jsimsa Fix use-after-free in MutableGraphView. Abseil containers do not guar... ... ✓ History

9 contributors

1619 lines (1416 sloc) | 60.3 KB ...

```
1  /* Copyright 2018 The TensorFlow Authors. All Rights Reserved.
2
3  Licensed under the Apache License, Version 2.0 (the "License");
4  you may not use this file except in compliance with the License.
5  You may obtain a copy of the License at
6
7      http://www.apache.org/licenses/LICENSE-2.0
8
9  Unless required by applicable law or agreed to in writing, software
10 distributed under the License is distributed on an "AS IS" BASIS,
11 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 See the License for the specific language governing permissions and
13 limitations under the License.
14 ==============================================================================*/
15
16 #include "tensorflow/core/grappler/mutable_graph_view.h"
17
18 #include <algorithm>
19 #include <utility>
20
21 #include "absl/container/flat_hash_map.h"
22 #include "absl/strings/str_cat.h"
23 #include "absl/strings/str_join.h"
24 #include "absl/strings/string_view.h"
25 #include "absl/strings/substitute.h"
26 #include "tensorflow/core/framework/function.h"
27 #include "tensorflow/core/framework/graph.pb.h"
28 #include "tensorflow/core/framework/node_def.pb.h"
29 #include "tensorflow/core/graph/graph.h"
```

```cpp
#include "tensorflow/core/graph/tensor_id.h"
#include "tensorflow/core/grappler/op_types.h"
#include "tensorflow/core/grappler/utils.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/lib/core/stringpiece.h"
#include "tensorflow/core/lib/gtl/map_util.h"
#include "tensorflow/core/platform/types.h"

namespace tensorflow {
namespace grappler {

namespace {

bool IsTensorIdPortValid(const TensorId& tensor_id) {
  return tensor_id.index() >= Graph::kControlSlot;
}

bool IsTensorIdRegular(const TensorId& tensor_id) {
  return tensor_id.index() > Graph::kControlSlot;
}

bool IsTensorIdControlling(const TensorId& tensor_id) {
  return tensor_id.index() == Graph::kControlSlot;
}

bool IsOutputPortControlling(const MutableGraphView::OutputPort& port) {
  return port.port_id == Graph::kControlSlot;
}

// Determines if node is an Identity where it's first regular input is a Switch
// node.
bool IsIdentityConsumingSwitch(const MutableGraphView& graph,
                               const NodeDef& node) {
  if ((IsIdentity(node) || IsIdentityNSingleInput(node)) &&
      node.input_size() > 0) {
    TensorId tensor_id = ParseTensorName(node.input(0));
    if (IsTensorIdControlling(tensor_id)) {
      return false;
    }

    NodeDef* input_node = graph.GetNode(tensor_id.node());
    return IsSwitch(*input_node);
  }
  return false;
}

// Determines if node input can be deduped by regular inputs when used as a
// control dependency. Specifically, if a node is an Identity that leads to a
// Switch node, when used as a control dependency, that control dependency
```

```
// should not be deduped even though the same node is used as a regular input.
bool CanDedupControlWithRegularInput(const MutableGraphView& graph,
                                     const NodeDef& control_node) {
  return !IsIdentityConsumingSwitch(graph, control_node);
}

// Determines if node input can be deduped by regular inputs when used as a
// control dependency. Specifically, if a node is an Identity that leads to a
// Switch node, when used as a control dependency, that control dependency
// should not be deduped even though the same node is used as a regular input.
bool CanDedupControlWithRegularInput(const MutableGraphView& graph,
                                     absl::string_view control_node_name) {
  NodeDef* control_node = graph.GetNode(control_node_name);
  if (control_node == nullptr) {
    return false;
  }
  return CanDedupControlWithRegularInput(graph, *control_node);
}

bool HasRegularFaninNode(const MutableGraphView& graph, const NodeDef& node,
                         absl::string_view fanin_node_name) {
  const int num_regular_fanins =
      graph.NumFanins(node, /*include_controlling_nodes=*/false);
  for (int i = 0; i < num_regular_fanins; ++i) {
    if (ParseTensorName(node.input(i)).node() == fanin_node_name) {
      return true;
    }
  }
  return false;
}

using FanoutsMap =
    absl::flat_hash_map<MutableGraphView::OutputPort,
                        absl::flat_hash_set<MutableGraphView::InputPort>>;

void SwapControlledFanoutInputs(const MutableGraphView& graph,
                                const FanoutsMap::iterator& control_fanouts,
                                absl::string_view to_node_name) {
  absl::string_view from_node_name(control_fanouts->first.node->name());
  string control = TensorIdToString({to_node_name, Graph::kControlSlot});
  for (const auto& control_fanout : control_fanouts->second) {
    const int start = graph.NumFanins(*control_fanout.node,
                                      /*include_controlling_nodes=*/false);
    for (int i = start; i < control_fanout.node->input_size(); ++i) {
      TensorId tensor_id = ParseTensorName(control_fanout.node->input(i));
      if (tensor_id.node() == from_node_name) {
        control_fanout.node->set_input(i, control);
        break;
      }
```

```
    }
  }
}

void SwapRegularFanoutInputs(FanoutsMap* fanouts, NodeDef* from_node,
                             absl::string_view to_node_name, int max_port) {
  MutableGraphView::OutputPort port;
  port.node = from_node;
  for (int i = 0; i <= max_port; ++i) {
    port.port_id = i;
    auto it = fanouts->find(port);
    if (it == fanouts->end()) {
      continue;
    }
    string input = TensorIdToString({to_node_name, i});
    for (const auto& fanout : it->second) {
      fanout.node->set_input(fanout.port_id, input);
    }
  }
}

using MaxOutputPortsMap = absl::flat_hash_map<const NodeDef*, int>;

void SwapFanoutInputs(const MutableGraphView& graph, FanoutsMap* fanouts,
                      MaxOutputPortsMap* max_output_ports, NodeDef* from_node,
                      NodeDef* to_node) {
  auto from_control_fanouts = fanouts->find({from_node, Graph::kControlSlot});
  if (from_control_fanouts != fanouts->end()) {
    SwapControlledFanoutInputs(graph, from_control_fanouts, to_node->name());
  }
  auto to_control_fanouts = fanouts->find({to_node, Graph::kControlSlot});
  if (to_control_fanouts != fanouts->end()) {
    SwapControlledFanoutInputs(graph, to_control_fanouts, from_node->name());
  }
  auto from_max_port = max_output_ports->find(from_node);
  if (from_max_port != max_output_ports->end()) {
    SwapRegularFanoutInputs(fanouts, from_node, to_node->name(),
                            from_max_port->second);
  }
  auto to_max_port = max_output_ports->find(to_node);
  if (to_max_port != max_output_ports->end()) {
    SwapRegularFanoutInputs(fanouts, to_node, from_node->name(),
                            to_max_port->second);
  }
}

void SwapFanoutsMapValues(FanoutsMap* fanouts,
                          const MutableGraphView::OutputPort& from_port,
                          const FanoutsMap::iterator& from_fanouts,
```

```
                          const MutableGraphView::OutputPort& to_port,
                          const FanoutsMap::iterator& to_fanouts) {
  const bool from_exists = from_fanouts != fanouts->end();
  const bool to_exists = to_fanouts != fanouts->end();

  if (from_exists && to_exists) {
    std::swap(from_fanouts->second, to_fanouts->second);
  } else if (from_exists) {
    fanouts->emplace(to_port, std::move(from_fanouts->second));
    fanouts->erase(from_port);
  } else if (to_exists) {
    fanouts->emplace(from_port, std::move(to_fanouts->second));
    fanouts->erase(to_port);
  }
}

void SwapRegularFanoutsAndMaxPortValues(FanoutsMap* fanouts,
                                        MaxOutputPortsMap* max_output_ports,
                                        NodeDef* from_node, NodeDef* to_node) {
  auto from_max_port = max_output_ports->find(from_node);
  auto to_max_port = max_output_ports->find(to_node);
  bool from_exists = from_max_port != max_output_ports->end();
  bool to_exists = to_max_port != max_output_ports->end();

  auto forward_fanouts = [fanouts](NodeDef* from, NodeDef* to, int start,
                                   int end) {
    for (int i = start; i <= end; ++i) {
      MutableGraphView::OutputPort from_port(from, i);
      auto from_fanouts = fanouts->find(from_port);
      if (from_fanouts != fanouts->end()) {
        MutableGraphView::OutputPort to_port(to, i);
        fanouts->emplace(to_port, std::move(from_fanouts->second));
        fanouts->erase(from_port);
      }
    }
  };

  if (from_exists && to_exists) {
    const int from = from_max_port->second;
    const int to = to_max_port->second;
    const int shared = std::min(from, to);
    for (int i = 0; i <= shared; ++i) {
      MutableGraphView::OutputPort from_port(from_node, i);
      auto from_fanouts = fanouts->find(from_port);
      MutableGraphView::OutputPort to_port(to_node, i);
      auto to_fanouts = fanouts->find(to_port);
      SwapFanoutsMapValues(fanouts, from_port, from_fanouts, to_port,
                           to_fanouts);
    }
```

```
226       if (to > from) {
227         forward_fanouts(to_node, from_node, shared + 1, to);
228       } else if (from > to) {
229         forward_fanouts(from_node, to_node, shared + 1, from);
230       }
231
232       std::swap(from_max_port->second, to_max_port->second);
233     } else if (from_exists) {
234       forward_fanouts(from_node, to_node, 0, from_max_port->second);
235
236       max_output_ports->emplace(to_node, from_max_port->second);
237       max_output_ports->erase(from_node);
238     } else if (to_exists) {
239       forward_fanouts(to_node, from_node, 0, to_max_port->second);
240
241       max_output_ports->emplace(from_node, to_max_port->second);
242       max_output_ports->erase(to_node);
243     }
244   }
245
246   bool HasFanoutValue(const FanoutsMap& fanouts, const FanoutsMap::iterator& it) {
247     return it != fanouts.end() && !it->second.empty();
248   }
249
250   Status MutationError(absl::string_view function_name, absl::string_view params,
251                        absl::string_view msg) {
252     return errors::InvalidArgument(absl::Substitute(
253         "MutableGraphView::$0($1) error: $2.", function_name, params, msg));
254   }
255
256   using ErrorHandler = std::function<Status(absl::string_view)>;
257
258   ErrorHandler UpdateFanoutsError(absl::string_view from_node_name,
259                                   absl::string_view to_node_name) {
260     return [from_node_name, to_node_name](absl::string_view msg) {
261       string params = absl::Substitute("from_node_name='$0', to_node_name='$1'",
262                                        from_node_name, to_node_name);
263       return MutationError("UpdateFanouts", params, msg);
264     };
265   }
266
267   Status CheckFaninIsRegular(const TensorId& fanin, ErrorHandler handler) {
268     if (!IsTensorIdRegular(fanin)) {
269       return handler(absl::Substitute("fanin '$0' must be a regular tensor id",
270                                       fanin.ToString()));
271     }
272     return Status::OK();
273   }
274
```

```
Status CheckFaninIsValid(const TensorId& fanin, ErrorHandler handler) {
  if (!IsTensorIdPortValid(fanin)) {
    return handler(absl::Substitute("fanin '$0' must be a valid tensor id",
                                    fanin.ToString()));
  }
  return Status::OK();
}

Status CheckAddingFaninToSelf(absl::string_view node_name,
                              const TensorId& fanin, ErrorHandler handler) {
  if (node_name == fanin.node()) {
    return handler(
        absl::Substitute("can't add fanin '$0' to self", fanin.ToString()));
  }
  return Status::OK();
}

Status CheckRemovingFaninFromSelf(absl::string_view node_name,
                                  const TensorId& fanin, ErrorHandler handler) {
  if (node_name == fanin.node()) {
    return handler(absl::Substitute("can't remove fanin '$0' from self",
                                    fanin.ToString()));
  }
  return Status::OK();
}

string NodeMissingErrorMsg(absl::string_view node_name) {
  return absl::Substitute("node '$0' was not found", node_name);
}

Status CheckNodeExists(absl::string_view node_name, NodeDef* node,
                       ErrorHandler handler) {
  if (node == nullptr) {
    return handler(NodeMissingErrorMsg(node_name));
  }
  return Status::OK();
}

Status CheckPortRange(int port, int min, int max, ErrorHandler handler) {
  if (port < min || port > max) {
    if (max < min) {
      return handler("no available ports as node has no regular fanins");
    }
    return handler(
        absl::Substitute("port must be in range [$0, $1]", min, max));
  }
  return Status::OK();
}

```

```
string SwapNodeNamesSwitchControlErrorMsg(absl::string_view node_name) {
  return absl::Substitute(
      "can't swap node name '$0' as it will become a Switch control dependency",
      node_name);
}

string GeneratedNameForIdentityConsumingSwitch(
    const MutableGraphView::OutputPort& fanin) {
  return AddPrefixToNodeName(
      absl::StrCat(fanin.node->name(), "_", fanin.port_id),
      kMutableGraphViewCtrl);
}

}  // namespace

void MutableGraphView::AddAndDedupFanouts(NodeDef* node) {
  // TODO(lyandy): Checks for self loops, Switch control dependencies, fanins
  // exist, and all regular fanins come before controlling fanins.
  absl::flat_hash_set<absl::string_view> fanins;
  absl::flat_hash_set<absl::string_view> controlling_fanins;
  int max_input_port = -1;
  int pos = 0;
  const int last_idx = node->input_size() - 1;
  int last_pos = last_idx;
  while (pos <= last_pos) {
    TensorId tensor_id = ParseTensorName(node->input(pos));
    absl::string_view input_node_name = tensor_id.node();
    bool is_control_input = IsTensorIdControlling(tensor_id);
    bool can_dedup_control_with_regular_input =
        CanDedupControlWithRegularInput(*this, input_node_name);
    bool can_dedup_control =
        is_control_input && (can_dedup_control_with_regular_input ||
                             controlling_fanins.contains(input_node_name));
    if (!gtl::InsertIfNotPresent(&fanins, input_node_name) &&
        can_dedup_control) {
      node->mutable_input()->SwapElements(pos, last_pos);
      --last_pos;
    } else {
      OutputPort output(nodes()[input_node_name], tensor_id.index());

      if (is_control_input) {
        fanouts()[output].emplace(node, Graph::kControlSlot);
      } else {
        max_input_port = pos;
        max_regular_output_port()[output.node] =
            std::max(max_regular_output_port()[output.node], output.port_id);
        fanouts()[output].emplace(node, pos);
      }
      ++pos;
```

```
373       }
374       if (is_control_input) {
375         controlling_fanins.insert(input_node_name);
376       }
377     }
378
379     if (last_pos < last_idx) {
380       node->mutable_input()->DeleteSubrange(last_pos + 1, last_idx - last_pos);
381     }
382
383     if (max_input_port > -1) {
384       max_regular_input_port()[node] = max_input_port;
385     }
386   }
387
388   void MutableGraphView::UpdateMaxRegularOutputPortForRemovedFanin(
389       const OutputPort& fanin,
390       const absl::flat_hash_set<InputPort>& fanin_fanouts) {
391     int max_port = max_regular_output_port()[fanin.node];
392     if (!fanin_fanouts.empty() || max_port != fanin.port_id) {
393       return;
394     }
395     bool updated_max_port = false;
396     for (int i = fanin.port_id - 1; i >= 0; --i) {
397       OutputPort fanin_port(fanin.node, i);
398       if (!fanouts()[fanin_port].empty()) {
399         max_regular_output_port()[fanin.node] = i;
400         updated_max_port = true;
401         break;
402       }
403     }
404     if (!updated_max_port) {
405       max_regular_output_port().erase(fanin.node);
406     }
407   }
408
409   void MutableGraphView::UpdateMaxRegularOutputPortForAddedFanin(
410       const OutputPort& fanin) {
411     if (max_regular_output_port()[fanin.node] < fanin.port_id) {
412       max_regular_output_port()[fanin.node] = fanin.port_id;
413     }
414   }
415
416   const absl::flat_hash_set<MutableGraphView::InputPort>&
417   MutableGraphView::GetFanout(const GraphView::OutputPort& port) const {
418     return GetFanout(MutableGraphView::OutputPort(const_cast<NodeDef*>(port.node),
419                                                   port.port_id));
420   }
421
```

```
absl::flat_hash_set<MutableGraphView::OutputPort> MutableGraphView::GetFanin(
    const GraphView::InputPort& port) const {
  return GetFanin(MutableGraphView::InputPort(const_cast<NodeDef*>(port.node),
                                              port.port_id));
}

const MutableGraphView::OutputPort MutableGraphView::GetRegularFanin(
    const GraphView::InputPort& port) const {
  return GetRegularFanin(MutableGraphView::InputPort(
      const_cast<NodeDef*>(port.node), port.port_id));
}

NodeDef* MutableGraphView::AddNode(NodeDef&& node) {
  auto* node_in_graph = graph()->add_node();
  *node_in_graph = std::move(node);

  AddUniqueNodeOrDie(node_in_graph);

  AddAndDedupFanouts(node_in_graph);
  return node_in_graph;
}

Status MutableGraphView::AddSubgraph(GraphDef&& subgraph) {
  // 1. Add all new functions and check that functions with the same name
  // have identical definition.
  const int function_size = subgraph.library().function_size();
  if (function_size > 0) {
    absl::flat_hash_map<absl::string_view, const FunctionDef*> graph_fdefs;
    for (const FunctionDef& fdef : graph()->library().function()) {
      graph_fdefs.emplace(fdef.signature().name(), &fdef);
    }

    for (FunctionDef& fdef : *subgraph.mutable_library()->mutable_function()) {
      const auto graph_fdef = graph_fdefs.find(fdef.signature().name());

      if (graph_fdef == graph_fdefs.end()) {
        VLOG(3) << "Add new function definition: " << fdef.signature().name();
        graph()->mutable_library()->add_function()->Swap(&fdef);
      } else {
        if (!FunctionDefsEqual(fdef, *graph_fdef->second)) {
          return MutationError(
              "AddSubgraph",
              absl::Substitute("function_size=$0", function_size),
              absl::StrCat(
                  "Found different function definition with the same name: ",
                  fdef.signature().name()));
        }
      }
    }
```

```
471   }
472
473   // 2. Add all nodes to the underlying graph.
474   int node_size_before = graph()->node_size();
475
476   for (NodeDef& node : *subgraph.mutable_node()) {
477     auto* node_in_graph = graph()->add_node();
478     node_in_graph->Swap(&node);
479     TF_RETURN_IF_ERROR(AddUniqueNode(node_in_graph));
480   }
481
482   // TODO(ezhulenev, lyandy): Right now AddAndDedupFanouts do not check that
483   // fanins actually exists in the graph, and there is already TODO for that.
484
485   for (int i = node_size_before; i < graph()->node_size(); ++i) {
486     NodeDef* node = graph()->mutable_node(i);
487     AddAndDedupFanouts(node);
488   }
489
490   return Status::OK();
491 }
492
493 Status MutableGraphView::UpdateNode(
494     absl::string_view node_name, absl::string_view op, absl::string_view device,
495     absl::Span<const std::pair<string, AttrValue>> attrs) {
496   auto error_status = [node_name, op, device, attrs](absl::string_view msg) {
497     std::vector<string> attr_strs;
498     attr_strs.reserve(attrs.size());
499     for (const auto& attr : attrs) {
500       string attr_str = absl::Substitute("('$0', $1)", attr.first,
501                                          attr.second.ShortDebugString());
502       attr_strs.push_back(attr_str);
503     }
504     string params =
505         absl::Substitute("node_name='$0', op='$1', device='$2', attrs={$3}",
506                          node_name, op, device, absl::StrJoin(attr_strs, ", "));
507     return MutationError("UpdateNodeOp", params, msg);
508   };
509
510   NodeDef* node = GetNode(node_name);
511   TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
512
513   MutableGraphView::OutputPort control_port(node, Graph::kControlSlot);
514   auto control_fanouts = GetFanout(control_port);
515   if (op == "Switch" && !control_fanouts.empty()) {
516     return error_status(
517         "can't change node op to Switch when node drives a control dependency "
518         "(alternatively, we could add the identity node needed, but it seems "
519         "like an unlikely event and probably a mistake)");
```

```
520     }
521
522     if (node->device() != device) {
523       node->set_device(string(device));
524     }
525     node->mutable_attr()->clear();
526     for (const auto& attr : attrs) {
527       (*node->mutable_attr())[attr.first] = attr.second;
528     }
529
530     if (node->op() == op) {
531       return Status::OK();
532     }
533
534     node->set_op(string(op));
535
536     if (CanDedupControlWithRegularInput(*this, *node)) {
537       for (const auto& control_fanout : control_fanouts) {
538         if (HasRegularFaninNode(*this, *control_fanout.node, node->name())) {
539           RemoveControllingFaninInternal(control_fanout.node, node);
540         }
541       }
542     }
543
544     return Status::OK();
545   }
546
547   Status MutableGraphView::UpdateNodeName(absl::string_view from_node_name,
548                                           absl::string_view to_node_name,
549                                           bool update_fanouts) {
550     auto error_status = [from_node_name, to_node_name,
551                          update_fanouts](absl::string_view msg) {
552       string params = absl::Substitute(
553           "from_node_name='$0', to_node_name='$1', update_fanouts=$2",
554           from_node_name, to_node_name, update_fanouts);
555       return MutationError("UpdateNodeName", params, msg);
556     };
557
558     NodeDef* node = GetNode(from_node_name);
559     TF_RETURN_IF_ERROR(CheckNodeExists(from_node_name, node, error_status));
560
561     if (node->name() == to_node_name) {
562       return Status::OK();
563     }
564     if (HasNode(to_node_name)) {
565       return error_status(
566           "can't update node name because new node name is in use");
567     }
568     auto max_output_port = max_regular_output_port().find(node);
```

```
569   const bool has_max_output_port =
570       max_output_port != max_regular_output_port().end();
571   auto control_fanouts = fanouts().find({node, Graph::kControlSlot});
572
573   if (update_fanouts) {
574     SwapControlledFanoutInputs(*this, control_fanouts, to_node_name);
575     if (has_max_output_port) {
576       SwapRegularFanoutInputs(&fanouts(), node, to_node_name,
577                               max_output_port->second);
578     }
579   } else if (has_max_output_port ||
580              HasFanoutValue(fanouts(), control_fanouts)) {
581     return error_status("can't update node name because node has fanouts");
582   }
583
584   nodes().erase(node->name());
585   node->set_name(string(to_node_name));
586   nodes().emplace(node->name(), node);
587   return Status::OK();
588 }
589
590 Status MutableGraphView::SwapNodeNames(absl::string_view from_node_name,
591                                        absl::string_view to_node_name,
592                                        bool update_fanouts) {
593   auto error_status = [from_node_name, to_node_name,
594                        update_fanouts](absl::string_view msg) {
595     string params = absl::Substitute(
596         "from_node_name='$0', to_node_name='$1', update_fanouts=$2",
597         from_node_name, to_node_name, update_fanouts);
598     return MutationError("SwapNodeNames", params, msg);
599   };
600
601   NodeDef* from_node = GetNode(from_node_name);
602   TF_RETURN_IF_ERROR(CheckNodeExists(from_node_name, from_node, error_status));
603   if (from_node_name == to_node_name) {
604     return Status::OK();
605   }
606   NodeDef* to_node = GetNode(to_node_name);
607   TF_RETURN_IF_ERROR(CheckNodeExists(to_node_name, to_node, error_status));
608
609   auto swap_names = [this, from_node, to_node]() {
610     nodes().erase(from_node->name());
611     nodes().erase(to_node->name());
612     std::swap(*from_node->mutable_name(), *to_node->mutable_name());
613     nodes().emplace(from_node->name(), from_node);
614     nodes().emplace(to_node->name(), to_node);
615   };
616
617   if (update_fanouts) {
```

```
618       SwapFanoutInputs(*this, &fanouts(), &max_regular_output_port(), from_node,
619                        to_node);
620       swap_names();
621       return Status::OK();
622     }
623
624     bool from_is_switch = IsSwitch(*from_node);
625     MutableGraphView::OutputPort to_control(to_node, Graph::kControlSlot);
626     auto to_control_fanouts = fanouts().find(to_control);
627     if (from_is_switch && HasFanoutValue(fanouts(), to_control_fanouts)) {
628       return error_status(SwapNodeNamesSwitchControlErrorMsg(from_node_name));
629     }
630
631     bool to_is_switch = IsSwitch(*to_node);
632     MutableGraphView::OutputPort from_control(from_node, Graph::kControlSlot);
633     auto from_control_fanouts = fanouts().find(from_control);
634     if (to_is_switch && HasFanoutValue(fanouts(), from_control_fanouts)) {
635       return error_status(SwapNodeNamesSwitchControlErrorMsg(to_node_name));
636     }
637
638     // Swap node names.
639     swap_names();
640
641     // Swap controlling fanouts.
642     //
643     // Note: To and from control fanout iterators are still valid as no mutations
644     // has been performed on fanouts().
645     SwapFanoutsMapValues(&fanouts(), from_control, from_control_fanouts,
646                          to_control, to_control_fanouts);
647
648     // Swap regular fanouts.
649     SwapRegularFanoutsAndMaxPortValues(&fanouts(), &max_regular_output_port(),
650                                        from_node, to_node);
651
652     // Update fanins to remove self loops.
653     auto update_fanins = [this](NodeDef* node, absl::string_view old_node_name) {
654       for (int i = 0; i < node->input_size(); ++i) {
655         TensorId tensor_id = ParseTensorName(node->input(i));
656         if (tensor_id.node() == node->name()) {
657           const int idx = tensor_id.index();
658           const int node_idx =
659               IsTensorIdControlling(tensor_id) ? Graph::kControlSlot : i;
660
661           MutableGraphView::OutputPort from_fanin(node, idx);
662           absl::flat_hash_set<InputPort>* from_fanouts = &fanouts()[from_fanin];
663           from_fanouts->erase({node, node_idx});
664           UpdateMaxRegularOutputPortForRemovedFanin(from_fanin, *from_fanouts);
665
666           MutableGraphView::OutputPort to_fanin(nodes().at(old_node_name), idx);
```

```
667         fanouts()[to_fanin].insert({node, node_idx});
668         UpdateMaxRegularOutputPortForAddedFanin(to_fanin);
669         node->set_input(i, TensorIdToString({old_node_name, idx}));
670       }
671     }
672   };
673   update_fanins(from_node, to_node->name());
674   update_fanins(to_node, from_node->name());
675
676   // Dedup control dependencies.
677   auto dedup_control_fanouts =
678       [this](NodeDef* node, const FanoutsMap::iterator& control_fanouts) {
679         if (CanDedupControlWithRegularInput(*this, *node) &&
680             control_fanouts != fanouts().end()) {
681           for (auto it = control_fanouts->second.begin();
682                it != control_fanouts->second.end();) {
683             // Advance `it` before invalidation from removal.
684             const auto& control_fanout = *it++;
685             if (HasRegularFaninNode(*this, *control_fanout.node,
686                                     node->name())) {
687               RemoveControllingFaninInternal(control_fanout.node, node);
688             }
689           }
690         }
691       };
692   auto dedup_switch_control = [this, dedup_control_fanouts](NodeDef* node) {
693     OutputPort port;
694     port.node = node;
695     const int max_port =
696         gtl::FindWithDefault(max_regular_output_port(), node, -1);
697     for (int i = 0; i <= max_port; ++i) {
698       port.port_id = i;
699       auto it = fanouts().find(port);
700       if (it == fanouts().end()) {
701         continue;
702       }
703       for (const auto& fanout : it->second) {
704         auto fanout_controls =
705             fanouts().find({fanout.node, Graph::kControlSlot});
706         dedup_control_fanouts(fanout.node, fanout_controls);
707       }
708     }
709   };
710
711   if (!from_is_switch) {
712     if (to_is_switch) {
713       dedup_switch_control(from_node);
714     } else {
715       // Fetch iterator again as the original iterator might have been
```

```
716       // invalidated by container rehash triggered due to mutations.
717       auto from_control_fanouts = fanouts().find(from_control);
718       dedup_control_fanouts(from_node, from_control_fanouts);
719     }
720   }
721   if (!to_is_switch) {
722     if (from_is_switch) {
723       dedup_switch_control(to_node);
724     } else {
725       // Fetch iterator again as the original iterator might have been
726       // invalidated by container rehash triggered due to mutations.
727       auto to_control_fanouts = fanouts().find(to_control);
728       dedup_control_fanouts(to_node, to_control_fanouts);
729     }
730   }
731
732   return Status::OK();
733 }
734
735 Status MutableGraphView::UpdateFanouts(absl::string_view from_node_name,
736                                        absl::string_view to_node_name) {
737   NodeDef* from_node = GetNode(from_node_name);
738   TF_RETURN_IF_ERROR(
739       CheckNodeExists(from_node_name, from_node,
740                       UpdateFanoutsError(from_node_name, to_node_name)));
741   NodeDef* to_node = GetNode(to_node_name);
742   TF_RETURN_IF_ERROR(CheckNodeExists(
743       to_node_name, to_node, UpdateFanoutsError(from_node_name, to_node_name)));
744
745   return UpdateFanoutsInternal(from_node, to_node);
746 }
747
748 Status MutableGraphView::UpdateFanoutsInternal(NodeDef* from_node,
749                                                NodeDef* to_node) {
750   VLOG(2) << absl::Substitute("Update fanouts from '$0' to '$1'.",
751                               from_node->name(), to_node->name());
752   if (from_node == to_node) {
753     return Status::OK();
754   }
755
756   // Update internal state with the new output_port->input_port edge.
757   const auto add_edge = [this](const OutputPort& output_port,
758                                const InputPort& input_port) {
759     fanouts()[output_port].insert(input_port);
760   };
761
762   // Remove invalidated edge from the internal state.
763   const auto remove_edge = [this](const OutputPort& output_port,
764                                   const InputPort& input_port) {
```

```
765       fanouts()[output_port].erase(input_port);
766     };
767
768     // For the control fanouts we do not know the input index in a NodeDef,
769     // so we have to traverse all control inputs.
770
771     auto control_fanouts =
772         GetFanout(GraphView::OutputPort(from_node, Graph::kControlSlot));
773
774     bool to_node_is_switch = IsSwitch(*to_node);
775     for (const InputPort& control_port : control_fanouts) {
776       // Node can't be control dependency of itself.
777       if (control_port.node == to_node) continue;
778
779       // Can't add Switch node as a control dependency.
780       if (to_node_is_switch) {
781         // Trying to add a Switch as a control dependency, which if allowed will
782         // make the graph invalid.
783         return UpdateFanoutsError(from_node->name(), to_node->name())(
784             absl::Substitute("can't update fanouts to node '$0' as it will "
785                              "become a Switch control dependency",
786                              to_node->name()));
787       }
788
789       NodeDef* node = control_port.node;
790       RemoveControllingFaninInternal(node, from_node);
791       AddFaninInternal(node, {to_node, Graph::kControlSlot});
792     }
793
794     // First we update regular fanouts. For the regular fanouts
795     // `input_port:port_id` is the input index in NodeDef.
796
797     auto regular_edges =
798         GetFanoutEdges(*from_node, /*include_controlled_edges=*/false);
799
800     // Maximum index of the `from_node` output tensor that is still used as an
801     // input to some other node.
802     int keep_max_regular_output_port = -1;
803
804     for (const Edge& edge : regular_edges) {
805       const OutputPort output_port = edge.src;
806       const InputPort input_port = edge.dst;
807
808       // If the `to_node` reads from the `from_node`, skip this edge (see
809       // AddAndUpdateFanoutsWithoutSelfLoops test for an example).
810       if (input_port.node == to_node) {
811         keep_max_regular_output_port =
812             std::max(keep_max_regular_output_port, output_port.port_id);
813         continue;
```

```
814       }
815
816       // Update input at destination node.
817       input_port.node->set_input(
818           input_port.port_id,
819           TensorIdToString({to_node->name(), output_port.port_id}));
820
821       // Remove old edge between the `from_node` and the fanout node.
822       remove_edge(output_port, input_port);
823       // Add an edge between the `to_node` and new fanout node.
824       add_edge(OutputPort(to_node, output_port.port_id), input_port);
825       // Dedup control dependency.
826       if (CanDedupControlWithRegularInput(*this, *to_node)) {
827         RemoveControllingFaninInternal(input_port.node, to_node);
828       }
829     }
830
831     // Because we update all regular fanouts of `from_node`, we can just copy
832     // the value `num_regular_outputs`.
833     max_regular_output_port()[to_node] = max_regular_output_port()[from_node];
834
835     // Check if all fanouts were updated to read from the `to_node`.
836     if (keep_max_regular_output_port >= 0) {
837       max_regular_output_port()[from_node] = keep_max_regular_output_port;
838     } else {
839       max_regular_output_port().erase(from_node);
840     }
841
842     return Status::OK();
843   }
844
845   bool MutableGraphView::AddFaninInternal(NodeDef* node,
846                                           const OutputPort& fanin) {
847     int num_regular_fanins =
848         NumFanins(*node, /*include_controlling_nodes=*/false);
849     bool input_is_control = IsOutputPortControlling(fanin);
850     bool can_dedup_control_with_regular_input =
851         CanDedupControlWithRegularInput(*this, *fanin.node);
852     // Don't add duplicate control dependencies.
853     if (input_is_control) {
854       const int start =
855           can_dedup_control_with_regular_input ? 0 : num_regular_fanins;
856       for (int i = start; i < node->input_size(); ++i) {
857         if (ParseTensorName(node->input(i)).node() == fanin.node->name()) {
858           return false;
859         }
860       }
861     }
862
```

```
  InputPort input;
  input.node = node;
  input.port_id = input_is_control ? Graph::kControlSlot : num_regular_fanins;

  node->add_input(TensorIdToString({fanin.node->name(), fanin.port_id}));
  if (!input_is_control) {
    const int last_node_input = node->input_size() - 1;
    // If there are control dependencies in node, move newly inserted fanin to
    // be before such control dependencies.
    if (num_regular_fanins < last_node_input) {
      node->mutable_input()->SwapElements(last_node_input, num_regular_fanins);
    }
  }

  fanouts()[fanin].insert(input);
  if (max_regular_output_port()[fanin.node] < fanin.port_id) {
    max_regular_output_port()[fanin.node] = fanin.port_id;
  }

  // Update max input port and dedup control dependencies.
  if (!input_is_control) {
    max_regular_input_port()[node] = num_regular_fanins;
    if (can_dedup_control_with_regular_input) {
      RemoveControllingFaninInternal(node, fanin.node);
    }
  }

  return true;
}

Status MutableGraphView::AddRegularFanin(absl::string_view node_name,
                                         const TensorId& fanin) {
  auto error_status = [node_name, fanin](absl::string_view msg) {
    string params = absl::Substitute("node_name='$0', fanin='$1'", node_name,
                                     fanin.ToString());
    return MutationError("AddRegularFanin", params, msg);
  };

  TF_RETURN_IF_ERROR(CheckFaninIsRegular(fanin, error_status));
  TF_RETURN_IF_ERROR(CheckAddingFaninToSelf(node_name, fanin, error_status));
  NodeDef* node = GetNode(node_name);
  TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
  NodeDef* fanin_node = GetNode(fanin.node());
  TF_RETURN_IF_ERROR(CheckNodeExists(fanin.node(), fanin_node, error_status));

  AddFaninInternal(node, {fanin_node, fanin.index()});
  return Status::OK();
}

```

```
912 Status MutableGraphView::AddRegularFaninByPort(absl::string_view node_name,
913                                                int port,
914                                                const TensorId& fanin) {
915   auto error_status = [node_name, port, fanin](absl::string_view msg) {
916     string params = absl::Substitute("node_name='$0', port=$1, fanin='$2'",
917                                      node_name, port, fanin.ToString());
918     return MutationError("AddRegularFaninByPort", params, msg);
919   };
920
921   TF_RETURN_IF_ERROR(CheckFaninIsRegular(fanin, error_status));
922   TF_RETURN_IF_ERROR(CheckAddingFaninToSelf(node_name, fanin, error_status));
923   NodeDef* node = GetNode(node_name);
924   TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
925   const int num_regular_fanins =
926       NumFanins(*node, /*include_controlling_nodes=*/false);
927   TF_RETURN_IF_ERROR(
928       CheckPortRange(port, /*min=*/0, num_regular_fanins, error_status));
929   NodeDef* fanin_node = GetNode(fanin.node());
930   TF_RETURN_IF_ERROR(CheckNodeExists(fanin.node(), fanin_node, error_status));
931
932   const int last_node_input = node->input_size();
933   node->add_input(TensorIdToString(fanin));
934   node->mutable_input()->SwapElements(num_regular_fanins, last_node_input);
935   for (int i = num_regular_fanins - 1; i >= port; --i) {
936     TensorId tensor_id = ParseTensorName(node->input(i));
937     OutputPort fanin_port(nodes()[tensor_id.node()], tensor_id.index());
938     absl::flat_hash_set<InputPort>* fanouts_set = &fanouts()[fanin_port];
939     fanouts_set->erase({node, i});
940     fanouts_set->insert({node, i + 1});
941     node->mutable_input()->SwapElements(i, i + 1);
942   }
943
944   OutputPort fanin_port(fanin_node, fanin.index());
945   fanouts()[fanin_port].insert({node, port});
946   UpdateMaxRegularOutputPortForAddedFanin(fanin_port);
947
948   max_regular_input_port()[node] = num_regular_fanins;
949   if (CanDedupControlWithRegularInput(*this, *fanin_node)) {
950     RemoveControllingFaninInternal(node, fanin_node);
951   }
952
953   return Status::OK();
954 }
955
956 NodeDef* MutableGraphView::GetControllingFaninToAdd(absl::string_view node_name,
957                                                     const OutputPort& fanin,
958                                                     string* error_msg) {
959   if (!IsSwitch(*fanin.node)) {
960     return fanin.node;
```

```
961     } else {
962       if (IsOutputPortControlling(fanin)) {
963         // Can't add a Switch node control dependency.
964         TensorId tensor_id(fanin.node->name(), fanin.port_id);
965         *error_msg = absl::Substitute(
966             "can't add fanin '$0' as it will become a Switch control dependency",
967             tensor_id.ToString());
968         return nullptr;
969       }
970       // We can't anchor control dependencies directly on the switch node: unlike
971       // other nodes only one of the outputs of the switch node will be generated
972       // when the switch node is executed, and we need to make sure the control
973       // dependency is only triggered when the corresponding output is triggered.
974       // We start by looking for an identity node connected to the output of the
975       // switch node, and use it to anchor the control dependency.
976       for (const auto& fanout : GetFanout(fanin)) {
977         if (IsIdentity(*fanout.node) || IsIdentityNSingleInput(*fanout.node)) {
978           if (fanout.node->name() == node_name) {
979             *error_msg =
980                 absl::Substitute("can't add found fanin '$0' to self",
981                                  AsControlDependency(fanout.node->name()));
982             return nullptr;
983           }
984           return fanout.node;
985         }
986       }
987
988       // No node found, check if node to be created is itself.
989       if (GeneratedNameForIdentityConsumingSwitch(fanin) == node_name) {
990         *error_msg = absl::Substitute("can't add generated fanin '$0' to self",
991                                       AsControlDependency(string(node_name)));
992       }
993     }
994     return nullptr;
995   }
996
997   NodeDef* MutableGraphView::GetOrCreateIdentityConsumingSwitch(
998       const OutputPort& fanin) {
999     // We haven't found an existing node where we can anchor the control
1000    // dependency: add a new identity node.
1001    string identity_name = GeneratedNameForIdentityConsumingSwitch(fanin);
1002    NodeDef* identity_node = GetNode(identity_name);
1003    if (identity_node == nullptr) {
1004      NodeDef new_node;
1005      new_node.set_name(identity_name);
1006      new_node.set_op("Identity");
1007      new_node.set_device(fanin.node->device());
1008      (*new_node.mutable_attr())["T"].set_type(fanin.node->attr().at("T").type());
1009      new_node.add_input(TensorIdToString({fanin.node->name(), fanin.port_id}));
```

```
1010      identity_node = AddNode(std::move(new_node));
1011    }
1012    return identity_node;
1013  }
1014
1015  Status MutableGraphView::AddControllingFanin(absl::string_view node_name,
1016                                               const TensorId& fanin) {
1017    auto error_status = [node_name, fanin](absl::string_view msg) {
1018      string params = absl::Substitute("node_name='$0', fanin='$1'", node_name,
1019                                       fanin.ToString());
1020      return MutationError("AddControllingFanin", params, msg);
1021    };
1022
1023    TF_RETURN_IF_ERROR(CheckFaninIsValid(fanin, error_status));
1024    TF_RETURN_IF_ERROR(CheckAddingFaninToSelf(node_name, fanin, error_status));
1025    NodeDef* node = GetNode(node_name);
1026    TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
1027    NodeDef* fanin_node = GetNode(fanin.node());
1028    TF_RETURN_IF_ERROR(CheckNodeExists(fanin.node(), fanin_node, error_status));
1029
1030    OutputPort fanin_port(fanin_node, fanin.index());
1031
1032    string error_msg = "";
1033    NodeDef* control_node = GetControllingFaninToAdd(
1034        node_name, {fanin_node, fanin.index()}, &error_msg);
1035    if (!error_msg.empty()) {
1036      return error_status(error_msg);
1037    }
1038    if (control_node == nullptr) {
1039      control_node = GetOrCreateIdentityConsumingSwitch(fanin_port);
1040    }
1041    AddFaninInternal(node, {control_node, Graph::kControlSlot});
1042
1043    return Status::OK();
1044  }
1045
1046  bool MutableGraphView::RemoveRegularFaninInternal(NodeDef* node,
1047                                                    const OutputPort& fanin) {
1048    auto remove_input = [this, node](const OutputPort& fanin_port,
1049                                     int node_input_port, bool update_max_port) {
1050      InputPort input(node, node_input_port);
1051
1052      absl::flat_hash_set<InputPort>* fanouts_set = &fanouts()[fanin_port];
1053      fanouts_set->erase(input);
1054      if (update_max_port) {
1055        UpdateMaxRegularOutputPortForRemovedFanin(fanin_port, *fanouts_set);
1056      }
1057      return fanouts_set;
1058    };
```

```
1059
1060    auto mutable_inputs = node->mutable_input();
1061    bool modified = false;
1062    const int num_regular_fanins =
1063        NumFanins(*node, /*include_controlling_nodes=*/false);
1064    int i;
1065    int curr_pos = 0;
1066    for (i = 0; i < num_regular_fanins; ++i) {
1067      TensorId tensor_id = ParseTensorName(node->input(i));
1068      if (tensor_id.node() == fanin.node->name() &&
1069          tensor_id.index() == fanin.port_id) {
1070        remove_input(fanin, i, /*update_max_port=*/true);
1071        modified = true;
1072      } else if (modified) {
1073        // Regular inputs will need to have their ports updated.
1074        OutputPort fanin_port(nodes()[tensor_id.node()], tensor_id.index());
1075        auto fanouts_set = remove_input(fanin_port, i, /*update_max_port=*/false);
1076        fanouts_set->insert({node, curr_pos});
1077        // Shift inputs to be retained.
1078        mutable_inputs->SwapElements(i, curr_pos);
1079        ++curr_pos;
1080      } else {
1081        // Skip inputs to be retained until first modification.
1082        ++curr_pos;
1083      }
1084    }
1085
1086    if (modified) {
1087      const int last_regular_input_port = curr_pos - 1;
1088      if (last_regular_input_port < 0) {
1089        max_regular_input_port().erase(node);
1090      } else {
1091        max_regular_input_port()[node] = last_regular_input_port;
1092      }
1093      if (curr_pos < i) {
1094        // Remove fanins from node inputs.
1095        mutable_inputs->DeleteSubrange(curr_pos, i - curr_pos);
1096      }
1097    }
1098
1099    return modified;
1100  }
1101
1102  Status MutableGraphView::RemoveRegularFanin(absl::string_view node_name,
1103                                              const TensorId& fanin) {
1104    auto error_status = [node_name, fanin](absl::string_view msg) {
1105      string params = absl::Substitute("node_name='$0', fanin='$1'", node_name,
1106                                       fanin.ToString());
1107      return MutationError("RemoveRegularFanin", params, msg);
```

```
1108    };
1109
1110    TF_RETURN_IF_ERROR(CheckFaninIsRegular(fanin, error_status));
1111    TF_RETURN_IF_ERROR(
1112        CheckRemovingFaninFromSelf(node_name, fanin, error_status));
1113    NodeDef* node = GetNode(node_name);
1114    TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
1115    NodeDef* fanin_node = GetNode(fanin.node());
1116    TF_RETURN_IF_ERROR(CheckNodeExists(fanin.node(), fanin_node, error_status));
1117
1118    RemoveRegularFaninInternal(node, {fanin_node, fanin.index()});
1119    return Status::OK();
1120  }
1121
1122  Status MutableGraphView::RemoveRegularFaninByPort(absl::string_view node_name,
1123                                                    int port) {
1124    auto error_status = [node_name, port](absl::string_view msg) {
1125      string params =
1126          absl::Substitute("node_name='$0', port=$1", node_name, port);
1127      return MutationError("RemoveRegularFaninByPort", params, msg);
1128    };
1129
1130    NodeDef* node = GetNode(node_name);
1131    TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
1132    const int last_regular_fanin_port =
1133        gtl::FindWithDefault(max_regular_input_port(), node, -1);
1134    TF_RETURN_IF_ERROR(
1135        CheckPortRange(port, /*min=*/0, last_regular_fanin_port, error_status));
1136
1137    TensorId tensor_id = ParseTensorName(node->input(port));
1138    OutputPort fanin_port(nodes()[tensor_id.node()], tensor_id.index());
1139    fanouts()[fanin_port].erase({node, port});
1140    auto mutable_inputs = node->mutable_input();
1141    for (int i = port + 1; i <= last_regular_fanin_port; ++i) {
1142      TensorId tensor_id = ParseTensorName(node->input(i));
1143      OutputPort fanin_port(nodes()[tensor_id.node()], tensor_id.index());
1144      absl::flat_hash_set<InputPort>* fanouts_set = &fanouts()[fanin_port];
1145      fanouts_set->erase({node, i});
1146      fanouts_set->insert({node, i - 1});
1147      mutable_inputs->SwapElements(i - 1, i);
1148    }
1149    const int last_node_input = node->input_size() - 1;
1150    if (last_regular_fanin_port < last_node_input) {
1151      mutable_inputs->SwapElements(last_regular_fanin_port, last_node_input);
1152    }
1153    mutable_inputs->RemoveLast();
1154
1155    const int updated_last_regular_input_port = last_regular_fanin_port - 1;
1156    if (updated_last_regular_input_port < 0) {
```

```
1157      max_regular_input_port().erase(node);
1158    } else {
1159      max_regular_input_port()[node] = updated_last_regular_input_port;
1160    }
1161
1162    return Status::OK();
1163  }
1164
1165  bool MutableGraphView::RemoveControllingFaninInternal(NodeDef* node,
1166                                                        NodeDef* fanin_node) {
1167    for (int i = node->input_size() - 1; i >= 0; --i) {
1168      TensorId tensor_id = ParseTensorName(node->input(i));
1169      if (tensor_id.index() > Graph::kControlSlot) {
1170        break;
1171      }
1172      if (tensor_id.node() == fanin_node->name()) {
1173        fanouts()[{fanin_node, Graph::kControlSlot}].erase(
1174            {node, Graph::kControlSlot});
1175        node->mutable_input()->SwapElements(i, node->input_size() - 1);
1176        node->mutable_input()->RemoveLast();
1177        return true;
1178      }
1179    }
1180    return false;
1181  }
1182
1183  Status MutableGraphView::RemoveControllingFanin(
1184      absl::string_view node_name, absl::string_view fanin_node_name) {
1185    auto error_status = [node_name, fanin_node_name](absl::string_view msg) {
1186      string params = absl::Substitute("node_name='$0', fanin_node_name='$1'",
1187                                       node_name, fanin_node_name);
1188      return MutationError("RemoveControllingFanin", params, msg);
1189    };
1190
1191    TF_RETURN_IF_ERROR(CheckRemovingFaninFromSelf(
1192        node_name, {fanin_node_name, Graph::kControlSlot}, error_status));
1193    NodeDef* node = GetNode(node_name);
1194    TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
1195    NodeDef* fanin_node = GetNode(fanin_node_name);
1196    TF_RETURN_IF_ERROR(
1197        CheckNodeExists(fanin_node_name, fanin_node, error_status));
1198
1199    RemoveControllingFaninInternal(node, fanin_node);
1200    return Status::OK();
1201  }
1202
1203  Status MutableGraphView::RemoveAllFanins(absl::string_view node_name,
1204                                           bool keep_controlling_fanins) {
1205    NodeDef* node = GetNode(node_name);
```

```
1206    if (node == nullptr) {
1207      string params =
1208          absl::Substitute("node_name='$0', keep_controlling_fanins=$1",
1209                           node_name, keep_controlling_fanins);
1210      return MutationError("RemoveAllFanins", params,
1211                           NodeMissingErrorMsg(node_name));
1212    }
1213
1214    if (node->input().empty()) {
1215      return Status::OK();
1216    }
1217
1218    const int num_regular_fanins =
1219        NumFanins(*node, /*include_controlling_nodes=*/false);
1220    RemoveFaninsInternal(node, keep_controlling_fanins);
1221    if (keep_controlling_fanins) {
1222      if (num_regular_fanins == 0) {
1223        return Status::OK();
1224      } else if (num_regular_fanins < node->input_size()) {
1225        node->mutable_input()->DeleteSubrange(0, num_regular_fanins);
1226      } else {
1227        node->clear_input();
1228      }
1229    } else {
1230      node->clear_input();
1231    }
1232    return Status::OK();
1233  }
1234
1235  Status MutableGraphView::UpdateFanin(absl::string_view node_name,
1236                                       const TensorId& from_fanin,
1237                                       const TensorId& to_fanin) {
1238    auto error_status = [node_name, from_fanin, to_fanin](absl::string_view msg) {
1239      string params =
1240          absl::Substitute("node_name='$0', from_fanin='$1', to_fanin='$2'",
1241                           node_name, from_fanin.ToString(), to_fanin.ToString());
1242      return MutationError("UpdateFanin", params, msg);
1243    };
1244
1245    TF_RETURN_IF_ERROR(CheckFaninIsValid(from_fanin, error_status));
1246    TF_RETURN_IF_ERROR(CheckFaninIsValid(to_fanin, error_status));
1247    NodeDef* node = GetNode(node_name);
1248    TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
1249    NodeDef* from_fanin_node = GetNode(from_fanin.node());
1250    TF_RETURN_IF_ERROR(
1251        CheckNodeExists(from_fanin.node(), from_fanin_node, error_status));
1252    NodeDef* to_fanin_node = GetNode(to_fanin.node());
1253    TF_RETURN_IF_ERROR(
1254        CheckNodeExists(to_fanin.node(), to_fanin_node, error_status));
```

```
1255
1256    // When replacing a non control dependency fanin with a control dependency, or
1257    // vice versa, remove and add, so ports can be updated properly in fanout(s).
1258    bool to_fanin_is_control = IsTensorIdControlling(to_fanin);
1259    if (to_fanin_is_control && IsSwitch(*to_fanin_node)) {
1260      // Can't add Switch node as a control dependency.
1261      return error_status(
1262          absl::Substitute("can't update to fanin '$0' as it will become a "
1263                           "Switch control dependency",
1264                           to_fanin.ToString()));
1265    }
1266    if (node_name == from_fanin.node() || node_name == to_fanin.node()) {
1267      return error_status("can't update fanin to or from self");
1268    }
1269
1270    if (from_fanin == to_fanin) {
1271      return Status::OK();
1272    }
1273
1274    bool from_fanin_is_control = IsTensorIdControlling(from_fanin);
1275    if (from_fanin_is_control || to_fanin_is_control) {
1276      bool modified = false;
1277      if (from_fanin_is_control) {
1278        modified |= RemoveControllingFaninInternal(node, from_fanin_node);
1279      } else {
1280        modified |= RemoveRegularFaninInternal(
1281            node, {from_fanin_node, from_fanin.index()});
1282      }
1283      if (modified) {
1284        AddFaninInternal(node, {to_fanin_node, to_fanin.index()});
1285      }
1286      return Status::OK();
1287    }
1288
1289    // In place mutation of regular fanins, requires no shifting of ports.
1290    string to_fanin_string = TensorIdToString(to_fanin);
1291    const int num_regular_fanins =
1292        NumFanins(*node, /*include_controlling_nodes=*/false);
1293    bool modified = false;
1294    for (int i = 0; i < num_regular_fanins; ++i) {
1295      if (ParseTensorName(node->input(i)) == from_fanin) {
1296        InputPort input(node, i);
1297
1298        OutputPort from_fanin_port(from_fanin_node, from_fanin.index());
1299        fanouts()[from_fanin_port].erase(input);
1300
1301        OutputPort to_fanin_port(to_fanin_node, to_fanin.index());
1302        fanouts()[to_fanin_port].insert(input);
1303
```

```
1304        node->set_input(i, to_fanin_string);
1305        modified = true;
1306      }
1307    }
1308
1309    // Dedup control dependencies and update max regular output ports.
1310    if (modified) {
1311      OutputPort from_fanin_port(from_fanin_node, from_fanin.index());
1312      UpdateMaxRegularOutputPortForRemovedFanin(
1313          {from_fanin_node, from_fanin.index()}, fanouts()[from_fanin_port]);
1314      if (max_regular_output_port()[to_fanin_node] < to_fanin.index()) {
1315        max_regular_output_port()[to_fanin_node] = to_fanin.index();
1316      }
1317      if (CanDedupControlWithRegularInput(*this, *to_fanin_node)) {
1318        RemoveControllingFaninInternal(node, to_fanin_node);
1319      }
1320    }
1321
1322    return Status::OK();
1323  }
1324
1325  Status MutableGraphView::UpdateRegularFaninByPort(absl::string_view node_name,
1326                                                    int port,
1327                                                    const TensorId& fanin) {
1328    auto error_status = [node_name, port, fanin](absl::string_view msg) {
1329      string params = absl::Substitute("node_name='$0', port=$1, fanin='$2'",
1330                                       node_name, port, fanin.ToString());
1331      return MutationError("UpdateRegularFaninByPort", params, msg);
1332    };
1333
1334    TF_RETURN_IF_ERROR(CheckFaninIsRegular(fanin, error_status));
1335    TF_RETURN_IF_ERROR(CheckAddingFaninToSelf(node_name, fanin, error_status));
1336    NodeDef* node = GetNode(node_name);
1337    TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
1338    const int last_regular_fanin_port =
1339        gtl::FindWithDefault(max_regular_input_port(), node, -1);
1340    TF_RETURN_IF_ERROR(
1341        CheckPortRange(port, /*min=*/0, last_regular_fanin_port, error_status));
1342    NodeDef* fanin_node = GetNode(fanin.node());
1343    TF_RETURN_IF_ERROR(CheckNodeExists(fanin.node(), fanin_node, error_status));
1344
1345    TensorId tensor_id = ParseTensorName(node->input(port));
1346    if (tensor_id == fanin) {
1347      return Status::OK();
1348    }
1349
1350    InputPort input(node, port);
1351    OutputPort from_fanin_port(nodes()[tensor_id.node()], tensor_id.index());
1352    absl::flat_hash_set<InputPort>* from_fanouts = &fanouts()[from_fanin_port];
```

```
1353    from_fanouts->erase(input);
1354    UpdateMaxRegularOutputPortForRemovedFanin(from_fanin_port, *from_fanouts);
1355
1356    OutputPort to_fanin_port(fanin_node, fanin.index());
1357    fanouts()[to_fanin_port].insert(input);
1358    UpdateMaxRegularOutputPortForAddedFanin(to_fanin_port);
1359
1360    node->set_input(port, TensorIdToString(fanin));
1361
1362    if (CanDedupControlWithRegularInput(*this, *fanin_node)) {
1363      RemoveControllingFaninInternal(node, fanin_node);
1364    }
1365
1366    return Status::OK();
1367  }
1368
1369  Status MutableGraphView::SwapRegularFaninsByPorts(absl::string_view node_name,
1370                                                    int from_port, int to_port) {
1371    auto error_status = [node_name, from_port, to_port](absl::string_view msg) {
1372      string params = absl::Substitute("node_name='$0', from_port=$1, to_port=$2",
1373                                       node_name, from_port, to_port);
1374      return MutationError("SwapRegularFaninsByPorts", params, msg);
1375    };
1376
1377    NodeDef* node = GetNode(node_name);
1378    TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));
1379    const int last_regular_fanin_port =
1380        gtl::FindWithDefault(max_regular_input_port(), node, -1);
1381    TF_RETURN_IF_ERROR(CheckPortRange(from_port, /*min=*/0,
1382                                      last_regular_fanin_port, error_status));
1383    TF_RETURN_IF_ERROR(CheckPortRange(to_port, /*min=*/0, last_regular_fanin_port,
1384                                      error_status));
1385
1386    if (from_port == to_port) {
1387      return Status::OK();
1388    }
1389    TensorId from_fanin = ParseTensorName(node->input(from_port));
1390    TensorId to_fanin = ParseTensorName(node->input(to_port));
1391    if (from_fanin == to_fanin) {
1392      return Status::OK();
1393    }
1394
1395    InputPort from_input(node, from_port);
1396    InputPort to_input(node, to_port);
1397    NodeDef* from_fanin_node = GetNode(from_fanin.node());
1398    absl::flat_hash_set<InputPort>* from_fanouts =
1399        &fanouts()[{from_fanin_node, from_fanin.index()}];
1400    from_fanouts->erase(from_input);
1401    from_fanouts->insert(to_input);
```

```
  NodeDef* to_fanin_node = GetNode(to_fanin.node());
  absl::flat_hash_set<InputPort>* to_fanouts =
      &fanouts()[{to_fanin_node, to_fanin.index()}];
  to_fanouts->erase(to_input);
  to_fanouts->insert(from_input);

  node->mutable_input()->SwapElements(from_port, to_port);

  return Status::OK();
}

Status MutableGraphView::UpdateAllRegularFaninsToControlling(
    absl::string_view node_name) {
  auto error_status = [node_name](absl::string_view msg) {
    string params = absl::Substitute("node_name='$0'", node_name);
    return MutationError("UpdateAllRegularFaninsToControlling", params, msg);
  };

  NodeDef* node = GetNode(node_name);
  TF_RETURN_IF_ERROR(CheckNodeExists(node_name, node, error_status));

  const int num_regular_fanins =
      NumFanins(*node, /*include_controlling_nodes=*/false);
  std::vector<OutputPort> regular_fanins;
  regular_fanins.reserve(num_regular_fanins);
  std::vector<NodeDef*> controlling_fanins;
  controlling_fanins.reserve(num_regular_fanins);

  // Get all regular fanins and derive controlling fanins.
  for (int i = 0; i < num_regular_fanins; ++i) {
    TensorId tensor_id = ParseTensorName(node->input(i));
    OutputPort fanin_port(nodes()[tensor_id.node()], tensor_id.index());

    string error_msg = "";
    NodeDef* control_node =
        GetControllingFaninToAdd(node_name, fanin_port, &error_msg);
    if (!error_msg.empty()) {
      return error_status(error_msg);
    }

    regular_fanins.push_back(fanin_port);
    controlling_fanins.push_back(control_node);
  }

  // Replace regular fanins with controlling fanins and dedup.
  int pos = 0;
  InputPort input_port(node, Graph::kControlSlot);
  absl::flat_hash_set<absl::string_view> controls;
  for (int i = 0; i < num_regular_fanins; ++i) {
```

```
1451      OutputPort fanin_port = regular_fanins[i];
1452      NodeDef* control = controlling_fanins[i];
1453      if (control == nullptr) {
1454        control = GetOrCreateIdentityConsumingSwitch(fanin_port);
1455      }
1456      fanouts()[fanin_port].erase({node, i});
1457      if (controls.contains(control->name())) {
1458        continue;
1459      }
1460      controls.insert(control->name());
1461      node->set_input(pos, AsControlDependency(control->name()));
1462      fanouts()[{control, Graph::kControlSlot}].insert(input_port);
1463      ++pos;
1464    }
1465
1466    // Shift existing controlling fanins and dedup.
1467    for (int i = num_regular_fanins; i < node->input_size(); ++i) {
1468      TensorId tensor_id = ParseTensorName(node->input(i));
1469      if (controls.contains(tensor_id.node())) {
1470        continue;
1471      }
1472      controls.insert(tensor_id.node());
1473      node->mutable_input()->SwapElements(pos, i);
1474      ++pos;
1475    }
1476
1477    // Remove duplicate controls and leftover regular fanins.
1478    node->mutable_input()->DeleteSubrange(pos, node->input_size() - pos);
1479    max_regular_input_port().erase(node);
1480
1481    return Status::OK();
1482  }
1483
1484  Status MutableGraphView::CheckNodesCanBeDeleted(
1485      const absl::flat_hash_set<string>& nodes_to_delete) {
1486    std::vector<string> missing_nodes;
1487    std::vector<string> nodes_with_fanouts;
1488    for (const string& node_name_to_delete : nodes_to_delete) {
1489      NodeDef* node = GetNode(node_name_to_delete);
1490      if (node == nullptr) {
1491        // Can't delete missing node.
1492        missing_nodes.push_back(node_name_to_delete);
1493        continue;
1494      }
1495      const int max_port = gtl::FindWithDefault(max_regular_output_port(), node,
1496                                                Graph::kControlSlot);
1497      for (int i = Graph::kControlSlot; i <= max_port; ++i) {
1498        auto it = fanouts().find({node, i});
1499        bool has_retained_fanout = false;
```

```
1500        if (it != fanouts().end()) {
1501          for (const auto& fanout : it->second) {
1502            // Check if fanouts are of nodes to be deleted, and if so, they can be
1503            // ignored, as they will be removed also.
1504            if (!nodes_to_delete.contains(fanout.node->name())) {
1505              // Removing node will leave graph in an invalid state.
1506              has_retained_fanout = true;
1507              break;
1508            }
1509          }
1510        }
1511        if (has_retained_fanout) {
1512          nodes_with_fanouts.push_back(node_name_to_delete);
1513          break;
1514        }
1515      }
1516    }
1517
1518    // Error message can get quite long, so we only show the first 5 node names.
1519    auto sort_and_sample = [](std::vector<string>* s) {
1520      constexpr int kMaxNodeNames = 5;
1521      std::sort(s->begin(), s->end());
1522      if (s->size() > kMaxNodeNames) {
1523        return absl::StrCat(
1524            absl::StrJoin(s->begin(), s->begin() + kMaxNodeNames, ", "), ", ...");
1525      }
1526      return absl::StrJoin(*s, ", ");
1527    };
1528
1529    if (!missing_nodes.empty()) {
1530      VLOG(2) << absl::Substitute("Attempting to delete missing node(s) [$0].",
1531                                  sort_and_sample(&missing_nodes));
1532    }
1533    if (!nodes_with_fanouts.empty()) {
1534      std::vector<string> input_node_names(nodes_to_delete.begin(),
1535                                           nodes_to_delete.end());
1536      string params = absl::Substitute("nodes_to_delete={$0}",
1537                                       sort_and_sample(&input_node_names));
1538      string error_msg =
1539          absl::Substitute("can't delete node(s) with retained fanouts(s) [$0]",
1540                           sort_and_sample(&nodes_with_fanouts));
1541      return MutationError("DeleteNodes", params, error_msg);
1542    }
1543
1544    return Status::OK();
1545  }
1546
1547  Status MutableGraphView::DeleteNodes(
1548      const absl::flat_hash_set<string>& nodes_to_delete) {
```

```
1549    TF_RETURN_IF_ERROR(CheckNodesCanBeDeleted(nodes_to_delete));
1550
1551    // Find nodes in internal state and delete.
1552    for (const string& node_name_to_delete : nodes_to_delete) {
1553      NodeDef* node = GetNode(node_name_to_delete);
1554      if (node != nullptr) {
1555        RemoveFaninsInternal(node, /*keep_controlling_fanins=*/false);
1556        RemoveFanoutsInternal(node);
1557      }
1558    }
1559    for (const string& node_name_to_delete : nodes_to_delete) {
1560      nodes().erase(node_name_to_delete);
1561    }
1562
1563    // Find nodes in graph and delete by partitioning into nodes to retain and
1564    // nodes to delete based on input set of nodes to delete by name.
1565    // TODO(lyandy): Use a node name->idx hashmap if this is a performance
1566    // bottleneck.
1567    int pos = 0;
1568    const int last_idx = graph()->node_size() - 1;
1569    int last_pos = last_idx;
1570    while (pos <= last_pos) {
1571      if (nodes_to_delete.contains(graph()->node(pos).name())) {
1572        graph()->mutable_node()->SwapElements(pos, last_pos);
1573        --last_pos;
1574      } else {
1575        ++pos;
1576      }
1577    }
1578    if (last_pos < last_idx) {
1579      graph()->mutable_node()->DeleteSubrange(last_pos + 1, last_idx - last_pos);
1580    }
1581
1582    return Status::OK();
1583  }
1584
1585  void MutableGraphView::RemoveFaninsInternal(NodeDef* deleted_node,
1586                                              bool keep_controlling_fanins) {
1587    for (int i = 0; i < deleted_node->input_size(); ++i) {
1588      TensorId tensor_id = ParseTensorName(deleted_node->input(i));
1589      bool is_control = IsTensorIdControlling(tensor_id);
1590      if (keep_controlling_fanins && is_control) {
1591        break;
1592      }
1593      OutputPort fanin(nodes()[tensor_id.node()], tensor_id.index());
1594
1595      InputPort input;
1596      input.node = deleted_node;
1597      input.port_id = is_control ? Graph::kControlSlot : i;
```

```
1598
1599      auto it = fanouts().find(fanin);
1600      if (it != fanouts().end()) {
1601        absl::flat_hash_set<InputPort>* fanouts_set = &it->second;
1602        fanouts_set->erase(input);
1603        UpdateMaxRegularOutputPortForRemovedFanin(fanin, *fanouts_set);
1604      }
1605    }
1606    max_regular_input_port().erase(deleted_node);
1607  }
1608
1609  void MutableGraphView::RemoveFanoutsInternal(NodeDef* deleted_node) {
1610    const int max_port =
1611        gtl::FindWithDefault(max_regular_output_port(), deleted_node, -1);
1612    for (int i = Graph::kControlSlot; i <= max_port; ++i) {
1613      fanouts().erase({deleted_node, i});
1614    }
1615    max_regular_output_port().erase(deleted_node);
1616  }
1617
1618  }  // end namespace grappler
1619  }  // end namespace tensorflow
```