Combined Code Motion and Register Allocation using the Value State Dependence Graph

Neil Johnson and Alan Mycroft

Computer Laboratory, University of Cambridge
William Gates Building, JJ Thompson Avenue,
Cambridge, CB3 0FD, UK
{`Neil.Johnson,Alan.Mycroft`}`@cl.cam.ac.uk`

Abstract. We define the Value State Dependence Graph (VSDG). The VSDG is a form of the Value Dependence Graph (VDG) extended by the addition of state dependence edges to model sequentialised computation. These express store dependencies and loop termination dependencies of the original program. We also exploit them to express the additional serialization inherent in producing final object code.

The central idea is that this latter serialization can be done incrementally so that we have a class of algorithms which effectively interleave register allocation and code motion, thereby avoiding a well-known phase-order problem in compilers. This class operates by first normalizing the VSDG during construction, to remove all duplicated computation, and then repeatedly choosing between: (*i*) allocating a value to a register, (*ii*) spilling a value to memory, (*iii*) moving a loop-invariant computation within a loop to avoid register spillage, and (*iv*) statically duplicating a computation to avoid register spillage.

We show that the classical two-phase approach (code motion then register allocation in both Chow and Chaitin forms) are examples of this class, and propose a new algorithm based on depth-first cuts of the VSDG.

1 Introduction

An important problem encountered by compiler designers is the *phase ordering* problem, which can be phrased as *"in which order does one schedule the register allocation and code motion phases to give the best target code?"*. These phases are antagonistic to each other—code motion may increase register pressure, while register allocation places additional dependencies between instructions, artificially constraining code motion. In this paper we show that a unified approach, in which both register allocation and code motion are considered together, sidesteps the problem of which phase to do first.

In support of this endeavour, we present a new program representation, the *Value State Dependence Graph* (VSDG), as an extension of the Value Dependence Graph (VDG) [21]. It is a simple unifying framework within which a wide range of code space optimizations can be implemented. We believe that the VSDG can be used in both intermediate code transformations, and all the way through to final target code generation.

Traditional register allocation has been represented as a graph colouring problem, originally proposed by Chaitin [5], and based on the Control Flow Graph (CFG). Unfortunately the CFG imposes an artificial ordering of instructions, constraining the register allocator to the given order.

The VDG represents programs as value dependencies—there is an edge (p, n), drawn as an arrow $n \to p$, if node n requires the value of p to compute its own value. This representation removes any specific ordering of instructions (nodes), but does not elegantly handle loop and function termination dependencies.

The VSDG introduces state dependency edges to model sequentialised computing. These edges also have the surprising benefit of generalising the VSDG: by adding sufficient serializing edges we can select any one of a number of CFGs. Our thesis is that relaxing the exact serialization of the CFG into the more general VSDG supports a combined register allocation and code motion algorithm.

1.1 Paper Structure

This paper is structured as follows. Section 2 describes the forms of nodes and edges in the VSDG, while Section 3 explores additional serialization and liveness within the VSDG. In Section 4 we describe the general approach to joint register allocation and code motion ($RACM$) as applied to the VSDG, and show that classical Chaitin/Chow-style register colouring specialises it. Section 5 introduces our new greedy register allocation algorithm. Section 6 provides context for this paper with a review of related work, with Section 7 concluding.

2 Formalism

The Value State Dependence Graph is a directed graph consisting of operation nodes, loop and merge nodes together with value- and state-dependency edges. Cycles are permitted but must satisfy various restrictions. A VSDG represents a single procedure; this matches the classical CFG but differs from the VDG in which loops were converted to tail-recursive procedures called at the logical start of the loop. We justify this because of our interest in performing RACM at the same time; inter-procedural motion and allocation issues are considered a topic for future work.

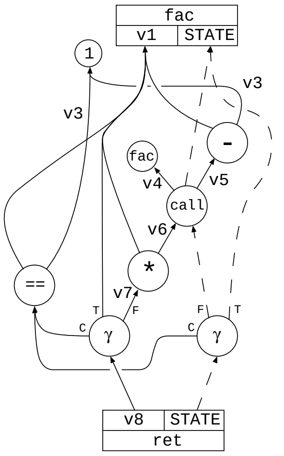
An example VSDG is shown in Fig. 1. In (a) we have the original C source for a recursive factorial function. The corresponding VSDG (b) shows both value and state edges and a selection of nodes.

2.1 Definition of the VSDG

Definition 1. *A VSDG is a labelled directed graph* $G = (N, E_V, E_S, \ell, N_0, N_\infty)$ *consisting of nodes* N *(with unique entry node* N_0 *and exit node* N_∞*), value-dependency edges* $E_V \subseteq N \times N$*, state-dependency edges* $E_S \subseteq N \times N$*. The labelling function* ℓ *associates each node with an operator (§2.2 for details).*

```
int fac( int n ) {
   int result;

   if ( n == 1 )
      result = n;
   else
      result = n * fac( n - 1 );
   return result;
}
```



(a) (b)

Fig. 1. A recursive factorial function, whose VSDG illustrates the key graph components—value dependency edges (solid lines), state dependency edges (dashed lines), a `const` node, a `call` node, two γ-nodes, a conditional node (`==`), and the function entry and exit nodes. The left-hand γ-node returns the original function argument if the condition is true, or that of the expression otherwise. The right-hand γ-node behaves similarly for the state edges, returning either the state on entry to the function, or that returned by the `call` node.

VSDGs have to satisfy two well-formedness conditions. Firstly ℓ and the (E_V) arity must be consistent, *e.g.* that a binary arithmetic operator must have two inputs; secondly (at least for the purposes of this paper) that the VSDG corresponds to a structured program, *e.g.* that there are no cycles in the VSDG except those mediated by θ (loop) nodes (see §3.2).

Value dependency (E_V) indicates the flow of values between nodes, and must be preserved during register allocation and code motion.

State dependency (E_S), for this paper, represents two things; the first is essential sequential dependency required by the original program, *e.g.* a given load instruction may be required to follow a given store instruction without being re-ordered, and a `return` node in general must wait for an earlier loop to terminate even though there might be no value-dependency between the loop and the `return` node. The second purpose, which in a sense is the centre of this work, is that state-dependency edges can be added incrementally until the VSDG corresponds to a unique CFG (§3.1). Such state dependency edges are called *serializing* edges.

An edge (n_1, n_2) represents the flow of data or control *from* n_1 to n_2, *i.e.* in the *forwards data flow direction*, so we will see n_1 as a predecessor of n_2. Similarly we will regard n_2 as a successor of n_1. If we wish to be specific we will write V-successors or S-successors for respectively E_V and E_S successors.

Similarly, we will write $succ_V(n)$, $pred_S(n)$ and the like for appropriate sets of successors or predecessors, and $dom(n)$ and $pdom(n)$ for sets of dominators and post-dominators respectively. We will draw pictures in the VDG form, with arrows following the *backwards data flow direction*, so that the edge (n_1, n_2) will be represented as an arrow *from* n_2 to n_1.

The VSDG inherits from the VDG the property that a program is implicitly represented in Static Single Assignment (SSA) form [8]: a given operator node, n, will have zero or more E_V-successors using its value. Note that, in implementation terms, a single register can hold the produced value for consumption at all successors; it is therefore useful to talk about the idea of an output *port* for n being allocated a specific register, r, to abbreviate the idea of r being used for each edge (n_1, n_2) where $n_2 \in succ(n_1)$. Similarly, we will talk about (say) the "right-hand input port" of a subtraction instruction, or of the R-input of a θ-node.

2.2 Node Labelling with Instructions

There are four main classes of VSDG nodes, based on those of the *tri*VM Intermediate Language [13]: value nodes (representing pure arithmetic), γ-nodes (conditionals), θ-nodes (loops), and state nodes (side-effects).

2.2.1 Value Nodes The majority of nodes in a VSDG generate a value based on some computation (add, subtract, etc) applied to their dependent values (constant nodes, which have no dependent nodes, are a special case).

2.2.2 γ-Nodes Our γ-node is similar to the γ-node of Gated Single Assignment form [2] in being dependent on a control predicate, rather than the control-independent nature of SSA ϕ-functions.

Definition 2. *A γ-node $\gamma(C, T, F)$ evaluates the condition dependency C, and returns the value of T if C is true, otherwise F.*

We generally treat γ-nodes as single-valued nodes (constrast θ-nodes, which are treated as tuples), with the effect that two separate γ-nodes with the same condition can be later combined (Section 4) into a tuple using a single test. Fig. 2 illustrates two γ-nodes that can be combined in this way.

2.2.3 θ-Nodes The θ-node models the iterative behaviour of loops, modelling loop state with the notion of an *internal value* which may be updated on each iteration of the loop. It has five specific ports which represent dependencies at various stages of computation.

Definition 3. *A θ-node $\theta(C, I, R, L, X)$ sets its internal value to initial value I then, while condition value C holds true, sets L to the current internal value and updates the internal value with the repeat value R. When C evaluates to false computation ceases and the last internal value is returned through the X port.*

A loop which updates k variables will have: a single condition port C, initial-value ports $I_1, \ldots, I_k$, loop iteration ports $L_1, \ldots, L_k$, loop return ports $R_1, \ldots, R_k$, and loop exit ports $X_1, \ldots, X_k$. The example in Fig. 3 shows a pair (2-tuple) of values being used for I, R, L, X, one for each loop-variant value.

For some purposes the L and X ports could be fused, as both represent outputs within, or exiting, a loop (the values are identical, while the C input merely selects their routing). We avoid this for two reasons: (*i*) we have operational semantics for VSDGs G and these semantics require separation of these concerns; and (*ii*) our construction of G^{noloop} (§3.2) requires it.

The θ-node directly implements pre-test loops (`while`, `for`); post-test loops (`do...while`, `repeat...until`) are synthesised from a pre-test loop preceded by a duplicate of the loop body. At first this may seem to cause unnecessary duplication of code, but it has two important benefits: (*i*) it exposes the first loop body iteration to optimization in post-test loops (*cf.* loop-peeling), and (*ii*) it normalizes all loops to one loop structure, which both reduces the cost of optimization, and increases the likelihood of two schematically-dissimilar loops being isomorphic in the VSDG.

2.2.4 State Nodes Loads, stores, and their volatile equivalents, compute a value and/or state (non-volatile loads return a value from memory without generating a new state). Accesses to volatile memory or hardware can change state independently of compiler-aware reads or writes (*cf.* IO-state [7]).

The `call` node takes both the name of the function to call and a list of arguments, and returns a list of results; it is treated as a state node as the function body may read or update state.

We maintain the simplicity of the VSDG by imposing the restriction that *all* functions have *one* return node (the exit node N_∞), which returns at least one result (which will be a state value in the case of `void` functions). To ensure that function calls and definitions are colourable, we suppose that the number of arguments to, and results from, a function is smaller than the number of physical registers—further arguments can be passed via a stack as usual.

```
a)   if (P)
         x = 2, y = 3;
     else
         x = 4, y = 5;

b)   if (P) x = 2; else x = 4;
         ...
     if (P) y = 3; else y = 5;
```

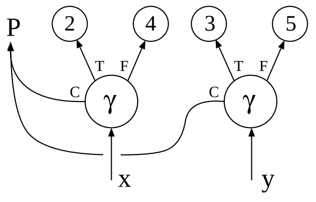


Fig. 2. Two different code schemes (a) & (b) map to the same γ-node structure.

```
j = ...
for( i = 0;
     i < 10;
     ++i )
    --j;

... = j;
```

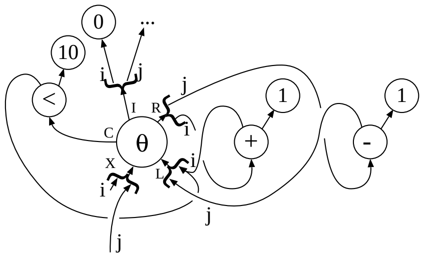


Fig. 3. A θ-node example showing a `for` loop. Evaluating the θ-node's **X** port triggers it to evaluate the **I** value (outputting the value on the **L** port). While **C** evaluates to true, it evaluates the **R** value (which in this case also uses the θ-node's **L** value). When **C** is false, it returns the final internal value through the **X** port. As `i` is not used after the θ-node loop then there is no dependency on the `i` port of **X**.

Note also that the VSDG neither forces loop invariant code into nor out-of loop bodies, but rather allows later phases to determine, by adding serializing edges, such placement of loop invariant nodes for later phases.

3 Applying the VSDG to RACM

3.1 Serialization

Weise *et al.* [21] observe that their mapping from CFGs to VDGs is many-one; that paper also suggests that "*Code motion optimizations are decided when the demand dependence graph is constructed from the VDG*"—*i.e.* that a VDG should be mapped back into a CFG for further processing—but does not give an algorithm or consider which of the many CFGs corresponding to a VSDG should be selected.

We identify VSDGs with 'enough' serializing edges with CFGs—such VSDGs can be simply transformed into CFGs if desired—the task of RACM then being to make the VSDG sufficiently sequential. The following informal definition captures this idea for the purposes of this paper.

Definition 4. *A* sequential *VSDG is one which has enough serializing edges to make it correspond to a single CFG.*

Here 'enough' means in essence that each node in the VSDG has a unique ($E_V \cup E_S$) immediate dominator which can be seen as its predecessor in the CFG. Exceptions arise for the start node (which has no predecessors in the VSDG or corresponding CFG), γ-nodes and θ-nodes. Given a γ-node, we interpret those nodes which the T port post-dominates as the *condition-true* sub-CFG and those which the F port post-dominates as the *condition-false* sub-CFG; a control-split node (corresponding to a CFG test node) is added to the VSDG as the immediate E_S-dominator of both sub-CFGs. For a θ-node, we recursively require this sequential property for its body, L, and interpret the "unique immediate dominator" property as a constraint on its I port.

3.2 VSDG Well-Formedness

As in the VDG we restrict attention to reducible graphs for the VSDG; recall that any CFG can be made reducible by duplicating code or by introducing additional boolean variables. For this paper we further restrict to programs whose only loops are `while`-loops and which exit them only by the condition becoming *false*, *i.e.* no `break` and the like (again this could be achieved with code duplication or with additional boolean variables).

In order to specify the "all cycles in a VSDG are mediated by θ-nodes" restriction, it is convenient to define a transformation on VSDGs.

Definition 5. *Given a VSDG, G, we define G^{noloop} to be identical to G except that each θ-node θ_i is replaced with two nodes, θ_i^{head} and θ_i^{tail}; edges to or from ports I or L of the original θ-node are redirected to θ_i^{head} whereas those to or from ports R, X, C are redirected to θ_i^{tail}.*

We then require G^{noloop} to be an acyclic graph.

When adding serializing edges we must maintain this acyclic property. To serialize nodes connected to a θ-node's X port we add serializing edges to θ^{tail}; all nodes within the body of the loop are on the sequential path from θ^{tail} to θ^{head}; all other nodes are serialized before θ^{head}. Definition 6 below sets out the conditions for a node to be within a loop.

Although this is merely a formal transformation, note that if we interpret θ^{tail} as a γ-node (or possibly a tuple thereof) and interpret θ^{head} as an identity operation then G^{noloop} represents a VSDG in which each loop is executed zero or one times according to the condition. Our θ^{head} and θ^{tail} nodes, while similar to GSA's μ- and η^F-functions [2], avoids the need for their "non-deterministic merge gate" to break cyclic dependencies.

The formal definition of a VSDG being well-formed is then:

Definition 6. *A VSDG, G, is well-formed if (*i*) G^{noloop} is acyclic and (*ii*) for each pair of $(\theta^{head}, \theta^{tail})$ nodes in G^{noloop}, θ^{tail} post-dominates all nodes in $succ^+_{V \cup S}(\theta^{head}) \cap pred^+_{V \cup S}(\theta^{tail})$.*

The second condition says that no value computed during a loop can be used outside the loop, except via the X port of a θ-node.

3.3 VSDG Normalization

The RACM algorithms below will assume (for maximal optimization potential rather than correctness) that the VSDG has been normalized, roughly in the way of 'hash-CONSing': any two nodes which have identical input nodes, will be assumed to have been replaced with a single node *provided that this does not violate well-formedness by creating a cycle* in the VSDG. Consider

```
int f( int v[], int i ) {
    int a = v[i+1];
    v[7] = 0;
    return v[i+1] + a;
}
```

There will only be one node for the constant 1, and one for the addition of this node to the second formal parameter (`i+1`) but two nodes for the *loading* from `v[i+1]` because sharing this node would lead to a cycle in E_S by being both a predecessor and successor of the *store* to `v[7]`.

Note that this is a safe form of CSE and loop invariant code lifting; this optimization is selectively undone (node cloning) during the joint RACM phase when required by register pressure.

3.4 Liveness in VSDGs

For the purposes of register allocation (*cf.* the register interference graph), we need to know which (output ports of) VSDG nodes may hold values simultaneously so we can forbid them being allocated the same register.

We define a *cut* to be a partition $N_1 \cup N_2$ of nodes in the VSDG with the property that there is no $E_V \cup E_S$ edge from N_2 to N_1 (excepting edges from L ports of θ-nodes—see the G^{noloop} construction).

We now define nodes n and n' to *interfere* if there is a cut $N_1 \cup N_2$ with $n, n' \in N_1$ and with both $succ(n)$ and $succ(n')$ having non-empty intersections with N_2.

This generalises the normal concept register of interference in a CFG; there a cut is just a program point and interference means "simultaneously live at any program point". Similarly "virtual register" corresponds to our "output port of a node". Note that we use the concept of "cut based on Depth From Root" in Section 5 for our new greedy algorithm.

4 Register Allocation and Code Motion

The goal of register allocation in the VSDG is to allocate one physical register (from a fairly small set) to each node's output ports. θ-nodes are a special case, as they require multiple registers on their tupled I, R, L and X ports.

Register requirements can be reduced by serializing computations (a register can be reused in two independent computations if we know that they do not interleave), or by reducing the range over which a value is live by duplicating a computation or by spilling a value to memory. In both cases the idea is that these operations reduce the register interference.

4.1 A Non-Deterministic Approach

Given a VSDG we repeatedly apply the following non-deterministic algorithm until all the nodes are coloured and the VSDG is sequential:

1. Colour a port with a physical register—provided no port it interferes with is already coloured with the same register;
2. Add a serializing edge to force one node before another—this removes edges from the interference graph by forbidding interleaving of computations;

3. Clone a node, *i.e.* recalculate it to reduce register pressure.
4. Tunnel values through memory by introducing store/load spill nodes.
5. Merge two γ-nodes a and b into a tuple, provided their C ports reference the same node and there is no path from a to b or from b to a.

The first action assigns a physical register to a port of the given node. The second moves the node, with the effect of changing the register usage; the choice of *which* node to move is determinined by specific algorithms (see §4.2 and Section 5).

Node cloning replaces a single instance of a node that has multiple uses, with multiple copies (clones) of the node, each with a subset of the original dependency edges. For example, a node n with two dependent nodes p and q, can be cloned into n' and n'', with p dependent on n' and q dependent on n''.

Spilling follows the traditional Chaitin-style register spilling where we add store and load nodes, together with some temporary storage in memory.

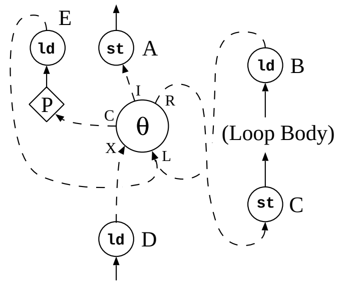
Finally, because the initial VSDG was normalized to ensure that each γ-node represented the merge of a *single* variable, given a VSDG such as that in Fig. 2, we can either arrange to serialize the two γ-nodes (action 2) resulting in two separate tests (or conditional move instructions) or to merge them (action 5) so that a single test is used (as in Fig. 2(a)).

The cost of spilling loop-variant variables is rather higher than the store-and-reload for a normal spill. For θ-nodes where the tuple is wider than the available target registers, we must spill one or more of the θ-node variables over the loop test code, not merely within the loop itself. At most this requires two stores and three loads for each variable spilled. Fig. 4 shows the location of the five spill nodes (a), with table (b) describing the use of each of the spill nodes.

4.2 The Classical Algorithms

We can phrase the classical Chaitin/Chow-style register allocators as instances of the above algorithm:

1. Perform all code motion transforms through adding serializing edges and merging γ-nodes if not already sequentialised;



(a)

Spill Node	Needed if variable...
A	Is initialised
B	Is used in Loop Body
C	Is defined in Loop Body
D	Is used after the loop
E	Is used in the condition predicate **P**

(b)

Fig. 4. Illustrating the locations of the five spill nodes associated with a θ-node.

2. Map the VSDG onto a CFG by adding additional serializing edges;
3. If there are insufficient physical registers to colour a node port, then:
   (a) Chaitin-style allocation [5]: spill nodes, with the restriction that the target register of the reload is the same as the source register of the store. Chaitin's cost estimates can be applied to determine which edge to spill;
   (b) Chow-style allocation [6]: spill nodes, but without the register restriction of Chaitin-style, thus splitting the live-range of the virtual register; use Chow's heuristics to decide which edge to split.

In both Chaitin and Chow instances post-code-motion transformations during register allocation are limited to inserting store and load nodes into the program.

5 A New Register Allocation Algorithm

The Chaitin/Chow algorithms do not make full use of the dependence information within the VSDG; they assume that a previous phase has performed code motion to produce a sequential VSDG—corresponding to a single CFG—on which traditional register colouring algorithms are applied.

We now present the central point of this paper—a register allocation algorithm specifically designed to maximise the usage of information within the VSDG. The algorithm consists of two distinct phases:

1. Starting at the exit node N_∞, walk up the graph edges calculating the maximal Depth From Root (DFR) of each node (see Definition 7); for each set of nodes of equal depth calculate their liveness width (the number of distinct values on which they depend, taking into account control flow).
2. Apply a forward "snow-plough"[1]-like graph reshaping algorithm, starting from N_∞ and pushing up towards N_0, to ensure that the liveness width is never more than the number of physical resisters. This is achieved by splitting, spilling or adding serializing edges in greedy way so that the previously smoothed-out parts of the graph (nearer the exit) are not re-visited.

The result is a colourable VSDG; colouring it constitutes register assignment completing the algorithm.

5.1 Partitioning the VSDG

The first phase annotates the VSDG with the maximal Depth From Root. The second phase then processes each cut of the VSDG in turn.

Definition 7. *The maximal Depth From Root, $\mathcal{D}(n)$, of a node $n \in N$ is the length of the longest path $p \in (E_V \cup E_S)^*$ from the root to n. Loop bodies are traversed once, such that a θ-node has two DFRs—one each for the θ^{head} and θ^{tail} nodes.*

---

[1] Imagine a snow plough pushing forward, scooping up excess snow, and depositing it where there is little snow. The goal is to even out the peaks and troughs.

Definition 8. *A* depth-first cut $\mathcal{S}_{=d}$ *is the set of nodes with the same DFR d:*

$$\mathcal{S}_{=d} = \{n \in N \mid \mathcal{D}(n) = d\}$$

It is convenient also to write

$$\mathcal{S}_{\leq d} = \{n \in N \mid \mathcal{D}(n) \leq d\}$$
$$\mathcal{S}_{>d} = \{n \in N \mid \mathcal{D}(n) > d\}$$

Note that the partition $(\mathcal{S}_{\leq d}, \mathcal{S}_{>d})$ is a *cut* according to the definition of §3.4.

Computing the DFR of a given VSDG is equivalent to computing the depth-first search of the graph—we simply start at the root node N_∞ and recursively walk along all dependency edges, setting each node to the larger of the node's current DFR and the new DFR, terminating either at the entry node N_0 or nodes with DFRs greater than the current DFR. It has a complexity of $O(N+E_V+E_S)$.

5.2 Calculating Liveness Width

We wish to transform each cut so that the number of nodes having edges passing through it is no greater than $\mathcal{R}$, the number of registers available for colouring.

For a cut of depth d the set of such live nodes is given by

$$\mathcal{W}_{in}(d) = \mathcal{S}_{>d} \cap pred_V(\mathcal{S}_{\leq d})$$

i.e. those nodes which are further than d from the exit node but whose values may be used on the path to the exit node. Note that only E_V and not E_S edges count towards liveness.

One might expect that $|\mathcal{W}_{in}(d)|$ is the number of registers required to compute the nodes in $\mathcal{S}_{\leq d}$ but this overstates the number of registers required for conditional nodes. γ-nodes have the property that the edges of each of their selection dependencies are disjoint—on any given execution trace, exactly one path to the γ-node will be executed at a time, and so therefore we can reuse the same registers to colour its True- and False-dominated nodes.

We identify the γ-node dependency register sets using the dominance property thus:

Definition 9. *A node $n \in N$ is a* predicated node *iff it is post-dominated by* either *the* True *or the* False *port of a γ-node, but not by both.*

Note that replacing nodes in either of the *True* or *False* regions with no-ops each gives a lower-bound to the liveness width of the cut[2]. Moreover, the greater of the liveness widths for these modified VSDGs gives the corrected liveness width for the original VSDG.

We prefer to formulate this in constraint form.

---

[2] Such no-ops are nodes with no value dependencies on input or output, but with state-dependences where previously there was either a value or state edge so that the DFR is not affected.

Definition 10. *A VSDG is* colourable *with $\mathcal{R}$ registers if either:*

1. *Every cut of depth d has $|\mathcal{W}_{in}(d)| \leq \mathcal{R}$; or*
2. *Each VSDG resulting from replacing either* True *or* False *regions with no-ops satisfies 1.*

5.3 Pass Through Edges

Some edges (*i.e.* of lifetime greater than one) pass *through* a cut. These pass-through (PT) edges may also interact with the cut. However, even the ordinary PT edges require a register, and so must be accommodated in any colouring scheme.

Definition 11. *The* lifetime *$\mathcal{L}$ of an edge (n, n') is the number of cuts over which it spans:*

$$\mathcal{L}(n, n') = \mathcal{D}(n) - \mathcal{D}(n')$$

Definition 12. *An edge $(n, n') \in E_V$ is a* Pass Through *(PT) edge over cut $\mathcal{S}$ of depth d when:*

$$\mathcal{D}(n) > d > \mathcal{D}(n')$$

A Used Pass Through *(UPT) edge is a PT edge from a node which is also used by one or more nodes in $\mathcal{S}$, i.e. there is $n'' \in \mathcal{S}$ with $(n, n'') \in E_V$.*

In particular, PT (and to a lesser extent UPT) edges are ideal candidates for spilling when transforming a cut. The next section discusses this further.

5.4 Register Allocation

In order to colour the graph successfully with $\mathcal{R}$ target machine registers no cut of the graph must be wider (*i.e.* the number of live registers) than the number of target registers available.

For every cut of depth d calculate $\mathcal{W}_{in}(d)$. Then, while $\mathcal{R} > |\mathcal{W}_{in}(d)|$ we apply three transformations to the VSDG in increasing order of cost: *(i)* node raising (code motion), *(ii)* node cloning (undoing CSE), or *(iii)* node spilling, where we first choose non-loop nodes followed by loop nodes.

The first—node raising—pushes a node up to the next cut by adding serializing edges from all other nodes in the cut. We repeat this until either the liveness width is less than the number of physical registers, or there is only one node left in the cut.

In node cloning, we take a node and generate copies (clones). Serializing edges are added to maintain the DFR of the the clones. A simple algorithm for this transformation is to produce as many clones as there are dependents of the node; a node recombining pass will recombine clones that end up in the same cut.

Node cloning is not always applicable as it may increase the liveness width of higher cuts (when the in-registers of the cloned node did not previously pass through the cut); placing a cloned node in a lower cut can increase the liveness width. But, used properly [18], node cloning can reduce the liveness width of lower cuts by splitting the live range of a node, which potentially has a lower cost than spilling.

Finally, when all other transformations are unable to satisfy the constraint, we must spill one or more edges to memory. PT edges are ideal candidates for spilling, as the lifetime of the edge affords good pipeline behaviour on superscalar RISC-like targets; likewise, UPT edges are similarly beneficial, but place some constraints on the location of the store node.

A related issue is the spilling of θ-nodes. As discussed previously the worst-case cost of spilling a loop-variant variable from a θ-node tuple is two stores and three loads, so these should always be done after spilling of PT nodes. By contrast, Chaitin/Chow colouring has to use approximate cost heuristics to decide to spill a variable in a loop or outside.

6 Related work

6.1 Benefits over other Program Graph Representations

The VSDG is based in part on the Value Dependence Graph (VDG) [21]. The VDG uses a λ-node to represent both functions and loop bodies, thereby combining loops and functions into one complex abstraction mechanism rather. In the VSDG we treat them separately with `call` and θ-nodes. One particular problem the VDG has is that of preserving the terminating properties of a program—"*Evaluation of the VDG may terminate even if the original program would not...*" [21].

Another significant issue with the VDG is the process of generating target code from the VDG. The authors describe converting the VDG into a demand-based Program Dependence Graph (dPDG)—a normal Program Dependence Graph [9] with additional edges representing demand dependence—then converting that into a traditional control flow graph (CFG) [1] before finally generating target code from the CFG with a standard back-end code generator; this is not as flexible (or as clearly specified) as the VSDG presented in this paper.

Many other program graphs (with many and varied edge forms) have been presented in the literature: the Program Dependence Graph [9], the Program Dependence Web [2], the System Dependence Graph [11] and the Dependence Flow Graph [15]. Our VSDG is both simpler—only two types of edge represent all of the above flow information—and more normalizing (§3.3).

6.2 Solving Phase Order Problems

The traditional view of register allocation as a graph colouring problem was proposed by Chaitin [5]. In §4.2 we generalise the both the Chaitin and Chow approaches.

Goodwin and Wilken [10] formulate global register allocation (including all possible spill placements) as a 0-1 integer programming problem. While they do achieve quite impressive results, the cost is very high: the complexity of their algorithm is $O(n^3)$ and for a given time period the allocator does not guarantee to allocate all functions.

Code motion as an optimization is not new (*e.g.* Partial Redundancy Elimination [14]). Perhaps the work closest in spirit to ours is that of Rüthing *et al.* [18] which presents algorithms for optimal placement of expressions and subexpressions, combining both raising and lowering of code within basic blocks.

Most work has concentrated on the instruction scheduling/register allocation phase order problem, which we now consider.

The CRAIG framework [4], implemented within the ROCKET compiler [19], takes a brute force approach:

1. attempt register allocation after instruction scheduling,
2. if the schedule cost is not acceptable (by some defined metric) attempt register allocation before scheduling,
3. then while the cost is acceptable (*i.e.* there is some better schedule) add back in information from the first pass until the schedule just becomes too costly.

Their experience with an instance of CRAIG (CRAIG_0) defines the metric as the existence of spill code. Their experimental results show improvements in execution time, but do not document the change in code size.

Rao [17] improves on CRAIG_0 with additional heuristics to allow *some* spilling, where it can be shown that spilling has a beneficial effect.

Touati's thesis [20] argues that register allocation is the primary determinant of performance, not scheduling. The goal of his thesis is again to minimize the insertion of spill-code, both through careful analysis of register pressure, and by adding serializing edges to each basic block data dependency DAG. It is basic-block-based.

An early attempt at combining register allocation with instruction scheduling was proposed by Pinter [16]. That work is based on an instruction level register-based intermediate code, and is preceded by a phase to determine data dependencies. This dependence information then drives the allocator, generating a *Parallelizable Interference Graph* to suggest possible register allocations. Further, the *Global Scheduling Graph* is then used to schedule instructions within a region.

Another region-based approach is that of Janssen and Corporaal [12], where regions correspond to the bodies of natural loops. They then use this hierarchy of nested regions to focus register allocation, with the inner-most regions being favoured by better register allocation (*i.e.* less spill code).

The Resource Spackling Framework of Berson *et al.* [3] applies a *Measure and Reduce* paradigm to combine both phases—their approach first measures the resource requirements of a program using a unified representation, and then moves instructions out of *excessive sets* into *resource holes*. This approach is basic-block-based: a local scheduler attempts to satisfy the target constraints without increasing the execution time of a block; the more complicated global scheduler moves instructions between blocks.

7 Conclusions and Further Work

In this paper we have defined the VSDG, an enhanced form of the VDG which includes state dependency edges to model sequentialized computation. By adding sufficient state-dependency edges we have shown that the VSDG is able to represent a single CFG; conversely fewer serializing edges relax the artificial constraints imposed by the CFG.

From this basis, we have shown that the VSDG framework supports a combined approach to register allocation and code motion, using an incremental algorithm which effectively interleaves the two phases, and thus avoiding the well-known phase-ordering problem. We have described an algorithm which, when given a well-formed, normalized VSDG then allocates registers, if necessary interleaving this with code motion, node splitting and register spilling.

The work presented here is the start of a larger project: an implementation of the algorithms in this paper is in progress.

Acknowledgements

The research reported here was supported with a grant from ARM Ltd. Thanks are due to Alistair Turnbull and Eben Upton for constructive debate on aspects of the VSDG, to Lee Smith for useful comments on an early draft of this paper, and to Mooly Sagiv and Tom Reps for helpful discussions at SAS'02.

References

1. Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.
2. Ballance, R. A., Maccabe, A. B., and Ottenstein, K. J. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. Conf. Prog. Lang. Design and Implementation (PLDI'90)* (June 1990), ACM, pp. 257–271.
3. Berson, D. A., Gupta, R., and Soffa, M. L. Resource spackling: A framework for integrating register allocation in local and global schedulers. Tech. rep., Dept. Computer Science, University of Pittsburgh, February 1994.
4. Brasier, T. S., Sweany, P. H., Beaty, S. J., and Carr, S. CRAIG: A practical framework for combining instruction scheduling and register assignment. In *Proc. Intl. Conf. Parallel Architectures and Compilation Techniques (PACT'95)* (Limassol, Cyprus, June 1995).
5. Chaitin, G. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices 17*, 6 (June 1982), 98–105.
6. Chow, F. C., and Hennessy, J. L. The priority-based coloring approach to register allocation. *ACM Trans. Prog. Lang. and Syst. 12*, 4 (October 1990), 501–536.
7. Click, C. From quads to graphs: an intermediate representation's journey. Tech. Rep. CRPC-TR93366-S, Center for Research on Parallel Computation, Rice University, October 1993.

8. Cytron, R. K., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing the static single assignment form and the control dependence graph. *ACM Trans. Programming Languages and Systems 12*, 4 (October 1991), 451–490.
9. Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. and Syst. 9*, 3 (July 1987), 319–349.
10. Goodwin, D. W., and Wilken, K. D. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software—Practice and Experience 26*, 8 (August 1996), 929–965.
11. Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Langs and Systems 12*, 1 (January 1990), 26–60.
12. Janssen, J., and Corporaal, H. Registers on demand: an integrated region scheduler and register allocator. In *Proc. Conf. on Compiler Construction* (April 1998).
13. Johnson, N. *tri*VM Intermediate Language Reference Manual. Tech. Rep. UCAM-CL-TR-529, University of Cambridge Computer Laboratory, 2002.
14. Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. *Comm. ACM 22*, 2 (February 1979), 96–103.
15. Pingali, K., Beck, M., Johnson, R., Moudgill, M., and Stodghill, P. Dependence flow graphs: An algebraic approach to program dependencies. In *Proc. 18th ACM Symp. on Principles of Prog. Langs (POPL)* (January 1991), ACM, pp. 67–78.
16. Pinter, S. S. Register allocation with instruction scheduling: A new approach. In *Proc. ACM SIGPLAN Conference on Prog. Lang. Design and Implementation* (Albuquerque, NM, June 1993), pp. 248–257.
17. Rao, M. P. Combining register assignment and instruction scheduling. Master's thesis, Michigan Technological University, 1998.
18. Rüthing, O., Knoop, J., and Steffen, B. Sparse code motion. In *Proc. 27th ACM SIGPLAN-SIGACT Symp. Principles of Prog. Langs (POPL)* (Boston, MA, 2000), ACM, pp. 170–183.
19. Sweany, P., and Beaty, S. Post-compaction register assignment in a retargetable compiler. In *Proc. 23rd Annual Workshop on Microprogramming and Microarchitecture* (November 1990), pp. 107–116.
20. Touati, S.-A.-A. *Register Pressure in Instruction Level Parallelism.* PhD thesis, Université de Versailles Saint-Quentin, June 2002.
21. Weise, D., Crew, R. F., Ernst, M., and Steensgaard, B. Value dependence graphs: Representation without taxation. In *ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Langs (POPL)* (January 1994), ACM.