Register Allocation: A Lagrangian Approach

David Koes
dkoes@cs.cmu.edu
Carnegie Mellon University

1 Motivation

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 registers 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 objective of this research is to formulate a new type of allocator that is tailored towards the demands of embedded architectures and is a more formal and less *ad hoc* approach then the current state of the art.

2 State of the Art: Graph Coloring

Traditional register allocators use a graph coloring model [7, 8]. These allocators construct an interference graph in which each node represents a variable and an edge indicates that two variables interfere (they cannot be allocated to the same register). The allocator then uses heuristic methods to color this graph with register "colors" so no node shares its color with any neighboring node. If the graph cannot be colored, heuristics are used to select a variable to spill to memory and the processes is repeated until a valid coloring (register assignment) is found.

Although the graph coloring representation of register allocation is a success for regular architectures, it is a poor representation of the register allocation problem for irregular architectures. The graph coloring model solves the register sufficiency problem, not the register allocation problem itself. That is, the algorithm determines if there are enough registers to allocate all the variables to registers; there is no explicit optimization of spill code. Since irregular architectures have few registers, variables are frequently spilled increasing the importance of good spill code generation. Graph allocators optimize spill code using simple heuristics [4] and through the use of a preconditioning passes that splits a variable into multiple variables [9, 3].

Irregular architectures typically have restrictions on where and how registers can be used. Although graph coloring can be modified to support these restrictions [5, 6, 18], the underlying problem representation is incapable of fully representing them. For example, although it is straightforward to model that a variable requires a particular type of register, it isn't clear how to precisely model the case where a variable can be validly allocated to any type of register but with different costs. Another deficiency of graph coloring allocation is that it assigns a single register to each variable. This can be undesirable if a variable's register preferences change or if allocating the variable to multiple registers would result in less spill code.

The deficiencies of the graph coloring model can be partially addressed by *ad hoc* modifications of the heuristics guiding the coloring and spilling decisions, but there are no principled extensions of the model that fully represent and exploit the features of irregular architectures.

3 Objective: A Principled Approach

A principled approach to register allocation for irregular architectures requires an underlying problem formulation that is capable of explicitly representing architectural irregularities and costs. In addition to being expressive, such

an approach must also be proper and progressive. The optimal solution to a proper problem formulation directly maps to an optimal register allocation. This is in contrast to the graph coloring approach where an optimal graph coloring does not necessarily result in an optimal register allocation since spill decisions are heuristic driven. Since optimal register allocation is NP-complete[17, 14], it is unlikely that efficient algorithms will exist for solving a proper problem formulation optimally. However, progressive algorithms might exist. Ideally, these algorithms should be able to quickly find a feasible solution and, as they are allowed more time for computation, converge on an optimal solution. Finally, in addition to being expressive, proper and progressive, a principled approach to register allocation for irregular architectures should also have a code quality to compile time ratio that is competitive with existing allocators.

Representing the problem as an integer linear programming problem, an approach that has been done before [11, 10] even for irregular architectures [16, 2, 13] is unfruitful. This approach, while both expressive and proper, is not progressive and has poor real world performance. Representing the problem as a multi-commodity network flow (MCNF) problem results in a more limited problem formulation that is still capable of representing the features of an irregular architecture but which allows for a more flexible solution procedure.

The MCNF model is described in Section 4. Several solution procedures are discussed in Section 5. The details of the `gcc` implementation are provided in Section 6. Results are given in Section 7 and Section 8 concludes.

4 Multi-commodity Network Flow

The multi-commodity network flow (MCNF) problem is to find the total minimum cost flow of commodities through a constrained network. The network is defined by nodes and edges where each edge has a cost and a capacity. The costs and capacities can be specific to each commodity, but edges also have bundle constraints which constrain the total capacity the edge. Each commodity has a source and sink such that the outflow of the source must equal the inflow of the sink. Although finding the minimum cost flow of a single commodity is readily solved in polynomial time, finding an solution to the multi-commodity network flow problem where all flows are integer is NP-complete [1].

Formally the MCNF problem is specified:

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

subject to

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

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

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

where x_{ij}^k is the flow of commodity k along edge (i, j), c^k is the cost vector containing the cost of each edge for commodity k, u_{ij} is the bundle constraint for edge (i, j), v_{ij}^k is an individual constraint on commodity k over edge (i, j), and $\mathcal{N}$ and b^k represent the network topology, sources and sinks.

In our MCNF representation of register allocation, the commodities represent variables. The design of the network and individual commodity constraints is dictated by the way in which variables are used. The bundle constraints enforce the limited number of registers available, and the edge costs are used to model the cost of spilling and register preferences. A node in the network represents an allocation class: a register, register class, or memory space that a variable can be allocated to. Nodes are grouped into instruction and 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 instruction groups constrain what 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 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 must flow through edges directly connecting crossbars.

A source node of a variable connects into the network at the defining instruction and the sink node of a variable removes the variable from the network at the last instruction to use the variable.

A simplified example of a register allocation MCNF problem 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 2 and the cost of moving between registers and memory is 4. The arguments `a` and `b` are passed on the stack and so are initially in memory. The SUB instruction can only support a single memory operand and the cost of using a memory operand is 2.

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 get overwritten. In order to model the persistence of data in memory, anti-variables 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 be 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 in order for these edges to be usable, the anti-variable must be evicted. This way a variable may flow from registers to memory multiple times and pay only the cost of a single store.

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

4.1 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. In this paper we measure cost strictly in terms of program size as this metric is straightforward to define and measure.

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 to see that if the MCNF problem is designed 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) as long as in the optimal solution no variable is allocated 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.

4.2 Expressiveness

The MCNF model is capable of expressing 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 or not 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.

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

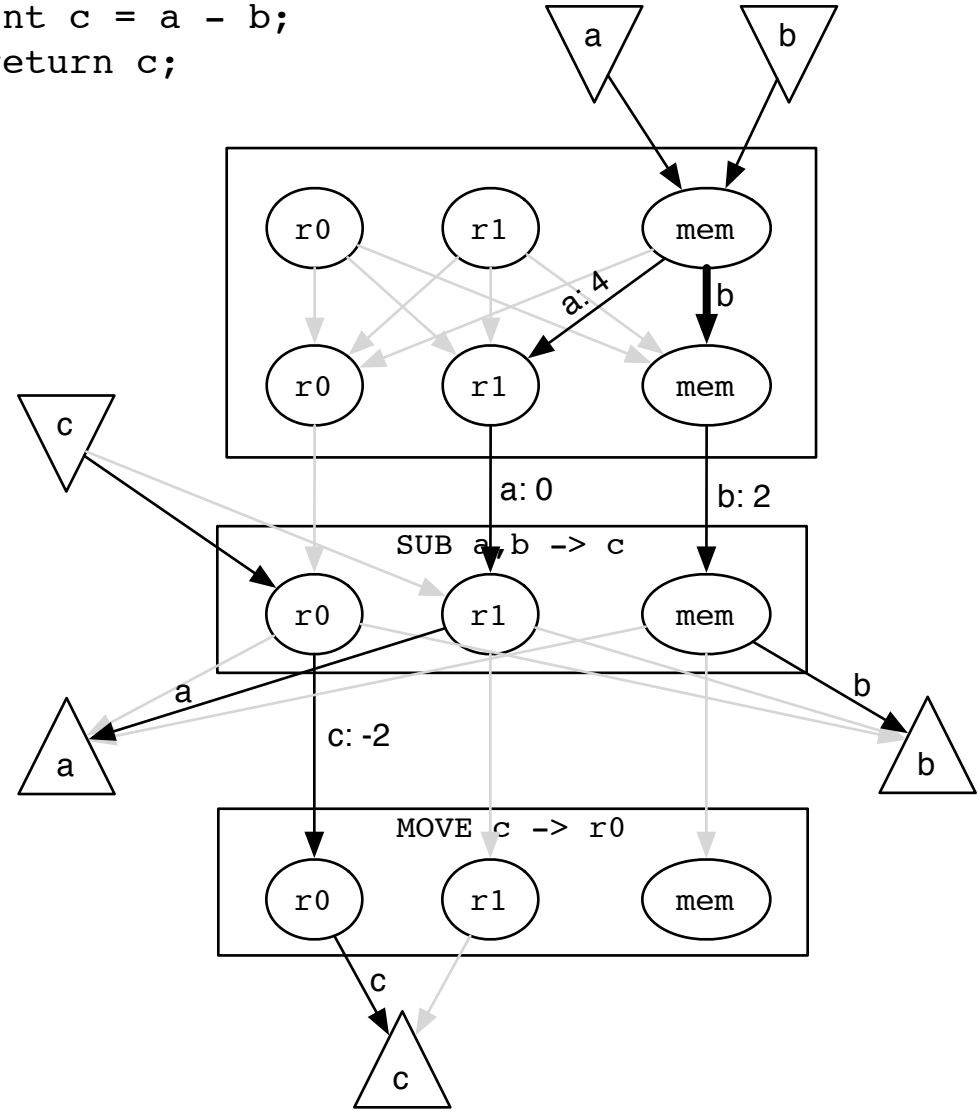

Figure 1: A simplified example register allocation formulated as a multi-commodity network flow problem. Thin edges have a capacity of one (as only one variable can be allocated to a register and the SUB instruction supports only a single memory operand). The thick arc indicates that memory is uncapacitated. For clarity, edges not used by the displayed solution are in gray. For the displayed solution, the cost of each used arc along with the commodity using it is displayed if the cost is nonzero. In this example the cost of a load is 4, the cost of using a memory operand in the SUB instruction is 2, and the benefit of allocating `c` to `r0` in the MOVE instruction is 2 since the move can be deleted in that case. The total cost of this solution is 4.

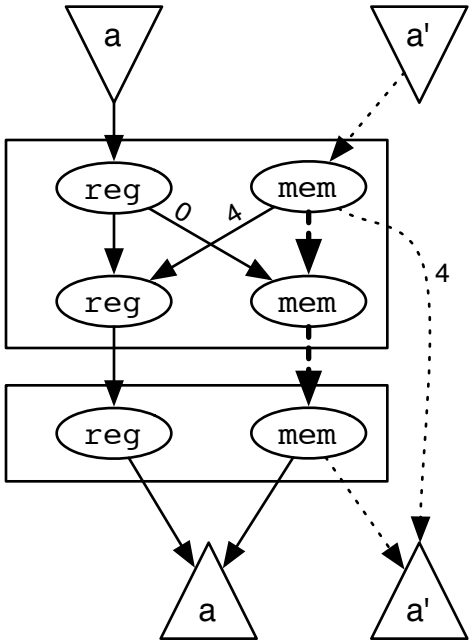

Figure 2: The anti-variable of a, a' is restricted to the memory subnetwork (dashed edges). A variable and its anti-variable are constrained such that only one or the other can use a memory edge. The cost of a variable flowing from a register to memory is zero, but in order for the variable to occupy the memory node the anti-variable must be evicted. The cost of doing so is simply the cost of performing a store. Once the ant-variable has been evicted there is no additional cost for the variable to be moved from a register to memory (although every flow back from memory to register must pay the cost of a load).

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 does not model the benefits of 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 should be provided an instruction stream with all possible moves coalesced. It will then insert splitting moves at the optimal places. Rematerialization of loads can be modeled by assigning a negative cost equal in magnitude to the cost of the load instruction when a variable enters the memory network at its definition point (a load instruction).

4.3 Progressiveness

In order to be useful, it must be possible to find a solution to the MCNF model quickly. 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 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 optimal. This method is not guaranteed to converge to an optimal solution but does establish a lower bound upon the optimal solution cost.

5 Solution Procedures

The specific form of our MCNF representation allows us to quickly find a feasible, if possibly very poor, solution. We build up a solution to the multi-commodity flow problem by solving the single commodity flow problem for each variable. This is a simple shortest path computation. Because the memory network is uncapacitated, if we every get stuck and can't find a path for a variable because the edges it needs to use are blocked by other variables, it will always be 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 very 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, it would be possible to build up a solution from a series of simple shortest path computations. Each individual variable's shortest path would somehow 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.

5.1 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:

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

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

subject to

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

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

The bundle constraints have been integrated into the objective function. If the 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 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 shown by 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:

- **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 complementary slackness condition then x^* is an optimal solution to the MCNF problem. The complementary slackness condition is necessary because of the inequalities in our MCNF formulations and requires that $w_{ij}\left(\sum_k x_{ij}^{*k} - u_{ij}\right) = 0$. That is, any edge which is not at full capacity in the solution should have a price of 0 in L^*.

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 prices 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:

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

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

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

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

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 void 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 θ with the formula:

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

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

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

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

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, which is for it to be allocated 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 overconstrained, this is the only edge to have 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 $L(w) = 4$ since this certifies that the first feasible solution we found was, in fact, optimal.

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

5.2 Feasible Solution Finding

The feasible solution finder step of the solution method constructs a feasible solution by allocating variables individually (using a simple 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.

Several different heuristics for constructing a good feasible solution were considered:

- **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 that path is chosen. Once as many variables are allocated as possible, the procedure is repeated with the shortest paths step avoiding already allocated edges. 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. Because every allocation modifies the network (by reducing the availability of registers), an all-variable shortest paths computation is performed after every allocation to calculate the next most expensive variable.

- **Fixed Iterative Shortest:** Similar to Iterative Shortest, but the ordering variables are allocated in is fixed by a single all-variables shortest paths computation at the onset, removing the need for further expensive all-variable computations.

- k**-Choice:** Similar in structure to Fixed Iterative Shortest, but a k-shortest paths computation is performed. A heuristic is used to choose among the k paths whose price is within a threshold of the shortest path. Especially since 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. Two heuristics were evaluated for choosing among paths:

  - **Lowest Cost:** Of the possible paths, the one with the lowest unpriced cost is chosen.
  - **Cost/History:** A history of the results of previous path choices is maintained. For each path the average and best result achieved using that path is calculated. When selecting a path, both the lowest cost and previous history are considered, with more weight given to the previous history as the best solution gets closer to the lower bound. The rational for this approach is that good solutions are expected to be incrementally similar to other good solutions and the history bias will exploit this property.

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.

6 Implementation

An MCNF allocation framework has been implemented 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 any remaining variables are allocated using a single pass graph coloring algorithm. Because we are only performing local register allocation, we do not need a separate allocation class for every register. Instead we can have an allocation class for every register class. Unfortunately, this distinction is meaningless for the x86 architecture. A preconditioning pass fixes poorly formed instructions (such as a two operand instruction which is internally being represented as a three operand instruction where all three operands are live out of the instruction) and coalesces moves. The solver finds the optimal spill code and register class assignments for each variable and inserts the appropriate moves, stores, and loads. Registers are then assigned to variables.

The MCNF model we implement uses a simple crossbar structure. This structure allows two register classes to both be saturated and swap the contents of two registers. Since a swap is not possible using just moves unless an additional temporary register is available, this crossbar representation does not exactly model the underlying architecture. Fortunately, both architectures we evaluate support a register exchange instruction. A single exchange instruction usually has a cheaper cost than two moves, a fact which is not modeled by our crossbar. Although it would be possible to make the crossbar more complicated in order to model only having moves or explicitly supporting the register exchange instruction, we prefer to keep the model simple and potentially under-utilize the register exchange instruction.

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.

7 Results

7.1 Example

Throughout this section, unless otherwise stated, the results are for the same small example compiling for the x86 architecture. The code for this example was isolated from a much larger example (the `squareEncrypt` function in the `pegwit_d` benchmark) and provides a reasonable register usage pattern that is complex enough to be interesting but simple enough to understand. The example has 60 instructions and 26 variables, six of which have long lifetimes. The minimum amount of overhead from register allocation in this example is five bytes. The register usage pattern is displayed graphically in Figure 3.

7.2 Convergence

In order for the solution procedure to be effective, the value of the Lagrangian subproblem $L(w)$ should converge quickly to the actual optimal solution. The convergence properties of the ratio method for determining the step size in each iteration are shown in Figure 4. The relatively large step sizes of this method allow for significant progress, but also result in a large amount of oscillation. Large initial step sizes result in very poor initial performance, so multiple starting points are evaluated.

The two delayed starting point methods perform similarly and significantly better than the undelayed method. Using a step size of $1/(k + 100)$, a solution $L(w)$ larger than 14 is found after 396 iterations (at which point, if a feasible solution of cost 15 has been found, there is proof of the optimality of the solution). After 3000 iterations, both delayed starting point methods have found a lower bound larger than 14.90.

The convergence properties of Newton's method are shown in Figure 5. The known optimal value is used as an upper bound in the step calculation and various values of δ are evaluated. Values larger than two are not guaranteed to converge, but are since they result in larger steps still manage to outperform the smaller δ values in this example. Larger δs result in a less stable convergence. Reasonable values of δ result in a fairly stable convergence that slows as the optimal solution is approached. Using a δ of two, a value of $L(w)$ larger than 14 is not obtained until iteration 1170 and the largest value obtained in 3000 iterations is 14.597.

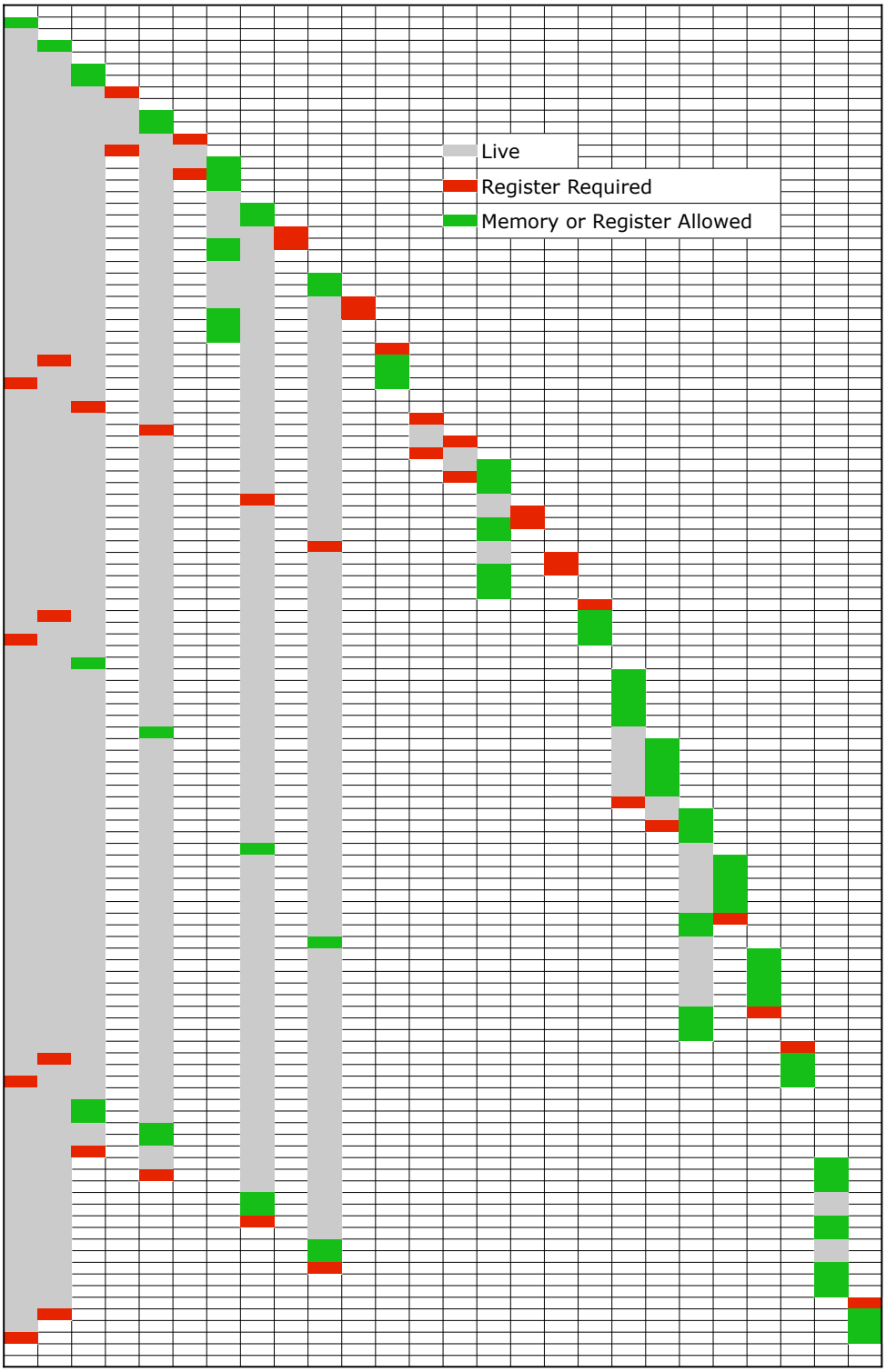

Figure 3: A graphical representation for a small but interesting function. Each column represents a variable and each row a program point (before and after an instruction). A block is colored in a column if the variable is live at that point. Red and green blocks indicate where a variable is defined or used with red indicating the variable must be in a register at that point and green that the variable can still be accessed even if it is in memory.

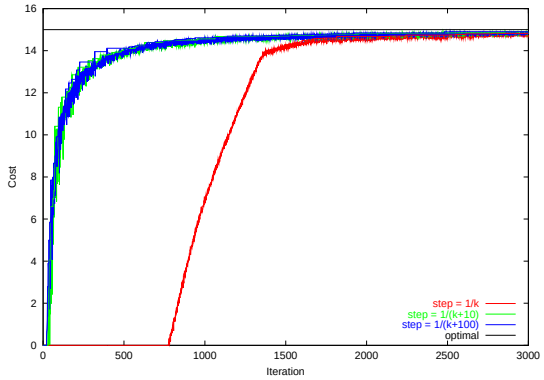

Figure 4: The convergence of the Lagrangian subproblem to the optimal value (15) over 3000 iterations using the ratio method for determining step sizes. The thicker lines are the "best so far" values for each approach while the thin lines are the actual values in each iteration.

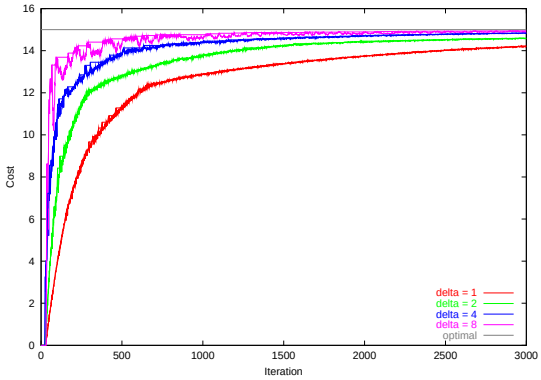

Figure 5: The convergence of the Lagrangian subproblem to the optimal value (15) over 3000 iterations using Newton's method for determining step sizes. The thicker lines are the "best so far" values for each approach while the thin lines are the actual values in each iteration.

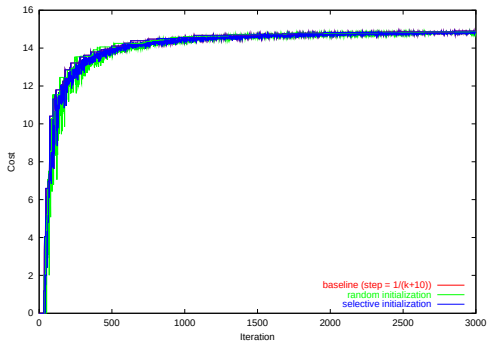

Figure 6: The convergence of the Lagrangian subproblem with the step size of the kth iteration being $1/(k + 10)$ evaluated with two different initialization schemes.

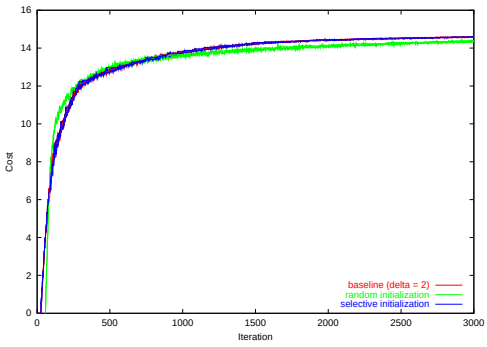

Figure 7: The convergence of the Lagrangian subproblem using Newton's method with a δ of 2 to compute the step size using two different initialization schemes.

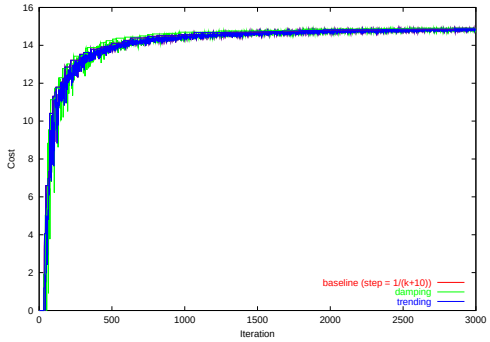

Figure 8: An evaluation of two techniques for speeding up convergence when the ratio method is used to determine step size.

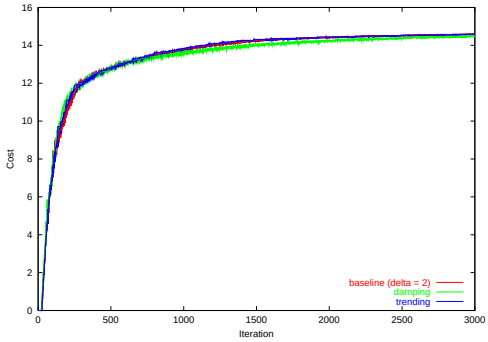

Figure 9: An evaluation of two techniques for speeding up convergence when Newton's method is used to determine step size.

Initially all prices are set to zero. If instead they were initialized to be close to the final prices the subproblem would converge faster. Two initialization strategies are shown in Figures 6 and 7. The random strategy simply initializes the prices to a random number between 0 and 1. The selective initialization strategy initializes the prices of edges going into memory nodes so that a variable pays a price for a store (the actual cost of which is absorbed by the anti-variable). This technique is successful, particularly with Newton's method.

Two attempts to speed up the convergence are shown in Figures 8 and 9. Both approaches keep a history of the direction that each price has been taking (increasing or decreasing). The trending approach doubles the step size if the price has been monotonically increasing or decreasing for the last 4 iterations. The damping method takes an average of old and new price if the price has been flip flopping between increasing and decreasing in an attempt to dampen oscillations. The trending technique is effective with Newton's method, but this gain is less than the gain of using selective initialization and the effects are not additive.

7.3 Heuristics

The purpose of the Lagrangian subproblem converging towards the optimum is to guide feasible solution finders. The $1/(k+10)$ ratio step selection and selective initialization techniques are used for evaluating all the solution finders.

The results of applying the Greedy Shortest, Iterative Shortest, and Fixed Iterative Shortest heuristics are shown in Figure 10. All these solutions consider only the shortest feasible path of each variable through the priced network. The quality of the the Fixed Iterative solution does not seem to suffer from not recalculating the next most expensive variable as in the Iterative Shortest solution. Figure 11 is a histogram of the number of times each procedure found a solution of a given cost. The iterative solutions perform significantly better than the simple greedy solution.

The k-Choice solution finders consider the k-shortest paths through the priced network when choosing an allocation for a variable. Of the k-shortest paths, only those with prices within a threshold, in this case the current step size multiplied by the lifetime of the variable, of the shortest path are considered. The lowest cost heuristic simply selects the path with the lowest unpriced cost. Its performance is shown in Figures 12 and 13. Having more paths to select from increases the effectiveness of the heuristic up to a point. As more paths are made available to choose from, the likelihood of the heuristic choosing poorly increases resulting in the decreased effectiveness of the 16-choice case.

The cost/history heuristic is evaluated in Figures 14 and 15. The addition of history information seems to have the largest effect on the 8-choice and 16-choice cases. While the heuristic is successful in increasing the effectiveness in the 8-choice case, it has a dramatic negative impact on the 16-choice case.

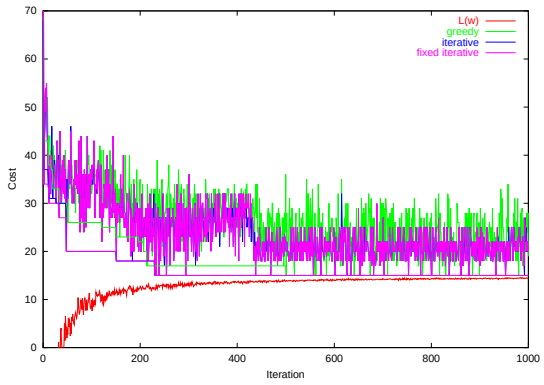

Figure 10: The results of three feasible solution finders over 1000 iterations. These solution procedures do not converge on the optimal, but do manage to repeatedly find optimal solutions.

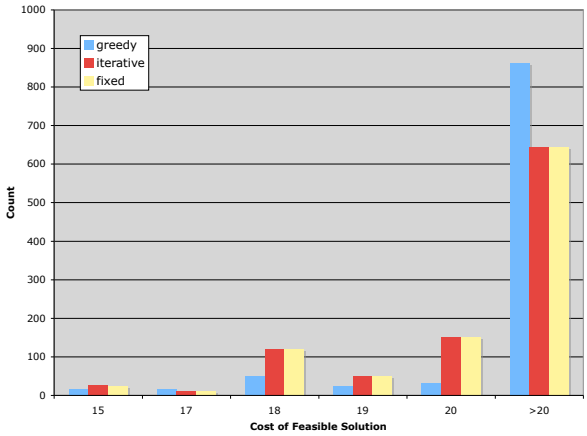

Figure 11: The number of times (out of 1000 iterations) each solution procedure found a solution of a given cost.

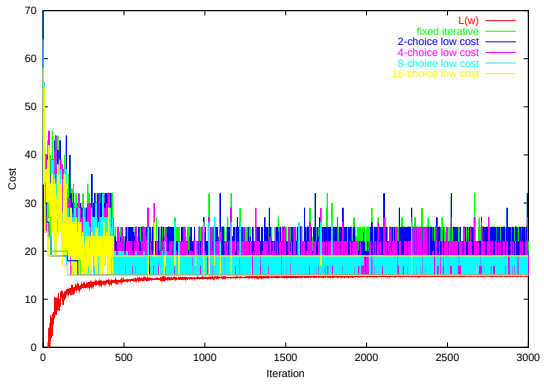

Figure 12: The performance of the lowest cost path selection heuristic over 3000 iterations with different limits on the number of paths considered.

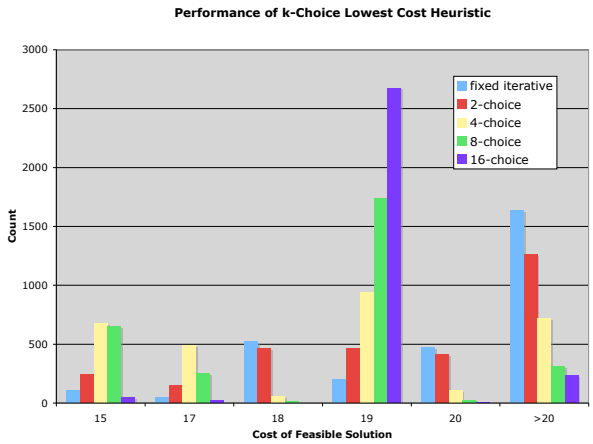

Figure 13: The number of times (out of 3000 iterations) each k-choice solution procedure found a solution of a given cost using the low cost heuristic.

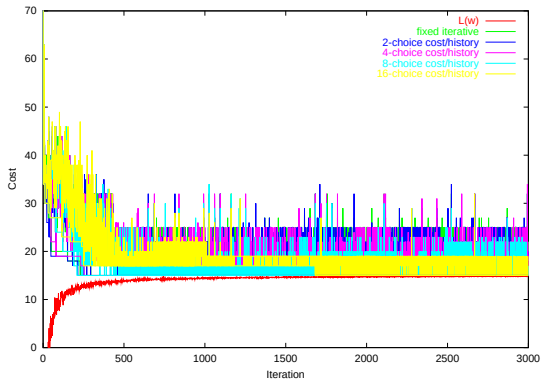

Figure 14: The performance of the cost/history path selection heuristic over 3000 iterations with different limits on the number of paths considered.

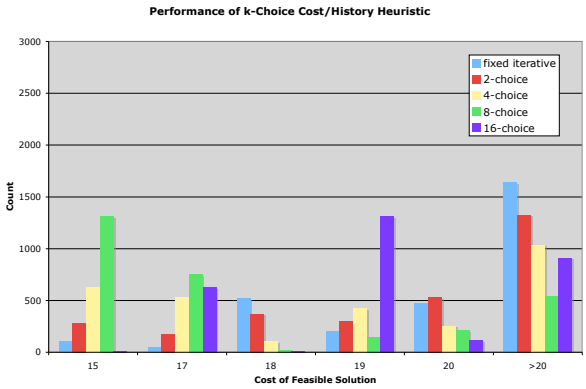

Figure 15: The number of times (out of 3000 iterations) each k-choice solution procedure found a solution of a given cost using the cost/history heuristic.

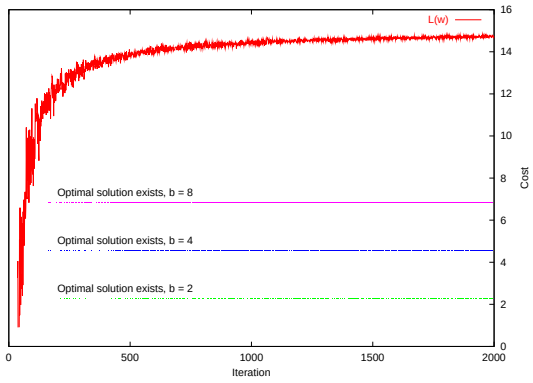

Figure 16: The existence of an optimal solution in the Lagrangian pruned search tree with different branching factors as a step function.

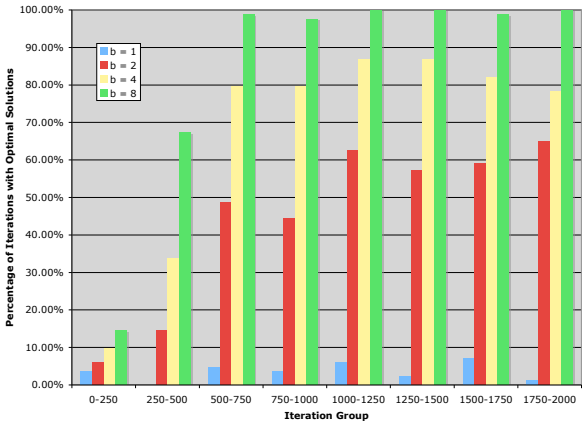

Figure 17: As the Lagrangian subproblem converges the likelihood of an optimal solution existing in the pruned search space increases and smaller branching factors are necessary to find an optimal solution.

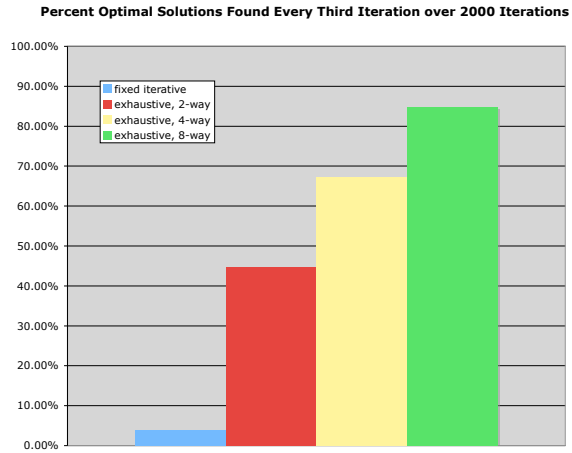

Figure 18: The percentage of searches that resulted in an optimal solution using different search strategies and sampling every 5 iterations over 2000 iterations.

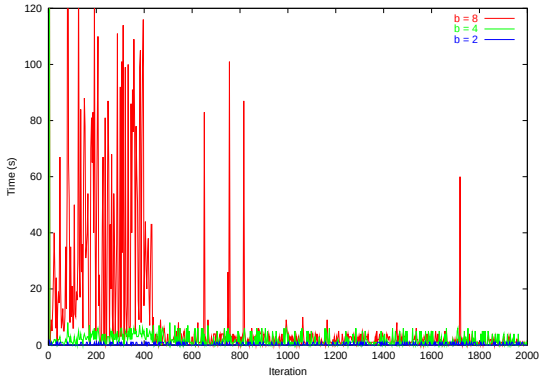

Figure 19: The time taken in seconds for an exhaustive search of the Lagrangian pruned search space to find an optimal solution.

7.4 Exhaustive Search

An exhaustive search of all possible allocations would be computationally intractable, even for a small example, but the priced network produced by the Lagrangian subproblem solution can be used to substantially prune this search. For a given branching factor b the search considers only the b shortest paths of each variable in the pruned network. If the Lagrangian approach is effective, then as $L(w)$ converges to the optimal solution a smaller and smaller branching factor should be necessary to find the optimal solution. The results of searches with different branching factors for every three iterations up to 2000 iterations are shown in Figure 16. As $L(w)$ converges, the optimal solution is increasingly likely to exist for smaller branching factors. Figure 17 shows how the percentage of iterations where an optimal solution exists increases as $L(w)$ converges.

Although the exhaustive search method is not always successful at finding an optimal solution, it does substantially better than the single path heuristic methods as shown in Figure 18. The fixed iterative search method is essentially an exhaustive search with a branching factor of one. The exhaustive search results can be viewed as an upperbound bound on the performance of k-choice heuristics.

7.5 Running Times

Technique	*Small Example Time (s)*	Large Example Time (s)
$L(w)$ maximization	.019	.28
Greedy Shortest	.52	3.7
Iterative Shortest	3.1	215
Fixed Iterative	.06	.90
2-Choice	.07	1.1
4-Choice	.08	1.2
8-Choice	.10	1.4
16-Choice	.14	1.8

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.

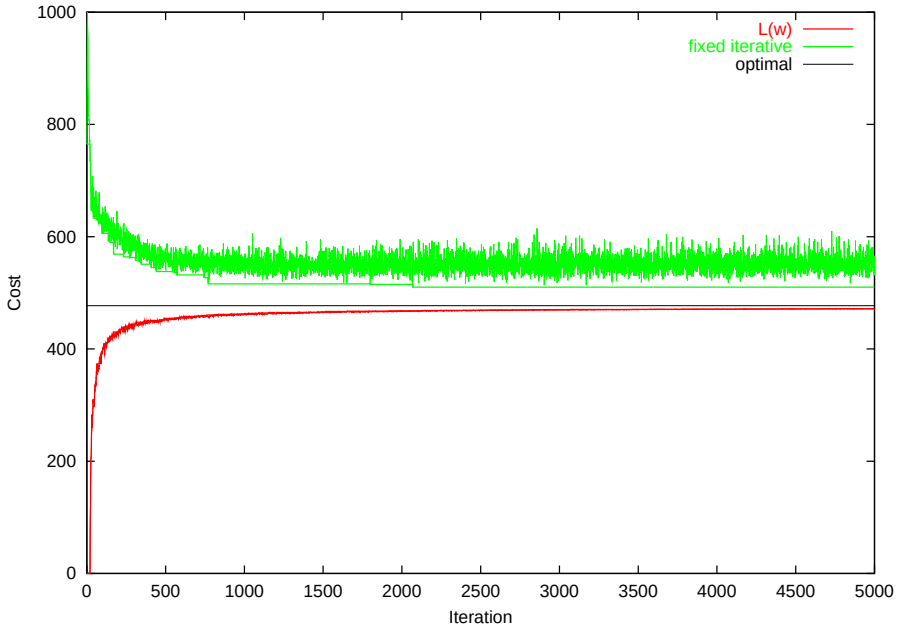

Figure 20: The behavior of the Lagrangian solution finder on a large example using the ratio method ($1/(k+10)$) and a fixed iterative solver.

In order for the Lagrangian approach to be useful for register allocation, it must offer performance competitive with traditional allocators and scale well as the problem size increases. We evaluate the performance of the various algorithms operating on both the small example from the previous sections and a larger function, the `squareEncrypt` function of `pegwit_d` benchmark. This function has 412 instructions and 150 variables. The value of the optimal solution is 477. The convergence properties and behavior of the fixed iterative feasible solution finder for this example are shown in Figure 20. After 5000 iterations, a solution of cost 510 that is guaranteed to be within 8.1% of optimal is found, but the slow rate of convergence at this point makes it unlikely that a much better solution will be found.

The running times of the various techniques per a single iteration are shown in Table 2. Using the 2-choice solution finder an optimal solution for the small example is found in iteration 216 (after about 15 seconds), but it isn't proven to be optimal until iteration 486 (about 35 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 745 (about 879 seconds). Although the cost of the Lagrangian maximization is necessary and unavoidable, the cost of the feasible solution finders can be mitigated by waiting until after a ramp up period to run them and running them less frequently

`gcc` Register Allocator	Small Example Time (s)	Large Example Time (s)
Local/Global (default)	.0085	.091
Briggs/Chaitin (-fnew-ra)	.065	.2

Table 3: The total time, in seconds, of the full register allocation pass for both of `gcc`'s allocators when run on a 1.8Ghz Pentium 4 system

Solver	Small Example Time (s)	Large Example Time (s)
`glpk`	553	∞
CPLEX 7.1	107 (191)	4653 (8295)

Table 4: The total time, in seconds, to determine the optimal solution to the MCNF register allocation problem formulation using the open-source `glpk` 4.4 solver[15] and the much more powerful commercial solver, CPLEX 7.1[12]. The CPLEX solver was run on a 1Ghz Pentium 3 and the `glpk` solver on a 1.8Ghz Pentium 4. 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. The `glpk` solver was not capable of solving the large example as it encountered unrecoverable instability after running for more than 10 days.

than once an iteration.

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 make no claims to optimality. Traditional integer linear program solvers can be used to solve for the MCNF formulation of the register allocation problem. The solution times for the GNU Linear Programming Kit (`glpk`)[15] and the powerful commercial CPLEX[12] solver are shown in Table 4. The Lagrangian approach compares favorably with the optimal solvers and has the added advantage of being progressive (capable of producing a suboptimal solution with optimality bounds guarantees at any point).

The times of performing exhaustive searches with various branching factors are shown in Figure 19. Once the ramp up period is over, the times are more or less stable. The relatively good performance of the exhaustive searches is due to the ability to stop searching once an upper bound is exceeded. For these searches an upper bound equal to the optimal solution was used as the goal was just to establish the existence of the optimal solution in the pruned search space, rather than to time to the cost of a full exhaustive search with no known previous upper bound.

7.6 Code Quality

	Small Example	Large Example
Theoretical Overhead	15	510
Actual Overhead	4	781
`gcc` Overhead	26	946

Table 5: The theoretical and actual overheads for the Lagrangian approach and the measured overhead of the default `gcc` allocator. The overheads include the cost of the preconditioning pass, which is not included in the optimum solution cost found by the solver.

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 MCNF formulation, the actual overhead of register allocation might be different than the theoretical overhead. Errors in the interface between the Lagrangian solver and `gcc`'s register allocator may also distort the results. Overhead is measured by taking the difference in code size from before and after register allocation. Some x86 instructions are difficult to size before register allocation as their size depends on which registers are allocated to what operands and in these cases the pre-register allocation tends to be conservative resulting in negative overhead.

The theoretical and actual overheads for the large and small example are shown in Table 5 along with the overheads incurred by the default `gcc` allocator. Since the optimal result for the large example is unknown, the best result found after 5000 iterations using the fixed iterative solver is shown. The large difference in theoretical and actual overheads for the large example is because the MCNF formulation currently does not correctly model the cost of stack accesses. 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.

8 Conclusion

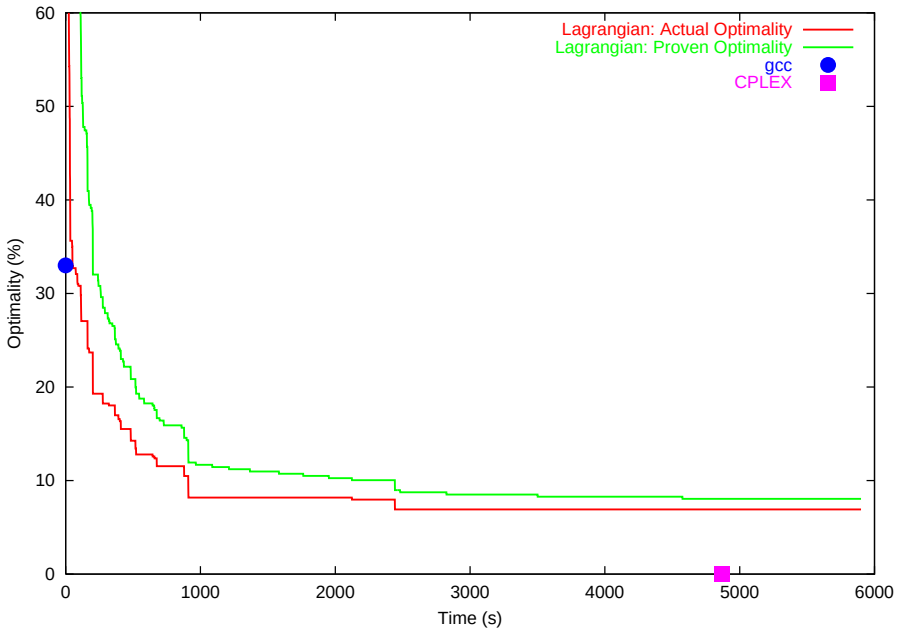

Figure 21: The progressive nature of the Lagrangian solution approach. The optimality percentage is the percent increase in overhead from the theoretical optimal solution. As the algorithm computes, a better solution and a better optimality bound on the solution are found.

Lagrangian relaxation is an effective way to guide heuristics and prune the search space of difficult multi-commodity flow problems. As currently implemented, this approach neither finds a decent solution as quickly as traditional allocator nor is guaranteed to find an optimal solution as with ILP-based allocators. However, as Figure 21 shows, this approach offers a compromise between those two allocators. A solution with provable optimality guarantees that is significantly better than traditional allocator solutions can be found substantially faster than with the ILP techniques.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., 1993.

[2] Andrew W. Appel and Lal 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] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill code minimization via interference region spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

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

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

[7] Preston Briggs, Keith D. Cooper, and Linda 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] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of the 1998 International Compiler Construction Converence*, 1998.

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

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

[13] Timothy Kong and Kent 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.

[14] Vincenzo Liberatore, Martin Farach, and Ulrich Kremer. Hardness and algorithms for local register allocation.

[15] Andrew Makhorin. *GNU Linear Programming Kit Reference Manual Version 4.4*. Free Software Foundation, Boston, MA, January 2004.

[16] Mayur Naik and Jens 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.

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

[18] Michael D. Smith and Glenn Holloway. Graph-coloring register allocation for irregular architectures.