6-1-1993

Register Allocation via Weighted Graph Coloring (Technical Summary)

Russell W. Quong
Purdue University School of Electrical Engineering

Shu-Ching Chen
Purdue University School of Electrical Engineering

REGISTER ALLOCATION VIA WEIGHTED GRAPH COLORING

RUSSELL W. QUONG
SHU-CHING CHEN

TR-EE 93-23
JUNE*1993*

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

Register Allocation via Weighted Graph Coloring (Technical Summary)

Russell W. Quong and Shu-Ching Chen
School of Electrical Engineering
Purdue University
W. Lafayette, IN 47906
(317) 494-1739
quong@ecn.purdue.edu

Abstract

Register allocation by coloring an interference graph is a common technique. We introduce the weighted interference graph (WIG) which improves upon previous approaches in the following ways: (i) the cost of a coloring accurately models the cost of the register assignment, (ii) arbitrary register spills are handled naturally, as the coloring implicitly determines when and what registers to spill, and (iii) the allocation granularity is freely scalable from an instruction level to a per variable level. In particular, at any granularity, a WIG models the degree of interference between two vertices, which improves upon the traditional interference graph.

The weights in a WIG incorporate usage counts to reflect the relative cost for spilling a quantity. We model the savings from keeping a variable in a register via negative edge weights. Our method is ideal when near-optimal register allocation is desired and spill code is unavoidable, such as for large frequently-executed basic blocks, as might be produced by inlining and loop unrolling. We show that a WIG gives better allocations than a traditional interference graph when it matters most, namely when there are not enough registers.

1 Introduction

Register allocation is perhaps the single most important compiler optimization [HP90]. Many current compilers use graph coloring to do register allocation. Despite the existence of reasonably good allocation algorithms, the need for improved algorithms persists. For example, global register allocation, procedure inlining, and loop unrolling all yield large interference graphs in which spilling may be unavoidable, a worst case scenario for many previous coloring algorithms.

Register allocation maps variables to physical registers so that memory references are minimized, subject to constraints of register availability. Unfortunately, the optimal register allocation problem RA is an NP-complete problem. Chaitin et al. [Cha82] popularized the idea of transforming register allocation into graph coloring, which is also NP-complete.

The graph coloring decision problem GC asks "Given an undirected unweighted graph G, is it possible to color the vertices of G using $\leq$ k colors such that no edge connects two vertices of the same color?" In register allocation, each vertex of the interference graph G represents definitions and uses of a variable. A color represents a physical register. Two vertices u and v in G have an edge if their corresponding variables are live simultaneously, because if both u and v are assigned to the same register, they will "interfere" with each other. A spill occurs when a variable must be moved between memory and registers.

In this paper, we question whether coloring an unweighted interference graph (UIG) is a good model for register allocation. Instead, we recommend the use of a weighted interference graph (WIG) that has both edge and vertex weights. A WIG vertex represents an arbitrary subset of the sites (uses and definitions) of a variable. All weights are scaled by the frequency of execution. A negative weight represents the relative savings of keeping a variable in a register versus memory.

Our WIG can be arbitrarily scaled for different allocation granularities, trading off coloring time for allocation quality. At the coursest extreme, a vertex represents all occurrences of a variable (e.g. the entire live range); at the other extreme, each vertex represents an operand (register or memory address) in a machine instruction. An uncolored vertex resides in memory. An edge exists between vertices v_x and v_y if (i) x is live at v_y and (ii) v_y is the next site of y after v,.

The weights are defined as follows.

- Vertex weight of v: $w(v)$ = the savings in keeping v in a register rather than memory, which would require a load or a store operation.
- Edge weights:
  - If vertices v_x and v_y represent different variables, x and y, the $w(v_x, v_y) =$ is the cost to store v_x and to load v_y from memory.
  - If v_x and v'_x represent the same variable, $w(v_x, v'_x) =$ the savings of not having to reload v'_x.

Positive edges include spill and load costs, but negative edges only include load costs. The edge-weight asymmetry and the vertex weights are necessary to model both register and memory operands. We shall prove coloring a WIG is an accurate model for the actual memory access costs of the corresponding register assignment, which is not true in UIGs.

In Section 3 we review previous work register allocation via graph coloring. In Section 3 we define the weighted interference graph. In Section 4 we prove a WIG is accurate in modelling the register allocation. In Section 5 we discuss scaling the granularity of the interference graph. Finally, in Section 6 we give preliminary results on coloring WIGs versus UIGs.

2 Previous Work

Despite the large number of allocation by graph coloring algorithms [BGG+89] [BCKT89] [CH90] [GSS89] [LH86] [CK91], most algorithms can be loosely categorized via the three characteristics: granularity, ranking, and spill hueristic.

The allocation granularity is the code granularity at which registers are assigned. Finer granularities potentially offer better allocations, but result in larger interference graphs requiring more compile time and space. Chaitin et al. [Cha82] and Briggs [BCKT89] use per-variable, the coursest granularity, where a vertex represents the entire lifetime of a temporary variable. A variable assigned to a register will always be in that register, and a spilled quantity is always in memory. Chow and Hennessey [CH90] do the same, but their lifetimes are sets of basic blocks called a "live-range". These algorithms have the advantage of relatively small graphs. Gupta et al. [GSS89] use clique separators to further decompose an interference graph into a series of smaller subgraphs.

As an example, Figure 1 shows a code fragment, in which rectangles represent basic blocks and arcs represent possible control flow paths. We have also labelled each control flow arc with a relative usage count, which will be used in later examples. The figure shows we will execute basic block B-2 20 times — once up entry and then 19 more times from the loop. Similarly, B-3 will execute five times. The usage counts can be either estimated or from profile information.

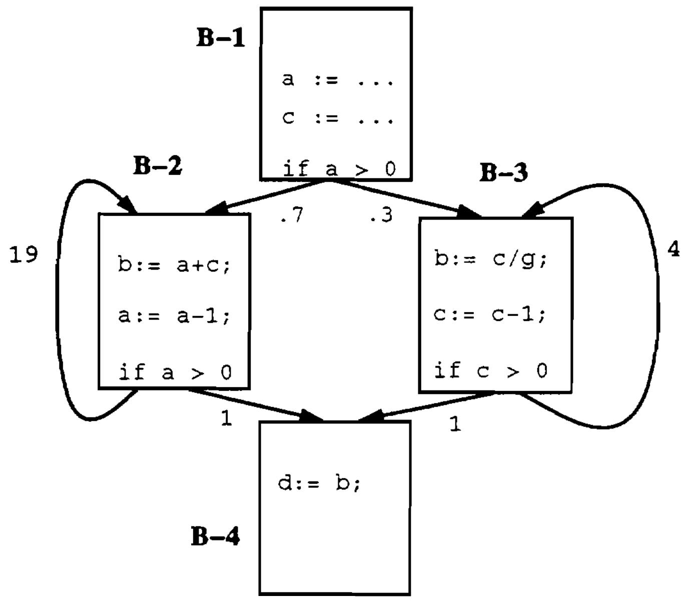

Figure 1: An example flow graph.

Figure 2 shows the interference graph for the code in Figure 1 using live-ranges as the granularity. Variable g is global and hence interferes with all other variables. There are edges between **a** and **c** because both are live at the exit of block **B-1**; similarly, the edges `(b, a)` and `(b,c)` are due to block B-2.

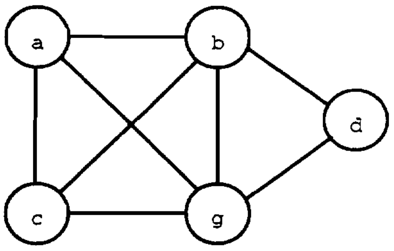

Figure 2: An interference graph based on live-ranges.

Callahan and Koblenz [CK91] use a tree of increasingly coarse granularities as the algorithm traverses from the leaves (inner loops) to the root (entire program). By allocating inner loops at fine granularity, this approach has the advantage of being "sensitive to local usage patterns while retaining a global perspective."

We define ***per-instruction*** granularity as the finest granularity possible, in which register assignments can change and spills can occur at any machine instruction. In theory, this granularity allows for the best possible allocation, in practice, a per-instruction interference graph for an entire program is simply too large. Previous coloring algorithms could not do per-instruction allocation, because there was no way to indicate a frequently used variable should stay in a register.

Ranking is how the coloring algorithm orders the vertices for coloring. Let C_{mem} be the estimated cost of not coloring a vertex, including all spill costs. Previous algorithms have used C_{mem} in some way. For example, Chow and Hennessey [CH90] rank each vertex using ($C_{mem}/$ size-of-live-range) as an estimate for the average savings gained by assigning that vertex to a register. Chaitin [Cha82] and Briggs [BCKT89] rank vertices via (C_{mem}/vertex-degree) so that vertices likely to interfere with many others are colored last. Bernstein et al. [BGG+89] pick the best coloring after trying several ranking methods of the form ($C_{mem}/$

fn(area,degree)), where the area of a vertex is an estimate of how much interference a vertex causes with emphasis given to nested loops and congested regions.

At some stage if the interference graph becomes uncolorable, traditional coloring algorithms use a ***spill heuristic*** to proceed further. The spill heuristic simplifies the graph, either by removing vertices [Cha82] [BCKT89] or by splitting a vertex into two vertices of lower degree [CH90]. Frequently the spilling heuristic and ranking are closely related. Most of the previously mentioned algorithms first disassemble the graph using the spill heuristic to repeatedly remove the lowest ranking vertex. Then the algorithm reconstructs the graph in the reverse order of the break-up, coloring vertices as they are added back.

One drawback of using traditional graph coloring is that it is a yes-no decision problem — if the graph is colorable then "success" otherwise "failure". A spill heuristics will be required sooner or later. However, register allocation is not a yes-no decision problem but an optimization problem. All register assignments have an associated run-time cost. Thus, it makes sense to model RA as an optimization problem. Also, from a theoretical standpoint, NP-complete decision problems do not have good approximate solutions, but NP-complete optimization problems frequently have good practical solutions, e.g. bin packing and the travelling salesman problem [Baa88] [PS82].

Our philosophy is similar to that of Proebsting and Fischer [PF92] in that we wish to avoid ad hoc spill heuristics, and we model the instruction-level interference between variables. However, their approach differs from ours because they do not do graph coloring but use a probablistic approach to determine register assignments.

3 The weighted interference graph

3.1 Definitions and assumptions

For the remander of this paper, we use the following terminology. M is the underlying machine. R is the set of registers of M; there are $|R|$ or k registers. Individual registers are denoted R_i. The ***run-time cost*** T_{mem} of a register assignment is the ***additional*** execution time for loads and stores for both (i) spills or (ii) accessing variables stored in memory. T_{mem} does not include the time to perform calculations. $T_{\text{mem}} = 0$ for a "perfect" allocation with unlimited registers. Program variables are denoted by x, y, or z.

A weighted interference graph G = (V, E) is a undirected ***vertex and edge weighted*** graph with n vertices and m edges. V is the vertex set; E is the edge set. Vertices are denoted by u or v, and may have subscripts. We implicitly assume u appears before v in the control flow graph. Each vertex v has a $w(v)$; each edge (u, v) has a weight denoted $w(u, v)$. Vertex and edge weights are real numbers. A ***coloring*** of G, $\text{Color}(G)$, assigns a color to each vertex; $\text{Color}(v)$ denotes the color assigned to vertex v.

The ***cost*** of an edge (u, v) in a particular coloring $\text{Color}(G)$ is (i) the weight of the edge if u and v are the same color or ***(ii)*** zero if u and v are different colors; this cost is denoted $\text{Cost}((u, v), \text{Color}(G))$ or $\text{Cost}(u, v)$ if the coloring is understood from context. An edge connecting same-colored vertices is ***active.*** Note if $w(u, v) < 0$, we are rewarded not penalized for coloring u and v the same color. The cost of a vertex v is $w(v)$ if the vertex is colored. The total cost of a coloring, $||\text{Color}(G)||$, is the sum of all vertex and edge costs $= \sum_V \text{Cost}(v) + \sum_E \text{Cost}(u, v)$. A coloring ***contradicts*** an edge (u, v) if $\text{Cost}(u, v) = \max(w(u, v), 0)$.

As in [CH90], the term ***variable*** represents any possible value that could reside in a register, including constants. Depending on when register allocation is done, variables can include intermediate results such as an address calculation of an array element. At some point in the flow graph, a value (or variable) is ***live***[1] if

[1] **The notion of liveness differs depending on whether values or variable names are used. Either definition can be used. In the examples from this paper, liveness refers to a value.**

it has a future use with no intervening definition; a value (variable) is dead otherwise. A ***thread*** is an acyclic path through the flow graph. Usage(u, v) is the usage count for the thread connecting u and v. Usage(u, v) can be either estimated or from previous profile information.

A ***site*** of variable x is either a definition or use of z. We assume each vertex represents a set of sites for a ***single*** variable. At one extreme, in per-variable granularity, a WIG vertex represents all the sites for a variable; at the other extreme, in per-instruction granularity, each vertex represents a single site (i.e. a machine instruction operand). A site can be assigned to at most one vertex. Let var(v) denote the variable for v. Site v for variable x is a ***next-site*** after site u for variable y, if there is a thread from u to v with no intervening uses or definitions of x or y. This definition applies even when x and y are the same variable. In Figure 3, the next-sites are connected by solid lines; v_1 is not a next-site for u_1, because u_2 intervenes. (Throughout the rest of this paper, we will align the sites for a variable in a column.) Because of branches, a site may have many next-sites. Finally, the phrase "assigning v to a register R_i" means assigning all sites represented by v to R_i.

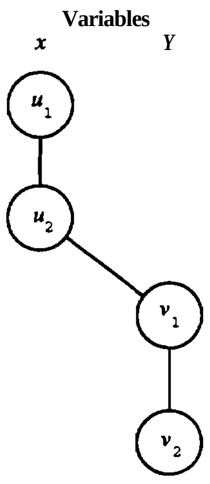

Figure 3: Edges showing the next-site relationship between variables x and y.

3.2 No WIG is a perfect model

Our goal is to have the cost of coloring a WIG accurately model the run-time costs T_{mem} of the corresponding register allocation. We use edge weights to account for the cost to spill interfering vertices. We assume that the coloring algorithm is not awful, because with arbitrarily bad colorings, the WIG becomes a poor model.

Claim 1 *No weighted interference graph can be 100% accurate for all colorings.*

Proof: We show that the presences of an edge is necessary for some colorings but erroneous for other colorings. Consider a thread with three vertices v_1, v_2, and v_3, for three different variables, x, y and z as shown in Figure 4. Edges correctly accounting for interference are bold. Clearly edges (v_1, v_2) and (v_2, v_3) are needed if v_2 interferes with v_1 or v_3, respectively. If Color(v_1) = Color(v_3) $\neq$ Color(v_2), then v_1 and v_3 interfere which requires a third edge (v_1, v_3) for correctness. But if Color(v_1) = Color(v_3) = Color(v_2), then the third edge gives an incorrect penalty, as the other two edges (v_1, v_2) and (v_2, v_3) already account for the spill cost. □

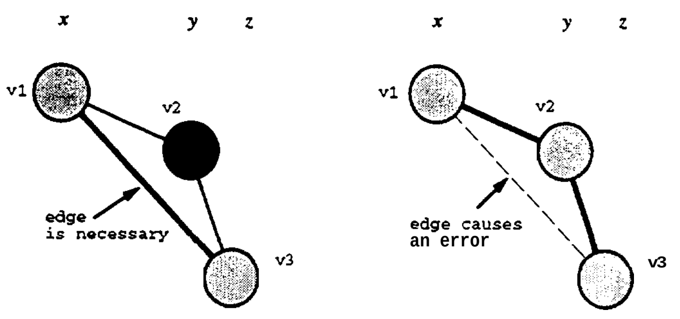

Figure 4: No WIG can be correct for all colorings.

3.3 Edges in the WIG

We define edges in a WIG according to the nezt-site edge rule: an edge (u, v) exists if (i) v is a next-site after u, and (ii) u is live at v. This rule applies for both cases, $\mathrm{var}(u) = \mathrm{var}(v)$ and $\mathrm{var}(u) \neq \mathrm{var}(v)$. As an example, Figure 5 shows the WIG for the basic block B-2 from Figure 1 at per-instruction granularity. We have labelled the vertices for variable **a,** as a–1, a–2, a–3 and a–4. There is an edge (**a-1**,**b**) because b is the next-site (for variable b) after vertex **a-1**. There is an edge (**a-1**,**a-2**) because **a-2** is the next-site (for a) after **a-1**. There is no edge **(a-2,a-3)** because the value a–2 is not live at a–3. The striped edges have negative weights.

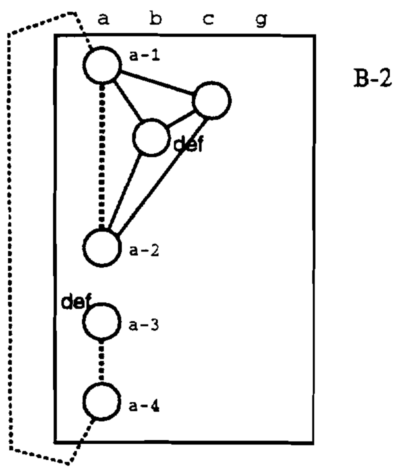

Figure 5: Example of edges in a WIG.

The edge weight $w(u, \mathrm{v})$ reflects the relative run-time cost if u and v were to occupy the same register. Positive edges include spill and load costs, but negative edges only include load costs.

Condition	Edge weight
$\mathrm{var}(u)$ # $\mathrm{var}(v)$	$w(u,\mathrm{v})$ =Usage(u,v) * (cost to spill u + load v)
$\mathrm{var}(u) = \mathrm{var}(v)$	$w(u,\mathrm{v})$ =–1 * Usage(u,v) * (cost to load v from memory)

If the machine M allows for memory operands (i.e. M is not load-store), we allow a vertex v to remain uncolored, which corresponds to leaving v in memory. An uncolored vertex v will not incur any edge costs, as it does not conflict with other register assignments. To model the memory-access cost of an uncolored vertex, the ***vertex weight*** is the savings from keeping v in a register versus in memory.

Condition	Vertex weight
v is a use	$w(v) =$ -1 * Usage(v) * (time to load v − time to get v from a register)
v is a definition	$w(v) =$ -1 * Usage(v) * (time to store v − time to put v in a register)

A vertex weight is always negative and is a differential cost. Thus, a vertex is rewarded for being in a register by $w(v)$ if it is colored (with any color). It seems more natural to charge a vertex $|w(v)|$ if v is uncolored, but this scheme breaks down if vertices represent more than one site.

The next-site edge rule keeps edges sparse, which in turn minimizes compiler storage requirements and speeds up coloring. Claim 1 shows that any definition of edges will introduce potential inaccuracies, but the next-site edge rule handles the common cases correctly.

Consider the case where the next-site edge rule breaks down. Let $\text{var}(u) =$ y and $\text{var}(v_1) = \text{var}(v_2) = x$, where v_1 is a next site after u, and v_2 is a later site of 2. If u is live at v_2, u and v_2 could interfere, but our WIG does not have edge (u, v_2). For example, in Figure 5, variable b is always live and could interfere with all the vertices of variable a, yet there is no (b,a-4) edge. Why?

Claim 2 ***The nezt-site edge rule is rarely inaccurate and the inaccuracies are small for reasonable colorings.***

Proof See Appendix A.

3.4 Modelling other factors

We believe that by modifying vertex and edge weights we will be able to model other factors involved in register allocation. For example, the effect of rematerialization [BCT92] can be modelled by decreasing the vertex weight.

4 Accuracy of edge costs

The weighted graph coloring optimization problem (WGC) is "Given a weighted undirected graph G, find a coloring of G using $\leq k$ colors with the minimum cost." WGC is NP-complete. [2] Let K_{min} be the optimal cost of coloring G with an unlimited number of colors. For a WIG, K_{min} is the cost of assigning each variable a separate color. Clearly, we are credited for all negative weight edges and are never penalized. K_{min} is a constant for any G. We now prove that $||\text{Color}(G)||$ is proportional to T_{mem}.

Theorem 1 ***Let T_{mem} = the actual run-time cost of a register assignment corresponding to a coloring of G, $Color(G)$. Then $||Color(G)|| - K_{\text{min}} \approx T_{\text{mem}}$.***

Proof See Appendix B.

5 Granularity

5.1 Instruction level allocation

Figure 6 shows the instruction-level WIG for the flow diagram in Figure 1. The variable g is global and is live upon entry to block B-1, resulting in the fictitious def node for it. As before, negative weight edges are

[2] A problem Π is NP-complete if (i) Π is at least as hard as another NP-complete problem and (ii) $\Pi \in NP$ via a succinct certificate [CLR90, pg. 927]. *WGC* **is clearly** *as* **hard as** *GC* **and for the corresponding decision problem, the coloring is a polynomial time verifiable certificate, showing** *WGC* $\in NP$.

striped; positive weight edges are solid. Unspecified edge weights are +2 or −1, assuming the run-time cost for both loads and stores is 1. We have omitted the vertex for variable d in the figure.

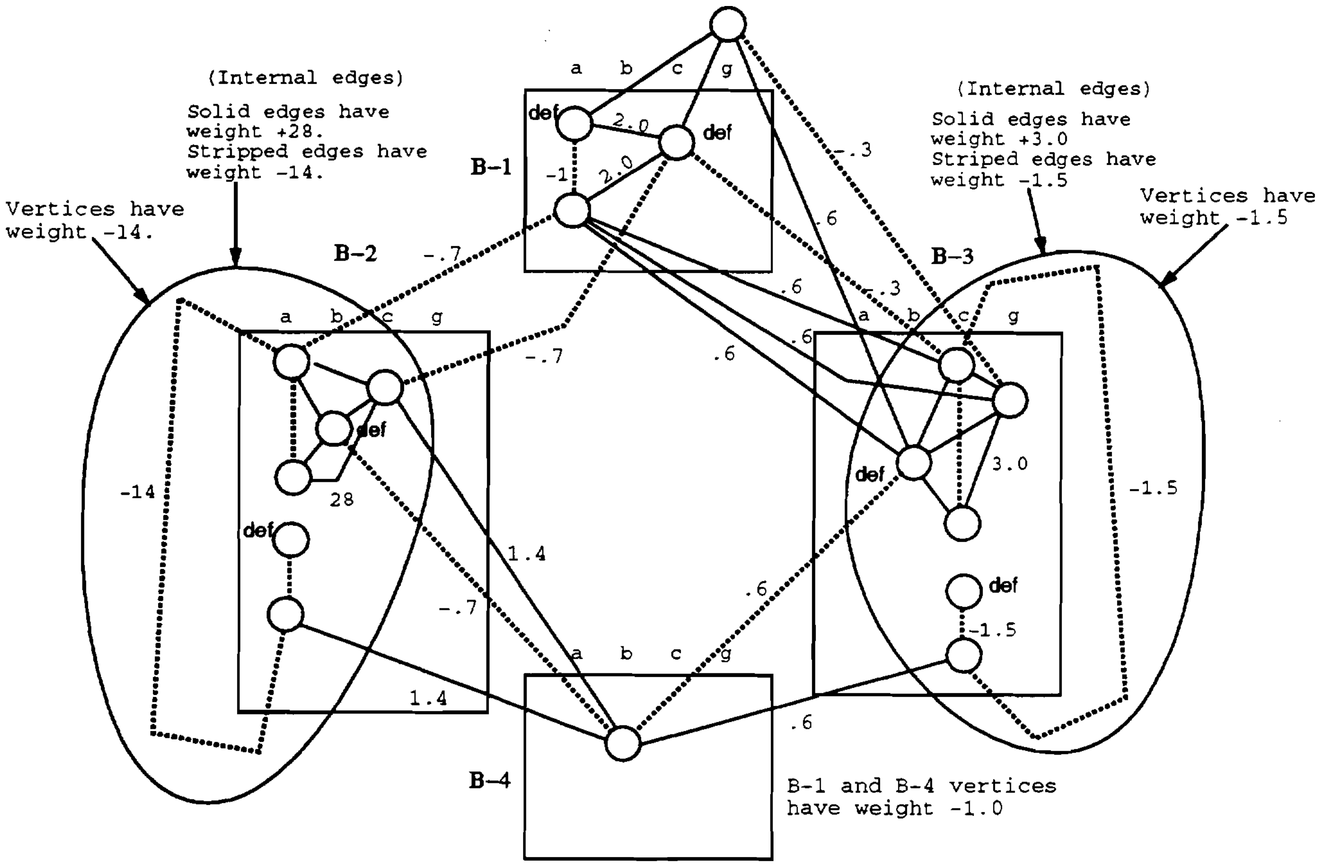
(Internal edges)
Solid edges have weight +28.
Stripped edges have weight -14.
Vertices have weight -14.
(Internal edges)
Solid edges have weight +3.0
Striped edges have weight -1.5
Vertices have weight -1.5
B-1
B-2
B-3
B-4
a b c g
def
2.0
-1
-.7
-.3
.6
1.4
28
-14
3.0
-1.5
B-1 and B-4 vertices have weight -1.0

Figure 6: The instruction-level WIG for Figure 1, with variable d omitted.

5.2 Per-variable allocation

We illustrate how to construct vertex and edge weights when a vertex represents several sites. Figure 7 shows the WIG at per-variable granularity for the code in Figure 1. Note that Figure 7 does not include site for global g before B-1. All sites for a variable are represented by a single vertex. The vertex and edge weights are simply the sum of the corresponding weights in the per instruction WIG.

$$w(a) = \sum_{\text{var}(v) = a} w(v) \qquad \text{and} \qquad w(a, b) = \sum_{\substack{\text{var}(u) = a \\ \text{var}(v) = b}} w(u, v).$$

In our example, vertex c has weight $w(c) = -25.0 = -21.0$ (from vertex weights) −4.0 (from edges of the form (u,, v_c)). In detail, the vertex weights for $w(c)$ are −1 (B-1) −14 (B-2) −4 ∗ 1.5 (B-3) = −21; the edge weights are −0.7 (B-1 to B-2) −0.3 (B-1 to B-3) + 2∗ −1.5 (internal B-3) = −4.0. Similarly, vertex a has weight $w(a) = -87.7 = -58$ (vertex weights) −29.70 (edges weights). Finally, the edge weight $w(a, c)$ is the run-time cost of having a and c share the same register. We get $w(a, c) = 60.6 = 4$ (internal B-1) + 0.6 (B-1(a) to B-3(c)) + 56.0 (internal B-2). This WIG is the roughly same size as traditional UIG.

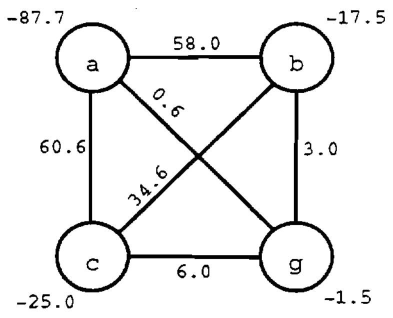

Figure 7: The WIG at per-variable granularity for Figure 1, with variable d omitted.

A per-variable WIG models the interference versus benefit between two vertices much better than a UIG. For example, in Figure 7 assume we have two available colors and we use them on a and c. Should we color g? We note that is $w(g)$ = -1.5 and $w(a, g)$ = 0.6 (the cost to store a and load g on the B-1/B-3 thread). Thus, the WIG indicates that it is better to color g with Color(a) than to leave g uncolored. Similarly, we are best off leaving b uncolored.

5.3 Heirarchical allocation

In a hierarchical approach, we make several passes over the code doing allocation on successively larger regions of code at increasingly rougher granularities. To merge the subregions together, we reconcile the subregions colorings via another coloring step. Our approach resembles that of Callahan and Koblenz [CK91], but we use WIGs to pass information between hierarchies, whereas they use "tile summaries"

We briefly outline how to combine subregions together. Let region r contain disjoint subregions $r_1, \ldots, r_n$ previously colored with $k_1, \ldots, k_n$ colors, respectively. Let p_i be an entry (or exit) point of subregion r_i. We form the graph G, as follows. The only vertices visible outside of r_i are the k_i *external* colored vertices at p_i. Initially, external vertices are uncolored in G,. The remaining live vertices at p_i are local to subregion r_i and do not have edges in G_r, as shown in Figure 8. Define edge weights between external vertices from different subregions as before. To match colors between the subregions, color G,.

As an example, suppose k = 3 and we hierarchically color the code in Figure 1 using two passes. In pass 1, we color each basic block individually; in pass 2 we combine the basic blocks. Assume after pass 1, B-1 has assigned a to a register at entry and assigns both a and c to registers at exit, namely B1:(a)→(a,c). Similarly assume B2:(a,b)→(a,b), B3:(b,c)→(b,c), and B4:(b)→(b). Note in B-4, we have used only one register for b. On pass 2, the basic blocks form the subregions. $B - 2$ vertices a and b are external; $B - 2$ vertices c and g are local and do not affect the graph. Coloring the resulting WIG determines which registers are maintained across basic blocks.

5.4 Determining Spills

There is no explicit step to handle spills, because the coloring itself contains all spill information. For example, if $\text{var}(u_x) = \text{var}(v_x)$ and $\text{Color}(u_x) \neq \text{Color}(v_x)$, then x must be spilled between u, and v_x. We insert spill code to store u along the cheapest thread between u and v.

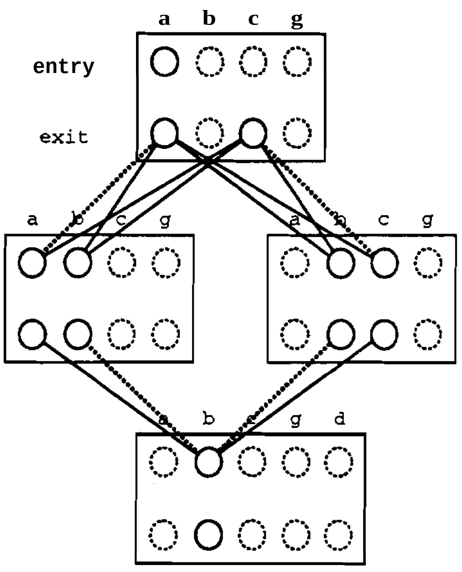

Figure 8: Weighted interference graph at the basic block hierarchy.

6 Preliminary experimental results

Our preliminary results give the following general observations.

- If there are enough or nearly enough registers, the use of a per-variable granularity WIG gives optimal or virtually optimal allocations.
- If there is a significant shortage of registers, coloring a instruction level WIG gives significantly better allocations than per-variable granularity graphs.
- Colorings from traditional interference graphs were good when there were enough registers, but relatively poor when there was a shortage of registers.

6.1 Coloring Algorithms

We colored the graphs with the following algorithms, assuming a load/store machine so that vertices could not remain uncolored. We use t_L = load cost = 1 and t_S = store cost = 1. The k colors were numbered $0, 1, \ldots, k-1$. We denote the ith vertex as v_i. In all the algorithms, vertices start uncolored. A ***prioritized coloring*** uses a metric $\mu(v)$ to order the vertices. We then color vertices in order from highest to lowest μ values. The ***best color*** for v is the color with the lowest cost given the current coloring using the specified metric. Thus, when coloring the vertices in a per-variable WIG in the order $v_1, v_2, \ldots, v_n$ the best color is an unused color. The algorithms are listed in approximate order of effectiveness, from worst to best. The full paper motivates and describes the coloring algorithms and test cases in detail.

Algorithm	Description
VertexNum	Color$(v_i) = i$ mod k. (A poor algorithm for reference.)
Vert Deg only	Prioritized best coloring, using $\mu(v)$=degree of v. I.e. color vertices with the highest degrees first, using the best color.
Vert Wgt/(1+deg)	Prioritized best coloring, using $\mu(v) = w(v)/(1 + \deg(v))$.
Vert Wgt	Prioritized best coloring, using $\mu(v) = w(v)$.
Greedy Edge	Sort the positive edges by descending weight. Assign different colors to edge endpoints, until all vertices have been colored. Do not recolor a vertex. (Attempts to minimize the penalty of the costliest edges.)
Greedy Vertex	Color $v_1, v_2, \ldots, v_n$ with the best color. using $\mu(v_i) = [w(v_i) + \sum_u w(v_i, u)]$
Rand t × p	A random coloring involving t trials of p passes. Each trial consists of randomly coloring the graph and then making p passes over the entire graph choosing the best color for each vertex. (Changes from a previous pass can affect later passes.) Choose the best coloring of the t trials. The coloring time is proportional to $O(tp)$. (For large pt products, this algorithm is impractical, but we wanted a good reference algorithm. This randomized heuristic was used by Lin [Lin65] on the traveling salesman problem, which is also NP-compete.)

The prioritized algorithms were meant to resemble traditional coloring strategies that do not have access to edge weights. In contrast, we call **Greedy Vertex** and **Rand t p** *edge-weight* algorithms. Also, we expect the performance of all the above algorithms to improve as we refine them in the future

6.2 Test cases

We are currently limited to small test cases. As a simple, first attempt to measure the effect of not having enough registers, we colored our test cases using artificially small values of k. For each algorithm, we colored a WIG and measured the fraction of memory accesses remaining. We created per-variable WIGs and per-instruction WIGs for two different tests.

In test **heap,** we used the main subroutine, **Heapify()** [AHU74] of heapsort, using profile data assuming $\approx 10^6$ keys were to be sorted. In test **randwig** we used randomly built interference graphs. We created these graphs to mimic real WIGs, by first generating a random control flow graph and then by filling in the basic blocks with random definitions and uses. The **randwig** graphs include loops and if-then-else branching.

There were 13 distinct variables and 46 vertices in the **heap** instruction-level WIG. Figure 9 shows that when coloring a per-variable WIG, the algorithms using edge-weights were significantly better than the other algorithms, especially when there was a shortage of registers. The algorithms using edge-weight data eliminated over 97% of the memory traffic with only 6 registers; the other algorithms required 8 or 9 colors for the same effectiveness. Using an instruction level WIG, 10 shows that edge-weight algorithms performed best when there was a severe shortage of registers $(3 \leq k \leq 5)$.

The **randwig** tests produced similar results the **heap** test, providing initial validation of this test method. In the **randwig** example in this paper, we generated an instruction-level WIG with 20 variables and 232 sitex over six large basic blocks, forming a difficult graph to color. The algorithms using edge-weight data again performed significantly better than the other algorithms, reducing memory usage to $\approx$ 15% at $k = 14$ and to under 10% at $k = 16$, as shown in Figure 11. In contrast, even at $k = 18$, the algorithms not using edge-weight data still had significant memory usage. Figure 12 shows that we only get significant improvement coloring at the instruction-level when there is severe register shortage, namely $k \leq 10$ in this case. For $k = 8$ registers, we reduce memory usage to 40% versus 58% for the best per-variable coloring.

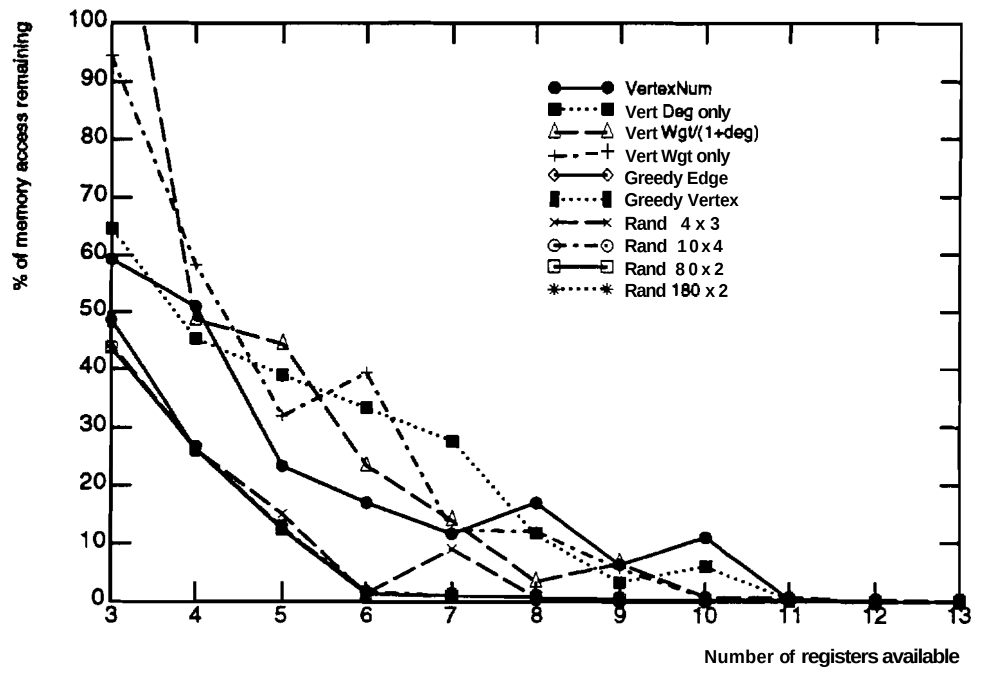

Figure 9: Results from coloring the **heap** test case at per-variable granularity.

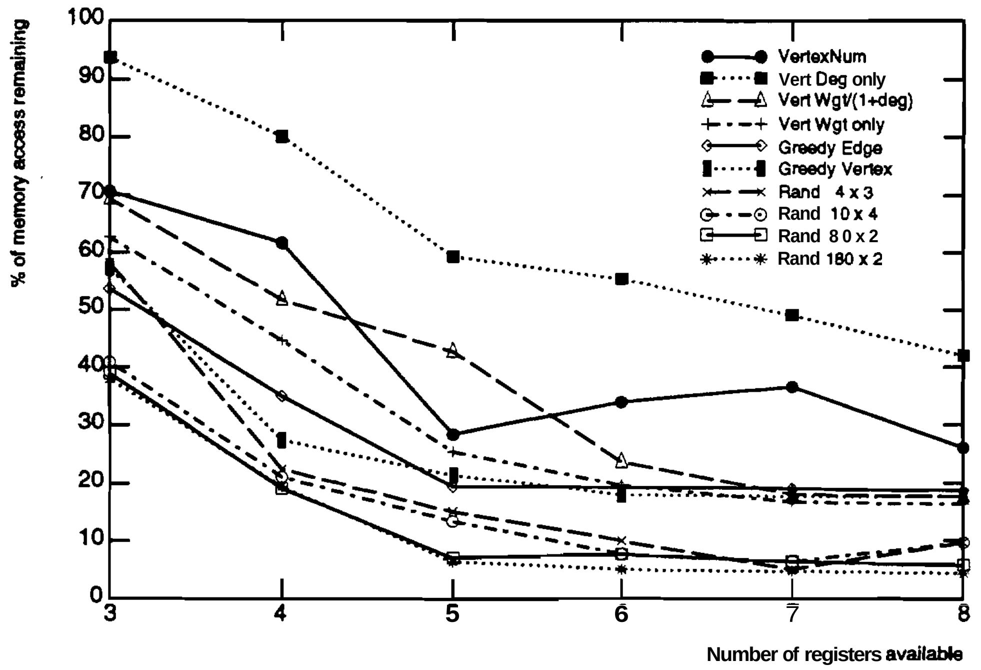

Figure 10: Results from coloring the **heap** test case at the machine instruction level

6.3 Implementation details

Because an instruction-level WIG becomes large quite quickly, we anticipate that this granularity will only be used for frequently executed basic blocks or procedures. Combine colorings vie the hierarchical approach outlined in Section 5. However, for most of the program, we suggest coloring an entire procedure using

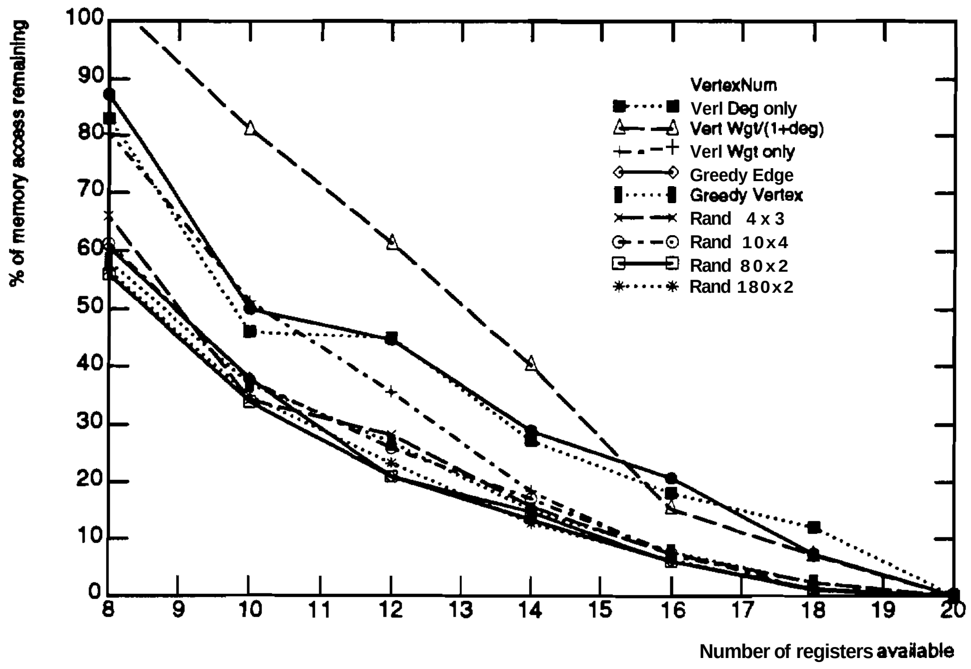

Figure 11: Results from coloring a 20 var, 232 vertex randwig graph at per-variable granularity.

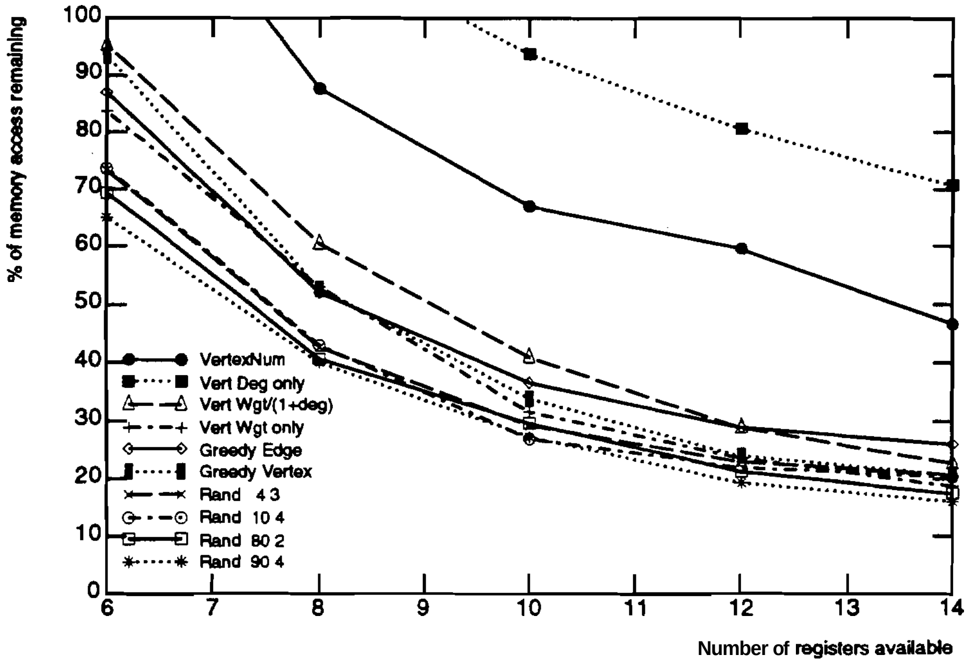

Figure 12: Results from coloring 20 var, 232 vertex randwig graph at the machine instruction level.

per-variable granularity. For best results, coloring the WIG should take place after optimizations which induce code expansion, such as inlining and loop unrolling.

7 Conclusion

We have described the use of a weighted interference graph (WIG) for register allocation. The WIG generalizes and improves upon previous graph-coloring based approaches because it allows an scalable granularity of coloring, it naturally handles arbitrary spill/load decisions, and it accurately models the underlying problem. Edge weights in a WIG indicate the precise degree of interference between two vertices. We have proven that coloring a WIG mirrors the run time cost of the corresponding register allocation under a wide variety of circumstances.

A per-variable WIG is the same size as a traditional UIG, but the WIG yields better register allocations than a UIG. The difference is most apparent when there are not enough registers. We have shown coloring an instruction-level WIG is ideally suited for collections of large, frequently-executed basic blocks when there is a serious register shortage.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. ***The Design and Analysis of Computer Algorithms.*** Addison-Wesley, Reading, MA, 1974.

[Baa88] Sara Baase. ***Computer Algorithms.*** Addison-Wesley, Reading, MA, 1988.

[BCKT89] Preston Briggs, Keith Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In ***ACM SIGPLAN Conference on Programming Language Design and Implementation,*** pages 275–284, Portland, OR, June 21–23 1989. ACM.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In ***ACM SIGPLAN Conference on Programming Language Design and Implementation,*** pages 311–321, San Francisco, CA, June 17–19 1992. ACM.

[BGG+89] D. Bernstein, D. Q. Golden, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and Pinter R. Y. Spill code minimization techniques for optimizing compilers. In ***ACM SIGPLAN Conference on Programming Language Design and Implementation,*** pages 258–263, Portland, OR, June 21–23 1989. ACM.

[CH90] Fredric Chow and John Hennessy. The priority-based coloring approach to register allocation. ***ACM Transactions on Programming Languages and Systems,*** 12:501–536, October 1990.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. In ***Proceedings*** 1982 ***Symposium on Compiler Construction,*** pages 98–105, Boston, MA, June 23–25 1982. ACM.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In ***ACM SIGPLAN Conference on Programming Language Design and Implementation,*** pages 192–203, Toronto, Canada, June 26–28 1991. ACM.

[CLR90] T. Cormen, C. E. Leiserson, and R. Rivest. ***Introduction to Algorithms.*** McGraw Hill, New York, NY, 1990.

[GSS89] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. In ***ACM SIGPLAN Conference on Programming Language Design and Implementation,*** pages 264–274, Portland, OR, June 21–23 1989. ACM.

[HP90] John L. Hennessy and David A. Patterson. ***Computer Architecture: A Quantitative Approach.*** Morgan Kaufmann, San Mateo, CA, 1990.

[LH86] James R. Larus and Paul N. Hilfinger. Register allocation in the spur lisp compiler. In ***Proceedings*** 1986 ***Symposium on Compiler Construction,*** pages 255–263, Palo Alto, CA, June 1986. ACM.

[Lin65] S. Lin. Computer solutions to the traveling salesman problem. BSTJ, 44(10):2245–2269, 1965.

[PF92] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In ***ACM SIGPLAN Conference on Programming Language Design and Implementation,*** pages 300–310, San Francisco, CA, June 17–19 1992. ACM.

[PS82] Christos Papadimitriou and Kenneth Steiglitz. ***Combinatorial Optimization.*** Prentice Hall, Englewood Cliffs, NJ, 1982.

A Proof of Claim 2

Claim 2 ***The nezt-site edge rule is rarely inaccurate and the inaccuracies are small for reasonable colorings.***

Proof: Define u, v_1 and v_2 as before. Figure 13 illustrates the three cases to consider. We show that defining an "extra edge" (u,v_2) is wrong more often than not. Negative weight edges are striped. For concreteness, in Figure 5 u = b, v_1 = a–2, and v_2 = a–4.

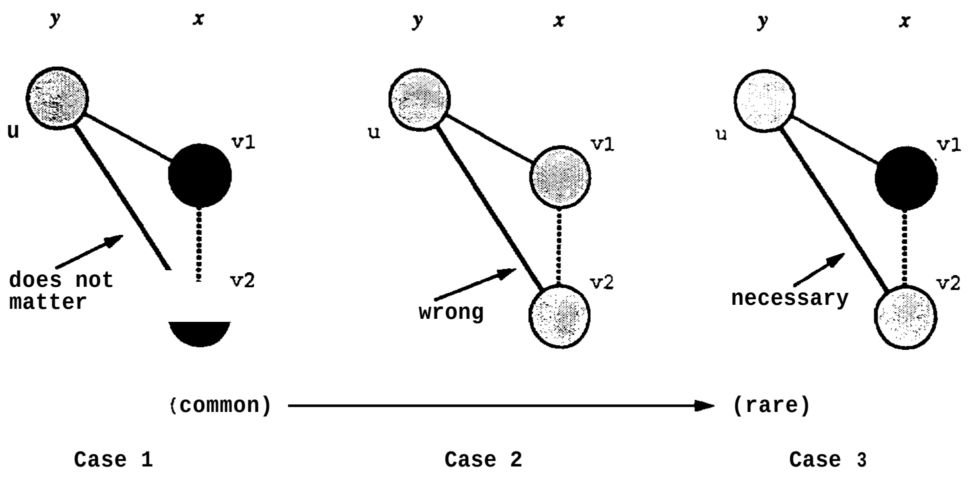

Figure 13: Justification for the next-site rule for edges, proving that edge (u,v_2) should not be present.

Case 1. Most allocations are fairly good so that variables need not be spilled. In this case, variables x and y do not interfere and so that edge (u,v_2) has no effect.

Case 2. Variables x and y conflict so that the extra edge (u,v_2) increases the cost of the coloring incorrectly. The two edges (u,v_1) and (v_1,v_2) account for all the run-time spill costs. This case occurs when variable y is spilled and replaced by variable z, which then remains in that register.

Case 3. The extra edge is necessary in the unlikely case that x at v_1 is spilled and then reloaded with Color(u). For this case to occur, both edges (v_1,v_2) and (u,v_2) must be relatively unimportant, as the coloring contradicts edges (u,v_2) and (v_1,v_2). Thus, we assume the error introduced from the absence of the extra edge is not likely to be significant. □

B Proof of Theorem 1

Theorem 1 Let T_{mem} = the actual run-time cost of a register assignment corresponding to a coloring of G, *Color*(G). Then $||Color(G)|| - K_{\min} \approx T_{\text{mem}}$.

Proof: We show that $||\text{Color}(G)|| - K_{\min} \approx T_{\text{mem}}$ under all coloring combinations. In a perfect coloring, by definition, $||\text{Color}(G)|| = K_{\min}$, so that $||\text{Color}(G)|| - K_{\min} = 0 = T_{\text{mem}}$. Thus, we only need to examine the cases when a run-time memory-access cost is incurred between vertices u and v.

There are ten cases which fall under two main categories depending on whether $\text{var}(u)$ equals $\text{var}(v)$. We sketch details for these cases at per-instruction level granularity. At other granularities the same arguments apply. In each category, the five cases are (a) $\text{Color}(u) = \text{Color}(v)$, (b) $\text{Color}(u) \neq \text{Color}(v)$, (c) only v is uncolored, (d) only u uncolored, and (e) both u and v are uncolored. Cases (a) and (b) are the common ones, and hence the most important to handle correctly. Let t_L and t_S be the run-time load and store costs, respectively. The term $t_{L\backslash S}$ represents either a load or store depending on whether the vertex u is a definition or use, respectively. (Vertex v must be a use, as were v a definition, there would be no edge (u, v) as u would be dead at v.)

Category I. Here, $\text{var}(u) \neq \text{var}(v)$, and the edge weight $w(u,v)$ includes the cost to spill u and load v. In all five cases, we find that the WIG models the runtime cost perfectly

Category **II**. If $\text{var}(u) = \text{var}(v)$, then $w(u,v)$ only includes the cost to load v from memory, ignoring the cost to store u. (Our definition is required in order to handle case (b) correctly.) Grinding through the details, we find errors in case (b), (c) and (e) as shown in the table below. The error in (c) is the difference $t_L - t_S$ which is likely to be small. The error in (e) is an extra t_L in the coloring cost. However, the fact that neither u or v was colored in case (e) probably means that u and v were relatively unimportant vertices, hence the error is not likely to be significant.

Case	$\lVert\text{Color}(G)\rVert - K_{\min}$ (LHS)	T_{mem} ***(RHS)***	|Error|
(a)	0	0	none
(b)	†	†	none or $\lvert 2t_S - t_L\rvert$
(c)	$2\,t_L$	$t_{L\backslash S}+ t_L$	none or $\lvert t_L - t_S\rvert$
(d)	$t_{L\backslash S}+ t_L$	$t_{L\backslash S}+ t_L$	none
(e)	$t_{L\backslash S}+ 2\,t_L$	$t_{L\backslash S}+ t_L$	t_L

(†) Case (b) is tricky because we need to consider a third vertex v' to explain why $\text{Color}(u) \neq \text{Color}(v)$. Let $\text{var}(u) = \text{var}(v) = x$. Assuming a reasonably good coloring, case (b) means that x was spilled between u and v, due to high demand for registers between u and v. There must be another vertex v_y (between u and v, with $\text{var}(v_y) \neq x$ and $\text{Color}(v_y) = \text{Color}(u)$. (If no such v_y existed, we would simply reuse $\text{Color}(u)$ for v.) Thus, we spill u to hold v_y, as shown in Figure 14.

Let the notation $X_S\backslash X_L$ represent X_S on a definition and X_L on a use of the specified vertex. Thus, $t_{L\backslash S} \equiv t_{L\backslash S}$. The vertex weight of v_y is $w(v_y) = t_S\backslash t_L$. The values for T_{mem} (the real load/store times), the coloring and $K_{\min}$ are given below. We get an error of $(2t_S - t_L)$ when v_y is a definition.

		u		v_y		v
T_{mem}	=	t_S	+	$0\backslash t_L$	+	t_L
$\lVert\text{Color}(u, v_y, v)\rVert$	=	$0\backslash t_S - t_L$	+	$t_S - t_L\backslash 0$	−	t_L
$-K_{\min}$	=	$t_S\backslash t_L$	+	$t_S\backslash t_L$	+	$2t_L$
error	=	0	+	$2t_S - t_L\backslash 0$	+	0

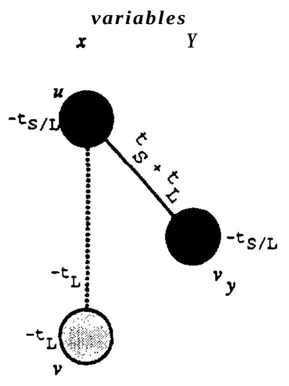

Figure 14: Run-time costs when a variable x is spilled between sites u and *v*.

We note that the cases with errors only occur when there is relatively heavy **register/memory** usage, so that the relative error to the total T_{mem} is likely to be small. Thus $||\text{Color}(G)|| - K_{\text{min}}$ is rarely inaccurate and is a good approximation to T_{mem}. □