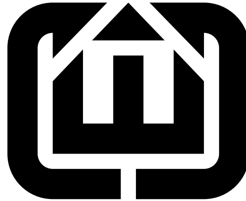
CEC C Generator



Stephen A. Edwards
Columbia University
sedwards@cs.columbia.edu

Contents

1 The generateC function **2**

2 fixGRCnode **8**

3 Check Acyclic **10**

4 Scheduling **11**

5 Simple Clustering **12**

6 Greedy Clustering **14**

7 Levelizing **17**

8 Split cluster nodes **19**

9 Top-Level Files **23**
9.1 GRCC2.hpp 23
9.2 GRCC2.cpp 23
9.3 cec-grcc2.cpp 24

1 ⟨*declarations* 1⟩≡

```
typedef map<GRCNode *, int> CFGmap;
typedef map<STNode *, int> STmap;
```

1 The generateC function

2 ⟨*declarations* 1⟩+≡

```
void generateC(std::ostream &o, Module &, string, bool);
```

3 ⟨*definitions 3*⟩≡

```
CFGmap cfgmap; // FIXME: for debugging

void generateC(std::ostream &o, Module &m, string basename, bool ansi)
{
  CPrinter::Printer p(o, m, false);

  GRCgraph *g = dynamic_cast<AST::GRCgraph*>(m.body);
  if (!g) throw IR::Error("Module is not in GRC format");

  o << "/* Generated by CEC/GRCC2";
  if (ansi) o << " in ANSI mode ";
  else o << " using the GCC computed-goto extension ";
  o << "*/\n";

  // CFGmap cfgmap;
  STmap stmap;
  g->enumerate(cfgmap, stmap);

  EnterGRC *eg = dynamic_cast<EnterGRC*>(g->control_flow_graph);
  assert(eg); // The top of the control-flow graph should be an EnterGRC
  assert(eg->successors.size() == 2); // It should have two successors
  GRCNode *firstNode = eg->successors.back();

  // Perform minor restructuring on the control-flow graph

  for ( CFGmap::iterator i = cfgmap.begin() ; i != cfgmap.end() ; i++ )
    fixGRCNode( (*i).first );

  // Check that the generated graph is acyclic

  checkAcyclic(firstNode);

  // Cluster the nodes in the graph

  vector<Cluster*> clusters;
  map<GRCNode*, Cluster*> clusterOf;
#ifdef SIMPLE_CLUSTER
  // Schedule the nodes in the control-flow graph

  vector<GRCNode*> schedule;
  calculateSchedule(firstNode, schedule);

  o << "/* Schedule:\n";
  for ( vector<GRCNode*>::iterator i = schedule.begin() ; i != schedule.end() ;
        i++ )
    o << cfgmap[*i] << ' ';
  o << "*/\n";
  cluster(schedule, clusters, clusterOf);
#else
```

```
  greedyCluster(firstNode, clusters, clusterOf);
#endif

#ifdef CLUSTER_DOT_OUTPUT
  o << "/* Clusters:\n";
  for ( vector<Cluster*>::iterator i = clusters.begin() ;
        i != clusters.end() ; i++ ) {
    o << "subgraph cluster" << i - clusters.begin()
      << " { style=filled color=lightgrey\n  ";
    for ( vector<GRCNode*>::iterator j = (*i)->nodes.begin() ;
          j != (*i)->nodes.end() ; j++ )
      o << 'n' << cfgmap[*j] << ' ';
    o << "}\n";
  }
  o << "*/\n";
#endif

  vector<Level*> levels;
  levelize(clusters, levels);

  // The top level should contain exactly one cluster that starts everything.
  assert(levels.front()->clusters.size() == 1);
  // The first level should be the first cluster
  assert(levels.front()->clusters.front() == clusters.front());

#ifdef LEVEL_DOT_OUTPUT
  o << "/* Levels:\n";
  for ( vector<Level*>::iterator i = levels.begin() ;
        i != levels.end() ; i++ ) {
    o << i - levels.begin() << ": ";
    for ( vector<Cluster*>::iterator j = (*i)->clusters.begin() ;
          j != (*i)->clusters.end() ; j++ ) {
      o << '(';
      for ( vector<GRCNode*>::iterator k = (*j)->nodes.begin() ;
            k != (*j)->nodes.end() ; k++ )
        o << cfgmap[*k] << ' ';
      o << ')';
    }
    o << '\n';
  }
  o << "*/\n";
#endif

  p.printDeclarations(basename);
  p.ioDefinitions();

  // Split apart the CFGs for each cluster

  for ( vector<Level*>::iterator i = levels.begin() ;
        i != levels.end() ; i++ )
```

```
  for ( vector<Cluster*>::iterator j = (*i)->clusters.begin() ;
        j != (*i)->clusters.end() ; j++ )
    split(*j, clusterOf, p.labelFor);

// Generate definitions for the code to schedule each cluster

for ( vector<Level*>::iterator i = levels.begin() ;
      i != levels.end() ; i++ )
  for ( vector<Cluster*>::iterator j = (*i)->clusters.begin() ;
        j != (*i)->clusters.end() ; j++ ) {
    Cluster *cluster = *j;
    for ( vector<GRCNode*>::iterator k = cluster->entries.begin() ;
          k != cluster->entries.end() ; k++ ) {
      GRCNode *node = *k;
      o << "#define _schedule_" << cfgmap[node]
        << " { _next_" << cluster->id << " = _level_"
        << cluster->level << "; _level_" << cluster->level << " = ";
      if (ansi) {
        o << cfgmap[node];
      } else {
        o << "&&" << p.labelFor[node];
      }
      o << "; }\n";
    }
  }

// Main "tick" function

o <<
  "\n"
  "int " << m.symbol->name << "(void)\n"
  "{\n"
  ;

if (ansi) o << "  unsigned int _next;\n";

for ( unsigned int i = 1 ; i < clusters.size() ; i++ ) {
  if (ansi) o << "  unsigned int ";
  else o <<  "  void *";
  o << "_next_" << i << ";\n";
}
o << '\n';
for ( unsigned int i = 1 ; i < levels.size() ; i++ ) {
  if (ansi) o << "  unsigned int ";
  else o << "  void *";
  o << "_level_" << i << " = ";
  if (ansi) o << '0';
  else o << "&&_LEVEL_" << i << "_END";
  o << ";\n";
}
```

```
o << '\n';

// For each level, for each cluster, print code

for ( unsigned int i = 0 ; i < levels.size() ; i++ ) {

  Level &level = *(levels[i]);

  if ( i > 0 ) {
    if (ansi) {
      o <<
        "  _next = _level_" << i << ";\n"
        "\n"
        "_LEVEL_" << i << "_START:\n"
        "  switch (_next) {\n";
      for ( vector<Cluster*>::iterator j = level.clusters.begin() ;
            j != level.clusters.end() ; j++ ) {
        Cluster *cluster = *j;
        for ( vector<GRCNode*>::iterator k = cluster->entries.begin() ;
              k != cluster->entries.end() ; k++ ) {
          GRCNode *node = *k;
          o << "  case " << cfgmap[node]
            << ": goto " << p.labelFor[node] << ";\n";
        }
      }
      o <<
        "  default: goto _LEVEL_" << i << "_END;\n"
        "  }\n";
    } else
      o << "  goto *_level_" << i << ";\n\n";
  }

  for ( unsigned int j = 0 ; j < level.clusters.size() ; j++ ) {
    Cluster *cluster = level.clusters[j];
    assert(cluster);
    unsigned int id = cluster->id;

    // Generate code for the body of the cluster

    assert(cluster->nodes.back()->successors.size() == 0);
    assert(cluster->nodes.back()->predecessors.size() > 0);
    p.printStructuredCode(cluster->nodes.back(), 1);

    if ( i > 0 ) {
      if (ansi) o <<
                  "  _next = _next_" << id << ";\n"
                  "  goto _LEVEL_" << i << "_START;\n"
                  "\n";
      else o <<
             "  goto *_next_" << id << ";\n"
```

```
                "\n";
        }
      }

      if ( i > 0 ) o <<
                    " _LEVEL_" << i << "_END:\n";
    }

    o << '\n';

    p.outputFunctions();
    p.resetInputs();

  #ifdef DEBUG
    // Print each of the state variables
    for ( STmap::const_iterator i = stmap.begin() ; i != stmap.end() ; i++ ) {
      STexcl *e = dynamic_cast<STexcl*>((*i).first);
      if (e) {
        o << "  printf(\"";
        p.stateVar(e);
        o << " = %d\\n\", ";
        p.stateVar(e);
        o << ");\n";
      }
    }
  #endif

    STexcl *excl = dynamic_cast<STexcl*>(g->selection_tree);
    assert(excl);
    assert(!excl->children.empty());

    STleaf *terminal_leaf = dynamic_cast<STleaf*>(excl->children.front());

    if (terminal_leaf && terminal_leaf->isfinal()) {
      o << "  return " << p.stateVar[excl] << " != 0;\n";
    } else {
      o << "  return 1;\n";
    }

    o << "}" "\n";

    // Reset function

    o <<
      "\n"
      "int " << m.symbol->name << "_reset(void)\n"
      "{\n";

    // Reset the topmost state variable
    o << "  " << p.stateVar[excl];
```

```
    o << " = " << (excl->children.size() - 1) << ";\n";

    p.resetInputs();

    o <<
      "  return 0;"    "\n"
      "}"              "\n";
  }
```

2 fixGRCnode

This remove control-flow successors from Terminate nodes that lead to Sync nodes and adds a control-flow successor to from each Fork to its Sync. This is done because in the control-flow graph, only Sync nodes ever have more than one active incoming control arc. The scheduling code assumes that each node is scheduled at most once.

It also remove successors from any EnterGRC node and predecessors from any ExitGRC node.

8 ⟨*declarations* 1⟩+≡

```
  void fixGRCNode(GRCNode *);
```

9 ⟨*definitions* 3⟩+≡

```
void fixGRCNode(GRCNode *n)
{
  Terminate *t = dynamic_cast<Terminate *>(n);
  if (t) {
    if ( t->successors.size() == 1) {
      Sync *s = dynamic_cast<Sync*>(t->successors.front());
      if (s) {
        // Remove the Terminate from the sync node's predecessors
        for ( vector<GRCNode*>::iterator i = s->predecessors.begin() ;
              i != s->predecessors.end() ; i++ )
          if ( (*i) == t ) {
            s->predecessors.erase(i);
            break;
          }
        // Remove the Terminate's successors
        t->successors.clear();
      }
    }
  }

  Fork *f = dynamic_cast<Fork *>(n);
  if (f && f->sync) {
    // Add the sync node as a successor to this Fork
    (*f) >> f->sync;
    // std::cerr << "Added " << cfgmap[f->sync] << " to fork node " << cfgmap[f] << std::endl;
  }

  if (dynamic_cast<EnterGRC*>(n)) {
    for ( vector<GRCNode*>::iterator j = n->successors.begin() ;
          j != n->successors.end() ; j++ )
      for ( vector<GRCNode*>::iterator k = (*j)->predecessors.begin() ;
            k != (*j)->predecessors.end() ; k++ )
        if (*k == n) {
          (*j)->predecessors.erase(k);
          break;
        }
    n->successors.clear();
  }

  if (dynamic_cast<ExitGRC*>(n)) {
    for ( vector<GRCNode*>::iterator j = n->predecessors.begin() ;
          j != n->predecessors.end() ; j++ )
      for ( vector<GRCNode*>::iterator k = (*j)->successors.begin() ;
            k != (*j)->successors.end() ; k++ )
        if (*k == n) {
          (*j)->successors.erase(k);
          break;
        }
    n->predecessors.clear();
```

```
    }
  }
```

3 Check Acyclic

This uses a simple DFS to verify that the given control-flow graph is acyclic.

10a ⟨*declarations* 1⟩+≡

```
  void checkAcyclic(GRCNode *);
```

10b ⟨*definitions* 3⟩+≡

```
  void checkAcyclic(GRCNode *n) {
    AcyclicChecker checker(n);
  }
```

10c ⟨*declarations* 1⟩+≡

```
  class AcyclicChecker {
    std::map<GRCNode*, bool> completed;
    bool visit(GRCNode *);
  public:
    AcyclicChecker(GRCNode* n) {
      if (visit(n)) {
        std::cerr << std::endl;
        throw IR::Error("CFG is cyclic");
      }
    }
  };
```

The DFS procedure `visit` returns true if a cycle was found.

11a ⟨*definitions* 3⟩+≡
```
bool AcyclicChecker::visit(GRCNode *n)
{
  if ( n == NULL ||
       (completed.find(n) != completed.end() && completed[n]) ) return false;

  if (completed.find(n) != completed.end() && !completed[n]) {
    std::cerr << "Cycle found, includes nodes " << cfgmap[n];
    return true;
  }

  completed[n] = false;

  for (vector<GRCNode*>::iterator i = n->successors.begin() ;
       i != n->successors.end() ; i++)
    if ( visit(*i) ) {
      std::cerr << ' ' << cfgmap[n];
      return true;
    }

  for (vector<GRCNode*>::iterator i = n->dataSuccessors.begin() ;
       i != n->dataSuccessors.end() ; i++)
    if ( visit(*i) ) {
      std::cerr << ' ' << cfgmap[n];
      return true;
    }

  completed[n] = true;

  return false;
}
```

4 Scheduling

This computes a schedule for the reachable nodes in the control-flow graph. This is a simple topological sort.

11b ⟨*declarations* 1⟩+≡
```
void calculateSchedule(GRCNode *, vector<GRCNode*> &);
```

11c ⟨*definitions* 3⟩+≡
```
void calculateSchedule(GRCNode *n, vector<GRCNode*> &s) {
  Scheduler scheduler(n, s);
}
```

12a ⟨*declarations* 1⟩+≡
```
class Scheduler {
  std::set<GRCNode*> visited;
  std::vector<GRCNode*> &schedule;
  void visit(GRCNode*);
public:
  Scheduler(GRCNode *n, vector<GRCNode*> &schedule) : schedule(schedule) {
    visit(n);
  }
};
```

12b ⟨*definitions* 3⟩+≡
```
void Scheduler::visit(GRCNode *n)
{
  if (n == NULL || visited.find(n) != visited.end()) return;

  visited.insert(n);

  for (vector<GRCNode*>::iterator i = n->successors.begin() ;
       i != n->successors.end() ; i++)
    visit(*i);

  for (vector<GRCNode*>::iterator i = n->dataSuccessors.begin() ;
       i != n->dataSuccessors.end() ; i++)
    visit(*i);

  schedule.insert(schedule.begin(), n);
}
```

5 Simple Clustering

This algorithm constructs clusters: a sequence of nodes in the schedule that do not have control or data predecessors from outside the cluster.

12c ⟨*declarations* 1⟩+≡
```
struct Cluster {
  unsigned int id;
  unsigned int level;
  vector<GRCNode*> nodes;
  std::set<Cluster*> successors;
  vector<GRCNode*> entries;
  Cluster(int id) : id(id) {}
};
```

12d ⟨*declarations* 1⟩+≡
```
void cluster(vector<GRCNode*> &, vector<Cluster*> &, map<GRCNode*, Cluster*> &);
```

13 ⟨*definitions* 3⟩+≡

```
void cluster(vector<GRCNode*> &s, vector<Cluster*> &c, map<GRCNode*, Cluster*> &m)
{
  set<GRCNode*> nodes;
  Cluster *cluster = NULL;
  unsigned int id = 0;

  for ( vector<GRCNode*>::const_iterator i = s.begin(); i != s.end() ; i++ ) {

    // Decide whether to start a new cluster

    bool startNew = (cluster == NULL);
    if (!startNew)
      for ( vector<GRCNode*>::const_iterator j = (*i)->predecessors.begin() ;
            j != (*i)->predecessors.end() ; j++ )
        if ( nodes.find(*j) == nodes.end() || dynamic_cast<Fork*>(*j) ) {
          startNew = true;
          break;
        }
    if (!startNew)
      for ( vector<GRCNode*>::const_iterator j = (*i)->dataPredecessors.begin() ;
            j != (*i)->dataPredecessors.end() ; j++ )
        if ( nodes.find(*j) == nodes.end() ) {
          startNew = true;
          break;
        }

    if (startNew) {
      // Start a new cluster: add a new vector, clear the set
      cluster = new Cluster(id++);
      c.push_back(cluster);
      nodes.clear();
    }

    nodes.insert(*i);
    cluster->nodes.push_back(*i);
    m[*i] = cluster;
  }
}
```

6 Greedy Clustering

The clustering algorithm. This takes a control-flow graph with information about control and data predecessors and successors and produces a set of clusters, each of which is a set of nodes that can be executed without interruption. Unlike the other clustering algorithm, this does not rely on a schedule; instead it effectively creates its own schedule by greedily trying to add nodes to a cluster until no more are possible.

14 ⟨*declarations* 1⟩+≡

```
void greedyCluster(GRCNode *, vector<Cluster*> &, map<GRCNode*, Cluster*> &);
```

15 ⟨*definitions 3*⟩+≡

```
void greedyCluster(GRCNode *root, vector<Cluster*> &clusters,
                   map<GRCNode *, Cluster*> &clusterMap)
{
  // std::cerr << "greedyCluster()\n";

  // Set of nodes that have predecessors in some cluster but are not in a cluster
  set<GRCNode*> frontier;

  frontier.insert(root);

  unsigned int clusterId = 0;

  while (!frontier.empty()) {
    // Select a node from the frontier at random
    // and consider it as a seed

    set<GRCNode*>::const_iterator seedi = frontier.begin();

    GRCNode *seed = *seedi;
    assert(seed);

    frontier.erase(seed);

    // std::cerr << "Seed is " << cfgmap[seed] << std::endl;

    Cluster *cluster = 0;

    set<GRCNode*> pending; // Nodes to be considered
    pending.insert(seed);

    while (!pending.empty()) {
      set<GRCNode*>::const_iterator candidatei = pending.begin();

      GRCNode *candidate = *candidatei;
      assert(candidate);

      // std::cerr << "Candidate is " << cfgmap[candidate] << std::endl;

      pending.erase(candidatei);

      bool addToCluster = clusterMap.find(candidate) == clusterMap.end();
      for ( vector<GRCNode*>::const_iterator i =
              candidate->predecessors.begin() ;
            i != candidate->predecessors.end() ; i++ )
        if ( clusterMap.find(*i) == clusterMap.end() ) {
          addToCluster = false;
          break;
        }
      if (addToCluster)
```

```
    for ( vector<GRCNode*>::const_iterator i =
            candidate->dataPredecessors.begin() ;
          i != candidate->dataPredecessors.end() ; i++ )
      if ( clusterMap.find(*i) == clusterMap.end() ) {
        addToCluster = false;
        break;
      }
  // std::cerr << addToCluster << std::endl;
  if (addToCluster) {

    if (!cluster) {
      cluster = new Cluster(clusterId++);
      clusters.push_back(cluster);
    }

    cluster->nodes.push_back(candidate);
    clusterMap[candidate] = cluster;
    // std::cerr << "Added " << cfgmap[candidate] << " to " << clusters.size() << std::end

    if (dynamic_cast<Fork*>(candidate) == NULL) {
      // Not a fork: add all its non-null successors
      for ( vector<GRCNode*>::const_iterator i =
              candidate->successors.begin() ;
            i != candidate->successors.end() ; i++ )
        if (*i) pending.insert(*i);
    } else {
      // A fork node: only add its first successor
      assert(!candidate->successors.empty());
      assert(candidate->successors.front());
      pending.insert(candidate->successors.front());
    }

    // Add both control and data successors to the frontier
    for ( vector<GRCNode*>::const_iterator i =
            candidate->successors.begin() ;
          i != candidate->successors.end() ; i++ )
      if (*i) {
        frontier.insert(*i);
        // std::cerr << "Added " << cfgmap[*i] << " to frontier" << std::endl;
      }

    for ( vector<GRCNode*>::const_iterator i =
            candidate->dataSuccessors.begin() ;
          i != candidate->dataSuccessors.end() ; i++ ) {
      frontier.insert(*i);
      // std::cerr << "Added " << cfgmap[*i] << " to frontier" << std::endl;
    }

    set<GRCNode*>::iterator candi = frontier.find(candidate);
    if (candi != frontier.end()) frontier.erase(candi);
```

```
      }
    }
  }
  // std::cerr << "Done with greedy clusters: " << clusterId << std::endl;
}
```

7 Levelizing

The clusters are divided into levels. Nodes within the clusters within a level do not communicate.

17a ⟨*declarations* 1⟩+≡

```
struct Level {
  vector<Cluster *> clusters;
};
```

17b ⟨*declarations* 1⟩+≡

```
void levelize(vector<Cluster*> &, vector<Level*> &);
```

18 ⟨*definitions* 3⟩+≡

```
void levelize(vector<Cluster*> &b, vector<Level*> &levels)
{
  assert(!b.empty()); // Need at least one cluster

  // Construct inter-cluster dependencies

  map<GRCNode*, Cluster*> clusterOfNode;
  for ( vector<Cluster*>::const_iterator i = b.begin() ; i != b.end() ; i++ ) {
    for ( vector<GRCNode*>::const_iterator j = (*i)->nodes.begin() ;
          j != (*i)->nodes.end() ; j++ ) {
      clusterOfNode[*j] = *i;
      for ( vector<GRCNode*>::const_iterator k = (*j)->predecessors.begin() ;
            k != (*j)->predecessors.end() ; k++ ) {
        if (clusterOfNode.find(*k) != clusterOfNode.end() && clusterOfNode[*k] != *i)
          clusterOfNode[*k]->successors.insert(*i);
      }
      for ( vector<GRCNode*>::const_iterator k = (*j)->dataPredecessors.begin() ;
            k != (*j)->dataPredecessors.end() ; k++ ) {
        if (clusterOfNode.find(*k) != clusterOfNode.end() && clusterOfNode[*k] != *i)
          clusterOfNode[*k]->successors.insert(*i);
      }
    }
  }

  // Calculate the level of each cluster through relaxation

  map<Cluster*, unsigned int> level;
  for ( vector<Cluster*>::const_iterator i = b.begin() ; i != b.end() ; i++ )
    level[*i] = 0;

  set<Cluster*> unvisited;
  unvisited.insert(b.front());

  unsigned maxlevel = 0;

  while (!unvisited.empty()) {
    Cluster *vb = *(unvisited.begin());
    assert(vb);
    unvisited.erase(unvisited.begin());

    assert(level.find(vb) != level.end());
    unsigned int nextlevel = level[vb] + 1;
    for ( set<Cluster*>::const_iterator i = vb->successors.begin() ;
          i != vb->successors.end() ; i++ )
      if ( level[*i] < nextlevel ) {
        level[*i] = nextlevel;
        if (nextlevel > maxlevel) maxlevel = nextlevel;
        unvisited.insert(*i);
      }
```

```
    }

    // Create the levels

    for (unsigned int i = 0 ; i <= maxlevel ; i++)
      levels.push_back(new Level());

    // Insert clusters in the levels

    for ( map<Cluster *, unsigned int>::const_iterator i = level.begin() ;
          i != level.end() ; i++ ) {
      Cluster *cluster = (*i).first;
      unsigned int level = (*i).second;
      levels[level]->clusters.push_back(cluster);
      cluster->level = level;
    }
  }
```

8 Split cluster nodes

The C generator wants a self-contained CFG; this breaks the control dependencies among nodes in different clusters to ensure this.

19 ⟨*declarations* 1⟩+≡

```
  void split(Cluster *, map<GRCNode*, Cluster*> &, map<GRCNode*, string> &);
```

20 ⟨*definitions* 3⟩+≡

```
void split(Cluster *cluster, map<GRCNode*, Cluster*> &clusterOf,
           map<GRCNode*, string> &labelFor)
{
  assert(cluster);
  // std::cerr << "Splitting cluster " << cluster->id << '\n';
  ExitGRC *exitNode = new ExitGRC();

  for ( vector<GRCNode*>::iterator i = cluster->nodes.begin() ;
        i != cluster->nodes.end() ; i++ ) {
    GRCNode *node = *i;
    assert(node);
    // std::cerr << "Examining a node \n";

    if ( dynamic_cast<Fork*>(node) != NULL ) {

      // A fork node: convert its successors to a sequence of
      // schedule statements
      // that branch to the exitNode of the cluster.

      GRCNode *nopchain = exitNode;

      // If there is exactly one successor in this same cluster,
      // make it the successor of the nopchain
      for ( vector<GRCNode*>::iterator j = node->successors.begin() ;
            j != node->successors.end() ; j++ ) {
        assert(*j);
        // The successor should have been put in some cluster
        if (clusterOf.find(*j) == clusterOf.end()) std::cerr << "node " << cfgmap[*j] << " not
        assert(clusterOf.find(*j) != clusterOf.end());
        if (clusterOf[*j] == cluster) {
          // Should not have already found a successor in the same cluster
          assert(nopchain == exitNode);
          nopchain = *j;
        }
      }

      // Build a chain of "Nop" nodes that schedule the successors of the
      // fork that are in other clusters

      for ( vector<GRCNode*>::iterator j = node->successors.begin() ;
            j != node->successors.end() ; j++ ) {
        assert(*j);
        assert(clusterOf.find(*j) != clusterOf.end());
        if ( clusterOf[*j] != cluster ) {
          Nop *nop = new Nop();
          char buf[30];
          sprintf(buf, "_schedule_%d", cfgmap[*j]);
          nop->body = buf;
          *nop >> nopchain;
```

```
        nopchain = nop;
        if (labelFor.find(*j) == labelFor.end()) {
          // Not yet marked as an entry point.
          char buf[30];
          sprintf(buf, "L%d", cfgmap[*j]);
          labelFor[*j] = buf;
          clusterOf[*j]->entries.push_back(*j);
        }
      }
    }

    // Disconnect the fork's successors

    for ( vector<GRCNode*>::iterator j = node->successors.begin() ;
          j != node->successors.end() ; j++ ) {
      for ( vector<GRCNode*>::iterator k = (*j)->predecessors.begin() ;
            k != (*j)->predecessors.end() ; k++ )
          if (*k == node) {
            (*j)->predecessors.erase(k);
            break;
          }
    }
    node->successors.clear();

    // Connect the fork to the beginning of the chain

    *node >> nopchain;

  } else {

    // A normal node

    for ( vector<GRCNode*>::iterator j = node->successors.begin() ;
          j != node->successors.end() ; j++ )
      if ( *j ) {
        // The successor should have been put in some cluster
        if (clusterOf.find(*j) == clusterOf.end()) std::cerr << "node " << cfgmap[*j] << " not in any cl
        assert(clusterOf.find(*j) != clusterOf.end());
        if (clusterOf[*j] != cluster ) {

          // std::cerr << "Fixing a node that branches to outside this cluster\n";

          // Control flows to a different cluster: change this successor to
          // a Nop node that schedules the cluster and branches to the exitNode

          // Delete the link back to us
          for ( vector<GRCNode*>::iterator k = (*j)->predecessors.begin() ;
                k != (*j)->predecessors.end() ; k++ )
            if (*k == node) {
              (*j)->predecessors.erase(k);
```

```
              break;
            }

          if (labelFor.find(*j) == labelFor.end()) {
            // Not yet marked as an entry point.
            char buf[30];
            sprintf(buf, "L%d", cfgmap[*j]);
            labelFor[*j] = buf;
            clusterOf[*j]->entries.push_back(*j);
          }

          Nop *nop = new Nop();
          char buf[30];
          sprintf(buf, "_schedule_%d", cfgmap[*j]);
          nop->body = buf;
          *j = nop;
          nop->predecessors.push_back(node);
          *nop >> exitNode;
        }
      }

      // A node with no successors? Send it to the exit node
      if (node->successors.empty())
        *node >> exitNode;
    }
  }

  cluster->nodes.push_back(exitNode);
  // Note: The list of nodes in the cluster does not include the Nop nodes
  // Moreover, the clusterOf map does not include the Nops or the exit node

  assert(exitNode->successors.size() == 0);
  assert(exitNode->predecessors.size() > 0);
}
```

9 Top-Level Files

9.1 GRCC2.hpp

23a ⟨*GRCC2.hpp* 23a⟩≡

```
#ifndef _GRCC2_HPP
#  define _GRCC2_HPP

#  include "AST.hpp"
#  include "CPrinter.hpp"

namespace GRCC2 {
  using namespace AST;
  ⟨declarations 1⟩
}

#endif
```

9.2 GRCC2.cpp

23b ⟨*GRCC2.cpp* 23b⟩≡

```
#include "GRCC2.hpp"
#include <set>
#include <vector>
#include <cstdio>

/* for debugging */
/* #define CLUSTER_DOT_OUTPUT */
/* #define LEVEL_DOT_OUTPUT */

namespace GRCC2 {
  using std::set;
  using std::vector;

  ⟨definitions 3⟩
}
```

9.3 cec-grcc2.cpp

24 ⟨*cec-grcc2.cpp* 24⟩≡

```
#include "AST.hpp"
#include "GRCC2.hpp"
#include <iostream>
#include <cassert>
#include <string>
#include <string.h>

struct Usage {};

int main(int argc, char *argv[])
{
  try {

    std::string basename;
    bool ansi = false;

    argc--, argv++;  // Skip program name

    while (argc > 0 && argv[0][0] == '-') {
      switch (argv[0][1]) {
      case 'a':
        ansi = true;
        break;
      case 'B':
        argc--, argv++;
        if (argc == 0) throw Usage();
        basename = argv[0];
        break;
      case 'h':
      default:
        throw Usage();
        break;
      }
      argc--, argv++;
    }

    if ( argc > 0 ) throw Usage();

    IR::XMListream r(std::cin);
    IR::Node *n;
    r >> n;

    AST::Modules *mods = dynamic_cast<AST::Modules*>(n);
    if (!mods) throw IR::Error("Root node is not a Modules object");

    assert(mods->modules.size() > 0);
    AST::Module *m = mods->modules.front();
```

```
    assert(m);

    if (basename.empty()) basename = m->symbol->name;

    GRCC2::generateC(std::cout, *m, basename, ansi);

  } catch (IR::Error &e) {
    std::cerr << e.s << std::endl;
    exit(-1);
  } catch (Usage &) {
    std::cerr << "Usage: cec-grcc2 [-a] [-B basename]" << std::endl;
    exit(-1);
  }
  return 0;
}
```