such as the 68k, ColdFire, x86, ARM Thumb, MIPS16, and NEC V800 architectures, tend to be irregular, CISC architectures. These architectures may have small register sets, restrictions on how and when registers can be used, support for memory operands within arbitrary instructions, variable sized instructions or other features that complicate register allocation. The irregularities of these architectures make the register allocation problem particularly difficult and increase the effect of register allocation on quality code generation.

A register allocator typically reduces the register allocation problem to a more readily solved class of problem (for example, graph coloring). Ideally, the reduction produces a problem that is proper, expressive, and progressive. We define these properties as follows - a register allocator is:

- **Proper** if an optimal solution to the reduced problem is also an optimal register allocation. Since optimal register allocation is NP-complete [25, 20], it is unlikely that efficient algorithms will exist for solving the reduced problem. The optimality criterion can be any metric that can be statically evaluated at compile time such as code size or compiler-estimated execution time.
- **Expressive** if the reduced problem is capable of explicitly representing architectural irregularities and costs. For example, such a register allocator would be able to utilize information about the cost of assigning a variable to different register classes.
- **Progressive** if the solution procedure for solving the reduced problem is capable of making progress as more time is allotted for computation. Ideally, a reasonable solution is found quickly and the best solution converges to optimal as more time is allotted. Existing register allocators are not progressive; they either quickly find a suboptimal solution using heuristics, or after extensive computation find an optimal solution. A progressive register allocator fills this gap.

A fully expressive, proper and progressive register allocator is more flexible and powerful than current register allocators. The progressive nature of the allocator results in comparable code quality to conventional register allocators without sacrificing compile time, yet when fast compiles are not necessary, it can generate substantially better quality code. We present an expressive, proper and progressive register allocator which reduces register allocation to the problem of finding the minimum flow of multiple commodities through a network.

Related work is presented in Section 2. The multi-commodity network flow (MCNF) model is described in

Section 3. Several solution procedures are discussed in Section 4. We evaluate our allocator as a substitute for gcc's local allocator. The details of the gcc implementation are provided in Section 5. Results are given in Section 6.

## 2. Related Work

Traditional graph coloring [7, 8] works well in practice for regular architectures, but lacks the expressiveness to fully model irregular architecture features. Various researchers have proposed extensions to traditional graph coloring register allocation to improve allocation on irregular architectures [5, 6, 26]. These approaches either modify the heuristics used to color, modify the spilling heuristics, or modify the interference graph to prevent illegal allocations. There is no explicit optimization of spill code, nor do these techniques address all the features of irregular architectures. Spill code optimization has been addressed by modifying the spilling heuristic [4] and by splitting the live range of a variable so that a variable will only be partially spilled to memory [9, 3]. Although these techniques can significantly improve the quality of the register allocator, they are limited in that they are based on graph coloring. They are not proper or progressive, nor do they fully represent all the features of irregular architecture.

Register allocators which are proper and solve the register allocation problem optimally have been implemented by (1) performing a guided exhaustive search through the configuration space, potentially using exponential space [15, 17], (2) formulating register allocation as an integer linear program and then solving this formulation optimally using powerful solution techniques [13, 21], and (3) formulating register allocation as a partitioned boolean quadratic optimization problem which can than be solved either approximately or optimally [24]. Both the integer linear programming and partitioned boolean quadratic optimization approaches have been shown to be capable of precisely modeling features of irregular architectures [22, 2, 18]. Unfortunately, these solution techniques are not progressive; the solvers do not quickly find feasible solutions and then improve upon them. ILP solvers first must solve the linear relaxation of the integer program, which, although polynomial in time complexity, can be time-consuming and generally does not result in a feasible (all integer) solution. Once the solution to the linear relaxation is found, an optimal feasible solution is searched for, potentially taking exponential time. Although increases in processing power and improvements in integer linear programming solution software have improved the performance of this approach by orders of magnitude [11], its performance is still far from being competitive with traditional allocators. In some cases it takes hours to allocate a single function.

In this paper we develop a multi-commodity network flow formulation of the register allocation problem which is both expressive and proper while allowing the use of progressive solution algorithms. Network flows have been used to model and solve a wide range of problems, from transportation and distribution problems [1] to communications and routing in specialized computing networks [23]. Single commodity network flows have been used to allocate registers for improved energy efficiency [12]. 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 [10].

## 3. Multi-commodity Network Flow

The multi-commodity network flow (MCNF) problem is: find the minimum cost flow of commodities through a constrained network. The network is defined by nodes and edges where each edge has costs and a capacity. The costs and capacities can be specific to each commodity, but edges also have bundle constraints which constrain the total capacity of the edge. Each commodity has a source and sink 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).

A simplified example of our MCNF representation of register allocation is shown in Figure 1. In this example there are two registers, r0 and r1, and a memory allocation class. The cost of moving between registers is two and the cost of moving between registers and memory is

```
int example(int a, int b)
{
  int c = a - b;
  return c;
}
```

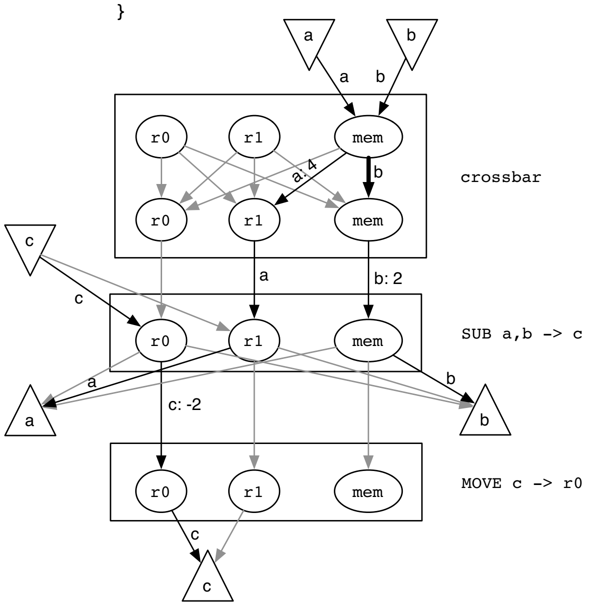

**Figure 1. A simplified example register allocation problem formulated as a multi-commodity network flow problem. Thin edges have a capacity of one (as only one variable can be allocated to a register and the SUB instruction supports only a single memory operand). The thick arc indicates that memory is uncapacitated. For clarity, edges not used by the displayed solution are in gray. 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 four, the cost of using a memory operand in the SUB instruction is two, and the benefit of allocating `c` to `r0` in the MOVE instruction is two since the move can be deleted in that case. The total cost of this solution is four.**

four. The arguments a and b are passed on the stack and thus are initially in memory. The SUB instruction can only support a single memory operand and the cost of using a memory operand is two. The cost of a commodity using an edge corresponds to the cost of the corresponding operation (move, load, store). For simplicity and ease of evaluation, we only consider the straightforward cost metric of program size (although any metric that can be evaluated at compile time could be used).

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

A node in the network represents an allocation class: a register, register class, or memory space to which a variable can be allocated. 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. 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 no variables are allowed to flow through the memory allocation class node. Variables used by an instruction must flow through the nodes of the corresponding instruction group. Crossbar groups are inserted between every instruction and allow variables to change allocation groups. For example, the ability to store a variable to memory is represented by an edge within a crossbar group from a register allocation class node to a memory allocation class node. Variables which are not used by an instruction bypass the corresponding instruction group and flow directly between the crossbars surrounding the instruction.

The cost of an operation, such as a move, can usually be represented by a cost on the edge that represents the move between allocation classes. However, this is not the correct model for 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 sub-network 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. The cost of evicting the anti-variable is just the cost of a single store. This way a variable may flow from registers to memory multiple times and yet pay the cost of only a single store (of course, every transition from memory

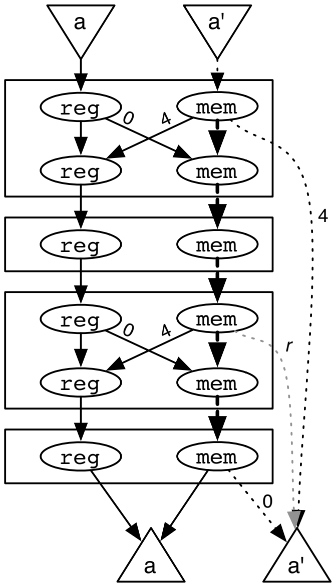

**Figure 2. An illustrative example of anti-variables. The anti-variable of $a$, $a'$, is restricted to the memory subnetwork (dashed edges). The $r$ edge is redundant and would not be in the actual graph. The cost of the second store can just as well be paid by the first edge. If the $r$ edge is left in the graph, it would have to have a cost of four 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.**

to a register pays the cost of a load). An actual store is only generated for the first move to memory.

The MCNF formulation of register allocation is proper, expressive, and has progressive solution procedures.

### 3.1. Properness

An optimal solution of the register allocation problem finds the assignment of registers and memory to variables at every program point for a given instruction stream that results in the minimum cost. The cost is a function of what the user wishes to optimize for. For example, in this paper we measure cost strictly in terms of program size as this metric is straightforward to define and measure. However, program execution time or energy usage could also be optimized, given some function that can statically evaluate these metrics at compile time.

It is important to note that our definition of optimality is restricted by the instruction stream provided to the register allocator and the limited types of operations the allocator is allowed to perform, such as moves, loads and stores. In some cases it is desirable for instruction selection decisions to be made during register allocation. It is possible to model these decisions, but the register allocation represented by the optimal solution to the corresponding MCNF problem is only optimal with respect to those instruction selection decisions we choose to model.

It is clear that if the MCNF problem is constructed properly the optimal solution will correspond to the optimal register allocation attainable using our restricted set of operations (inserting moves, loads, and stores between instructions and some limited local instruction selection decisions), so as long as the optimal solution never allocates the same variable to multiple registers at the same program point. This is the case because there is a direct correspondence between the flow of a variable through the MCNF problem and a variable's allocation 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, but can be removed by using a technique similar to the anti-variables used to model stores.

### 3.2. Expressiveness

The MCNF model is capable of expressing many of the pertinent features of irregular architectures. Since movement among registers and memory is precisely and flexibly modeled, spill code placement is explicitly optimized. Requirements on what types of registers an instruction can support and whether memory operands can be supported are also straightforward to model. If a variable must reside in a certain register class in an instruction, only edges to that register class node in the instruction group have any capacity for that variable. If an instruction can only support a certain number of memory operands, then the bundle constraint for the edge entering the memory node of the instruction group is set to limit the number of variables that can be in memory when the instruction is executed. In addition, if using an operand in memory rather than a register has some cost associated with it (for example, if it increases the size of the instruction), this can be modeled with a cost along this edge. For example, the SUB instruction in Figure 1 has a cost of 2 associated with using a memory operand.

The MCNF model is also capable of modeling register preferences. If an instruction can use any register, but some are more expensive than others (for example, several x86 instructions are only one byte if one of the operands is in `eax`), this can be represented with costs along the edges leading into those register nodes in the instruction group.

In some cases, instruction selection is influenced by register allocation. For example, in the 68k architecture a move with sign extend is implemented with two instructions if the destination is in a data register but can be implemented with a single move if the destination is an address register. A MCNF allocator can represent this decision as a simple preference for an address register where the cost of using a data register is equal to the cost of using the larger instruction sequence.

The MCNF model is not as fully expressive as a full ILP formulation of the problem. In particular, dependent constraints, where the cost of a specific variable allocation depends upon the allocation of other variables, are not easily modeled. For example, consider an add, $a = b + c$, on the x86 architecture. The cost of allocating $a$ to the register `eax` is zero if either $b$ or $c$ gets allocated to `eax`, but is one otherwise since a `lea` instruction, which is one byte larger than an add, would be used to perform the addition in this case. This situation cannot be modeled by a simple edge cost since the cost depends upon which variables use the edge in the final solution.

Contrary to good practice in graph coloring allocators, the MCNF model does not attempt to perform copy coalescing (where the source and destination of a move instruction can be allocated to the same register allowing for the deletion of the move). Instead, the allocator is provided with an instruction stream with all possible moves coalesced. It will then insert splitting moves at the optimal places. That is, because the MCNF model naturally expresses the costs associated with inserting moves into the instruction stream, moves can be aggressively coalesced with the expectation that the register allocator will correctly deal with the resulting extended lifetimes. Load and constant propagation can be modeled by assigning a negative cost equal in magnitude to the cost of the memory/constant load instruction when the variable being defined enters the memory network.

### 3.3. Progressiveness

To be useful, it must be possible to quickly find a feasible solution to the MCNF model. Ideally, it would be possible to steadily improve upon this solution until eventually an optimal solution is found. Such a solution procedure would allow the user to consciously trade compile time for code quality. Existing integer linear programming solvers do not have this property since they do not immediately find a feasible solution and only search for integer, as opposed to fractional, solutions at the end of the solution procedure.

The MCNF-based solution procedures evaluated in this paper combine the Lagrangian relaxation method of solving MCNF problems with algorithms for finding feasible solutions to our MCNF problems. These methods immediately find a feasible solution and then attempt to improve upon it as the Lagrangian relaxation converges to the optimal value. Although this method is not guaranteed to find an optimal solution, it can determine a bound on how close the solution is to optimal.

## 4. Solution Procedures

The specific form of our MCNF representation allows us to quickly find a feasible, though possibly low quality, solution. We build up a solution to the multi-commodity flow problem by solving the single commodity flow problem for each variable. This is simply a shortest-path computation. We will always be able to find a feasible solution because the memory network is uncapacitated. If no path is available for a variable (because the edges are all already allocated to other variables), it is always possible to locally evict another variable to memory (or another register) and continue to make progress.

Alternatively, we can constrain our shortest path finding algorithm to conservatively ignore paths that potentially will make the network infeasible for the variables that still need to be allocated. For example, if an instruction requires its operand to be in a register and that operand has not yet been allocated 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. The variables we allocate first will stay in registers the longest. The shortest path computations we perform have no way of being influenced by the costs to other variables. Ideally, we would like to build a solution from a series of 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 specific allocation with respect to all the other variables. Lagrangian relaxation provides a formal way of computing these marginal costs.

### 4.1. Lagrangian Relaxation

Lagrangian relaxation is a general solution technique [1]. It works by removing one or more constraints from the problem and integrating them into the objective function using Lagrangian multipliers resulting in a more easily solved Lagrangian subproblem. In the case of MCNF, the Lagrangian subproblem is to find a price vector $w$ such that $L(w)$ is maximal, where $L(w)$ is defined:

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

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

subject to

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

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

The bundle constraints have been integrated into the objective function. If an edge $x_{ij}$ is over-allocated then the term $\sum_k x^k_{ij} - u_{ij}$ will increase the value of the objective function, making it less likely that an over-allocated edge will exist in the 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 being minimized over. The subproblem is still subject to the 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.

The function $L(w)$ has several useful properties [1]:

- **Lagrangian Bounding Principle**. For any set of prices $w$, the value of $L(w)$ is a lower bound on the optimal value of the objective function of the original MCNF problem.
- **Weak Duality**. Let $L^* = max_w L(w)$. $L^*$ is always a lower bound on the optimal objective function of the original MCNF problem.
- **Optimality Test**. Let $x^*$ be a solution to $L^*$. If $x^*$ is also a feasible solution to the original MCNF problem (does not violate the bundle constraints) and satisfies the condition $w_{ij}\left(\sum_k x^{*k}_{ij} - u_{ij}\right) = 0$ (only fully utilized edges in the solution have nonzero prices in $L^*$), then $x^*$ is an optimal solution to the MCNF problem.

In short, the Lagrangian relaxation provides a strong theoretical lower bound for the optimal solution value. Solutions to the relaxed subproblem which are feasible in the original MCNF problem are likely to be optimal and, under certain conditions, can be proven optimal.

A reasonable solution procedure is to find the price vector which maximizes $L(w)$ and then construct a feasible solution that is also a solution to $L^*$ (or at least close to it). First we must 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. We then update $w$ using the rule:

$$w^{q+1}_{ij} = \max\left(w^q_{ij} + \theta_q\left(\sum_k y^k_{ij} - u_{ij}\right), 0\right)$$

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

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

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

| **q** | a | b | c | $w_{SUB_{mem}}$ | $L(w^q)$ |
|---|---|---|---|---|---|
| **0** | 2 | 2 | -2 | 0 | 2 |
| **1** | 3 | 3 | -2 | 1 | 3 |
| **2** | 4 | 4 | -2 | 2 | 4 |

**Table 1. The price of the shortest paths for each variable, the price of the edge entering the SUB instruction's memory node, and $L(w^q)$ for each iteration $q$ of the iterative subgradient optimization algorithm. For ease of explanation, in this example the step size is fixed to 1 and prices are all initialized to 0.**

We evaluate two methods which meet these conditions for calculating the step size:

- **Ratio Method** The step size at iteration $q$ is simply $\theta_q = 1/q$. To avoid large initial step sizes, different starting points can be considered, such as $\theta_q = 1/(q + 10)$.
- **Newton's Method** A variation of Newton's method is used to choose $\theta$ with the formula:

  $$\theta_q = \frac{\delta_q[UB - L(w^q_{ij})]}{\sum_{i,j}\left(\sum_k x^{*k}_{ij} - u_{ij}\right)^2}$$

  where $UB$ is an upper bound on the value of the objective function. An upper bound can be calculated using any feasible solution finder.

As an example, consider the simple network in Figure 1. The allocator would first find a feasible solution. Assuming we process the variables in the order $(c, b, a)$, we first find the shortest path for $c$, allocating it to `r0`, then the shortest path for $b$, which leaves $b$ in memory, and then the shortest path for $a$, which would require a load since $b$ has saturated the edge into the memory node of the SUB instruction (this is the solution shown in the figure). The total cost of this solution is 4, which is optimal in this case. However, the algorithm has not yet proven that this result is optimal.

In the next step, the solution to the Lagrangian subproblem is found by finding the shortest paths ignoring the bundle constraints. We initialize our prices to be zero. As a result, the paths for a and b both have a cost of 2 and the path for c has a cost of -2. This gives us a total cost for $L(w^0)$ of 2. The prices are then updated. Since only one edge (the memory to memory edge into the SUB instruction) is over-constrained, this is the only edge that has its price change. For this example we use a fixed step size of 1 resulting in a new edge price of 1 for that edge. The algorithm is then repeated with the results shown in Table 1. We stop the algorithm when $L(w^3) = 4$ since this proves that the first feasible solution we found was, in fact, optimal.

Note that there are multiple solutions to the Lagrangian subproblem which have the optimal value 4, but not all of these solutions are feasible.

### 4.2. Feasible Solution Finding

The feasible solution finder constructs a feasible solution by allocating variables individually (using a shortest paths computation). As each variable is allocated, the shortest paths of the remaining variables are constrained, but care is taken to ensure there will always be some (possibly very expensive) allocation available for the remaining variables. The order in which variables are allocated is therefore important.

We considered several different heuristics for constructing a good feasible solution:

- **Greedy Shortest:** The shortest path through the priced graph is computed simultaneously for all variables. Variables are inspected starting with the variables with the most expensive paths. If the variable's path would not make further allocation infeasible and does not overlap with an already allocated variable, then that path is chosen. Once as many variables are allocated as possible, the shortest paths for the remaining, unallocated, variables are recomputed avoiding the already allocated edges, and the procedure is repeated until all variables are allocated. The assumption behind this heuristic is that it is beneficial to allocate as many variables as possible to paths that are identical to the paths in the relaxed solution. If the relaxed solution is feasible or very close to being feasible, then this heuristic should do well.

- **Iterative Shortest:** This heuristic allocates the variables with the most expensive allocations first. The order variables are allocated is fixed by a single all-variable shortest paths computation at the onset. A single variable shortest path computation is performed before a variable is allocated. This finds the current shortest feasible path, which may be different from the path found by the initial all-variable shortest path computation due to the previous allocation of other variables. This heuristic gives allocation preference to the most expensive variables and does not require multiple all-variable shortest path computations.

- **Lowest Cost $k$-Choice:** Similar in structure to Iterative Shortest, but a $k$-shortest paths computation is performed. Because the prices are only converging to the optimum values and are not actually optimal, paths with costs close to the shortest are likely to be good choices. Of the $k$ paths whose price is within a threshold of the shortest path, the path with the lowest unpriced cost is chosen.

An alternative to the heuristic based feasible solution finders is to exhaustively search the space of allocations with good prices. The exhaustive search procedure fixes an order of variables (based on their unconstrained shortest paths cost). The k-shortest priced paths are computed for a variable, then for each of these paths the remaining variables are recursively allocated. Thus the Lagrangian prices are used to narrow the search space of allocations of variables. Instead of considering all possible allocations of each variable, only a maximum of k allocations are considered. Although the search is exponential, if an upper bound is known then it is possible to avoid visiting the entire search tree.

## 5. Implementation

We have implemented our MCNF allocation framework as a replacement for the local register allocator in gcc 3.4. The 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.

Before allocation, we execute a preconditioning pass which coalesces moves and translates instructions that are not in an allocable form. For example, internally instructions are represented as three operand instructions even when 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 will translate such instructions so that two of the three operands are the same variable.

Then, for each basic block, we build an MCNF model that models all the variables, both global and local, that are used in the block. After running our solver on the model, we insert the appropriate moves, stores, and loads and assign registers to the local variables.

The MCNF model is simplified without loss of generality by only allowing loads of a variable before instructions that use the variable and only allowing stores after instructions that define it.

The solver first finds a feasible solution both in the unpriced MCNF model and in an MCNF model where the first store to memory is priced the cost of a store. For these initial solution variables are allocated in the same order used by the gcc local allocation heuristic.

## 6. Results

### 6.1 Example

Throughout this section, unless otherwise stated, the results are for the same small example compiled for the x86

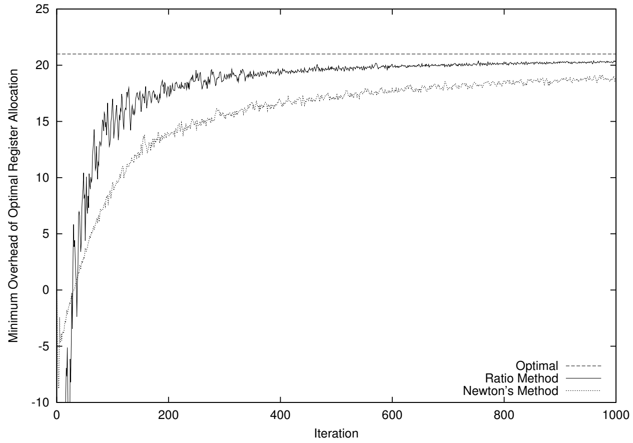

**Figure 3. The convergence of the Lagrangian subproblem to the optimal value over 1000 iterations using two different methods for calculating the step size.**

architecture. All code is compiled with the `-Os` option (optimize for size). The code for this example was isolated from a much larger example (the `squareEncrypt` function in the `pegwit` benchmark from the MediaBench benchmark suite [19]) and provides a reasonable register usage pattern that is complex enough to be interesting but simple enough to understand. The example has 56 instructions and 26 variables of which a maximum of 8 are simultaneously live at any program point. The minimum possible register allocation for this example is 21 bytes and the code size after register allocation is 248 bytes.

### 6.2. Convergence

For the solution procedure to be effective, the value of the Lagrangian subproblem $L(w)$ should converge quickly towards the actual optimal value. The convergence properties of both the ratio method and Newton's method of determining step sizes are shown in Figure 3. Although the ratio method begins poorly because of the large initial step sizes, it quickly converges towards the optimal solution of 21. By iteration 466 it has found a lower bound larger than 20 which is evidence that a feasible solution of cost 21 is optimal. Newton's method, by contrast, does not find a lower bound greater than 20 until iteration 1926. In general, Newton's method performed more poorly than the ratio method; for the remainder of this section we provide results for the ratio method only.

### 6.3. Heuristics

As the Lagrangian subproblem converges towards the optimum value it is expected to provide better guidance to the feasible solution finders. The behavior of the Iterative Shortest heuristic is shown in Figure 4. All the solution finders have a similar pattern of behaving erratically locally, but showing a general trend of improvement. A good solution finder will find the optimal solution and solutions close to optimal multiple times. The various solution finders are evaluated by this metric in Figure 5.

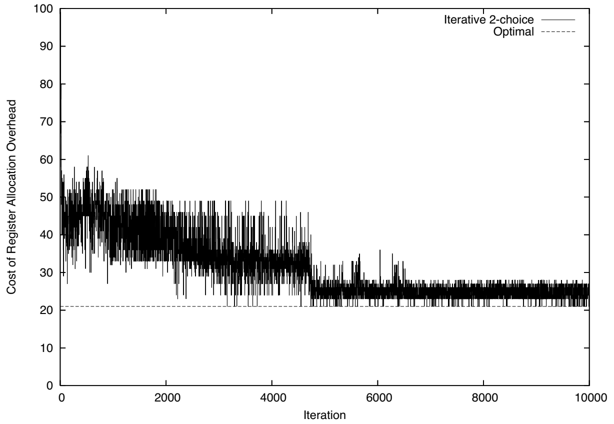

**Figure 4. The behavior of the Iterative Shortest feasible solution finder over 10000 iterations. As the Lagrangian subproblem converges, the quality of the found solutions improve.**

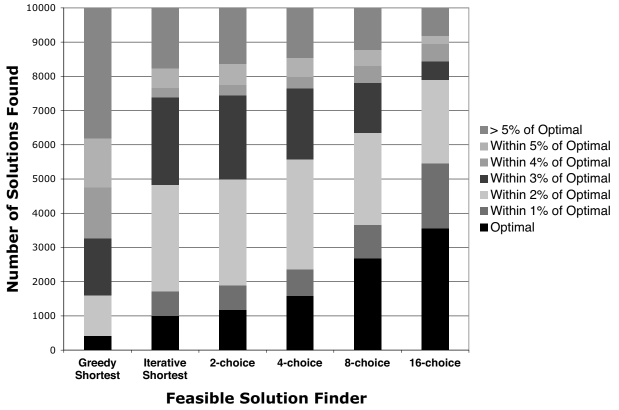

**Figure 5. The performance of different feasible solution finders as measured by how often optimal and near optimal solutions are found over 10000 iterations.**

The Greedy Shortest and Iterative Shortest heuristics consider only the shortest feasible path of each variable through the priced network. The iterative solution performs significantly better than the simple greedy solution. The $k$-Choice solution finders generally do better as $k$ is increased, but at the expense of increased execution time.

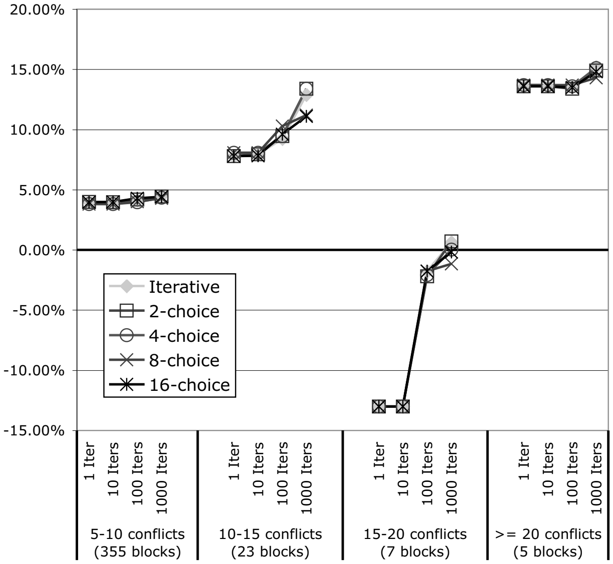

**Figure 6. The overall percentage improvement in total size of basic blocks, grouped based on the number of local conflicts, as the maximum number of iterations of the solver is increased for different solution finders.**

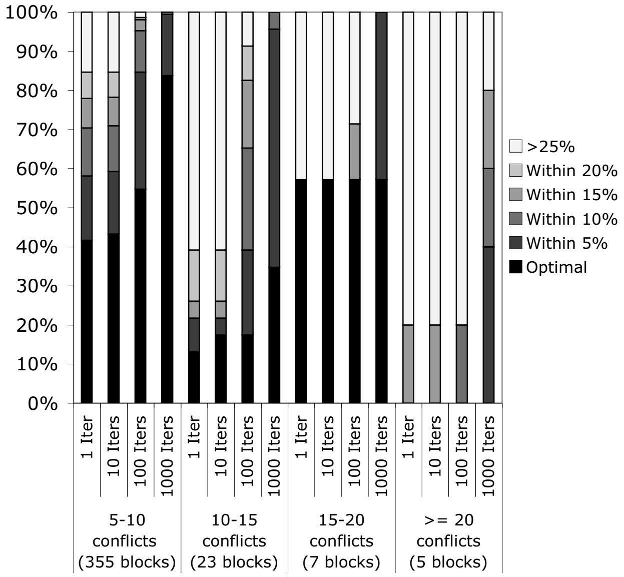

**Figure 7. The percentage of basic blocks, grouped by the maximum number of conflicts between local variables, whose solution can be proven to be a certain percentage from optimal. As more iterations are used, the algorithm finds better solutions and better bounds on the optimality of the solutions. The 2-choice solution finder was used; the results are representative of all the iterative solution finders.**

### 6.4. Code Quality

The solutions found by these procedures are, at best, only optimal in the narrowly defined context of the MCNF formulation of the register allocation problem. Because not all possible register related transformations are modeled in the implemented MCNF formulation, the actual overhead of register allocation might be different than the theoretical overhead. For example, in the model all accesses to the stack are assumed to only cost one byte which is only true if fewer than 32 variables are spilled to the stack. A more sophisticated model might have two memory classes to address this issue. In addition, errors in the interface between the Lagrangian solver and `gcc`'s register allocator may also distort the results.

Despite the many sources of noise, the Lagrangian solver generates significantly smaller code than `gcc`'s default allocator. We evaluate the allocator using basic blocks from MiBench [14], MediaBench [19], SpecInt 95, and SpecInt 2000 [27, 28]. The base for comparison is the default `gcc` allocator operating on an uncoalesced version of the preconditioned instruction stream used by the Lagrangian solver (the results are similar if the unconditioned instruction stream is used for a base of comparison). All results are based on the code size immediately after register allocation. Because we are only interested in local allocation, the allocator is only run on the 390 blocks which contain five or more simultaneously live local variables. The code improvement for blocks containing fewer simultaneously live local variables was slight (less than 1% improvement). Furthermore, more than 90% of these blocks had provably optimal solutions within 10 iterations. The results using different feasible solution finders are shown in Figure 6. Somewhat surprisingly, there is not a large difference between the various solution finders. As expected, the code quality improves as more time is allotted for execution. In addition, as shown in Figure 7, many of these solutions can be shown to be optimal or near-optimal.

The poor results for the 15-20 conflicts case are an artifact of the current implementation's preconditioning pass. We currently do not coalesce variables which have different sizes. Although this is not a common occurrence, it is vital for one of the blocks in this category.

### 6.5. Running Times

If the Lagrangian approach is to be useful for register allocation, it must offer both competitive performance and scale as the problem size increases. We evaluate the performance of the various solution finders operating on both the small example from the previous sections and the function, `squareEncrypt`, from which it is derived. This large

| Register Allocator | *Small Example Time (s)* | *Large Example Time (s)* |
|---|---|---|
| `gcc` default (local/global) | $<.01$ | .02 |
| `gcc -fnew-ra` (Briggs/Chaitin) | .01 | .15 |
| Lagrangian: 15% from Optimal | 1.8 | 59.4 |
| Lagrangian: 10% from Optimal | 2.94 | 85.2 |
| Lagrangian: 5% from Optimal | 14.64 | 528.6 |
| Lagrangian: Proven Optimal | 104.6 | N/A |
| ILP Solver, CPLEX 7.1 | 3.14 (5.62) | 949 (1699) |

**Table 3. The total time, in seconds, of the register allocation pass for various allocators on a 1.8Ghz Pentium 4 system. Times are given for both of `gcc`'s allocators, for the Lagrangian based approach using two different stopping criterion, and for the CPLEX 7.1 [16] ILP solver. The CPLEX solver was run on a 1Ghz Pentium 3. To make a more accurate comparison, the CPLEX times were extrapolated to the expected values on the faster system using the bogomips ratio of the two machines. The original times are shown in parentheses.**

| Technique | *Small* | *Large* |
|---|---|---|
| $L(w)$ maximization | .016 | .17 |
| Greedy Shortest | .05 | 2.6 |
| Iterative Shortest | .04 | .5 |
| 2-Choice | .05 | .6 |
| 4-Choice | .06 | .7 |
| 8-Choice | .07 | .9 |
| 16-Choice | .09 | 1.1 |

**Table 2. The time, in seconds, of performing a single iteration of Lagrangian maximization and of each feasible solution finder for both a small and large example on a 1.8Ghz Pentium 4 system.**

example has 380 instructions and 150 variables of which as many as 14 are simultaneously live. The value of the optimal solution is 428. The running times of the various techniques per an iteration are shown in Table 2.

An adequate lower bound for proving optimality of the small example is found in iteration 466 (28 seconds) but using the 2-choice solution finder an optimal feasible solution is not found until iteration 1744 (105 seconds). However, a solution that is proven to be within 10% of optimal and is actually within 5% of optimal is found within 3 seconds. The large example does not find an optimal solution within 5000 iterations, but will be able to provide a solution that is guaranteed to be within 15% of optimal at iteration 99 (about 60 seconds).

To put these times in perspective, the total register allocation times for both of `gcc`'s allocators are shown in Table 3. These allocators can produce decent allocations well before the Lagrangian approach has had a chance to ramp up, but they can make no claims about optimality. Traditional integer linear program solvers can be used to solve for the MCNF formulation of the register allocation problem. The solution time for the commercial CPLEX [16] solver is also shown in Table 3. The Lagrangian approach compares favorably with the optimal solver and has the added advantage of being progressive (that is, capable of producing a suboptimal solution with guaranteed optimality bounds at any point).

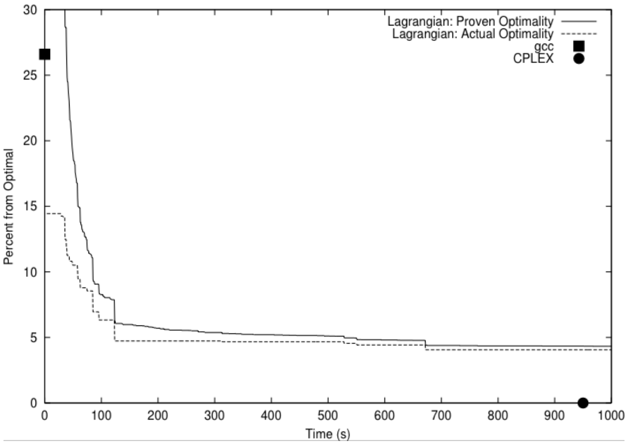

**Figure 8. The progressive nature of the Lagrangian solution approach run on the `squareEncrypt` function. As the algorithm computes, a better solution and a better optimality bound on the solution are found.**

## 7. Conclusion

As shown in Figure 8, the current implementation of our register allocator effectively bridges the gap between `gcc`, which is fast, but neither expressive nor proper, and the slow, but fully expressive and proper, ILP approach. The Lagrangian approach quickly finds a solution comparable with the gcc solution and continues to improve the result as more time passes. The Largrangian approach is not guaranteed to find an optimal solution and, even when an optimal solution is found, it may take longer than a commercial ILP solver. However, unlike with the ILP solver, a valid intermediate result is always available.

In conclusion, we have developed an approach to register allocation which allows a user to make a conscious tradeoff between compile-time and code quality. Our allocator, based on reducing the register allocation problem to a multicommodity network flow problem, is expressive, proper, and progressive; no other allocator combines all three properties.

## 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 *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 243–253. ACM Press, 2001.

[3] P. Bergner, P. Dahl, D. Engebretsen, and M. T. O'Keefe. Spill code minimization via interference region spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[4] D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263. ACM Press, 1989.

[5] P. Briggs. Register allocation via graph coloring. Technical Report CRPC-TR92218, Rice University, Houston, TX, April 1992.

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

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

[8] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

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

[10] M. Farach and V. Liberatore. On local register allocation. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.

[11] C. Fu and K. Wilken. A faster optimal register allocator. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256. IEEE Computer Society Press, 2002.

[12] C. H. Gebotys. Low energy memory and register allocation using network flow. In *Proceedings of the 34th annual conference on Design automation conference*, pages 435–440. ACM Press, 1997.

[13] D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software: Practice and Experience*, 26(8):929–965, 1996.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[15] W.-C. 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.

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

[17] 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.

[18] T. Kong and K. D. Wilken. Precise register allocation for irregular architectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 297–307. IEEE Computer Society Press, 1998.

[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro*, pages 330–335, 1997.

[20] V. Liberatore, M. Farach, and U. Kremer. Hardness and algorithms for local register allocation.

[21] V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. In *Compiler Construction, 8th International Conference, CC'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1575 of *Lecture Notes in Computer Science*. Springer, 1999.

[22] M. Naik and J. Palsberg. Compiling with code-size constraints. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 120–129. ACM Press, 2002.

[23] A. E. Ozdaglar and D. P. Bertsekas. Optimal solution of integer multicommodity flow problems with application in optical networks. In *Proc. of Symposium on Global Optimization*, June 2003.

[24] B. Scholz and E. Eckstein. Register allocation for irregular architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 139–148. ACM Press, 2002.

[25] R. Sethi. Complete register allocation problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 182–195. ACM Press, 1973.

[26] M. D. Smith and G. Holloway. Graph-coloring register allocation for irregular architectures.

[27] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[28] Standard Performance Evaluation Corp. *SPEC CPU2000 Benchmark Suite*, 2000.