Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

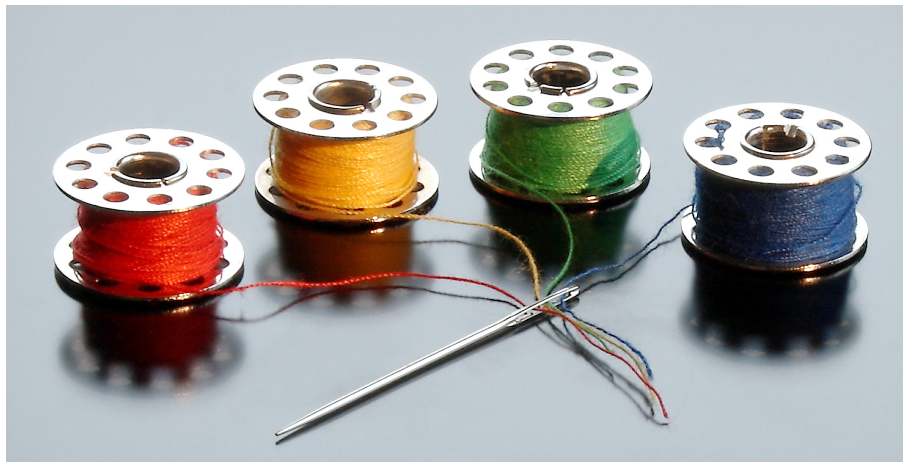
Lehrstuhl Prof. Dr.-Ing. Snelting

Generalized Jump Threading in libFIRM

Masterarbeit von

Joachim Priesner

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr.-Ing. Jörg Henkel
Betreuender Mitarbeiter:	Dipl.-Inform. Andreas Zwinkau

Bearbeitungszeit: 5. Oktober 2016 – 23. Januar 2017

Zusammenfassung/Abstract

Jump Threading (dt. „Sprünge fädeln“) ist eine Compileroptimierung, die statisch vorhersagbare bedingte Sprünge in unbedingte Sprünge umwandelt. Bei der Ausführung kann ein Prozessor bedingte Sprünge zunächst nur heuristisch mit Hilfe der Sprungvorhersage auswerten. Sie stellen daher generell ein Performancehindernis dar.

Die Umwandlung ist insbesondere auch dann möglich, wenn das Sprungziel nur auf einer Teilmenge der zu dem Sprung führenden Ausführungspfade statisch bestimmbar ist. In diesem Fall, der den überwiegenden Teil der durch Jump Threading optimierten Sprünge betrifft, muss die Optimierung Grundblöcke duplizieren, um jene Ausführungspfade zu isolieren.

Verschiedene aktuelle Compiler enthalten sehr unterschiedliche Implementierungen von Jump Threading. In dieser Masterarbeit wird zunächst ein theoretischer Rahmen für Jump Threading vorgestellt. Sodann wird eine allgemeine Fassung eines Jump-Threading-Algorithmus entwickelt, implementiert und in diverser Hinsicht untersucht, insbesondere auf Wechselwirkungen mit anderen Optimierungen wie If Conversion. Anhand von Benchmarks werden schließlich die Optimierungsparameter festgelegt und es wird verifiziert, dass Jump Threading sich auch auf modernen Prozessoren noch lohnt. Auf einem aktuellen Intel-Prozessor verzeichnen wir beim SPECint2000-Benchmark eine Beschleunigung von bis zu 3 %.

Jump Threading is a compiler optimization which transforms statically predictable conditional branches into unconditional branches. During the execution, a processor can initially evaluate conditional branches only heuristically using branch prediction. Therefore, they generally form a performance obstacle.

In particular, the transformation is possible even if the branch target is only statically determinable on a subset of the execution paths leading to the branch. In those cases, which constitute the predominant part of the branches optimized by Jump Threading, the optimization must duplicate basic blocks in order to isolate those execution paths.

Diverse current compilers contain very different implementations of Jump Threading. In this master's thesis, we first introduce a theoretical framework for Jump Threading. Then, we develop and implement a generalized version of a Jump Threading algorithm and analyze it in various respects, particularly on its interaction with other optimizations like If Conversion. Finally, using benchmarks, we set the optimization parameters and verify that Jump Threading is still profitable on modern processors. On a current Intel processor, we observe a speedup of up to 3 % in the SPECint2000 benchmark.

Erklärung

Hiermit erkläre ich, Joachim Priesner, dass ich die vorliegende Masterarbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, wörtlich oder inhaltlich übernommene Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

Contents

1. Introduction **7**

2. Basics and Related Work **11**
2.1. Branch prediction mechanisms . . . 11
2.2. Related work . . . 12

3. Design **13**
3.1. Preliminaries . . . 13
3.1.1. Limitations on the CFG structure . . . 13
3.1.2. CFG execution semantics . . . 13
3.1.3. Duplicating blocks . . . 14
3.2. Threading Opportunities . . . 15
3.2.1. Definition . . . 15
3.2.2. Critical edges . . . 16
3.2.3. Irreducible control flow . . . 17
3.2.4. Threading multiple TOs . . . 19
3.3. Finding TOs . . . 25
3.3.1. Data flow analysis . . . 25
3.3.2. Properties . . . 26
3.3.3. Reconstructing TOs . . . 27
3.4. Interaction with other optimizations . . . 31
3.4.1. If Conversion . . . 31
3.4.2. Block scheduling . . . 36

4. Analysis of existing algorithms **39**
4.1. LLVM . . . 39
4.2. GCC . . . 39
4.3. libFirm . . . 41
4.3.1. Confirm nodes and optimization pass . . . 41
4.3.2. The existing Jump Threading algorithm . . . 42
4.3.3. Limitations . . . 45
4.3.4. Non-termination . . . 46

5. Implementation **47**
5.1. General structure . . . 47
5.2. Wrap-around intervals . . . 47

5.3. Implementation details 51
5.3.1. Use of bit sets for cond sets 51
5.3.2. Edge annotations 51
5.3.3. Usage of the link field 51
5.4. Validation 52
5.5. SSA reconstruction 52
5.6. Cost model 53

6. Evaluation **55**
6.1. Experimental setup and methodology 55
6.1.1. Platform 55
6.1.2. Compiler 55
6.1.3. Benchmarks 56
6.1.4. Methodology 56
6.2. Experimental results and discussion 56
6.2.1. FSM microbenchmark 56
6.2.2. Execution time 58
6.2.3. Opcode mix 59
6.2.4. Compile time 62
6.2.5. Number of conds 62

7. Conclusion and Further Work **65**

A. Jump Threading source code **69**

1. Introduction

In order to increase throughput, all modern processors use a *pipelined* execution model, where processor instructions are not executed sequentially, but multiple consecutive instructions are in different stages of the pipeline at the same time. The elaborate pipelines of today's processors might contain more than a dozen stages.

One obstacle of the pipelined execution model is branching. The instruction to be executed after a branch instruction[1] is not known for sure until the branch instruction is decoded (for unconditional branches) or its data dependencies are evaluated (for conditional and indirect branches).

To avoid holding off the instruction fetch until then (called *stalling* the pipeline), a predictor first guesses the branch target, which is then fetched and executed speculatively until the branch target is known for sure. If the guess was correct, the results of the speculative execution are retired, i.e. made permanent; if not, the pipeline has to be flushed and the pipeline restarted at the correct branch target.

The latter case has a negative impact on application performance. A question on the programming Q&A site StackOverflow, "Why is it faster to process a sorted array than an unsorted array?"[2], illustrates this. In fact, this is by far the most popular question of all time on StackOverflow, with 22 596 upvotes at the time of writing. The problematic code looks roughly like this:

```
for element in array:
  if element > threshold:
    do something
```

Sorting the array results in a very regular pattern for the condition (always false at first, then always true), which causes the branch predictor to guess correctly almost all of the time. If the array is unsorted, the pattern is essentially random, and the branch misprediction penalty increases the running time of the program almost sixfold in this particular case.

Even though this anecdotal evidence is a somewhat extreme case, it might be desirable to eliminate branches statically, i.e. at compile time, wherever possible. Especially conditional branches should be avoided, and if possible converted into unconditional ones.

One optimization that does this is Jump Threading. In contrast to Constant Folding, the target needs to be known only on some of the code paths leading to

[1] We distinguish *unconditional* branches, which always jump to a constant location; *conditional* branches, which jump to a constant location but depending on a condition are taken or not; and *indirect* branches, which jump to a dynamically calculated location.

[2] `http://stackoverflow.com/q/11227809`, retrieved 2017-01-06

the conditional branch. Jump Threading duplicates these paths, and then converts conditional into unconditional jumps, as shown in Figure 1.1.

The elimination of conditional branches by Jump Threading comes at the cost of increased code size. Furthermore, the change in code structure might influence other optimizations.

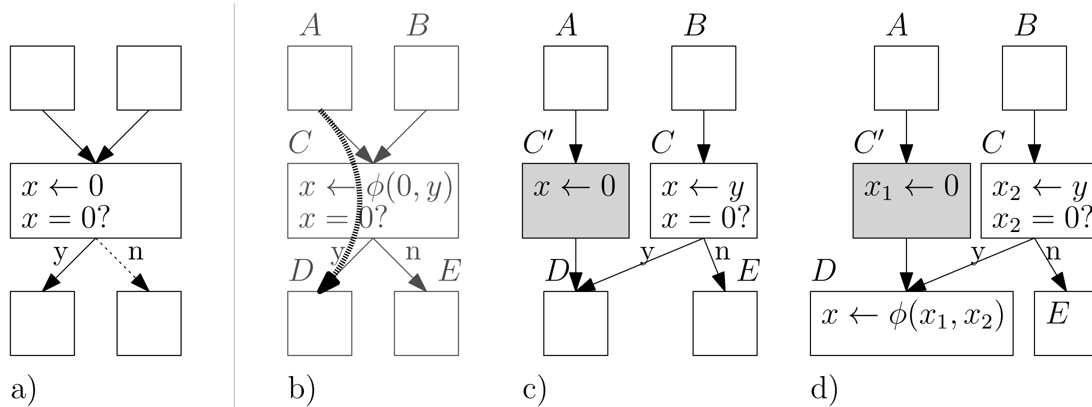


Figure 1.1. Simple Jump Threading example.
a) The outcome of the conditional branch is statically known on all code paths. Using Constant Folding and subsequent simplification, the dotted edge can be removed, transforming the branch into an unconditional one.
b) The outcome of the conditional branch is statically known only on the code paths that enter C via A. The jump $A \to C$ will always be followed by $C \to D$: they can be *threaded together*. We say ACD is a Threading Opportunity.
c) Jump Threading has duplicated block C, separating the code path on which the branch is always taken. In the duplicate, the conditional branch has been replaced by an unconditional branch. Throughout this thesis, we will designate duplicated blocks by shading them gray.
d) Duplicating C has introduced a second definition of x. To preserve the SSA property, a ϕ function is inserted at the iterated dominance frontier of C' and C.

Compilers have been implementing Jump Threading for a long time, and little scientific attention has been devoted to it because it is a seemingly basic optimization. However, what compiler authors mean by Jump Threading differs wildly between compilers.

The aim of this thesis is as follows:

- to provide a theoretical framework in which different Jump Threading variants can be analyzed,
- to develop a generalized Jump Threading algorithm, based closely on that theoretical model, that captures the theoretical maximum of what Jump Threading can do,

- to analyze this Jump Threading variant under various aspects, in particular the interaction with other optimizations,
- to implement this Jump Threading variant in the libFirm compiler library,
- to benchmark the implemented algorithm in real-world situations, in order to find which of the theoretical possibilities are useful in practice and how parameters should be tuned; also, in order to determine whether this optimization is still worth the trade-offs considering that branch predictors get more and more sophisticated.

2. Basics and Related Work

In this chapter, we present basic information about branch prediction, as well as previous work related to Jump Threading.

2.1. Branch prediction mechanisms

Branch prediction is a performance critical part of a modern microprocessor, and its inner workings are (in the best case) heavily patented or (in the worst case) heavily guarded secrets. Additionally, with every subsequent processor architecture generation, they are optimized further and sometimes fundamentally redesigned. However, some basic mechanisms remain the same.

We briefly summarize the introduction to [1, Chapter 3], which gives a good overview of branch prediction mechanisms in current processors.

We distinguish the related, but distinct concepts of *branch target prediction* and *branch prediction*, which may be implemented in the same unit of a processor. Branch target prediction, also called fetch prediction, uses a cache called a *Branch Target Buffer (BTB)* to detect whether the current instruction—which at this point is not even decoded yet—is likely to be a branch, and if yes, what the branch target will be if it is taken. This allows the pipeline to fetch and decode the instructions following an unconditional branch and is especially important in super-scalar processors with multiple instruction fetch and decoding units.

A BTB entry contains the target address of a taken jump and is created every time a branch is taken. Indirect branches with multiple possible targets may occupy multiple BTB entries. As the BTB has a limited set of entries, different branches may replace each other's BTB entries, causing a BTB miss or *misfetch*. Thus also unconditional branches and—depending on how the BTB is addressed—even the location of branches in the compiled code can affect performance, although a misfetch penalty is typically lower than a misprediction penalty.

The second concept, branch prediction, applies only to conditional branches and predicts whether the branch will be taken or not. A variety of information can be used to make the guess as educated as possible. Static information that can be used may include

- the direction of the branch (whether it jumps forwards or backwards)
- the opcode of the branch instruction
- hints inserted by the programmer or the compiler ("likely"/"unlikely" instruction prefixes)

and the (perhaps more important) dynamic information may include

- branch history: whether or not it has been taken in the past, even factoring in multiple recent executions of the branch instruction. The branch history may be local to that specific branch instruction, or global across all branches, or a combination thereof.
- loop counters, counting the number of iterations in previous executions of a loop, and predicting the same number of executions

The operating principles of a branch prediction unit are good to keep in mind when interpreting the evaluation results later. For example, restructuring the code might trigger cache effects such as cache conflicts in the BTB and thus may affect performance.

2.2. Related work

Few papers have been published that deal with the theoretical aspects of Jump Threading. Mueller et al. [2] present a basic version of Jump Threading. They describe an algorithm for finding threading paths as well as for creating block duplicates. However, they concentrate exclusively on finding threading paths within a loop, processing the graph from the innermost to the outermost loop. They also study the optimization on its own and not in interaction with other optimizations.

Bodík et al. [3] extend this to an interprocedural analysis, creating multiple entry and exit points of a procedure if needed.

Finally, compilers like GCC and LLVM contain Jump Threading implementations that are tuned very well and consider many special cases. Anyone wishing to study Jump Threading more closely will find these implementations very valuable.

3. Design

In this chapter, we describe patterns in control-flow graphs (CFG) that are relevant for Jump Threading, as well as the transformation of a CFG by Jump Threading, in a formal way. This theoretical framework serves as a basis for analyzing the different flavors of Jump Threading.

3.1. Preliminaries

3.1.1. Limitations on the CFG structure

Throughout this thesis, we will assume that a sequence of CFG blocks uniquely determines the path taken. This is not strictly necessary, but facilitates notation; otherwise we would have to define Threading Opportunities, executions, etc. in terms of edges instead of blocks, impairing readability.

Therefore we forbid the existence of more than one edge between two blocks P and Q of a CFG. Otherwise, the execution sequence (P, Q) would be ambiguous as to which output edge of P and which input edge of Q was taken, which is relevant e.g. for the evaluation of ϕ functions in Q.

This condition is fulfilled if the graph contains no critical edges, as each of the multiple edges between P and Q would be critical. The implementation of Jump Threading presented in Chapter 5 does not have that limitation and can deal with critical edges and even multiple edges between two blocks, as it stores information on a per-edge basis.

3.1.2. CFG execution semantics

We use an intuitive small-step operational semantics to describe executions in a CFG G. An *state* $\sigma \in \Sigma$ is some abstract entity that captures all local and global state of the program. The *transfer function* $\text{transfer} : V(G) \to \Sigma \to \Sigma$ describes the effect of a block's instructions. The *successor function* $\text{succ} : V(G) \to \Sigma \to V(G)$ determines the successor block of a CFG block given the state at the end of the block—on a real-world processor, this could e.g. correspond to reading the flags register in order to determine whether to take a conditional branch that terminates a block.

The *execution* of a CFG with initial state σ is a sequence $(P_i)_{i \in \mathbb{N}}$ of blocks in the CFG so that $P_{i+1} = \text{succ}(P_i, \sigma_i)$, where $\sigma_0 = \sigma$ and $\sigma_i = \text{transfer}(P_i, \sigma_{i-1})$. We are normally only interested in the block sequence in the CFG rather than the actual state modifications that lead to it, thus the exact format of σ is not relevant to us.

By definition of transfer and succ, an execution is deterministic and depends only on the initial state.

3.1.3. Duplicating blocks

Throughout this section, we will apply a transformation to a CFG $G = (V, E)$, yielding a CFG $G' = (V', E')$. One operation we perform often is duplicating blocks.

Definition 3.1.1. We say G' is obtained from G by *duplicating* a block P if

$V' = V \cup P'$, with $\text{transfer}(P') = \text{transfer}(P)$ and $\text{succ}(P') = \text{succ}(P)$

$E' = E \cup \{P'Q_i \mid PQ_i \in E\}$ ◊

We say that P' is a duplicate of P and an edge $P'Q$ is a duplicate of the corresponding edge PQ. Note that P' initially has no incoming edges and the same outgoing edges as P.

In SSA-based CFGs, we have to take additional steps to preserve semantics. Firstly, duplicating blocks means introducing additional definitions of that block's variables. This in turn requires SSA form to be reconstructed; this will be detailed in Section 5.5. Secondly, adding or removing an incoming edge to/from a block requires modifications of that block's ϕ functions. Clearly, this must be handled by the implementation (and it is); however, in this chapter, we assume that all edge manipulation operations do the right thing automatically. That is, if PQ is the i'th predecessor edge of block Q and is rerouted to become the j'th predecessor edge of block Q', then the j'th input of a ϕ function in Q' is the i'th input of the corresponding ϕ function in Q.

We require a block P and its duplicate P' to have the same transfer function, i.e. the same execution semantics. This is an actual constraint, albeit not a severe one: imagine that an instruction that accesses the instruction pointer and does arithmetic with it—its results would be different depending on whether it resides in the original or duplicate block. However, such an instruction would not be expected to survive an optimizing compiler anyway.

We distinguish *original blocks*, which are all blocks of the original graph G, and *duplicate blocks*. We will only create new blocks by duplicating existing blocks, so all elements of $V(G') \setminus V(G)$ will be duplicate blocks. The function $orig : V(G') \to V(G)$ links an original or duplicate block to itself or its original block respectively: We have $orig|_{V(G)} = id$, and when duplicating a block P', for the resulting block Q' we have $orig(Q') = orig(P')$. The function $Orig$ is the lifting of $orig$ to sequences of blocks.

A word on notation: Throughout this chapter, we write P for an original block and P' for a block that is either an original or duplicate block. We also omit the $orig$ function by just writing P for the original block of P'.

3.2. Threading Opportunities

We introduce the concept of Threading Opportunities, which are paths in the CFG that, when followed during an execution, guarantee that a specific successor of a block will be taken. The task of the Jump Threading optimization pass is *a)* to find such Threading Opportunities by performing static analysis on the CFG, and *b)* to transform the CFG in order to take advantage of the information contained in them.

3.2.1. Definition

Definition 3.2.1 (Threading Opportunity)**.** Let $n \geq 2$. A walk[1] $T = P_0 P_1 \dots P_n$ in G is called a **Threading Opportunity (TO)** if

(T1) no execution $(P_i)_{i\in\mathbb{N}}$ of G contains a subsequence $P_0 P_1 \dots P_{n-1} X$ for $X \neq P_n$. That is, after executing $P_0, P_1, \dots, P_{n-1}$ in that order, independent of the program state or the execution path taken before reaching P_0, the program will always jump to P_n.

(T2) if $n \neq 2$, P_1 does not have exactly one incoming edge

(T3) P_{n-1} has more than one outgoing edge

We call P_0 the **start block**, P_{n-1} the **condition block**, P_n the **target block** of T. The edge $P_{n-1}P_n$ is called the **target edge**. ♢

While (T1) is the actual characterization of a TO, (T2) and (T3) serve only to eliminate trivial or redundant TOs: If a walk $P_0 P_1 \dots P_n$ does not fulfil (T2), i.e. P_1 has only one incoming edge and $n > 2$, then the sub-walk $P_1 \dots P_n$ fulfils (T1) as well since P_1 must be preceded by P_0 in any execution of the program. If (T3) is not fulfilled, then there is nothing left for Jump Threading to do; the branch of P_{n-1} is already unconditional.

If $n = 2$ and $P_1 = P_{n-1}$ has only one incoming edge (but multiple outgoing edges), then (T2) is fulfilled and removing all of P_{n-1}'s outgoing edges except the target edge does not change the program's semantics. This is however a very special case, and we would like to take advantage of the TOs in the general case by making the branch that a TO predicts unconditional:

Definition 3.2.2 (Respecting a TO)**.** Let $T = P_0 P_1 \dots P_n$ be a TO in G. We say that the transformed graph G' *respects* T if for any subsequence $P'_0 P'_1 \dots P'_{n-1}$ of an execution of G' (where the P'_i are duplicates of the respective P_i), P'_{n-1} has only one outgoing edge.

G' respects a set $\mathcal{T}$ of TOs if it respects every element of $\mathcal{T}$. ♢

For one TO, this is very straightforward to fulfil using path duplication:

[1] A *path* in a graph is a sequence of connected nodes with no repeated nodes, while a *walk* may contain nodes more than once.

Definition 3.2.3 (Threading algorithm for a single TO)**.** $T = P_0P_1 \dots P_n$ be a TO in $G = (V, E)$. We say the CFG $G' = (V', E')$ is obtained from G by *threading* T if

$V' = V \cup \{P_1^*, \dots, P_{n-1}^*\}$, where P_i^* is a duplicate of P_i.

$E' = (E \setminus \{P_0P_1\}) \cup \{P_0P_1^*\} \cup \{P_i^*P_{i+1}^* \mid i = 0, \dots, n-2\} \cup \{P_i^*Q \mid i = 0, \dots, n-2, P_iQ \in E, Q \neq P_{i+1}\} \cup \{P_{n-1}^*P_n\}$ ◊

Threading T preserves a CFG's semantics and the resulting graph respects T:

Theorem 3.2.4. *Let G' be the CFG obtained from G by threading a TO $T := P_0P_1 \dots P_n$. Let $(Q_i)_{i\in\mathbb{N}}$ be an execution of G, and let $(Q'_i)_{i\in\mathbb{N}}$ be an execution of G' with the same initial state σ. Then for the intermediate states σ_i, σ'_i of the executions, we have $\sigma_i = \sigma'_i$ for all i.*

Furthermore, G' respects T.

Proof. Semantics preservation: We show that for $i \in \mathbb{N}$, Q'_i is a duplicate of Q_i. Then $\text{transfer}(Q'_i) = \text{transfer}(Q_i)$, and the result follows from the definition of the execution semantics.

We perform induction over i. For $i = 1$, we have $Q_1 = \mathsf{Start}_G$ and $Q'_1 = \mathsf{Start}_{G'} = \mathsf{Start}_G$: Note that the Start block is never duplicated by threading, as only the blocks $P_1, \dots, P_{n-1}$ are duplicated. These all have a predecessor (else T would not be a walk in G) and thus are unequal to the Start block.

For $i > 1$, assume that Q'_i is a duplicate of Q_i. If $Q_i = P_j$ for $0 \leq j \leq n-1$, it follows from the definition of E' in Definition 3.2.3 that the successor of Q'_i is a duplicate of Q_{i+1} (for $j = n-1$ or $Q_{i+1} \neq P_{i+1}$, it is Q_{i+1} itself, otherwise the duplicate P_{i+1}^* of P_{i+1}).

If Q_i is not one of these blocks, its successor edges are not changed by threading T, therefore $Q'_{i+1} = Q_{i+1}$.

G' respects T: We show that for $1 \leq i \leq n-1$, if $(Q_i)_{i\in\mathbb{N}}$ contains a subsequence $P'_0 \dots P'_i$, then $P'_i = P_i^*$. The result then follows because P_{n-1}^* has only one control flow successor.

We perform induction over i. For $i = 1$, the statement holds because $P_0P_1^*$ is the only edge from P_0 to a duplicate of P_1. For $i > 1$, assume that $P'_i = P_i^*$. Then $P'_{i+1} = P_{i+1}^*$ because $P_i^*P_{i+1}^*$ is the only edge from P_i^* to a duplicate of P_{i+1}. (Here we use that the successor edge of a block is uniquely determined by its successor block, as noted in Section 3.1.1.) □

3.2.2. Critical edges

Note that in G' the edge $P_{n-1}P_n$ will be a critical edge: P_{n-1} has multiple outgoing edges by definition of T, and P_n has at least $P_{n-1}P_n$ and $P_{n-1}^*P_n$ as incoming edges.

Critical edges interfere with SSA deconstruction and are therefore undesirable to have in a CFG. A critical edge PQ can be removed by *splitting* it, that is by inserting a new, empty block S into the CFG and replacing the edge PQ by edges PS and SQ. Splitting an edge modifies the TOs containing that edge, but do not create or destroy TOs:

Lemma 3.2.5. *Let G be a CFG, and let G' be the CFG obtained from G by splitting an edge PQ in G. Then there is a one-to-one mapping between TOs in G and TOs in G'.*

Proof. The new block S has one control flow predecessor and one control flow successor. Thus in any execution of G' it is always preceded by P and succeeded by Q. Its transfer function is the identity function. Replacing all occurrences of the subsequence PSQ in an execution of G' by PQ therefore yields an equivalent execution of G and vice versa.

Thus transforming a TO T in G as follows yields a TO T' in G':

- If the TO begins with $PQ\ldots$, change the first block to S.
- If the TO ends with $\ldots PQ$, change the last block to S.
- Replace all occurrences of PQ in the TO by PSQ.

This operation does not change the second block P_1 or the second-to-last block P_{n-1} of the TO and therefore preserves the restrictions on those blocks made in Definition 3.2.1. The resulting walk T' is still a TO in G', as an execution of G' that contradicts T' could be transformed into an execution of G that contradicts T.

It is easy to see that this operation yields a bijective function between TOs in G and TOs in G'. □

3.2.3. Irreducible control flow

In some cases, such as in Figure 3.1, Jump Threading can create irreducible control flow, which might prevent other optimizations from being applied.

Currently, no optimization in libFIRM depends on the CFG being reducible, and loops are represented internally using strongly connected components, which also cover irreducible loops. Other compilers, however, represent loops as natural loops using loop headers and backedges, and some optimizations demand reducible control flow. In these compilers, Jump Threading usually avoids creating irreducible loops.

In this section, we give a necessary condition for Jump Threading introducing irreducible control flow. First, we recall the definition of reducibility.

A depth-first search (DFS) path $P_0 \ldots P_n$ of a graph G is a path from the Start node in which $P_0, \ldots, P_{n-1}$ are mutually distinct (as a depth-first search stops at already seen nodes). If P_n is equal to one of $P_0, \ldots, P_{n-1}$, then $P_{n-1}P_n$ is called a *retreating edge* of the DFS (more commonly a backedge, but using this term would create confusion with backedges in the CFG).

There are multiple equivalent definitions of reducibility of a CFG G, the one we use here is:

Definition 3.2.6 (Reducibility of a CFG)**.** A CFG G is *reducible* if any retreating edge PQ in a DFS path in G is a backedge of G, i.e. Q dominates P. ◊

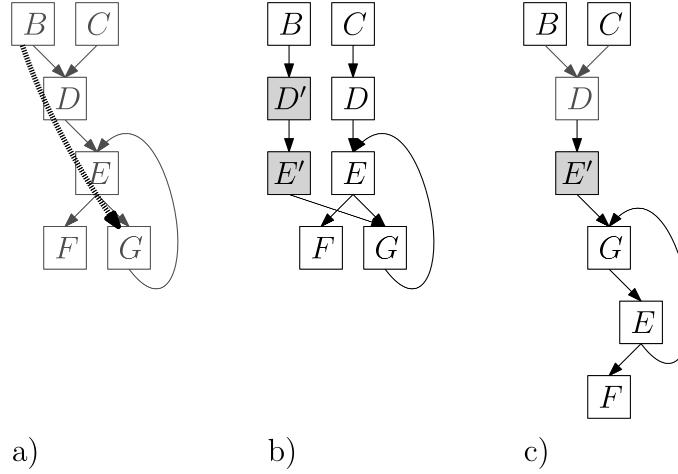


Figure 3.1. Example of an irreducible loop created by Jump Threading
a) A graph which contains a natural loop consisting of the blocks E (loop header) and G.
b) After threading the TO $BDEG$, an irreducible loop is formed; the loop now has two entry points.
c) Duplicating a loop header does not always create an irreducible loop. After threading the TO DEG in graph *a)*, the resulting graph's control flow is still reducible.

We now assume that G is reducible and applying Jump Threading results in an irreducible graph G'.

Lemma 3.2.7. *If threading a TO in a reducible CFG G creates an irreducible CFG G', then G' contains two blocks that are a duplicate of a loop header block in G.*

Proof. As G' is irreducible, there exists a depth-first search path $P' := P'_0 \dots P'_i \dots P'_n P'_i$ in G' with $0 < i < n$ so that P'_i does not dominate P'_n. (We know that $0 < i$ because $P_0 = \mathsf{Start}_{G'}$ has no incoming edges, and $i < n$ because if $i = n$, then $P'_i = P'_n$ would dominate itself. In particular, $P'_i \neq P'_n$.)

We examine the corresponding path $P := Orig(P') = P_0 \dots P_i \dots P_n P_i$ in G and distinguish two cases.

Case 1: P is not a depth-first search path in G.

Let $Q := P_0 \dots P_k$ for $0 < k < n$ be the longest prefix of P that is a DFS path in G. Q ends in a retreating edge (else it could be extended), so $Q = P_0 \dots P_j \dots P_{k-1} P_k$ with $P_j = P_k$ for some $j < k$.

Because G is reducible, $P_{k-1} P_k$ is a backedge in G, therefore $P_k \operatorname{dom} P_{k-1}$ and P_k is a loop header. Furthermore, because $P'_j \neq P'_k$, P'_j is a duplicate of P_k that is distinct from P'_k.

Case 2: P is a depth-first search path in G.

Because P ends in a retreating edge $P_n P_i$ and G is reducible, we have $P_i \operatorname{dom} P_n$. In particular $P_n P_i$ is a backedge in G and P_i is a loop header.

On the other hand, $\neg(P'_i \operatorname{dom} P'_n)$, so there exists a path Q' from $\mathsf{Start}_{G'}$ to P'_n that does not contain P'_i. However, Q' does contain some duplicate P''_i of P_i, as the corresponding path $Q = Orig(Q')$ in G is a path from Start_G to P_n and thus contains P_i.

In both cases, there exist two blocks in G' that are a duplicate of a loop header block in G. (One of these block could be the original loop header, as we view an original block as being a duplicate of itself.) □

The two cases in the proof of Lemma 3.2.7 can be seen in Figure 3.1 *b)*: The path $\mathsf{Start} \ldots BDE'G'EG$ is a DFS path leading to Case 1, while the DFS path $\mathsf{Start} \ldots CDEGE$ leads to Case 2.

Lemma 3.2.7 yields a necessary condition for the introduction of irreducible control flow. The condition is not a sufficient one, as illustrated in Figure 3.1 *c)*. Nevertheless, it can be used as a heuristics to avoid irreducible loops. This is for example done in LLVM.

3.2.4. Threading multiple TOs

A single CFG may contain multiple TOs, and while threading them one after another certainly preserves the program's semantics, it might create more block duplicates than necessary, as demonstrated in Figure 3.2. It is therefore beneficial to consider (finite) sets of TOs together.

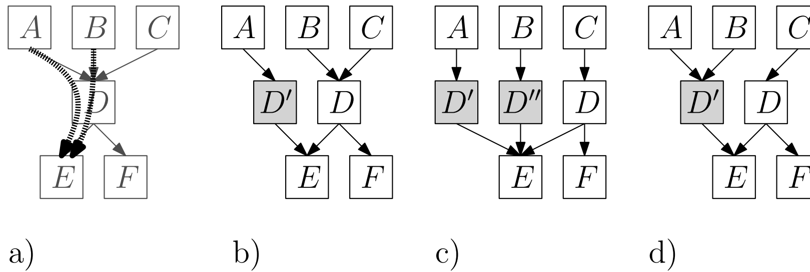


Figure 3.2. Disadvantage of threading TOs one after another
a) The original graph, with TOs $\{ADE, BDE\}$.
b) By threading $\{ADE\}$, a duplicate of D is created. BDE is a TO in the resulting graph.
c) Threading BDE in graph *b)* results in another duplicate of D being created.
d) If the set $\{ADE, BDE\}$ is instead threaded in one step, the TOs can share the duplicate block D. The resulting graph is semantically equivalent to graph *c)*.

Restrictions on sets of TOs

We do not want a set of TOs to contain conflicting or redundant information:

Definition 3.2.8. Let T_1 and T_2 be suffixes of TOs. T_1 *forbids* T_2 if T_2 contains an edge that is forbidden by T_1, i.e. if $T_1 = P_0P_1 \dots P_{n-1}P_n$ and T_2 contains a subsequence $P_0P_1 \dots P_{n-1}X$ for $X \neq P_n$. $\diamondsuit$

We impose the following restrictions on a set $\mathcal{T}$ of TOs we consider:

(A1) No element of $\mathcal{T}$ is a suffix of another element of $\mathcal{T}$.

(A2) For all $P_0P_1 \dots P_n \in \mathcal{T}$, a predecessor Q of P_1 exists with $QP_1 \dots P_n \notin \mathcal{T}$.

(A3) No element of $\mathcal{T}$ forbids another element of $\mathcal{T}$.

Because of the semantics associated with a TO, a set of TOs by definition cannot contain conflicting information. Therefore, (A1) implies the following stronger condition:

(A1') For all $P_0P_1 \dots P_n, Q_0Q_1 \dots Q_n \in \mathcal{T}$, $P_0 \dots P_{n-1}$ is not a suffix of $Q_0 \dots Q_{n-1}$.

Figure 3.3 illustrates these conditions.

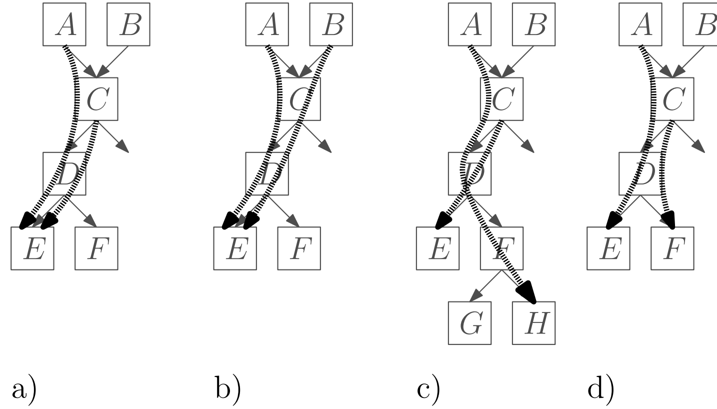


Figure 3.3. Examples of forbidden sets of TOs.
a) The set $\{ACDE, CDE\}$ is forbidden by (A1), since CD is a suffix of ACD. The TO $ACDE$ is redundant.
b) The set $\{ACDE, BCDE\}$ is forbidden by (A2), since it is equivalent to $\{CDE\}$.
c) The set $\{ACDFH, CDE\}$ is forbidden by (A3), since $ACDFH$ contains the edge DF which is forbidden by CDE.
d) The set $\{ACDE, CDF\}$ is not a valid set of TOs because it does not fulfil (A1'), since CD is a suffix of ACD. The TO CDF conflicts with $ACDE$.

Annotation of CFGs

Given a graph G and a set $\mathcal{T}$ of TOs, our goal is to transform G into a semantically equivalent CFG that respects $\mathcal{T}$ (Definition 3.2.2).

To better understand when two TOs can share a duplicated block, we take a step back and look at what the block duplication in Definition 3.2.3 actually accomplishes. As the original's and the duplicate's transfer functions are equivalent, some other kind of information must be (implicitly) associated with the duplicated block. That information, for a block P_i of a TO, is the statement "In any execution, when this block is followed by $P_{i+1}, \dots, P_{n-1}$, then the next block after that will be P_n".

This statement holds unconditionally for the original block P_0 (by definition of the TO). It does not hold in general for $P_1, \dots, P_{n-1}$, which is why the the duplicate block P_i' is needed that has the same execution semantics but may assume that the statement holds.

When threading multiple TOs, we make this information carried by a duplicated block explicit. This is done by means of the *annotation function* $\mathcal{S}$ (for *suffixes*), which assigns to every block P' in G' a set $\mathcal{S}(P')$ containing walks in G that are guaranteed by this duplicate of P. That is, if $P_0 P_1 \dots P_{n-1} P_n \in \mathcal{S}(P_0')$, then any execution subsequence of G' that starts at P_0' and continues with duplicates of P_1, $\dots$, P_{n-1} will be continued with a duplicate of P_n.

$\mathcal{S}(P')$ will contain proper suffixes of TOs, thus by (A1), $\mathcal{T} \cap \mathcal{S}(P') = \emptyset$ for all P'.

Furthermore, we write $\mathcal{T}|_P := \{T \in \mathcal{T} \mid T \text{ starts with } P\}$. Those are the walks guaranteed by *every* duplicate of P (including P itself). Then for any block P' of G', $\mathcal{S}(P') \cup \mathcal{T}|_P = \mathcal{S}(P') \,\dot\cup\, \mathcal{T}|_P$ is the set of guarantees made by the duplicate P'. In order to produce the minimum amount of duplicate blocks, we additionally filter out redundant information, yielding

$$\mathcal{S}'(P') := \mathcal{S}(P') \cup \{T \in \mathcal{T}|_P \mid \text{ no element of } \mathcal{S}(P') \text{ forbids } T\}. \tag{3.1}$$

(See Definition 3.2.8 for the definition of one TO suffix forbidding another.)

Propagation of TO suffixes

Next, we discuss how sets of TO suffixes are propagated across control flow edges. Note that if $\mathcal{S}'(P')$ contains a two-element walk PQ, then all outgoing edges of P' except the one to a duplicate of Q can be removed without changing the CFG's semantics.

We define a function *ShortenFilter* which propagates sets of TO suffixes along a control flow edge PQ. Given a set of TO suffixes in P, we filter out the TO suffixes that concern edges not leading to Q. The remaining TO suffixes are shortened, i.e. their first element, which is always P, is removed.

For a TO suffix $T := PP_1 \dots P_n$, and for a set S of TO suffixes, we define

$$\mathit{ShortenFilter}_Q(T) := \begin{cases} P_1 \dots P_n & \text{if } n > 2 \wedge P_1 = Q \\ \bot & \text{otherwise} \end{cases} \tag{3.2}$$

$$\mathit{ShortenFilter}_Q(S) := \begin{cases} \emptyset & \text{if } \exists PX \in S : X \neq Q \\ \{\mathit{ShortenFilter}_Q(T) \mid T \in S\} \setminus \{\bot\} & \text{otherwise} \end{cases} \tag{3.3}$$

The first case of the second definition ensures that we do not try to propagate information across an edge that will be removed.

We note the following necessary condition for the preservation of the CFG's semantics: For any edge $P'Q'$ in the transformed graph G', we need

$$\mathcal{S}(Q') \subseteq \mathit{ShortenFilter}_Q(\mathcal{S}(P') \,\dot\cup\, \mathcal{T}|_P). \tag{3.4}$$

Otherwise there exists a walk $W := QP_1 \dots P'_n$ so that *a)* $W \in \mathcal{S}(Q')$ but *b)* $PQP_1 \dots P_n \notin \mathcal{S}(P') \cup \mathcal{T}|_P$, i.e. it is not guaranteed by P'. We know from *b)* that an execution of G' may contain a subsequence $P'Q'P'_1 \dots X'$ for $X' \neq P'_n$, however *a)* forbids this.

On the other hand, the desired property of a CFG that respects $\mathcal{T}$ is that the guarantees made by a block duplicate preserve as much information as possible from its predecessors, i.e. for any edge $P'Q'$ in G',

$$\mathcal{S}'(Q') \supseteq \mathit{ShortenFilter}_Q(\mathcal{S}'(P')). \tag{3.5}$$

From (A1) we have $\mathcal{T}|_Q \cap \mathit{ShortenFilter}_Q(\mathcal{S}'(P')) = \emptyset$, as *ShortenFilter* produces suffixes of TOs. Using Equation (3.1), we can therefore replace $\mathcal{S}'(Q')$ by $\mathcal{S}(Q')$ on the left hand side of the equation, and with Equation (3.4) we obtain that after threading, G' should fulfil

$$\forall P'Q' \in E(G') : \mathcal{S}(Q') = \mathit{ShortenFilter}_Q(\mathcal{S}'(P')). \tag{3.6}$$

Data-flow analysis

We formulate the problem of finding the necessary block duplicates as a data flow problem. The analysis is a forward may analysis, with the data flow equations

$$\begin{aligned} \mathit{out}_P &= f_P(\mathit{in}_P) \\ \mathit{in}_P &= \bigsqcup_{Q \in \mathit{pred}_P} \mathit{out}_Q. \end{aligned}$$

For each block $P \in V(G)$, we want to obtain $\mathcal{S}'(P')$ for all block duplicates P'. Thus the value lattice is the power set of the set of suffixes of TOs in $\mathcal{T}$, with the natural partial order "$\subseteq$" and the join operator "$\bigcup$". This lattice has finite height because all TOs are finite and $\mathcal{T}$ contains finitely many elements.

The transfer function f_P of a block P is derived from Equations (3.1) and (3.6) and defined by

$$\begin{aligned} f_P(in_P) &= \{g_P(S) \mid S \in in_P\}, \text{ where} \\ g_P(S) &= \mathit{ShortenFilter}_P(S) \cup \\ &\quad \{T \in \mathcal{T}(P) \mid \text{ no } T' \in \mathit{ShortenFilter}_P(S) \text{ forbids } T\}. \end{aligned}$$

Threading algorithm

The data flow equations lead to Algorithm Thread, which performs a fixpoint iteration to compute the block duplicates and performs the necessary control flow changes in the same step:

Algorithm 1: Algorithm Thread

input : a graph G and a set $\mathcal{T}$ of TOs in G

1 **while** $\exists P'Q' \in E(G') : \mathcal{S}(Q') \neq \mathit{ShortenFilter}_Q(\mathcal{S}'(P'))$ **do**
2   **if** $\exists$ *duplicate* Q^* *of* Q *with* $\mathcal{S}(Q^*) = \mathit{ShortenFilter}_Q(\mathcal{S}'(P'))$ **then**
3         $Q^* \leftarrow$ that duplicate
4   **else**
5   Create a duplicate Q^* of Q with $\mathcal{S}(Q^*) = \mathit{ShortenFilter}_Q(\mathcal{S}'(P'))$;
          /* Q^* has no incoming edges and an outgoing edge Q^*R for all edges QR in G */
6   **end**
7   Replace the edge $P'Q'$ by $P'Q^*$;
8 **end**
9 **foreach** *block* P' *with a two-element walk* $PQ \in \mathcal{S}'(\mathcal{P}')$ **do**
10   Remove all outgoing edges of P' except the one to (a duplicate of) Q ;
11 **end**

The caveats in connection with rerouting edges when executing this algorithm in an SSA-based CFG have been mentioned above.

Note that we can postpone the removal of outgoing edges until after the While loop: For each edge $P'Q'$ so that $PX \in \mathcal{S}'(P')$ for $X \neq Q$, we have $\mathit{ShortenFilter}(\mathcal{S}'(P')) = \emptyset$ by Equation (3.3). Hence this edge does not cause any block duplicates to be created.

When executed with a one-element set $\mathcal{T}$, this algorithm yields the same result as Definition 3.2.3.

Lemma 3.2.9. *Algorithm* Thread *terminates.*

Proof. The algorithm terminates iff the While loop terminates. Note that in Line 7 the edge PQ that fulfilled the loop condition is removed and a new edge is added that does not fulfil the loop condition. Newly created edges that fulfil the loop condition can therefore only result from the block duplication in Line 5. Thus it suffices to

show that the algorithm creates finitely many blocks, which follows from the fact that the value lattice, which contains the possible values for $\mathcal{S}(Q')$, is finite. □

As in the case of threading a single TO, this algorithm optimizes the graph without changing its execution semantics. We prepare two auxiliary lemmas:

Lemma 3.2.10 (Start and End duplication)**.** *Algorithm* Thread *does not duplicate the* Start *or* End *block of the input graph.*

Proof. A block Q' is duplicated only if it has a predecessor P' so that $\mathcal{S}(Q') \neq \mathit{ShortenFilter}_Q(\mathcal{S}'(P'))$.

The Start block has no predecessors and is thus never duplicated.

For any block Q, $\mathit{ShortenFilter}_Q(\cdot)$ contains walks of length ≥ 2 that start with Q. Any element of $\mathit{ShortenFilter}_{\mathsf{End}}(\cdot)$ would therefore contain a successor of End as second element, a contradiction. Thus $\mathit{ShortenFilter}_{\mathsf{End}}(\cdot) = \emptyset$ and no duplicate of End is created.

□

Lemma 3.2.11 (Following TO suffixes along a path)**.** *Let G' be a CFG that fulfils Equation* (3.6)*, let P_0^* be a duplicate of P_0 with $P_0 P_1 \dots P_{n-1} P_n \in \mathcal{S}'(P_0^*)$.*

If there is an execution of G' that contains a subsequence $P_0^* P_1' \dots P_{n-1}'$, where P_i' is a duplicate of P_i for all i and $P_0' = P_0^*$, then for $0 \leq i \leq n-1$ the suffix $P_i \dots P_{n-1} P_n$ is an element of $\mathcal{S}'(P_i')$.

Proof. Induction over i. The case $i = 0$ follows from the assumption.

For $1 < i < n-1$, assume that $U := P_i \dots P_{n-1} P_n \in \mathcal{S}'(P_i')$. As the edge $P_i' P_{i+1}'$ is contained in the execution of G' and therefore is an edge in G', we have $P_i X \notin \mathcal{S}'(P_i')$ for $X \neq P_{i+1}$. Because G' fulfils Equation (3.6), $\mathit{ShortenFilter}_{P_{i+1}}(U) = P_{i+1} \dots P_{n-1} P_n \in \mathcal{S}(P_{i+1}') \subseteq \mathcal{S}'(P_{i+1})$. □

Now we are ready to prove the main result:

Theorem 3.2.12. *The graph G' created by Algorithm* Thread *has the same execution semantics as the original graph G and respects $\mathcal{T}$.*

Proof. Semantics preservation: Let $(Q_i)_{i \in \mathbb{N}}$ and $(Q_i')_{i \in \mathbb{N}}$ be executions of G and G' respectively with same initial state. As in Theorem 3.2.4, it suffices to show that for $i \in \mathbb{N}$, Q_i' is a duplicate of Q_i.

We perform induction over i. The case $i = 1$ follows from Lemma 3.2.10. For $i > 1$, assume Q_i' is a duplicate of Q_i. As all duplicates created by Algorithm Thread have the same outgoing edges as their original block, and Algorithm Thread only reroutes outgoing edges to duplicates of their target blocks, Q_{i+1}' is a duplicate of Q_{i+1}.

G' respects $\mathcal{T}$: Let $T := P_0 P_1 \dots P_n \in \mathcal{T}$ be a TO. Consider any execution of G' that contains a subsequence of the form $P_0' P_1' \dots P_{n-1}'$.

We show that $T \in \mathcal{S}'(P_0')$. Then by Lemma 3.2.11, $P_{n-1}P_n \in \mathcal{S}'(P_{n-1}')$, therefore by Line 10 of Algorithm THREAD, P_{n-1}' has only one outgoing edge.

As we have $T \in \mathcal{T}|_{P_0}$, by Equation (3.1) it remains to show that no element of $\mathcal{S}(P_0')$ forbids T. Assume that there is such an element. It has the form $P_0 \dots P_i X$, with $0 \leq i < n-1$ and $X \neq P_{i+1}$. By Lemma 3.2.11, we have $P_i X \in \mathcal{S}'(P_i')$, thus P_i' has no outgoing edge to a duplicate of P_{i+1}. However, we assumed that the execution of G' contains the edge $P_i' P_{i+1}'$, a contradiction.

□

Theorem 3.2.13. *Algorithm* THREAD *computes an optimal solution to the data flow problem, and the order in which the edges are processed does not matter.*

Proof. As Algorithm THREAD performs a fixpoint iteration, its result does not depend on the order in which the edges are processed (however, its running time does).

Furthermore, as the transfer functions f_P are defined element wise on the input set, we have $f_P(in_1 \cup in_2) = f_P(in_1) \cup f_P(in_2)$, i.e. they are distributive. By a result of Kam and Ullman [4], the fixpoint iteration computes the optimal solution (meet-over-all-paths solution). □

3.3. Finding TOs

Until now, we have assumed a graph together with a set of TOs as input to our algorithms. We have not yet considered the problem of actually finding these TOs.

It is important to note that static analysis can only ever find a subset of all theoretically possible TOs in a graph. We describe an algorithm that finds a reasonably large subset of TOs.

3.3.1. Data flow analysis

We formulate the problem as a data flow analysis problem. The analysis is a backwards may analysis, with the data flow equations

$$\begin{aligned} out_P &= \bigsqcup_{Q \in succ_P} in_Q \\ in_P &= f_P(out_P) \end{aligned}$$

The value lattice is the power set of $Conds := \{x \diamond c \mid x \in Vars, \diamond \in Rels, c \in Consts\}$, with partial order "$\subseteq$" and join operator "$\bigcup$". Here $Vars$ are the variables in the CFG, $Consts$ is the set of all constant values (not just those occurring in the CFG) and $Rels$ is the set of all comparison relations that may occur in a CFG.

The global transfer function f_P of a block P is defined in terms of the the local transfer functions of its instructions, i.e. $f_P = f_0 \circ \dots \circ f_n$, where f_i is the local transfer function of instruction i.

The local transfer function $f_i : \mathcal{P}(\mathit{Conds}) \longrightarrow \mathcal{P}(\mathit{Conds})$ of an instruction i returns its input $C \subseteq \mathit{Conds}$ modified as follows:

- If i is a conditional branch, then for each branch target block X so that the edge PX is taken iff a condition $x \diamond c \in \mathit{Conds}$ is fulfilled, add $x \diamond c$ to C.

  (For a simple conditional branch depending on $x \diamond c$, we would add $x \diamond c$ and $x \diamond' c$, where $\diamond'$ is the inverse relation if $\diamond$. More complicated examples are possible when Switch statements are involved.)

- If i has the form "x `:=` e", where e is any expression, for any condition $x \diamond c \in C$, remove $x \diamond c$ from C.

  If e satisfies $x \diamond c$, we have found a TO starting point. In particular, this is the case if e is a constant value c' that satisfies $c' \diamond c$, or if e is an arbitrary value y annotated with an assertion specifying that it fulfils some condition C, and the condition "y fulfils C" implies $y \diamond c$.[2]

  Else, if $e = f(y)$ for a bijective unary function f and a variable y, add $y \diamond f^{-1}(c)$ to C. This especially covers simple assignments x `:=` y (for $f = id$), but also functions of the form $f(x) = x + c$, $f(x) = c - x$, etc.

3.3.2. Properties

This analysis finding a "reasonably large subset of TOs" is justified as follows: When encountering a conditional branch, we insert a condition that is fulfilled *iff* the branch is taken. When encountering an assignment to a variable, we insert a new condition C_2 that is fulfilled *iff* an existing condition C_1 for that variable is fulfilled.

For correctness, we clearly need $C_2 \Rightarrow C_1$, but requiring only this property would allow $C_2 = \textbf{false}$, which would be correct but yield no Threading Opportunities. We want C_2 to be fulfilled for as many values as possible, and because f is bijective, this is the case if $C_1 \Rightarrow C_2$. In a sense, we do not throw away information when transforming conditions.

It would be easy to extend the algorithm to include non-bijective functions. For example, for the condition $x = 0$ and the instruction x `:=` $y \cdot z$, the conditions $y = 0$ and $z = 0$ could be inserted.

The data flow analysis is distributive: For a local transfer function f_i, we have $f_i(C_1 \cup C_2) = f_i(C_1) \cup f_i(C_2)$, as $f_i(C)$ operates element-wise on C. Therefore the global transfer functions are distributive as well. Likewise, the transfer functions are monotone, that is $X \subseteq Y \longrightarrow f_P(X) \subseteq f_P(Y)$.

As the sets *Vars* and *Rels* are finite, the set *Conds* is finite (and has finite height) iff *Consts* is finite. Together with the monotonicity of f_P, this yields that a fixpoint iteration computing the solution of the data flow equations terminates iff *Consts* is finite.

[2]The latter case corresponds to the `Confirm` nodes of libFIRM which are presented in Section 4.3.1.

If in a theoretical analysis of this algorithm, one assumes that *Consts* contains e.g. $\mathbb{Z}$, and is therefore infinite, we could restrict the size of *Consts* by letting f_i consider only assignments x `:=` y instead of allowing functions on the right hand side. Then it would suffice to consider only the constants c that appear in instructions of the form `if` $x \diamond c$. However, this would unnecessarily limit the usefulness of the analysis.

For a more practical analysis, the set *Const* would consist of target machine values and therefore be finite. However, the lattice's height would still be enormous. If we use fixpoint iteration for the analysis, we should therefore limit it according to some cost function, e.g. by terminating it after a fixed number of iteration steps.

3.3.3. Reconstructing TOs

Reconstructing TOs from the conditions the above algorithm finds seems straightforward. One would simply reverse the effects of the local transfer functions f_i, building the TOs along the way. In our case, as the original analysis is a backward analysis, this would entail a forward analysis constructing prefixes of TOs:

- At each TO starting point (an instruction where we determined a condition $x \diamond c$ to be always fulfilled), create a new TO prefix containing only the instruction's block and mark it as belonging to the condition $x \diamond c$.

- For an assignment instruction x `:=` $f(y)$ and a TO prefix belonging to $y \diamond c$, mark the TO prefix as belonging to $x \diamond f^{-1}(c)$.

- For an edge AB and a TO prefix $P_0 \dots P_n A$ belonging to $x \diamond c$, extend the prefix to form the TO prefix $P_0 \dots P_n AB$. This should only be performed if the condition in question is relevant to a path containing B, which is the case if $x \diamond c \in in_B$.

- For a conditional branch in a block B which jumps to a block C if a condition $x \diamond c$ is fulfilled, and a TO prefix $P_0 \dots P_n B$ belonging to that condition, add $P_0 \dots P_n BC$ to the set $\mathcal{T}$ of complete TOs.

This algorithm certainly computes correct TOs. However, the naïve approach fails if the CFG contains loops, such as in Figure 3.4. If the start block of a TO lies before a loop, and its condition block lies after it, the algorithm will create a TO for each possible number of loop iterations, which could be infinitely many. In fact, in Figure 3.4 there *are* infinitely many TOs, yet by duplicating the loop completely we can thread all of them in finitely many steps.

We thus need a better representation of TOs. The set *Conds* used in the data flow analysis lends itself for that purpose because there is a straightforward correspondence between elements of *Conds* and TOs. We will show this after introducing some notation:

Given an execution of the above data flow analysis, we write $y \diamond c' \xrightarrow{i} x \diamond c$ if the local transfer function f_i of an instruction i transforms the condition $x \diamond c$ into $y \diamond c'$.

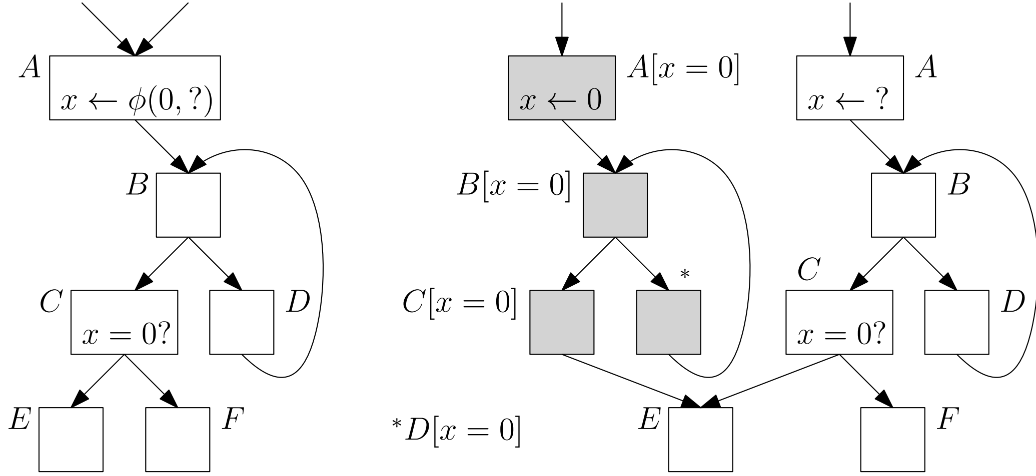


Figure 3.4. Example of a CFG with infinitely many TOs. The set $\{ABCE, ABDBCE, ABDBDBCE, \ldots\}$ is an infinite set of TOs in the left graph. Still, we can thread all these TOs in a finite number of steps by duplicating the loop completely, as shown in the right graph. To this end, we annotate the blocks using the conditions found by the data flow analysis of Section 3.3.1.

Regarding a program execution, this can be read as "If $y \diamond c'$ holds for an execution state σ, then executing i in state σ yields a state in which $x \diamond c$ holds".

For example, if i has the form `x := f(y)` we have $y \diamond f^{-1}(c) \xrightarrow{i} x \diamond c$ for all conditions concerning variable x, while for all other conditions c we have $c \xrightarrow{i} c$.

Similarly, we write $x \diamond c \xrightarrow{i} X$ for a condition inserted by f_i because of a conditional branch that jumps to block X if $x \diamond c$ is fulfilled, and $\mathbf{true} \xrightarrow{i} x \diamond c$ for a condition deemed always true by f_i. This covers all three cases in the local transfer function of the above data flow analysis.

As we prefer not to deal with individual instructions, we lift this notation to basic blocks: We write $y \diamond c' \xrightarrow{P} x \diamond c$ if the global transfer function f_P of block P transforms $x \diamond c$ into $y \diamond c'$ (possibly via some intermediate steps). We write $x \diamond c \xrightarrow{P} X$ if there is an instruction i in P so that $x \diamond c \xrightarrow{i} X$, and analogously for $\mathbf{true} \xrightarrow{P} x \diamond c$.

Again, this can be read as "If $y \diamond c'$ (or **true**) holds for an execution state σ, then $x \diamond c$ holds for transfer(σ)" and "…, then succ(transfer(P, σ)) $= X$" respectively.

Relationship between conditions and TOs

We are now ready to examine the link between conditions and TOs. We claim that a condition represents a (possibly infinite) set of TO suffixes. As a condition may be present in in_P for several blocks P, we additionally need to specify the start block in order to obtain those suffixes:

Definition 3.3.1 (TO suffixes for a condition in a block)**.** Let P be a block and let $A \in \mathit{Conds} \cup \{\mathbf{true}\}$. Then we define a set $T(A, P)$ of walks in G as follows:

$$
\begin{aligned}
T(A,P) = \{PX \mid A \xrightarrow{P} X\} \\
\cup \bigcup\{P + T(x \diamond c, Q) \mid A \xrightarrow{P} x \diamond c \wedge Q \in \mathit{succ}_P \wedge x \diamond c \in \mathit{in}_Q\}
\end{aligned}
$$

where $P + S = \{PP_1 \dots P_n \mid P_1 \dots P_n \in S\}$ for a set S.

The restriction $x \diamond c \in \mathit{in}_Q$ ensures that we only traverse paths on which the condition is relevant. $\Diamond$

We can use this definition to obtain the set of TOs starting at a block, given an execution of the above data flow analysis:

Lemma 3.3.2. *For any block P_0 of G, the set $T(\mathbf{true}, P_0)$ is a set of TOs in G.*

Proof. Let $W \in T(\mathbf{true}, P_0)$. We observe that any element of $T(\cdot, P_0)$ contains P_0 as its first element, so $W = P_0 \dots P_n X$.

From Definition 3.3.1 we obtain that W induces $C_1, \dots, C_n \in \mathit{Conds}$ so that

$$
\mathbf{true} \xrightarrow{P_0} C_1 \xrightarrow{P_1} \dots \xrightarrow{P_{n-1}} C_n \xrightarrow{P_n} X. \tag{3.7}
$$

We have to show that any execution of G that contains $P_0 \dots P_n$ as a subsequence will continue to X. We prove the following statement: For all i, if $P_{n-i} \dots P_n$ is a subsequence of an execution where C_{n-i} holds for the state σ_{n-i} at the start of the execution of P_{n-i}, then the execution will continue to X. The result then follows because **true** holds for all states, in particular for σ_0.

We perform induction over i. For $i = 0$, i.e. $n - i = n$, assume that C_n holds at the start of the execution of P_n. From Equation (3.7), we know that $C_n \xrightarrow{P_n} X$, thus after executing P_n, the condition that causes the branch to X to be taken will be fulfilled.

For $i > 0$, assume as induction hypothesis that if C_{n-i} holds at the start of the execution of P_{n-i}, the execution will continue to X. Now assume that $C_{n-(i+1)}$ holds at the start of the execution of $P_{n-(i+1)}$. From Equation (3.7) we have that C_{n-i} holds after the execution of $P_{n-(i+1)}$ and hence at the start of P_{n-i}, and the result follows with the induction hypothesis. $\square$

Threading algorithm

The reward for this very technical work is that we can now use infinite sets of TO suffixes in Algorithm THREAD by representing them using finitely many conditions.

Instead of a set $\mathcal{S}(P')$ of TO suffixes, we annotate each block duplicate with a set $\mathcal{C}(P')$ of conditions fulfilled at the start of this block. We define an equivalent to $\mathcal{S}'(P')$ that takes into account the TOs starting in P:

$$
\mathcal{C}'(P') = \mathcal{C}(P') \cup \{c \mid \mathbf{true} \xrightarrow{P} c\} \tag{3.8}
$$

Finally, we redefine the *ShortenFilter* function which transforms sets of TO suffixes along a control flow edge to deal with sets of conditions instead:

$$\mathit{ShortenFilter}'_{PQ}(C) = \begin{cases} \emptyset & \text{if } \exists c \in C : c \xrightarrow{P} X \wedge X \neq Q \\ \{c' \mid c \in C \wedge c \xrightarrow{P} c' \wedge c' \in \mathit{in}_Q\} & \text{otherwise} \end{cases} \tag{3.9}$$

This leads to Algorithm THREAD', which is the algorithm we implemented in libFIRM and which we describe in Chapter 5:

Algorithm 2: Algorithm THREAD'

input : a graph G and the results of the data flow analysis of Section 3.3.1

```
1  while ∃P'Q' ∈ E(G') : C(Q') ≠ ShortenFilter'_PQ(C'(P')) do
2  |  if ∃ duplicate Q* of Q with C(Q*) = ShortenFilter'_PQ(C'(P')) then
3  |  |  Q* ← that duplicate
4  |  else
5  |  |  Create a duplicate Q* of Q with C(Q*) = ShortenFilter'_PQ(C'(P')) ;
   |  |  /* Q* has no incoming edges and an outgoing edge Q*R for
   |  |     all edges QR in G                                        */
6  |  end
7  |  Replace the edge P'Q' by P'Q* ;
8  end
9  foreach block P' with ∃c ∈ C'(P') : c -P-> Q do
10 |  Remove all outgoing edges of P' except the one to (a duplicate of) Q ;
11 end
```

As with Algorithm THREAD, the definition of *ShortenFilter'* allows us to postpone the edge removal until after the While loop.

In order to demonstrate that Algorithms THREAD and THREAD' perform a similar transformation, we show a connection between the two variants of *ShortenFilter* (Equations (3.3) and (3.9)), namely that they are interchangeable with regard to the TO suffix function $T(c, P)$ of Definition 3.3.1:

Lemma 3.3.3. *For an edge PQ in G, we have*

$$T(\mathit{ShortenFilter}'_{PQ}(C), Q) = \mathit{ShortenFilter}_Q(T(C, P)),$$

where we use the shorthand notation $T(C, Q) := \bigcup_{c \in C} T(c, Q)$.

Proof. For $X \neq Q$ we have $\exists c \in C : c \xrightarrow{P} X \iff \exists c \in C : PX \in T(c, P)$, thus the case distinctions in Equations (3.3) and (3.9) both yield the same case.

For the first case both definitions yield the empty set. For the second case, let $c \in C$ and show $T(\mathit{ShortenFilter}'_{PQ}(\{c\}), Q) = \mathit{ShortenFilter}_Q(T(c, P))$, i.e. consider C element-wise.

Unfolding Definition 3.3.1, we have

$$\mathit{ShortenFilter}_Q(T(c, P)) = \bigcup \{T(c', Q) \mid c \xrightarrow{P} c' \wedge c' \in \mathit{in}_Q\}, \tag{3.10}$$

as $\mathit{ShortenFilter}_Q$ removes two-element walks, removes the first element of each walk and filters out walks whose second element is not Q.

On the other hand, eliminating the shorthand notation from above, we have

$$T(\mathit{ShortenFilter}'_{PQ}(C), Q) = \bigcup \{T(c', Q) \mid c' \in \mathit{ShortenFilter}'_{PQ}(\{c\})\} \tag{3.11}$$

and using the second case of Equation (3.9), we see that the right-hand sides of Equations (3.10) and (3.11) are equal. □

3.4. Interaction with other optimizations

Jump Threading is not the only optimization applied by a modern compiler. It is therefore important to examine how this optimization interacts with other optimizations applied before or after it. In this chapter, the theoretical aspect of the interactions is emphasized. Nevertheless, for practical reasons, we study the optimization variants implemented in libFirm.

3.4.1. If Conversion

Many instruction set architectures support *predicated instructions*, which are executed only if some condition is met (e.g. a processor flag is set), otherwise ignored. In particular, they can be used as an alternative to conditional branches if the code that is executed conditionally is short enough.

Intel x86 and x64 do not allow every instruction to be predicated, but provide the `CMOVcc` (conditional move) and `SETcc` (conditional set bit) instruction classes. These two predicated instruction types can be used to implement *speculative execution*, where the two possible values of a variable are both calculated, then the predicate is calculated and the correct value is moved into the variable's storage position. This requires that both values are computable without side effects.

In libFirm, the `Mux` node type is used for a value that takes one of two possible values depending (directly) on a condition. A ϕ node can be seen as a more generalized form of this node, as it takes one of several values depending on control flow, which in turn may depend on a condition. If the control flow is simple enough, a ϕ node can be converted to a `Mux` node.

An If Conversion opportunity consists of a "diamond structure": a block B with two predecessors B_1 and B_2 that are both control dependent on the same block C. Additionally, C must contain a conditional branch with condition *cond* (and thus have two successors), and B, B_1, B_2 need to be mutually distinct.

If all ϕ nodes in B can be converted to `Mux` nodes[3], they are replaced by `Mux` nodes in block C that depend on *cond*. The conditional branch of C can be removed and B, B_1, B_2, C merged into one block. The transformation is illustrated in Figure 3.5.

We recall the definition of the control dependence relation between two blocks X and Y in a CFG:

Definition 3.4.1. *Y is control-dependent on X if Y does not not postdominate X and there exists a path P from X to Y so that Y postdominates all elements of P except X.* ♢

In our case there exist two such paths, one from C to B_1 and one from C to B_2. They lead over different successor blocks of C, else one successor of C would be postdominated by both B_1 and B_2, thus $B_1 \operatorname{postdom} B_2$ or $B_2 \operatorname{postdom} B_1$, a contradiction because B_1 and B_2 are two predecessors of the same block.

For $i \in \{1, 2\}$, let C_i be the successor of C over which the path to B_i leads. Then we have $B_1 \operatorname{postdom} C_1$ and $B_2 \operatorname{postdom} C_2$. Furthermore, all C_1-B_1-paths and all C_2-B_2 paths are disjoint.

Interaction between Jump Threading and If Conversion

Both If Conversion and Jump Threading seek to reduce the number of conditional branches, but the methods used are different: If conversion converts control dependencies to data dependencies, allowing it to eliminate control flow and ϕ functions, which usually reduces code size. Although the conditional branch is eliminated, its condition must still be evaluated for the predicated move instruction. Jump Threading usually increases code size, but removes conditional branches along with their condition, at least on some (but sometimes on all) code paths.

The most straightforward way to influence the interaction of the two optimizations is to change the order in which they are applied. However, both optimizations, when applied first, affect the respective other one:

Threading opportunities destroyed by If Conversion We examine which FIRM nodes removed or modified by If Conversion would have been optimized away by Jump Threading.

Firstly, If Conversion removes the conditional branch in block C, but retains the condition *cond* on which it depends (e.g. a comparison) as input to the `Mux` node. The condition will now be evaluated on all code paths containing C, even if Jump Threading could have made the evaluation redundant on some of these paths. Figure 3.5 shows such a situation.

If *cond*'s evaluation is costly, this might have a negative effect on performance. However, values which Jump Threading can statically evaluate are normally calculated by simple operations (e.g. addition or subtraction of a constant). Additionally, If

[3] Whether a conversion is possible is backend specific, although to satisfy the "no side effects" condition, ϕ nodes with mode `M` can never be converted.

Conversion removes the conditional branch on all code paths, while Jump Threading might remove it only for a subset thereof. It seems therefore reasonable that these kinds of CFG structures should be handled by If Conversion.

Secondly, If Conversion modifies the ϕ nodes of block A, replacing them by `Mux` nodes in C. The two possible values of the `Mux` now lie on the same code path and are no longer distinguishable using control flow information. Therefore, Jump Threading usually cannot thread across `Mux` nodes. If some other conditional branch depended on such a ϕ node, e.g. by comparing its value with a constant, a Threading Opportunity might be lost.

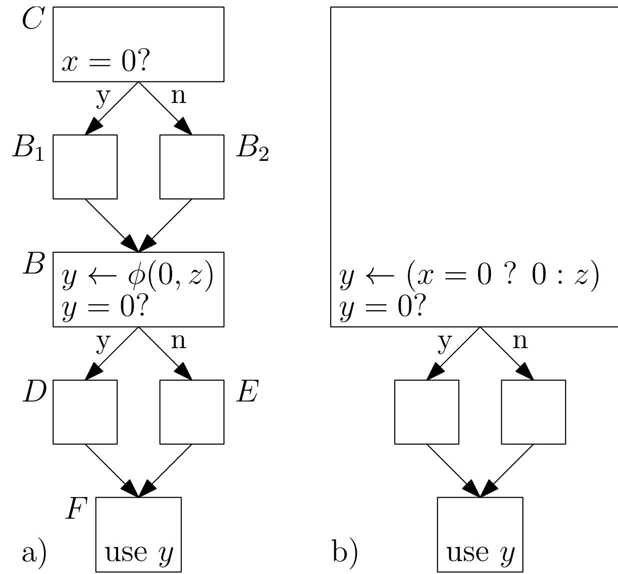


Figure 3.5. Example of If Conversion destroying a Threading Opportunity
a) In the original graph, the predecessors B_1 and B_2 of B are both control dependent on C. This constitutes an If Conversion opportunity. Additionally, the graph contains a TO B_1BD.
b) Applying If Conversion to the graph transforms the ϕ in B into a conditional move in C and merges the blocks between B and C. The two possible values of y are not discernible via control flow any more, thus the graph contains no TO any more.

If Conversion opportunities destroyed by Jump Threading Jump Threading can destroy opportunities for If Conversion either by modifying control dependence or by introducing side effects. Given an If Conversion opportunity with blocks B, B_1, B_2, C, we consider the following situations (omitting obvious symmetrical cases for each situation), which are illustrated in Figure 3.6:

Situation 1: C is the condition block of a TO. Then Jump Threading creates a duplicate C' and removes one of the two outgoing edges of C'. Wlog. let C_1 be the remaining successor of C'.

B_1 and B_2 are both still control dependent on C. But C_1, having gained a new predecessor C', now lies in the dominance frontier of $\{C, C'\}$. Because the predecessor B_2 of B is reachable from C_1 but not from C_2, B lies in the iterated dominance frontier of $\{C, C'\}$.

When reconstructing SSA form, Jump Threading therefore inserts ϕ functions in B for all variables defined in C (and possibly in other duplicated blocks) if such ϕ functions are not already present.

If one of the blocks duplicated by Jump Threading contains side effects, a Memory ϕ must be inserted into B, preventing all ϕ nodes in B from being optimized by If Conversion.

Situation 2: There is a TO that starts with the edge B_1B. Then Jump Threading reroutes the edge B_1B to a duplicate B' of B. Thus B_1 and B_2 are no longer predecessors of the same block.

The ϕ functions in B are removed because B and B' have only one predecessor each. Instead, new ϕ functions for the corresponding variables are inserted at the iterated dominance frontier of $\{B, B'\}$.

While it is possible that If Conversion can optimize those ϕ functions instead, this is not guaranteed: Firstly, the predecessors of a block in which they are inserted need not be control dependent on the same node. Secondly, if B or some other block of the TO contains a side effect, a Memory ϕ will have to be inserted into the block, again preventing If Conversion to act on the block.

Situation 3: There is a TO whose start block lies between C_1 (exclusive) and B_1 and which contains B_1 (except as target block).

Then Jump Threading creates a duplicate B_1' of B_1, and both B_1 and B_1' are reachable from C_1. Therefore, neither postdominates C_1 and neither is control dependent on C, preventing ϕ nodes in B from being optimized by If Conversion.

Conclusion We have shown that Jump Threading and If Conversion influence each other no matter which one is performed first. In Chapter 6, we have therefore included both variants in the benchmarks.

If one instead wanted to modify Jump Threading to avoid TOs that interfere with If Conversion, one would have to duplicate almost the entire logic of If Conversion including the backend-specific parts. In this case, combining the two optimizations could be a more favorable solution.

From a theory standpoint, If Conversion lends itself to be performed first: Because of the restrictions on the control flow it considers, especially the exclusion of side effects on either control flow path, it shows its strengths best inside small loops. These are usually the most performance critical parts of the code[4] and both the control flow elimination and the code size reduction performed by If Conversion should be most beneficial there.

[4]The anecdotal "90—10 rule" of programming states that 90 % of processor time is spent in loops. Modern processors are also optimized for them, for example by including a small μop cache for loops.

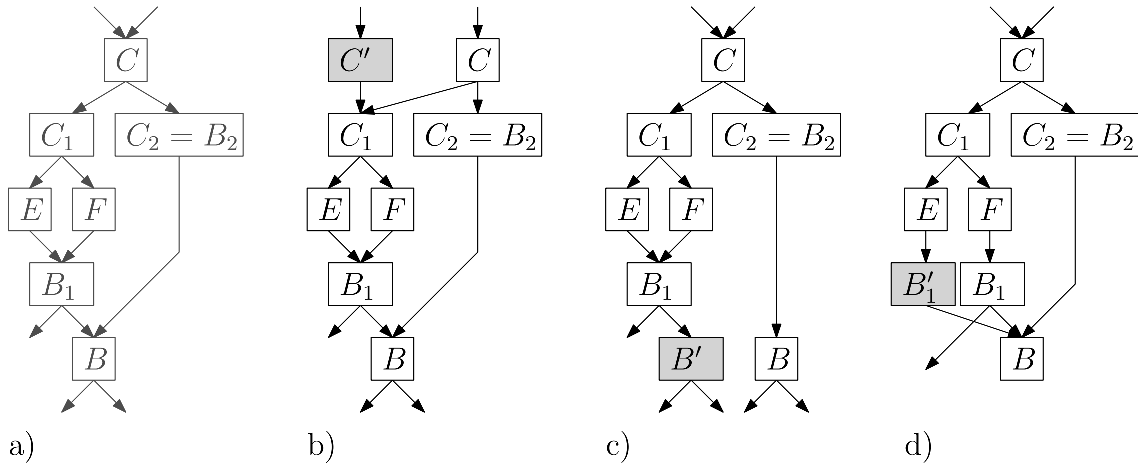


Figure 3.6. Examples of Jump Threading destroying If Conversion opportunities
a) The original graph. The predecessors B_1 and B_2 of B are both control dependent on C, in particular they each postdominate one of C's successors C_1 and C_2.
b) Threading a TO ending with $\ldots CC_1$ duplicates C. For any variable definition in C that is duplicated into C', a ϕ must be inserted in B. If C contains side effects, a Memory ϕ is inserted into B.
c) Threading a TO starting with $B_1B\ldots$ duplicates B. B_1 and B_2 are no longer predecessors of the same block.
d) Threading the TO EB_1B duplicates B_1. B_1 is no longer control dependent on C, as it does not postdominate any successor of C.

3.4.2. Block scheduling

The block scheduling phase of a compiler computes a linearization of the blocks in the CFG. As Jump Threading modifies control flow, and in particular adds new blocks, it affects block scheduling.

When computing the block schedule, one quantity we can try to maximize is the number of fall-throughs, i.e. the number of times a block is placed directly before a control flow successor. In this case, an unconditional branch at the end of the block can be eliminated.

Naturally, at most one predecessor of a block can be scheduled in this way. A positive effect of Jump Threading is that it often reduces the number of predecessors in both the original and the duplicate blocks. We consider the case of threading a single TO from Definition 3.2.3:

Observation 3.4.2. *When threading a TO $T := P_0 P_1 \dots P_n$ using Definition 3.2.3, the duplicate blocks $P_1^*, \dots, P_{n-1}^*$ each have only one control flow predecessor.*

Additionally, threading T reduces the number of control flow predecessors of the original block P_1 by one. For example, if P_1 previously had two control flow predecessors P_0 and P, both P_1 and P_1^* now have one control flow predecessor, and both P_0 and P can now fall through to their respective successor.

On the other hand, Jump Threading adds an additional predecessor to blocks that lie outside the TO but have a predecessor that lies in the TO. Nevertheless, there are cases in which block duplication can increase the number of fall-throughs, as shown in Figure 3.7.

The block duplication and edge rerouting performed by Jump Threading may also result in edges leading from a block P' with one successor to a block Q' with one predecessor. In this case, P' and Q' can be merged to form a single basic block, which is done by the control flow optimization pass. Nonetheless, we mention this in the section on block scheduling because even if the control flow optimization pass did not run, the mergeable blocks would be scheduled after another.

As shown in Figure 3.7, this phenomenon may affect not only duplicate blocks, but also original blocks, in particular the original block P_1 which has one less predecessor after threading.

We conclude by showing a condition under which Jump Threading does not need to duplicate any blocks, but can merge the duplicated instructions right into the start block of the TO.

Lemma 3.4.3. *Let $T := P_0 P_1 \dots P_n$ be a TO in G, and let G' be the graph obtained by threading T according to Definition 3.2.3.*

Then the duplicate blocks $P_1^*, \dots, P_{n-1}^*$ in G' each have one control flow predecessor and one control flow successor if and only if $P_1, \dots, P_{n-2}$ each have one control flow successor in G.

Proof. The number of control flow predecessors follows from Observation 3.4.2. From the definition of $E(G')$ in Definition 3.2.3, we see that $P_1^*, \dots, P_{n-2}^*$ have the same

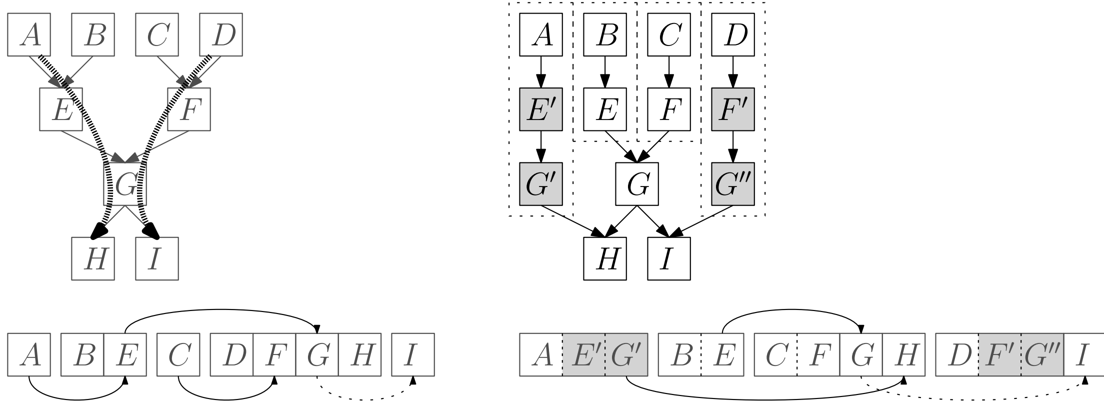


Figure 3.7. Left: A graph with two TOs ($AEGH$ and $DFGI$), and a linearization of its blocks that maximizes the number of fall-throughs. Dotted lines indicate conditional branches, while solid lines indicate unconditional branches.
Right: The same graph after threading the two TOs. All duplicate blocks have only one control flow predecessor. The original blocks E and F that previously had two predecessors each now have only one predecessor, while H and I each gain an additional predecessor. In the linearization, we can save one unconditional branch. Additionally, several basic blocks can now be merged, as indicated by a dotted line, reducing the number of basic blocks from 9 to 7.

number of successors as their respective original block. The duplicate P^*_{n-1} of the condition block always has only one successor. □

Corollary 3.4.4. *If and only if $P_0, P_1 \dots P_{n-2}$ each have one successor in G, all duplicate blocks created by Definition 3.2.3 can be merged into the start block P_0.*

Proof. With Lemma 3.4.3, we have that all duplicate blocks can be merged to form a block P with one control flow predecessor. If P_0 has one control flow successor, P and P_0 can be merged as well. □

This effectively eliminates the need to create any block duplicates for certain TOs. The current Jump Threading implementation of libFirm makes use of that, as we will see in Section 4.3.

4. Analysis of existing algorithms

In this section, we sketch the Jump Threading implementations of the currently biggest two compilers and examine the implementation in libFirm more closely.

4.1. LLVM

LLVM implements a conservative version of Jump Threading. As explained in the source comment, it "looks at blocks that have multiple predecessors and multiple successors. If one or more of the predecessors of the block can be proven to always jump to one of the successors, we forward the edge from the predecessor to the successor by duplicating the contents of this block." [5, l. 64–68] In terms of our theoretical framework, the length of TOs that are being considered is restricted to 3.

LLVM avoids creating infinite loops by not threading if the target block is equal to the condition block [5, l. 1456].

It also does not create irreducible loops from natural loops. To this end, any TO whose condition block is a loop header (as identified by a backedge leading to it) is discarded [5, l. 1464]. This guarantees that no loop header is duplicated and hence no irreducible loop is created, as seen in Lemma 3.2.7. However, the authors are aware that this is a conservative heuristics and that there are cases in which threading across a loop header would be profitable [5, l. 332–339].

The cost model is based on the estimated cost of duplicating the condition block (which in this case is the only block that needs to be duplicated). The maximum block size is a very conservative 6 instructions. However, threading a Switch or an indirect jump incurs a bonus of 6 and 8 instructions respectively that is subtracted from the cost.

The optimization pass consists of a loop that terminates only after the graph has not been changed during an iteration. However, due to the conservative nature of the threading, there is no obvious danger of an endless loop.

4.2. GCC

GCC's jump threading implementation for SSA-based graphs is implemented in three source files, of which [6] is the main file concerned with finding threading paths. Frequently, the original node "FSM (finite state automaton) jump threading" of the optimization is present in the code. This hints at one application for Jump Threading, which is to optimize a certain implementation pattern of finite state machines shown in Figure 4.1.

```
i = 0; /* index in the input string */
state = INITIAL_STATE;
while (state != END_STATE) {
    switch (state) {
        case 1:
           state = ...;
           break;
        case 2:
           state = ...;
           break;
        ...
    }
    i++;
}
```

Figure 4.1. Example for the implementation of a Finite State Machine in C. Without Jump Threading, the `break` statements jump to the condition block of the loop. With Jump Threading, not only can the condition `state != END_STATE` possibly be statically evaluated, but even the **`switch`** statement, effectively making the **`break`** statements jump directly to the next **`case`**.

GCC's implementation differs from LLVM's in a number of ways, most prominently it does not limit itself to threading paths of length 3. It also does not change the graph iteratively but collects threading paths, then in a second pass modifies the graph. The threading paths are sequences of basic blocks as described in Chapter 3.

Similar to our implementation and the one in libFirm, GCC follows data dependency edges. However, it requires that only a single path exists between a definition and a usage of a variable, i.e. it does not duplicate control flow. In this regard it is stricter than our implementation, which allows arbitrary control flow between definition and usage, but less strict than the current libFirm implementation, which allows no blocks at all between definition and usage.

The cost model is also based on the estimated code size increase. There are several parameters used to control the amount of code duplicated by Jump Threading, among which are the maximum number of threading paths (default 50) and the maximum number of blocks (10) and statements (100) to duplicate.

Although there is a limit on the number of threading paths, the optimization does not try to find the most profitable paths first; instead, the graph is traversed block by block, looking for threading paths to the current block.

GCC, like LLVM but unlike libFirm, stores loops as natural loops internally. Thus creating irreducible loops would pose a disadvantage to later optimizations. GCC's Jump Threading therefore employs a heuristics to detect possible irreducible loops, preventing them from being created except in special cases where it deems the loss negligible:

Assume that each loop has exactly one backedge LH, where L is called the loop *latch* and H is called the loop *header*. (The natural loop is then formed by all blocks that can reach L without going through H.) Then a threading path would create an irreducible loop if *a)* it contains the backedge LH of a loop, and *b)* its target block lies inside the loop, and *c)* its target block does not dominate the loop latch [6, l. 282–284].

This alone would not be sufficient to detect an irreducible loop being created. For example, the TO in Figure 3.1 *a)* does not contain the backedge of a loop but still causes an irreducible loop. However, GCC additionally requires that threading paths do not cross loop boundaries, i.e. all blocks of a threading path except for the start block must lie in the same loop. These conditions together are sufficient to detect the creation of irreducible loops.

4.3. libFirm

Jump Threading was added to libFIRM in September 2006 (the initial libFIRM revision dates from May 2000), and was then simply called “partial condition evaluation”. In July 2009, it was renamed to “Jump Threading”. Its basic functionality has not changed much since the initial version. In the following we will also refer to the existing implementation as JT_{Old}.

4.3.1. Confirm nodes and optimization pass

Both JT_{Old} and the new implementation make use of `Confirm` nodes, a special feature of libFIRM. A `Confirm` node annotates an arbitrary value with a condition it is guaranteed to fulfil in all basic blocks dominated by the `Confirm` node’s block.

A dedicated optimization pass inserts these nodes, replacing usages of the value node by usages of the `Confirm` node. For example:

```
if (i == 2) {
    foo(i);
} else {
    bar(i);
}
baz(i);
```

would be turned into

```
if (i == 2) {
    foo(Confirm(i, = 2))
} else {
    bar(Confirm(i, ≠ 2))
}
baz(i); // unchanged: neither Confirm node dominates this block
```

Optimization passes which do not need the additional semantics of a `Confirm` node simply ignore it by viewing the data dependency edge leading to a `Confirm` node as going to its value. A cleanup phase removes all `Confirm` nodes before code generation.

4.3.2. The existing Jump Threading algorithm

JT_{Old} transforms the graph in an iterative fashion, finding and threading one TO to each block in each iteration and terminating when no TO was found in one iteration. The essential parts of the implementation are presented in a simplified fashion in Algorithms Iterate, ThreadJumpsToBlock and FindAndThreadTO.

Algorithm 3: Algorithm Iterate, implemented in the `opt_jumpthreading` function [7, l. 635–665]

```
1  Ensure that graph contains no critical edges;
2  repeat
3  |  changed ← false;
4  |  foreach block ← graph.basicblocks do
5  |  |  changed ← changed ∨ ThreadJumpsToBlock(block)
6  |  end
7  until ¬ changed;
```

Algorithm 4: Algorithm ThreadJumpsToBlock, implemented in the `thread_jumps` function [7, l. 560–633]

```
   input  : block, the candidate for a target block of a TO
   output : Whether the graph was changed
1  if block has more than one control flow predecessor then
2  |  return false                                  // Limitation
3  end
4  if predecessor of block is not a ProjX with a Cond then
5  |  return false                  // Limitation:  Switch nodes
6  end
7  cond ← the condition causing the jump to block to be taken ;
8  var ← the variable compared against cond;
9  startblock ← FindAndThreadTO(var, cond, block.predecessor, block);
10 if startblock ≠ ⊥ then
11 |  Keep block alive ;
   |  // might have created an infinite loop
12 |  return true
13 end
14 return false
```

Algorithm 5: Algorithm FindAndThreadTO, implemented in the `find_candidate` [7, l. 467–545] and `find_const_or_confirm` [7, l. 397–465] functions

```
   input  : var, the variable to be compared against cond
   input  : cond, the condition to compare against
   input  : block, the currently first block of the TO suffix we are constructing
   input  : target-block, the target block of the TO suffix we are constructing
   output : The start block of the constructed TO, or ⊥ if no TO is found
1  if var has been visited before then
2  |  return ⊥ // Avoid endless loops
3  end
4  switch var do
5  |  case Const or Confirm that fulfils cond do
   |  |  // block is the start block of a TO
6  |  |  Reroute the outgoing edge of block to target-block;
7  |  |  Split the original predecessor edge of target-block, which may have
   |  |    become critical;
8  |  |  return block
9  |  end
10 |  case ϕ with > 1 argument in block do
11 |  |  foreach argument no. i of the ϕ do
12 |  |  |  start-block ← FindAndThreadTO(argument i of the ϕ, cond, pred. i of
   |  |  |    block, target-block);
13 |  |  |  if start-block ≠ ⊥ then
14 |  |  |  |  Copy contents of block into start-block;
   |  |  |  |  // Limitation: Do not look at any more predecessors
   |  |  |  |     of the ϕ during this iteration
15 |  |  |  |  return start-block
16 |  |  |  end
17 |  |  |  return ⊥
18 |  |  end
19 |  end
20 end
21 return ⊥
```

Analyzing Algorithm FindAndThreadTO, we see that once a recursive call returns a block other than $\perp$, the recursion is terminated and that block is returned to the caller of the outermost invocation. That return value is the start block of the TO this algorithm implicitly constructs:

Lemma 4.3.1. *If an execution of Algorithm* FindAndThreadTO *returns a block $P \neq \perp$, then P is the start block of a TO, and that TO is induced by the recursive calls of Algorithm* FindAndThreadTO*.*

Proof. The only base case not returning $\perp$ is in Line 8, while the only recursive invocation is in Line 12.

Let ThreadJumpsToBlock(X) $\rightarrow$ FindAndThreadTO(v_n, P_n) $\rightarrow \ldots \rightarrow$ FindAndThreadTO(v_0, P_0) be the recursive invocations leading to the base case FindAndThreadTO(v_0, P_0), along with their respective variable and block parameters. We only look at the invocation sequence, ignoring the control flow changes made by the algorithm.

We show that $P_0 \ldots P_n X$ is a TO. To this end, assume $P_0 \ldots P_n$ occurs as a subsequence in an execution of the CFG. We claim that for $0 \leq i \leq n$, after executing block P_i, v_i fulfils the condition cond that causes the branch in P_n to lead to X. The result then follows for $i = n$ because v_n is the variable used in the conditional branch.

Note that cond is passed along as an input parameter to all calls of FindAndThreadTO. Thus for $i = 0$, v_0 fulfils cond because of the `If` statement guarding Line 8.

For $i > 0$, assume that after executing block P_{i-1}, v_{i-1} fulfils cond. The call FindAndThreadTO(v_{i-1}, P_{i-1}) is a recursive call and thus happens in Line 12. Hence, v_i is a ϕ whose argument for predecessor block P_{i-1} is v_{i-1}. In the execution, P_i is entered via the edge $P_{i-1}P_i$, therefore we have $v_i = v_{i-1}$ and v_i fulfils cond as well. □

This TO is however never made explicit (unlike for example in GCC) as it is threaded immediately.

JT_{Old} also requires that the CFG is free of critical edges at all times. This is vital for correctness: In Line 14 of Algorithm FindAndThreadTO, the contents of the current block are duplicated not into a new block, but into the start block of the TO. This is correct if and only if the conditions of Corollary 3.4.4 are met, which is the case:

Lemma 4.3.2. *If the graph is free of critical edges, then in any TO $P_0 P_1 \ldots P_n$ found by JT_{Old}, all blocks P_i except the condition block P_{n-1} and the target block P_n have only one control flow successor.*

Proof. Suppose for the sake of contradiction that there is one block P_i, $i < n - 1$, with multiple control flow successors. Let P_j be the successor block from which FindAndThreadTO was called recursively. As FindAndThreadTO is only

recursively called when the current node is a ϕ with more than one argument[1], there is such a ϕ node in P_j. Therefore, P_j has more than one predecessor, and P_iP_j is a critical edge, a contradiction. □

We need to verify that the graph is free of critical edges at all times. Algorithm Iterate ensures this only at the beginning; it remains to show that this property holds after each execution of FindAndThreadTO:

Lemma 4.3.3. FindAndThreadTO *does not introduce new critical edges in a graph.*

Proof. The only control flow change is the rerouting of the start block's outgoing edge to the target block. The rerouted edge is not critical as the start block has only one successor. As the target block previously had only one predecessor, the only edge that could have become critical is the edge from that predecessor to the target block, which is split. □

From Lemma 3.2.5, we know that splitting the critical edge does not influence the number of TOs in G. However, it does influence the number of TOs that Algorithm FindAndThreadTO finds, as we will see in Section 4.3.4.

4.3.3. Limitations

There are a number of restrictions on the TOs that JT_{Old} finds:

- The target block of a TO must not have more than one control flow predecessor. This is to make sure that there are no ϕ functions in the target block that would have to be modified (only new ones are created).
- When finding TOs, the algorithm follows only data dependency edges that have a corresponding control flow edge; i.e. it does not skip over blocks. This makes the algorithm much easier to implement because no control flow needs to be duplicated, but also restricts it quite a bit.
- Each invocation of FindAndThreadTO returns once it has found a TO start block. For two TOs with a shared suffix but with different start blocks, the contents of all blocks (including those in the shared suffix) are therefore duplicated twice, as they are copied once into each start block. This corresponds to threading the TOs one after another as illustrated in Figure 3.2 of Section 3.2.4.

Some less severe limitations that could be overcome without drastically modifying the algorithm:

[1] At the time of writing, the implementation does not check that the ϕ has more than one argument. However, ϕ nodes with one argument only occur in degenerate graphs.

- The algorithm does not process addition, subtraction and/or multiplication nodes.
- The algorithm does not deal with Switch branches, only with simple if-then-else branches.

4.3.4. Non-termination

The biggest issue with JT_{Old} (and the initial motivation to start this thesis) is that it is not guaranteed to terminate. More specifically, the algorithm will sometimes unroll endless loops. There is a very simple example program that causes JT_{Old} to loop endlessly when compiled with `cparser -O0 -fthread-jumps`:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i = rand();

    if (i < 10) {
        i = 42;
        goto inloop;
    }

    while (i < 100) {
inloop:
        printf(".");
    }
}
```

Interestingly enough, not even one iteration of the loop in Algorithm Iterate is completed. Instead, it is the iteration over all blocks inside that loop which does not terminate: When splitting the critical edge in Line 7 of Algorithm FindAndThreadTO, a new block is created which in this case happens to be the target block of a TO, creating yet another critical edge split block, etc.

However, not splitting critical edges is not an alternative, as we have seen that the algorithm's correctness depends on the absence of critical edges.

In the end, the infinite loop is only a symptom of the algorithm's failure to restrict the increase in code size, as it can easily be seen that this happens only if the original graph contained infinitely many TOs to begin with.

5. Implementation

The Jump Threading algorithm developed in Chapter 3 was implemented in libFIRM in about 1200 lines of C. Some specifics of the implementation are noted here.

5.1. General structure

The data flow analysis introduced in Section 3.3.1 is performed by the `get_conds` function. Each condition is represented by an instance of the `jt_cond` structure. Because the libFIRM intermediate representation is based on explicit data dependency edges, we do not need to perform a classical data flow analysis. Instead, we start the analysis at conditional branches and follow data dependency edges until either reaching a node where a condition is fulfilled or until exceeding a cost threshold (see Section 5.6).

For finding the necessary block duplicates, we implement Algorithm THREAD' introduced in Section 3.3.3. The algorithm is implemented in the `get_succ_duplicates` function. The *ShortenFilter* function is integrated into this function, using the `relevant_conds` and `forbidden_conds` annotations of control flow edges.

5.2. Wrap-around intervals

To represent conditions, we would like to re-use the lattice $\mathit{Conds} := \{x \diamond c \mid x \in \mathit{Vars}, \diamond \in \mathit{Rels}, c \in \mathit{Consts}\}$ from Section 3.3.1. We need a way to represent elements thereof, in particular elements of *Rels*, in a meaningful way.

We have to take into account that libFIRM's data types are machine-oriented and have defined wrap-around semantics. For example, assuming $\mathbb{Z}$ as a data type, as one would do in a theoretical model, the local transfer function of the instruction $x \leftarrow y - 42$ would transform the cond $C_1 : x > 0$ into $C_2 : y > 42$, and we would have $C_1 \Leftrightarrow C_2$ as desired.

However, in reality x and y are of a datatype $\mathbb{I}$ that wraps around at $\mathbb{I}_{max}$ to some constant $\mathbb{I}_{min}$. It is still true that $C_2 \Rightarrow C_1$, the necessary condition for correctness, but equivalence now holds only for $C_2' := y > 42 \vee y < \mathbb{I}_{min} + 42$. In order to fully process addition, subtraction and similar instructions, we need a better representation for relations.

Definition

In the following, we restrict ourselves to integral datatypes with wrap-around semantics, such as integral numbers or pointers. We also consider Boolean values, which are a native data type of libFirm, but do not consider any operations on them, thus wrap-around semantics are not needed for them.

The exclusion of floating-point numbers is reasonable, as their semantics is much more complicated than those of integers and even depends on runtime factors (the FPU control word controls e.g. arithmetic precision).

Let the data type's value range be an interval $\mathbb{I} := [\mathbb{I}_{min}, \mathbb{I}_{max}]$, with $\mathbb{I}_{min} < \mathbb{I}_{max}$, and let arithmetic operations $\oplus$, $\ominus$ be defined on $\mathcal{I}$ so that $\mathbb{I}_{max} \oplus 1 = \mathbb{I}_{min}$ and $\mathbb{I}_{min} \ominus 1 = \mathbb{I}_{max}$.

We represent each Cond as $x \in [\![a, b]\!]$, where $[\![a, b]\!]$ is a *wrap-around interval*. The set of wrap-around intervals is

$$Int_{Wrap} := \{[a, b] \mid \mathbb{I}_{min} \leq a \leq b \leq \mathbb{I}_{max}\} \cup \{[\mathbb{I}_{min}, a] \dot\cup [b, \mathbb{I}_{max}] \mid \mathbb{I}_{min} \leq a < b \leq \mathbb{I}_{max}\}.$$

We write both types of intervals as $[\![a, b]\!]$ with

$$[\![a, b]\!] = \begin{cases} [a, b] & \text{if } a \leq b \\ [\mathbb{I}_{min}, b] \dot\cup [a, \mathbb{I}_{max}] & \text{if } a > b \end{cases} \tag{5.1}$$

This allows the intervals to be represented internally in an efficient way, using just two values of the same data type.

Note that the empty set is not contained in Int_{Wrap}. This is a side effect of the internal representation. The empty set corresponds to an "always false" condition in the program, which we can handle separately, thus an explicit representation of the empty set is not needed.

As comparison relations, we allow any subset of $\{<, =, >, \text{unordered}\}$, as libFirm does. Intervals are easily constructed from comparisons against a constant. For example, the set of values that fulfil the condition $x < 10$ are represented by the interval $[\![\mathbb{I}_{min}, 9]\!]$. The function `get_interval`, given a comparison against a constant value, computes the interval of values for which the comparison evaluates to `true`.

Membership

Equation (5.1) translates directly to the following interval membership function for a single value $x \in \mathbb{I}$:

$$x \in [\![a, b]\!] \Leftrightarrow \begin{cases} a \leq x \leq b & \text{if } a \leq b \\ x \leq b \lor x \geq a & \text{if } a > b \end{cases}$$

This function is implemented as `is_tv_in_interval`.

Subset relation

To determine whether one jump subsumes another, we need a subset relation on Int_{Wrap}. For example, if a condition $x \geq 0$ is reached from the true branch of a jump that is guarded by the condition $x \geq 10$, we know that the jump will be taken on this path because $[10, \mathbb{I}_{max}] \subseteq [0, \mathbb{I}_{max}]$.

In general, given two intervals $[\![a, b]\!]$ and $[\![a', b']\!]$, we want to determine whether $[\![a, b]\!] \subseteq [\![a', b']\!]$. Unfolding Equation (5.1) yields four cases:

Case 1: $a \leq b \wedge a' \leq b'$.

$[a, b] \subseteq [a', b'] \iff a' \leq a \wedge b \leq b'$:

"$\Leftarrow$" Let $x \in [a, b]$. From $a \leq x$ and $a' \leq a$ we have $a' \leq x$. From $x \leq b$ and $b \leq b'$ we have $x \leq b'$. Thus $x \in [a', b']$.

"$\Rightarrow$" Assume $a' > a$ or $b > b'$. In the first case, $a \in [a, b]$, but $a \notin [a', b']$. In the second case, $b \in [a, b]$, but $b \notin [a', b']$.

Case 2: $a \leq b \wedge a' > b'$.

$[a, b] \subseteq [\mathbb{I}_{min}, b'] \dot{\cup} [a', \mathbb{I}_{max}] \iff a' \leq a \vee b \leq b'$: As the union is disjoint, the left-hand side is equivalent to $[a, b] \subseteq [\mathbb{I}_{min}, b'] \vee [a, b] \subseteq [a', \mathbb{I}_{max}]$. From the first case, we know this is equivalent to $(\mathbb{I}_{min} \leq a \wedge b \leq b') \vee (a' \leq a \wedge b \leq \mathbb{I}_{max})$. Two of these inequalities are always true and can be removed, yielding the desired result.

Case 3: $a > b \wedge a' > b'$.

$[\mathbb{I}_{min}, b] \dot{\cup} [a, \mathbb{I}_{max}] \subseteq [\mathbb{I}_{min}, b'] \dot{\cup} [a', \mathbb{I}_{max}] \iff a' \leq a \wedge b \leq b'$: As both unions are disjoint, and because each two of the intervals share an endpoint, the left-hand side is equivalent to $[\mathbb{I}_{min}, b] \subseteq [\mathbb{I}_{min}, b'] \wedge [a, \mathbb{I}_{max}] \subseteq [a', \mathbb{I}_{max}]$. Using the result of the first case and removing inequalities that are always true yields the right-hand side.

Case 4: $a > b \wedge a' \leq b'$.

$[\mathbb{I}_{min}, b] \dot{\cup} [a, \mathbb{I}_{max}] \subseteq [a', b'] \iff [a', b'] = \mathbb{I}$:

"$\Rightarrow$" Both $\mathbb{I}_{min}$ and $\mathbb{I}_{max}$ are contained in the interval union on the left. From the subset relation, we have $\mathbb{I}_{min} \in [a', b']$ and $\mathbb{I}_{max} \in [a', b']$. As $\mathbb{I}_{min} \leq x$ for all $x \in \mathbb{I}$, we have $a' = \mathbb{I}_{min}$. Similarly, we have $b' = \mathbb{I}_{max}$.

"$\Leftarrow$" Obviously, any interval or union thereof is a subset of $\mathbb{I}$.

Two of the cases yield identical results, thus we have

$$[\![a, b]\!] \subseteq [\![a', b']\!] \Leftrightarrow \begin{cases} a' \leq a \wedge b \leq b' & \text{if } (a \leq b \leftrightarrow a' \leq b') \\ a' \leq a \vee b \leq b' & \text{if } a \leq b \wedge a' > b' \\ [\![a', b']\!] = \mathbb{I} & \text{otherwise} \end{cases} \tag{5.2}$$

This function is implemented as `is_interval_subseteq`.

Set complement

As $\mathbb{I}$ is a member of Int_{Wrap} but the empty set is not, Int_{Wrap} is not closed under set complement. However, that is the only exception, and $Int_{Wrap} \setminus \mathbb{I}$ is closed under set complement:

Let $a \leq b$ with $(a, b) \neq (\mathbb{I}_{min}, \mathbb{I}_{max})$. Then

$$\begin{aligned} \mathbb{I} \setminus [\![a, b]\!] = \{x \in \mathbb{I} \mid \neg(a \leq x) \wedge \neg(x \leq b)\} &= \{x \in \mathbb{I} \mid \mathbb{I}_{min} \leq x < a \wedge b < x \leq \mathbb{I}_{max}\} \\ &= [\mathbb{I}_{min}, a-1] \,\dot\cup\, [b+1, \mathbb{I}_{max}] = \cdots \end{aligned}$$

When making the strict inequalities non-strict, we use that $\mathbb{I}$ is an integral datatype. Note that the addition and subtraction operations used here are not those of $\mathbb{I}$, hence we cannot guarantee that $a - 1$ and $b + 1$ are members of $\mathbb{I}$. Three cases remain to be proven:

Case 1: $a \neq \mathbb{I}_{min} \wedge b \neq \mathbb{I}_{max}$. Then $a - 1 = a \ominus 1 \in \mathbb{I}$ and $b + 1 = b \oplus 1 \in \mathbb{I}$, thus $\cdots = [\![b \oplus 1, a \ominus 1]\!]$

Case 2: $a = \mathbb{I}_{min} \wedge b \neq \mathbb{I}_{max}$. Then $[\mathbb{I}_{min}, a - 1] = \emptyset$, thus $\cdots = x \in [b + 1, \mathbb{I}_{max}] = [\![b \oplus 1, a \ominus 1]\!]$

Case 3: $a \neq \mathbb{I}_{min} \wedge b = \mathbb{I}_{max}$. Then $[b + 1, \mathbb{I}_{max}] = \emptyset$, thus $\cdots = x \in [\mathbb{I}_{min}, a - 1] = [\![b \oplus 1, a \ominus 1]\!]$

The case $a > b$ is proven similarly, and we have $[\![a, b]\!]^{\complement} = [\![b \oplus 1, a \ominus 1]\!]$ for $[\![a, b]\!] \neq \mathbb{I}$. This partial function is implemented as `interval_complement`, where the partial nature of the function is reflected by an assertion.

Arithmetic operations

Finally, we need to make sure these intervals are compatible with basic arithmetic operations.

Lemma 5.2.1. *If* $[\![a, b]\!] \neq \mathbb{I}$, *then* $x \in [\![a, b]\!] \iff x \oplus c \in [\![a \oplus c, b \oplus c]\!]$.

Proof. It suffices to show the case $c = 1$. If $a \oplus 1$ and $b \oplus 1$ do not wrap around, i.e. both a and b are unequal to $\mathbb{I}_{max}$, the statement is easily verified for both interval types. The wrap-around case $\mathbb{I}_{max} \in \{a, b\}$ is verified by case distinction:

Case 1: $a = b = \mathbb{I}_{max}$. Then $x \in [a, b] \Leftrightarrow x = \mathbb{I}_{max} \Leftrightarrow x \oplus 1 = \mathbb{I}_{min} \Leftrightarrow x \oplus 1 \in [\mathbb{I}_{max} \oplus 1, \mathbb{I}_{max} \oplus 1]$.

Case 2: $a < b = \mathbb{I}_{max}$. Then $x \in [a, \mathbb{I}_{max}] \Leftrightarrow x \oplus 1 \in [a \oplus 1, \mathbb{I}_{max}] \vee x \in [\mathbb{I}_{min}, \mathbb{I}_{min}] \Leftrightarrow x \oplus 1 \in [\![a \oplus 1, \mathbb{I}_{min}]\!]$

Case 3: $b < a = \mathbb{I}_{max}$. Then $x \in [\mathbb{I}_{min}, b] \,\dot\cup\, [\mathbb{I}_{max}, \mathbb{I}_{max}] \Leftrightarrow x \oplus 1 \in [\mathbb{I}_{min} \oplus 1, b \oplus 1] \,\dot\cup\, [\mathbb{I}_{min}, \mathbb{I}_{min}] \Leftrightarrow x \oplus 1 \in [\mathbb{I}_{min}, b \oplus 1]$

□

Note that the case $[\![a, b]\!] = \mathbb{I}$ is handled quite easily, as $x \in \mathbb{I} \Leftrightarrow x \oplus c \in \mathbb{I}$.
Thus if $x = y \oplus c$, we have $x \in [\![a, b]\!] \Leftrightarrow y \in [\![a \ominus c, b \ominus c]\!]$.

As subtraction is usually represented in libFirm as addition of a negative number, that case does not need to be handled separately.

5.3. Implementation details

In the following, we note some specifics of the implementation that might be of interest.

5.3.1. Use of bit sets for cond sets

In order to perform fast operations on sets of conds, we use the built-in bit set data type of libFirm to represent such sets. Common set operations such as set difference can be executed in a few clock cycles if the number of conds is smaller than the word size of the processor.

There is one drawback however—as bit sets are statically allocated, we need to know the number of conds before writing any edge information. This means that we cannot annotate edges with a cond right after creating it, but instead must do so in a separate pass.

5.3.2. Edge annotations

As control flow edges in libFirm are not explicit, we need a way to identify a given control flow edge in order to store associated information. Using a tuple (block, predecessor index) would be enough, but cumbersome to implement using libFirm's predefined data structures.

Instead, we assume that every control flow node has only one user. This way, we can associate a control flow edge with the node from which it originates. This assumption is reasonable for all control flow nodes except for the `IJmp` node, which is used for indirect jumps to a computed address. In C code, this node is used to implement the "labels as values" GCC extension.

We therefore do not currently thread across edges to IJmp nodes. This is accomplished by stopping the recursion in `annotate_edges_rec` if a control flow node with more than one user is encountered. However, if the edge information representation was changed, it would be possible to thread across indirect jumps as well.

5.3.3. Usage of the link field

Each node (including each block) in libFirm has a pointer which is usable for any purpose. Jump Threading uses the link field as follows:

- On `Block` nodes, the link field points to a `jt_block_dupl` structure containing annotations of this block—namely the set of conds that are associated with it, corresponding to the set $\mathcal{S}'$, as well as its original block.
- On all other nodes, the link field points to the original node from which it was duplicated. That original node serves as the representative of the set of its duplicates and is used e.g. as key in the hash map of dominating definitions per block.

5.4. Validation

In terms of correctness, the implementation was validated using libFIRM's internal test suite. The unmodified version of cparser failed 106 tests with Jump Threading enabled, while the new version failed 103, none of which were new failures. Without Jump Threading enabled, cparser fails 102 tests. The additional failure is triggered by enabling either the old or new Jump Threading and occurs in the Inlining optimization pass.

5.5. SSA reconstruction

As briefly mentioned in Chapter 3, duplicating a block creates multiple definitions of that block's variables. The resulting graph does not fulfil the SSA property. In order to restore it, we must update all usages of a variable to point to a definition of the variable whose block dominates the usage block. If this is not possible, we need to insert ϕ functions at the iterated dominance frontiers of the definitions.

However, recomputing the dominator tree after each control flow change would be too expensive. Braun et al. [8] presented a SSA reconstruction algorithm that does not depend on dominance information. This algorithm is presented in a more detailed fashion in [9, chapter 5]. JT_{Old} also uses this algorithm for SSA reconstruction. We briefly sketch it here.

Given a definition *def* in block *def.block*, and a usage block *useblock* with predecessors $pred_1, \ldots, pred_n$, the following function yields a definition that dominates *useblock*:

$$\text{dominating-def}(def, useblock) = \begin{cases} def & \text{if } def.block = useblock \\ \text{the copy of } def \text{ in } useblock & \text{if } useblock \text{ is a duplicate of } def.block \\ \phi(d_1, \ldots, d_n) & \text{otherwise} \end{cases}$$
$$\text{where } d_i = \text{dominating-def}(def, pred_i).$$

Some of the ϕ functions generated by the third case will be superfluous. This is the case especially if a ϕ function's arguments consist only of itself, the undefined

value $\perp$ and one other value v, in which case it can be replaced by v. However, we do not know in advance whether a ϕ function will be superfluous. Therefore, if *useblock* has more than one predecessor we insert a *pending ϕ function* into *useblock*, then after computing its arguments we check whether it is superfluous.

This approach is not perfect, as it depends on the order in which blocks are visited and does not eliminate all superfluous ϕ functions. However, removing all superfluous ϕ functions even in irreducible control flow graphs is a task warranting a master's thesis of its own [10], one result of which is a dedicated optimization pass for libFIRM which could be run after Jump Threading if desired.

5.6. Cost model

Like with most other implementations, our cost model is based on the estimated code duplication cost. The search for TOs is aborted if its cost exceeds a configurable threshold. The duplication cost is estimated based on the number of nodes in the blocks between two conditions.

More fine-grained cost models are conceivable, for example sorting the TOs according to their estimated cost-benefit ratio and duplicating only the first few. However, interactions between TOs are quite complex. For example, if a set of TOs, taken together, determines the target of a branch instruction for all predecessors of its containing block, it makes sense to thread these TOs together, eliminating the conditional branch in all duplicates of the block.

In general however, considering all possible combinations of TOs for threading is not feasible from a performance standpoint.

6. Evaluation

6.1. Experimental setup and methodology

6.1.1. Platform

The main evaluation system (Haswell system) uses a 64-bit Intel Core i5-4460 CPU of the “Haswell” microarchitecture generation, with 4 physical cores, each running at a frequency of 3.20 GHz. When using only 1 or 2 cores, as is the case with the SPECint2000 benchmarks, “TurboBoost” technology can increase the frequency to 3.40 GHz. Each core contains two 32 KB 8-way set associative L1 caches for instructions and data respectively, as well as a 256 KB 8-way set associative L2 cache. Shared between the code is a 6 MB 12-way set-associative L3 cache.[1] Additionally, a small μop cache, acting as a L0 cache, stores decoded instructions. The μop cache was introduced in the Sandy Bridge microarchitecture.

Not much is known about Haswell’s branch prediction. Agner Fog [1] suspects that it contains “two branch prediction methods: a fast method tied to the μop cache and the instruction cache, and a slower method using a branch target buffer” [1, p. 29]. The branch misprediction penalty varies from 15–20 clock cycles.

The system is equipped with 8 GB (2×4 GB) of DDR3 RAM with a clock speed of 1600 MHz, and with a 250 GB SSD. All benchmarks were performed under Linux, Kernel version 3.16.7.

For some benchmarks, a Nehalem generation system (Intel Core i5 750 CPU, 20 GB of RAM) was used. On this system, no root privileges were available, thus some temci options could not be activated.

6.1.2. Compiler

The compiler used in all benchmarks is cparser, version cparser-1.22.0-192-g4ebb0b8, using libFIRM version libfirm-1.22.0-509-g64e6f36. The new Jump Threading algorithm was implemented in libFIRM on top of this version, replacing the old algorithm. The unmodified cparser version was used both for benchmarks using the old Jump Threading algorithm and for benchmarks without Jump Threading.

[1] CPU data taken from `http://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5-4460.html`, retrieved 2016-12-30.

6.1.3. Benchmarks

We mainly used the SPECint2000[2] benchmarking suite. It consists of twelve 32-bit applications that test a system's integer performance on real-work workloads such as data compression, combinatorial optimization, and compiling source code with GCC[3]. The "252.eon" benchmark was excluded as it requires a C++ compiler while cparser supports only C.

As mentioned in Section 4.2, one application of Jump Threading is the optimization of finite state machines. As a motivational example and microbenchmark, we used an implementation of a FSM with 6 states analogous to Figure 4.1. The FSM checks if a given string is a decimal, octal or hexadecimal number, or not a number at all. Each invocation of the program runs 1000 iterations of the state machine on an in-memory string containing the first million digits of π. The program was compiled with all optimizations but If Conversion enabled.

6.1.4. Methodology

We ran all benchmarks using temci [11, 12]. This tool facilitates benchmarking by conducting statistical tests on the benchmark results and formatting them for easier processing, as well as trying to minimize external factors like caching effects and other programs running on the same machine. An overview of temci options used is given in Figure 6.1.

Quantities such as branch misses and execution time were read from hardware counters by the `perf` tool. Information on compile time and compilation statistics were taken from cparser's statistics output (`--time` and `--statev` command line options). The static and dynamic opcode mixes were obtained using the Intel Pin dynamic instrumentation tool [13].

6.2. Experimental results and discussion

Out of the many possible variants of the new Jump Threading implementation, we used loop-ifconv-t30 and loop-no-ifconv-t30. This means that TOs may cross loop boundaries and each TO is restricted to duplicate at most 30 instructions (Firm nodes). In order to test the interaction with If Conversion, we distinguish two variants: ifconv, where an additional If Conversion pass is performed before Jump Threading, and no-ifconv, which employs the standard order of optimizations.

6.2.1. FSM microbenchmark

As apparent from Figure 6.2, Jump Threading can certainly have a big performance effect even on current processors, simply because it eliminates some branch instruc-

[2] `https://www.spec.org/cpu2000/CINT2000/`

[3] Of course, the "176.gcc" benchmark contains GCC's Jump Threading implementation, and yes, Jump Threading finds Threading Opportunities in it.

Option	Explanation [11, `temci exec --help`]
`--stop_start`	Stops almost all other processes running on the system.
`--sleep --sleep_seconds 10`	Sleep for 10 seconds before each benchmark run, in order to ensure that the system is quiescent.
`--nice --other_nice`	Increases the scheduling priority for the benchmarking process and decreases it for other processes of the system. The default values were kept, which are 19 for other processes and −15 for the benchmarking process. (Needs root privileges)
`--disable_hyper_threading`	Disable hyper-threaded cores on the CPU (while this particular CPU does not support hyper-threading anyway, this option would make sense on a CPU that does, as the benchmarks used are CPU bound). (Needs root privileges)
`--drop_fs_caches`	Drops the page cache, directory entry cache and inode cache before each benchmarking run. (Needs root privileges)
`--disable_swap`	Disables swapping on the system before benchmarking (and re-enables it afterwards). (Needs root privileges)
`--discarded_runs 1`	Performs an additional benchmarking run at the beginning whose results are discarded.
`--preheat`	Before benchmarking, pre-heats the CPU for 10 seconds using a CPU bound task
`--shuffle`	Randomize the order of the benchmarks in each run
`--cpuset_active --cpuset_parallel 0`	Pins the benchmarked programs to one CPU and all other tasks to the rest; disables parallel execution of benchmarks. (Needs root privileges)

Figure 6.1. Command line options for temci used for the benchmarks in this chapter. The options in the bottom section are enabled by default in temci, the options in the top section are not. The options that need root privileges were not used on the Nehalem test system. The `sleep` option was not used for the FSM microbenchmark.

tions (whether conditional or not) altogether. The decrease in execution time is correlated with the overall decrease in branch instructions. While branch prediction seems to have improved in Haswell, it is not a limiting factor at all on either of the systems.

The decrease in unconditional branches also supports our theoretical results about block scheduling from Section 3.4.2. In this case, Jump Threading very aggressively splits blocks at which control flow converges, enabling many opportunities to merge blocks with their predecessors and resulting in a program that jumps almost exclusively from one conditional branch to the next.

Quantity	no JT	JT_{Old} speedup	new JT speedup
Instructions	94.994 50 G	79.995 93 G 15.79 %	44.999 34 G 52.63 %
Branches	39.996 98 G	34.997 45 G 12.50 %	14.999 41 G 62.50 %
Percentage of cond. branches	87.5 %	85.7 %	100.0 %
Branch misses (Nehalem)	74 k	65 k (11.76 %)	39 k 47.09 %
Execution time (Nehalem)	12.54 s	10.97 s 12.52 %	4.72 s 62.32 %
Branch misses (Haswell)	23.3 k	22.9 k (1.63 %)	21.8 k 6.39 %
Execution time (Haswell)	5.932 s	5.192 s 12.48 %	2.227 s 62.45 %

Figure 6.2. Benchmark results for the Finite State Machine microbenchmark on both the Nehalem and Haswell test system. The dynamic instruction and branch count are the same on both systems. Numbers that lie in the uncertainty range (0–15 %) of the t-test are printed in parentheses.

6.2.2. Execution time

The execution time results for the SPECint2000 benchmark are shown in Figure 6.3, and the change in the number of branches and branch misses is shown is Figure 6.4.

On all benchmarks but `perlbmk` and `twolf`, Jump Threading achieves a speedup relative to the variant without Jump Threading of at least 0.48 % for loop-ifconv-t30 and 0.86 % for loop-no-ifconv-t30. In most of the cases we also either outperform JT_{Old} or achieve a speedup in the same range.

The `perlbmk` and `twolf` benchmarks each have characteristics that make optimizing them particularly difficult. As `twolf` is heavily cache dependent, duplicating code in the wrong place can have negative effects. The performance of `perlbmk` depends crucially on the regular expression match function where approximately 50 % of the benchmark's execution time is spent. It contains a small inner loop that provides both If Conversion and Jump Threading opportunities. Here running If Conversion first is preferable, as it reduces both the number of branches and the code size, as conjectured in Section 3.4.1. However, in most other cases executing If Conversion before Jump Threading seems to impair performance.

The speedups are also not directly correlated with the change in branch misses. The total number of branches executed seems to play a more important role. In the next section we will break down these branches by type.

6.2.3. Opcode mix

In order to verify that Jump Threading fulfils its main promise, which is to reduce the number of dynamically executed conditional branches, we evaluated both the static and dynamic opcode mix. The results are presented in Figure 6.5.

As expected, the static number of conditional branches increases, as our variant of Jump Threading duplicates control flow structures.

The dynamic number of conditional branches decreases in all benchmarks but `twolf`. This fits in with the fact that `twolf` is the only benchmark in which Jump Threading produces a slowdown. It seems that the conditional branches that were eliminated lay on seldom used code paths, so that the increase in code size is not offset by an improvement in performance.

Jump Threading also decreases the number of unconditional branches (both static and dynamic). As seen in Section 3.4.2, the path duplication performed by Jump Threading provides opportunities for the block scheduler to produce more fall-throughs and often even allows blocks both on the original and the duplicated code paths to be merged.

To verify this, we measured the percentage of duplicated blocks that can be merged with their predecessor. We did this by running a control flow optimization pass directly before and directly after Jump Threading, counting the number of merged blocks for the second run. The result is shown in Figure 6.5. We see that on average, a large number of blocks can be merged.

Note that no statement is made about whether the merge possibility affected an original or a duplicate block. We therefore included the maximum percentage of blocks merged relative to the number of block duplicates. In some cases, this is greater than 100 %, meaning that the merge possibilities affected more blocks than were duplicated.

Benchmark	no JT		JT_{Old}		speedup	loop-ifconv-t30		speedup	loop-no-ifconv-t30		speedup
164.gzip	66.1 s	±0.18 %	66.2 s	±0.36 %	=	65.6 s	±0.28 %	0.88 %	65.6 s	±0.13 %	0.78 %
175.vpr	50.5 s	±0.23 %	49.7 s	±0.12 %	1.64 %	49.9 s	±0.3 %	1.18 %	50.1 s	±0.35 %	0.88 %
176.gcc	23.00 s	±0.08 %	22.62 s	±0.07 %	1.64 %	22.71 s	±0.08 %	1.23 %	22.31 s	±0.09 %	2.98 %
181.mcf	23.7 s	±0.61 %	23.7 s	±0.31 %	=	23.5 s	±0.45 %	0.86 %	23.3 s	±0.44 %	1.57 %
186.crafty	28.4 s	±0.20 %	28.4 s	±0.26 %	=	28.28 s	±0.1 %	0.48 %	28.1 s	±0.19 %	0.86 %
197.parser	63.4 s	±0.11 %	63.32 s	±0.07 %	0.25 %	62.83 s	±0.06 %	1.03 %	62.4 s	±0.10 %	1.66 %
253.perlbmk	51.12 s	±0.05 %	52.09 s	±0.07 %	−1.88 %	51.1 s	±0.1 %	=	53.0 s	±0.12 %	−3.84 %
254.gap	28.5 s	±0.27 %	27.9 s	±0.29 %	1.95 %	28.0 s	±0.33 %	1.71 %	28.0 s	±0.30 %	1.66 %
255.vortex	43.5 s	±0.77 %	43.1 s	±0.75 %	(0.93 %)	42.8 s	±0.49 %	1.68 %	42.4 s	±0.28 %	2.42 %
256.bzip2	53.1 s	±0.47 %	52.5 s	±0.39 %	1.18 %	52.3 s	±0.19 %	1.40 %	52.5 s	±0.54 %	1.08 %
300.twolf	70.1 s	±0.15 %	70.2 s	±0.28 %	=	70.8 s	±0.24 %	-0.99 %	69.9 s	±0.20 %	(0.24 %)
average					0.52 %			0.86 %			0.94 %

Figure 6.3. Benchmark results for SPECint2000 on the Haswell system. Only the significant digits are shown. Speedup values for which the t-test returned an uncertainty value $>$ 15 % are denoted by "=", while values in the uncertainty range of 5–15 % are printed in parentheses.

Benchmark	loop-ifconv-t30 [%] branches	loop-ifconv-t30 [%] branch-misses	loop-no-ifconv-t30 [%] branches	loop-no-ifconv-t30 [%] branch-misses
164.gzip	−0.67	+1.29	−0.67	+1.97
175.vpr	+0.87	−2.44	+0.87	−2.18
176.gcc	−3.33	−3.77	−3.09	−3.84
181.mcf	−5.56	+2.82	−13.15	+4.52
186.crafty	−1.17	−0.76	−1.12	−0.16
197.parser	−5.41	+0.41	−5.48	−0.97
253.perlbmk	−0.22	−0.83	−0.04	−1.10
254.gap	−3.82	−16.63	−3.81	−16.92
255.vortex	−8.41	=	−8.39	=
256.bzip2	−3.01	=	−3.70	−1.25
300.twolf	(0.00)	−1.36	=	−2.17

Figure 6.4. Changes in the dynamic number of branches and branch misses for loop-ifconv-t30 and loop-no-ifconv-t30 relative to no JT.

Benchmark	branch insts static [±%] cond.	branch insts static [±%] uncond.	branch insts dynamic [±%] cond.	branch insts dynamic [±%] uncond.
164.gzip	+2.19	−8.47	−0.60	−2.10
175.vpr	+1.73	−1.22	−0.91	+4.40
176.gcc	+0.56	−10.48	−2.52	−12.89
181.mcf	+2.86	−1.69	−3.92	+13.42
186.crafty	+2.93	−2.91	−1.00	−3.10
197.parser	+4.04	−9.36	−5.58	−11.36
253.perlbmk	+0.03	−3.29	−0.26	−0.42
254.gap	+3.28	−9.45	−2.98	−13.42
255.vortex	+0.60	−1.49	−10.47	−14.08
256.bzip2	+2.46	−4.19	−2.72	−6.06
300.twolf	+1.02	−1.78	−0.00	+0.03

Figure 6.5. Comparison of the opcode mix for loop-ifconv-t30 relative to no JT. Static values refer to the instructions as present in the text section of the binary, while dynamic values refer to the actual number of instructions executed during a run of the benchmark.

Benchmark	Block duplicates	avg. mergeable	max. mergeable
164.gzip	208	17.80 %	100 %
175.vpr	250	30.84 %	100 %
176.gcc	6234	52.92 %	200 %
181.mcf	24	32.5 %	100 %
186.crafty	552	25.10 %	100 %
197.parser	777	61.35 %	200 %
253.perlbmk	1155	66.90 %	200 %
254.gap	2363	64.46 %	200 %
255.vortex	617	43.18 %	200 %
256.bzip2	166	20.07 %	100 %
300.twolf	435	35.27 %	150 %

Figure 6.6. Total number of blocks duplicated by Jump Threading (loop-no-ifconv-t30), and number of blocks mergeable with control flow predecessor after Jump Threading, relative to the number of block duplicates created by Jump Threading.

6.2.4. Compile time

The compile time for all benchmarks is listed in Figure 6.7. As expected, the new Jump Threading pass, doing more work, is also slower.

An analysis of the Jump Threading pass using the callgrind tool of valgrind shows that the SSA reconstruction employed by the new implementation is particularly costly, consuming up to 65 % of the time spent in Jump Threading. The SSA reconstruction used is not very optimized, as it traverses the whole program graph once. Some performance optimization in this area is certainly possible and would reduce the gap between the two implementations.

As Jump Threading increases the code size, other optimizations naturally have to process more code and are thus slowed down as well. This leads to an overall slowdown of less than 5 % on average.

It is therefore justified that Jump Threading is enabled by default only when compiling with `-O3`, which is described as “Aggressively optimize at the expense of compilation speed and code size”.

6.2.5. Number of conds

Finally, in Figure 6.8 we give an overview of the number of `jt_cond`s found by the analysis. We see that in approximately 80 % of all invocations, Jump Threading finds no threading opportunities. However, Jump Threading is run twice on each function that is not completely inlined into other functions, and in the second pass might not find new threading opportunities. By summing up the number of conds found during both passes for each function, it can be seen that the percentage of

	Jump Threading [s]		Σ optimization phases [s]	
Benchmark	JT_{Old}	loop-ifconv-t30	JT_{Old}	loop-ifconv-t30
164.gzip	0.0490	0.0838	5.710	6.102
175.vpr	1.228	1.743	15.545	16.255
176.gcc	2.356	4.145	189.3	199.3
181.mcf	0.010 63	0.0164	0.978	0.956
186.crafty	0.1423	0.294	21.3	22
197.parser	0.2317	0.384	18.38	19.53
253.perlbmk	0.8524	1.245	66.8	70.4
254.gap	0.7340	1.233	76.79	80.81
255.vortex	0.4835	0.754	35.43	37.51
256.bzip2	0.0383	0.0647	4.77	5.02
300.twolf	0.3096	0.3668	37.00	40.30

Figure 6.7. Compile time: Time spent in the Jump Threading optimization pass and in all optimization passes during a compilation of the whole benchmark suite. The average slowdown of the Jump Threading pass relative to JT_{Old} is $61.17\,\%$ or $311.89\,\mathrm{ms}$ while the average slowdown of the whole compilation is $4.94\,\%$ or $2.451\,\mathrm{s}$. On average, $1.01\,\%$ of compile time is spent in the old Jump Threading, while with the new Jump Threading it is $1.55\,\%$.

functions that remain unchanged by Jump Threading is about 70 %.

We now consider only the executions in which Jump Threading finds something. In more than 50 % of these cases it finds 1 or 2 conditions. As expected, If Conversion destroys some Jump Threading opportunities, therefore loop-ifconv-t30 has more instances of 1–3 conditions being found, while for 4 or more conditions, loop-no-ifconv-t30 starts finding more.

In Section 5.3.1 we mentioned the use of bit sets for performance reasons. Operations on these bit sets take time linear in the maximum size of the set, which in this case is the number of conds. In the overwhelming majority of cases, the number of conds is 32 or less, meaning that the bitsets almost always fit into a single processor word even on 32-bit systems. Correspondingly, the set operations are the fastest part of the implementation.

Number of conds	Occurrences (loop-no-ifconv-t30)		Occurrences (loop-ifconv-t30)	
0	9732	79.26 %	9855	80.24 %
1	652	25.61 %	675	27.81 %
2	669	26.27 %	667	27.48 %
3	232	9.11 %	241	9.93 %
4	307	12.06 %	250	10.30 %
5	133	5.22 %	124	5.11 %
6	125	4.91 %	101	4.16 %
...				
1–32	2520	98.98 %	2411	99.34 %
33–64	24	0.94 %	15	0.66 %
72	1			
80			1	
87	1			

Figure 6.8. All executions of the Jump Threading optimization pass during a full build of the SPECint2000 suite grouped by the number of conds they produce. In the topmost row (number of conds = 0), the percentages are relative to all executions of Jump Threading. For all other rows, they are relative to all executions of Jump Threading that find one or more conds.

7. Conclusion and Further Work

In this thesis, we presented a theoretical analysis of the Jump Threading compiler optimization, as well as a generalized algorithm for Jump Threading and an implementation thereof.

We have seen how Jump Threading interacts—both positively and negatively—with other compiler optimizations, especially with If Conversion. Simple changes like changing the order of optimizations can drastically change performance, in one case by more than 3 %.

Using the SPECint2000 benchmark suite, we have shown that Jump Threading is still a profitable optimization. Our implementation outperforms the existing libFIRM implementation in many of the benchmarks, achieving a higher average speedup of 0.86 % vs. 0.52 % and, depending on the configuration, a much higher maximum speedup of 2.98 % vs. 1.95 %.

Further work on the Jump Threading optimization itself would include the improvement of the cost model. There are a number of factors that could be taken into account to determine whether a TO is profitable to thread:

- Profiling information (either estimated or using profile-guided optimization) could be used to determine the reduction in execution frequency of the conditional branch, or to estimate whether the branch is easy or hard to predict.
- More structural information of the CFG could be taken into account; for example the optimization could detect whether it is currently unrolling a loop.
- The heuristics for the duplication costs could be improved, taking into account that control flow nodes are eliminated.

Finding optimal values for all these parameters seems a challenging task. Simpler improvements could include finding more Threading Opportunities, such as including supporting "$x \times y = 0$" conditions where neither x nor y is a constant, or supporting comparisons against non-constant values.

As we have seen, the interaction between compiler optimizations is very relevant to performance, and further research could for example devise a way to make Jump Threading and If Conversion coexist more productively. One could also try to determine whether register allocation can take advantage of the block annotations provided by Jump Threading, which are currently discarded.

Bibliography

[1] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs," *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering*, 2016.

[2] F. Mueller and D. B. Whalley, "Avoiding conditional branches by code replication," in *ACM SIGPLAN Notices*, vol. 30, pp. 56–66, ACM, 1995.

[3] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," in *ACM SIGPLAN Notices*, vol. 32, pp. 146–158, ACM, 1997.

[4] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica*, vol. 7, no. 3, pp. 305–317, 1977.

[5] "Source code of the LLVM compiler infrastructure." `http://llvm.org/viewvc/llvm-project/llvm/trunk/lib/Transforms/Scalar/JumpThreading.cpp?revision=287488&view=markup`. Revision 287488, 20 Nov. 2016.

[6] "Source code of the GNU compiler collection (GCC)." `https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/tree-ssa-threadbackward.c;hb=5fd049`. Revision 5fd049, 15 Nov. 2016.

[7] "Source code of libFirm." `http://pp.ipd.kit.edu/git/libfirm/tree/ir/opt/jumpthreading.c?id=aa7c5`. Revision aa7c5, 19 Sep. 2016.

[8] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, "Simple and efficient construction of static single assignment form," in *Compiler Construction* (R. Jhala and K. Bosschere, eds.), vol. 7791 of *Lecture Notes in Computer Science*, pp. 102–122, Springer, 2013.

[9] F. Rastello, *SSA-based Compiler Design*. Springer Publishing Company, Incorporated, 2016.

[10] M. Wagner, "Minimal static single assignment form," Master's thesis, Karlsruhe Institute of Technology (KIT), Nov. 2016. `http://pp.ipd.kit.edu/publication.php?id=wagner16masterarbeit`.

[11] J. Bechberger, "temci documentation." `http://temci.readthedocs.org/en/latest/`.

[12] J. Bechberger, “Besser benchmarken,” Bachelor’s thesis, Karlsruher Institut für Technologie (KIT), Apr. 2016. http://pp.ipd.kit.edu/publication.php?id=bechberger16bachelorarbeit.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Acm sigplan notices*, vol. 40, pp. 190–200, ACM, 2005.

A. Jump Threading source code

```c
/*
 * This file is part of libFirm.
 * Copyright (C) 2017 Karlsruhe Institute of Technology.
 */

/**
 * @file
 * @brief   Path-Sensitive Jump Threading
 * @date    2017-01-23
 * @author  Joachim Priesner
 */
#include "debug.h"
#include "irargs_t.h"
#include "iredges.h"
#include "irnode_t.h"
#include "irgmod.h"
#include "irgwalk.h"
#include "iroptimize.h"
#include "irtools.h"
#include "raw_bitset.h"
#include "tv.h"

DEBUG_ONLY(static firm_dbg_module_t *dbg;)

// Various ways of controlling the types of threading opportunities found
// #define JT_NO_ADD                /* Ignore Add nodes */
// #define JT_NO_CF                 /* Do not duplicate control flow */
// #define JT_NO_CRITICAL_EDGES     /* Require the graph to contain no critical edges before optimizing */
// #define JT_NO_SWITCH              /* Do not thread through Switch nodes */
// #define JT_NO_LOOP_BOUNDARIES    /* Start and target block of a TO must lie in the same loop */

// #define JT_DUMP_COND_GRAPH

#ifndef JT_COST_THRESHOLD
#define JT_COST_THRESHOLD 30
#endif

/* Represents the condition that the given Phi node's value is in the interval [[a, b]]. */
typedef struct jt_cond {
  ir_node *node;
  ir_tarval *a, *b;

  struct jt_cond **preds; /* List of predecessor conds by control flow predecessor of the Phi */

  unsigned int max_cost;  /* The maximum cost for which this cond has been evaluated */
  ir_visited_t visited;   /* Visited counter */
  size_t index;           /* Index in the global list of conds */
} jt_cond;

#define COND_UNKNOWN ((jt_cond*)NULL)

static void debug_cond(jt_cond *cond)
{
  if (cond == COND_UNKNOWN) {
    DB((dbg, LEVEL_1, "COND_UNKNOWN"));
  } else if (cond->node == NULL) {
```

```
57      DB((dbg, LEVEL_1, "COND_ALWAYS_TRUE"));
58    } else {
59      DB((dbg, LEVEL_1, "%+F \\in [[%+F, %+F]]", cond->node, cond->a, cond->b));
60    }
61  }
62
63  /* Global state for one Jump Threading pass */
64  typedef struct jt_environment {
65    ir_graph *irg;
66    jt_cond cond_always_true; /**< special predecessor condition that is always true */
67    jt_cond **conds;
68    unsigned orig_last_idx;   /**< original value of get_irg_last_idx(irg) --
69                                   nodes with index >= orig_last_idx are duplicates */
70    struct obstack obst;      /**< obstack for temporary data */
71    pmap *cf_edge_info;       /**< edge annotations, type node -> jt_edge_info*
72                                   (each node represents the CF edge starting at that node) */
73    pmap *thread_successors;  /**< Successor edge -> cond that guarantees that edge's execution */
74    pmap *block_dupls;   /**< block -> [jt_block_dupl*] **/
75    pmap *dominating_defs;    /**< node -> [block -> node] **/
76  #ifdef JT_DUMP_COND_GRAPH
77    FILE *cond_graph;
78  #endif
79    int num_node_dupls_exact;
80  } jt_environment;
81
82  static void debug_cond_set(jt_environment *env, unsigned *cond_set)
83  {
84    if (cond_set != NULL) {
85      rbitset_foreach(cond_set, ARR_LEN(env->conds), cond_idx) {
86        debug_cond(env->conds[cond_idx]);
87        DB((dbg, LEVEL_1, ", "));
88      }
89    }
90  }
91
92  /** Information about a block duplicate */
93  typedef struct jt_block_dupl {
94    unsigned *conds;          /**< The conds valid in this block duplicate */
95    ir_node *orig_block;      /**< The Block node of the original block */
96    ir_node *dupl_block;      /**< The Block node of the duplicate block */
97  } jt_block_dupl;
98
99  typedef struct jt_edge_info {
100   unsigned *relevant_conds;    /**< Conds to propagate along this edge */
101   unsigned *always_true_conds; /**< Conds that are true when entering the successor block along this edge */
102   unsigned *forbidden_conds;   /**< Conds that cause another edge of the same block to be taken */
103 } jt_edge_info;
104
105 /** Computes a closed interval [[@p a, @p b]] that contains exactly the ir_tarvals
106  * of the given mode that satisfy @p relation with regard to the constant @p cnst_val. */
107 static bool get_interval(ir_relation relation, ir_tarval *cnst_val, ir_tarval **a, ir_tarval **b)
108 {
109   ir_mode *mode = get_tarval_mode(cnst_val);
110   ir_tarval *one = new_tarval_from_str("1", 1, mode == mode_P ? mode_Is : mode);
111   assert(tarval_is_one(one));
112   *a = get_mode_min(mode);
113   *b = get_mode_max(mode);
114   switch (relation & ~ir_relation_unordered) {
115   case ir_relation_equal:
116     *a = cnst_val;
117     *b = cnst_val;
118     break;
119   case ir_relation_less:
120     *b = tarval_sub(cnst_val, one);
121     break;
122   case ir_relation_greater:
```

```
123      *a = tarval_add(cnst_val, one);
124      break;
125    case ir_relation_less_equal:
126      *b = cnst_val;
127      break;
128    case ir_relation_greater_equal:
129      *a = cnst_val;
130      break;
131    case ir_relation_less_greater:
132      *a = tarval_add(cnst_val, one);
133      *b = tarval_sub(cnst_val, one);
134      break;
135    case ir_relation_true:
136      break;
137    default:
138      return false;
139    }
140    assert(get_tarval_mode(*a) == mode);
141    assert(get_tarval_mode(*b) == mode);
142    return true;
143  }
144
145  /** Returns whether the condition of the given @p cmp is expressible as "@p var in [[@p a, @p b]]
146   * and if yes, fills the output parameters accordingly. */
147  static bool get_cmp_interval(ir_node *cmp, ir_node **var, ir_tarval **a, ir_tarval **b)
148  {
149    assert(is_Cmp(cmp));
150
151    *var = get_Cmp_left(cmp);
152    ir_relation rel = get_Cmp_relation(cmp);
153    ir_node *cnst = get_Cmp_right(cmp);
154    if (is_Const(*var)) {
155      ir_node *tmp = cnst; cnst = *var; *var = tmp;
156      rel = get_inversed_relation(rel);
157    }
158    if (!is_Const(cnst)) {
159      return false;
160    }
161
162    ir_mode *mode = get_irn_mode(*var);
163    return (mode_is_int(mode) || mode == get_modeP()) && get_interval(rel, get_Const_tarval(cnst), a, b);
164  }
165
166  #define is_tv_leq(tv1, tv2) ((tarval_cmp(tv1, tv2) & ir_relation_less_equal) != 0)
167  #define is_tv_geq(tv1, tv2) ((tarval_cmp(tv1, tv2) & ir_relation_greater_equal) != 0)
168  #define is_wraparound(a, b) (!is_tv_leq(a, b))
169
170  /* Returns true if @p tv is guaranteed to be in the interval [[@p a, @p b]].
171   * If a > b, this is interpreted as ">= a || <= b",
172   * otherwise (as usual) as ">= a && <= b".
173   */
174  static bool is_tv_in_interval(ir_tarval *tv, ir_tarval *a, ir_tarval *b)
175  {
176    assert(get_tarval_mode(tv) == get_tarval_mode(a));
177    assert(get_tarval_mode(tv) == get_tarval_mode(b));
178    bool geq_a = is_tv_geq(tv, a);
179    bool leq_b = is_tv_leq(tv, b);
180    return is_wraparound(a, b) ? geq_a || leq_b : geq_a && leq_b;
181  }
182
183  /* Returns whether [[@p a1, @p b1]] is a subset of [[@p a2, @p b2]] */
184  static bool is_interval_subseteq(ir_tarval *a1, ir_tarval *b1, ir_tarval *a2, ir_tarval *b2)
185  {
186    bool a2_leq_a1 = is_tv_leq(a2, a1);
187    bool b1_leq_b2 = is_tv_leq(b1, b2);
188    if (is_wraparound(a1, b1) == is_wraparound(a2, b2)) {
```

```
189      return a2_leq_a1 && b1_leq_b2;
190    } else if (!is_wraparound(a1, b1)) {
191      return a2_leq_a1 || b1_leq_b2;
192    } else {
193      return a1 == get_mode_max(get_tarval_mode(a1)) && b1 == get_mode_min(get_tarval_mode(b1));
194    }
195  }
196
197  /* Returns whether the value of the given @p node is guaranteed to be in the interval [[@p a, @p b]]. */
198  static bool is_const_or_confirm_in_interval(ir_node *node, ir_tarval *a, ir_tarval *b)
199  {
200    if (is_Const(node)) {
201      return is_tv_in_interval(get_Const_tarval(node), a, b);
202    } else {
203      assert(is_Confirm(node));
204      ir_node *bound = get_Confirm_bound(node);
205      if (!is_Const(bound)) {
206        return false;
207      }
208      ir_tarval *confirm_a, *confirm_b;
209      return get_interval(get_Confirm_relation(node), get_Const_tarval(bound), &confirm_a, &confirm_b)
210             && is_interval_subseteq(confirm_a, confirm_b, a, b);
211    }
212  }
213
214  /* Transforms the interval [[@p a, @p b]] into the interval [[@p b + 1, @p a - 1]],
215   * which is its inverse. The input interval may not be equal to [mode_min, mode_max],
216   * i.e. the whole universe. */
217  static void interval_complement(ir_tarval **a, ir_tarval **b)
218  {
219    ir_mode *mode = get_tarval_mode(*a);
220    assert(mode == get_tarval_mode(*b));
221    assert(*a != get_mode_min(mode) || *b != get_mode_max(mode));
222    // Use mode_Is for mode_P because subtracting mode_P from another mode_P
223    // yields a tarval of "reference offset" mode, which is something we don't want.
224    ir_tarval *one = new_tarval_from_str("1", 1, mode == mode_P ? mode_Is : mode);
225    ir_tarval *tmp = *a;
226    *a = tarval_add(*b, one);
227    *b = tarval_sub(tmp, one);
228    assert(get_tarval_mode(*a) == get_tarval_mode(*b));
229  }
230
231  static int get_chop_size_rec(ir_node *def_block, ir_node *cur_block)
232  {
233    int result = get_irn_n_edges(cur_block);
234    if (cur_block != def_block) {
235      for (int i = 0, len = get_Block_n_cfgpreds(cur_block); i < len; i++) {
236        ir_node *pred_block = get_Block_cfgpred_block(cur_block, i);
237        if (pred_block != NULL && !irn_visited_else_mark(pred_block)) {
238          result += get_chop_size_rec(def_block, pred_block);
239        }
240      }
241    }
242    return result;
243  }
244
245  /** Returns the number of nodes whose block lies on a path between @p def_block and @p use_block,
246   *  not including @p use_block itself. */
247  static int get_chop_size(ir_node *def_block, ir_node *use_block)
248  {
249    if (def_block == use_block || def_block == NULL || use_block == NULL) {
250      return 0;
251    }
252    inc_irg_visited(get_irn_irg(use_block));
253    mark_irn_visited(use_block);
254    int result = 0;
```

```
255   for (int i = 0, len = get_Block_n_cfgpreds(use_block); i < len; i++) {
256     ir_node *pred_block = get_Block_cfgpred_block(use_block, i);
257     if (pred_block != NULL && !irn_visited_else_mark(pred_block)) {
258       result += get_chop_size_rec(def_block, pred_block);
259     }
260   }
261   return result;
262 }
263
264 /** Returns the number of nodes whose block lies on a path between @p def_block and @p use_block. */
265 static int get_chop_size_incl(ir_node *def_block, ir_node *use_block)
266 {
267   return get_irn_n_edges(use_block) + get_chop_size(def_block, use_block);
268 }
269
270 /** Returns a jt_cond that represents the condition "@p node \\in [[@p a, @p b]]" if there is at least
271  *  one path on which the condition is true whose cost together with @p cost does not exceed JT_THRESHOLD.
272  *  @p start is a node in the condition block of the TO to be constructed. */
273 static jt_cond *get_cond(jt_environment *env, ir_node *node, ir_tarval *a, ir_tarval *b, unsigned int cost,
274                          ir_node *start)
275 {
276   assert(node != NULL);
277
278   DB((dbg, LEVEL_1, "-get_cond(%+F \\in [%+F, %+F], cost %d\n", node, a, b, cost));
279
280   // Base cases
281   if (is_Const(node)) {
282     if (is_const_or_confirm_in_interval(node, a, b)) {
283       DB((dbg, LEVEL_1, "Found a Const node: comparison %+F \\in [[%+F, %+F]] is true\n", node, a, b));
284       return &env->cond_always_true;
285     } else {
286       DB((dbg, LEVEL_1, "Found a Const node: comparison %+F \\in [[%+F, %+F]] is false\n", node, a, b));
287       return COND_UNKNOWN;
288     }
289   } else if (is_Confirm(node) && is_const_or_confirm_in_interval(node, a, b)) {
290     DB((dbg, LEVEL_1, "Found a Confirm node %+F: {x | x %s %+F} \\subseteq [[%+F, %+F]] is true\n", node,
291         get_relation_string(get_Confirm_relation(node)), get_Confirm_bound(node), a, b));
292     return &env->cond_always_true;
293   }
294
295   // Recursive cases
296   if (cost > JT_COST_THRESHOLD) {
297     DB((dbg, LEVEL_1, "Condition %+F \\in [[%+F, +%F]]: Cost %d >= threshold\n", node, a, b, cost));
298     return COND_UNKNOWN;
299   }
300
301 #ifndef JT_NO_ADD
302   if (is_Add(node)) {
303 #else
304   if (false) {
305 #endif
306     ir_node *var = get_Add_left(node), *add_cnst = get_Add_right(node);
307     if (is_Const(var)) {
308       ir_node *tmp = add_cnst; add_cnst = var; var = tmp;
309     }
310     if (is_Const(add_cnst)) {
311 #ifdef JT_NO_CF
312       if (!is_Const(var) && !is_Confirm(var) && get_nodes_block(var) != get_nodes_block(node)) {
313         return COND_UNKNOWN;
314       }
315 #endif
316       DB((dbg, LEVEL_1, "Found an Add node %+F with const: %+F\n", node, add_cnst));
317       ir_tarval *new_a = a, *new_b = b;
318       if (a != get_mode_min(get_tarval_mode(a)) || b != get_mode_max(get_tarval_mode(b))) {
319         new_a = tarval_sub(a, get_Const_tarval(add_cnst));
320         new_b = tarval_sub(b, get_Const_tarval(add_cnst));
```

```
321         }
322         return get_cond(env, var, new_a, new_b,
323                         cost + get_chop_size(get_nodes_block(var), get_nodes_block(node)), start);
324       } else {
325         return COND_UNKNOWN;
326       }
327
328     } else if (is_Confirm(node)) {
329       // We have a Confirm node that does not guarantee the condition: simply continue searching.
330   #ifdef JT_NO_CF
331       if (get_nodes_block(get_Confirm_value(node)) != get_nodes_block(node)) {
332         return COND_UNKNOWN;
333       }
334   #endif
335       ir_node *val = get_Confirm_value(node);
336       return get_cond(env, val, a, b,
337                       cost + get_chop_size(get_nodes_block(val), get_nodes_block(node)), start);
338
339     } else if (!is_Phi(node)) {
340       return COND_UNKNOWN;
341     }
342
343     assert(is_Phi(node));
344     jt_cond *cond = NULL;
345     for (size_t i = 0, len = ARR_LEN(env->conds); i < len; i++) {
346       jt_cond *entry = env->conds[i];
347       assert(entry != NULL);
348       if (entry->node == node && entry->a == a && entry->b == b) {
349         cond = entry;
350         break;
351       }
352     }
353
354     bool new_cond = cond == NULL;
355     if (new_cond) {
356       cond = OALLOC(&env->obst, jt_cond);
357       cond->index = ARR_LEN(env->conds);
358       cond->node = node;
359       cond->a = a;
360       cond->b = b;
361       cond->preds = OALLOCN(&env->obst, jt_cond *, get_irn_arity(node));
362       cond->max_cost = 99999;
363       cond->visited = 0;
364
365       DB((dbg, LEVEL_1, "Adding new condition \\in [%+F, %+F] to node %+F\n", a, b, node));
366       ARR_APP1(jt_cond *, env->conds, cond);
367     } else if (cond->max_cost <= cost) {
368       return cond;
369     }
370
371     // Search for predecessor conditions
372     cond->max_cost = cost;
373
374     DB((dbg, LEVEL_1, "Found a Phi node %+F\n", node));
375     bool always_true = true;
376     bool always_unknown = true;
377     for (int j = 0, len = get_irn_arity(node); j < len; j++) {
378       ir_node *pred = get_irn_n(node, j);
379       ir_node *cfpred = get_Block_cfgpred_block(get_nodes_block(node), j);
380       if (cfpred == NULL) {
381         // do not set always_true to false: if the predecessor is a Bad, we may ignore its value
382         cond->preds[j] = COND_UNKNOWN;
383         continue;
384       }
385   #ifdef JT_NO_CF
386       if (!is_Const(pred) && !is_Confirm(pred) && get_nodes_block(pred) != cfpred) {
```

```
387 #else
388     if (false) {
389 #endif
390       cond->preds[j] = COND_UNKNOWN;
391     } else {
392       cond->preds[j] = get_cond(env, get_irn_n(node, j), a, b,
393                                 cost + get_chop_size_incl(get_nodes_block(pred), cfpred), start);
394     }
395
396 #ifdef JT_NO_LOOP_BOUNDARIES
397     if (cond->preds[j] == &env->cond_always_true &&
398         get_loop_depth(get_irn_loop(cfpred)) != get_loop_depth(get_irn_loop(get_nodes_block(start)))) {
399       cond->preds[j] = COND_UNKNOWN;
400     }
401 #endif
402     always_true &= cond->preds[j] == &env->cond_always_true;
403     always_unknown &= cond->preds[j] == COND_UNKNOWN;
404   }
405
406   if ((always_unknown || always_true) && new_cond) {
407     /* Free all conds obtained in recursive calls, i.e. allocated after this cond,
408      * as those are always unknown/always true as well.
409      * However, we can do this only if we created the cond in this call. */
410     debug_cond(cond);
411     if (always_unknown) {
412       DB((dbg, LEVEL_1, " is always unknown, removing\n"));
413     } else {
414       DB((dbg, LEVEL_1, " is always true, removing\n"));
415     }
416     ARR_RESIZE(jt_cond *, env->conds, cond->index);
417     obstack_free(&env->obst, cond); /* frees cond and all conds+pred-arrays that were allocated after it */
418     return always_unknown ? COND_UNKNOWN : &env->cond_always_true;
419   }
420
421   assert(cond->index <= ARR_LEN(env->conds));
422
423 #ifdef JT_DUMP_COND_GRAPH
424   if (cond->node != NULL) {
425     ir_obst_vprintf(env, "node: {title: \"n%p\" label: \"%+F \\in [%+F, %+F]\"}\n",
426                     cond, cond->node, cond->a, cond->b);
427     obstack_1grow(&env->obst, '\0');
428     char *res = (char *)obstack_finish(&env->obst);
429     fprintf(env->cond_graph, "%s", res);
430     obstack_free(&env->obst, res);
431
432     for (size_t i = 0, len = get_irn_arity(cond->node); i < len; i++) {
433       if (cond->preds[i] != COND_UNKNOWN) {
434         fprintf(env->cond_graph, "edge: {sourcename: \"n%p\" targetname: \"n%p\" label: \"%d\"}\n",
435                 cond, cond->preds[i], (int)i);
436       }
437     }
438   }
439 #endif
440
441   return cond;
442 }
443
444 /** A block walker which finds threading opportunities that end in @p block. */
445 static void init_and_find_cmps(ir_node *block, void *_env)
446 {
447   jt_environment *env = _env;
448   set_irn_link(block, NULL);
449
450   for (int i = 0, n_cfgpreds = get_Block_n_cfgpreds(block); i < n_cfgpreds; i++) {
451     ir_node *projx = get_Block_cfgpred(block, i);
452     if (!is_Proj(projx)) {
```

```
453         continue;
454       }
455 
456       jt_cond *cond = NULL;
457       ir_node *cond_node = get_Proj_pred(projx);
458       if (is_Cond(cond_node)) {
459         if (get_Proj_num(projx) != pn_Cond_true && get_Proj_num(projx) != pn_Cond_false) {
460           continue;
461         }
462         ir_node *selector = get_Cond_selector(cond_node);
463 #ifdef JT_NO_CF
464         if (!is_Const(selector) && get_nodes_block(selector) != get_nodes_block(cond_node)) {
465           continue;
466         }
467 #endif
468         if (is_Cmp(selector)) {
469           ir_node *var;
470           ir_tarval *a, *b;
471           if (!get_cmp_interval(selector, &var, &a, &b)) {
472             continue;
473           }
474 
475 #ifdef JT_NO_CF
476           if (get_nodes_block(var) != get_nodes_block(selector)) {
477             continue;
478           }
479 #endif
480 
481           ir_mode *mode = get_irn_mode(var);
482           if (a == get_mode_min(mode) && b == get_mode_max(mode)) {
483             cond = get_Proj_num(projx) == pn_Cond_true ? &env->cond_always_true : COND_UNKNOWN;
484           } else {
485             if (get_Proj_num(projx) == pn_Cond_false) {
486               interval_complement(&a, &b);
487             }
488 
489             DB((dbg, LEVEL_1, "Found a Cmp node %+F, var %+F, interval [[%+F, %+F]], dest %+F\n",
490                 selector, var, a, b, projx));
491             cond = get_cond((jt_environment *)env, var, a, b,
492                             get_chop_size_incl(get_nodes_block(var), get_nodes_block(projx)), projx);
493           }
494         } else if (is_Const(selector)) {
495           ir_tarval *needed_tv = (get_Proj_num(projx) == pn_Cond_true ? tarval_b_true : tarval_b_false);
496           if (get_Const_tarval(selector) == needed_tv) {
497             DB((dbg, LEVEL_1, "Found a Const b node %+F\n", selector));
498             cond = &env->cond_always_true;
499           }
500         } else if (is_Phi(selector)) {
501           ir_tarval *needed_tv = get_Proj_num(projx) == pn_Cond_true ? tarval_b_true : tarval_b_false;
502           DB((dbg, LEVEL_1, "Found a Phi b node %+F\n", selector));
503           cond = get_cond(env, selector, needed_tv, needed_tv,
504                           get_chop_size_incl(get_nodes_block(selector), get_nodes_block(projx)), projx);
505         }
506 
507 #ifndef JT_NO_SWITCH
508       } else if (is_Switch(cond_node)) {
509         ir_node *selector = get_Switch_selector(cond_node);
510         if (!is_Phi(selector) && !is_Const(selector) && !is_Confirm(selector)) {
511           continue;
512         }
513         ir_mode *mode = get_irn_mode(selector);
514         if (!mode_is_int(mode) && mode != get_modeP()) {
515           continue;
516         }
517 #ifdef JT_NO_CF
518         if (get_nodes_block(selector) != get_nodes_block(cond_node)) {
```

```
519         continue;
520       }
521 #endif
522
523       ir_switch_table *switch_table = get_Switch_table(cond_node);
524       for (size_t j = 0, len = ir_switch_table_get_n_entries(switch_table); j < len; j++) {
525         if (ir_switch_table_get_pn(switch_table, j) == get_Proj_num(projx)) {
526           continue;
527         }
528         DB((dbg, LEVEL_1, "Found a Switch node %+F, var %+F, interval [%+F, %+F], dest %+F\n",
529             cond_node, selector, ir_switch_table_get_min(switch_table, j),
530             ir_switch_table_get_max(switch_table, j), projx));
531         cond = get_cond(env, selector, ir_switch_table_get_min(switch_table, j),
532                         ir_switch_table_get_max(switch_table, j),
533                         get_chop_size_incl(get_nodes_block(selector), get_nodes_block(cond_node)), projx);
534       }
535 #endif
536     }
537
538     if (cond != COND_UNKNOWN) {
539       pmap_insert(env->thread_successors, projx, cond);
540
541 #ifdef JT_DUMP_COND_GRAPH
542       ir_obst_vprintf(env, "node: { title: \"n%p\" label: \"%+F\" }\n", projx, projx);
543       obstack_1grow(&env->obst, '\0');
544       char *res = (char *)obstack_finish(&env->obst);
545       fprintf(env->cond_graph, "%s", res);
546       obstack_free(&env->obst, res);
547       fprintf(env->cond_graph, "edge: { sourcename: \"n%p\" targetname: \"n%p\" }\n", projx, cond);
548 #endif
549     }
550   }
551 }
552
553 static jt_edge_info *get_edge_info(jt_environment *env, ir_node *edge)
554 {
555   jt_edge_info *edge_info = pmap_get(jt_edge_info, env->cf_edge_info, edge);
556   if (edge_info == NULL) {
557     edge_info = OALLOC(&env->obst, jt_edge_info);
558     edge_info->relevant_conds = rbitset_obstack_alloc(&env->obst, ARR_LEN(env->conds));
559     edge_info->always_true_conds = rbitset_obstack_alloc(&env->obst, ARR_LEN(env->conds));
560     edge_info->forbidden_conds = rbitset_obstack_alloc(&env->obst, ARR_LEN(env->conds));
561     pmap_insert(env->cf_edge_info, edge, edge_info);
562   }
563
564   return edge_info;
565 }
566
567 /** Adds @p cond to the "relevant_conds" field of all edges between @p block and
568  * the block of cond->node, then calls itself recursively for all predecessor conds. */
569 static void annotate_edges_rec(jt_environment *env, ir_node *block, jt_cond *cond)
570 {
571   if (block == NULL) {
572     return;
573   }
574
575   for (int i = 0, len = get_Block_n_cfgpreds(block); i < len; i++) {
576     ir_node *cfpred = get_Block_cfgpred(block, i);
577     if (block == get_nodes_block(cond->node)) {
578       jt_cond *pred_cond = cond->preds[i];
579       if (pred_cond == &env->cond_always_true) {
580         rbitset_set(get_edge_info(env, cfpred)->always_true_conds, cond->index);
581       } else if (pred_cond != COND_UNKNOWN && pred_cond->visited < get_irg_visited(env->irg)) {
582         pred_cond->visited = get_irg_visited(env->irg);
583         annotate_edges_rec(env, get_nodes_block(cfpred), pred_cond);
584       }
```

```
585     } else if (get_irn_n_edges(cfpred) <= 1 /* "> 1" happens for IJmp nodes, we skip those */
586                && !rbitset_is_set(get_edge_info(env, cfpred)->relevant_conds, cond->index)) {
587       rbitset_set(get_edge_info(env, cfpred)->relevant_conds, cond->index);
588       annotate_edges_rec(env, get_nodes_block(cfpred), cond);
589     }
590   }
591 }
592 
593 /** Annotates all edges in the graph with the conds that are relevant for them
594  * (i.e. should be propagated along them). */
595 static void annotate_edges(jt_environment *env)
596 {
597   foreach_pmap(env->thread_successors, entry) {
598     ir_node *projx = (ir_node *)entry->key;
599     ir_node *projx_block = get_nodes_block(projx);
600     if (projx_block == NULL) {
601       continue;
602     }
603 
604     jt_cond *cond = (jt_cond *)entry->value;
605     foreach_block_succ_safe(projx_block, edge) {
606       ir_node *dst = get_irn_n(get_edge_src_irn(edge), get_edge_src_pos(edge));
607       if (dst != projx) {
608         if (cond == &env->cond_always_true) {
609           // Edges that are never taken are removed right away.
610           set_irn_n(get_edge_src_irn(edge), get_edge_src_pos(edge), new_r_Bad(env->irg, mode_X));
611           add_End_keepalive(get_irg_end(env->irg), projx_block);
612         } else {
613           rbitset_set(get_edge_info(env, dst)->forbidden_conds, cond->index);
614         }
615       }
616     }
617 
618     if (cond->node != NULL) {
619       annotate_edges_rec(env, projx_block, cond);
620     }
621   }
622 }
623 
624 /** Returns whether the duplicate of predecessor @p pred_no of @p succ_block_dupl that is
625  * annotated with @p pred_conds is a predecessor of @p succ_block_dupl.
626  */
627 static bool is_predecessor(jt_environment *env, jt_block_dupl *succ_block_dupl, int pred_no,
628                            unsigned *pred_conds)
629 {
630   jt_edge_info *edge_info = get_edge_info(env, get_Block_cfgpred(succ_block_dupl->dupl_block, pred_no));
631   if (rbitsets_have_common(pred_conds, edge_info->forbidden_conds, ARR_LEN(env->conds))) {
632     return false;
633   }
634 
635   unsigned *succ_conds_expected = rbitset_alloca(ARR_LEN(env->conds));
636   rbitset_copy(succ_conds_expected, pred_conds, ARR_LEN(env->conds));
637   rbitset_and(succ_conds_expected, edge_info->relevant_conds, ARR_LEN(env->conds));
638 
639   /* From here on, only new bits are set in succ_conds_expected. Exit early if we don't have
640    * a subset of succ_block_dupl->conds. */
641   if (!rbitset_contains(succ_conds_expected, succ_block_dupl->conds, ARR_LEN(env->conds))) {
642     return false;
643   }
644 
645   rbitset_or(succ_conds_expected, edge_info->always_true_conds, ARR_LEN(env->conds));
646 
647   /* For each cond in pred_conds that is a predecessor of a cond in the successor block,
648    * activate that cond if the corresponding predecessor cond is activated. */
649   for (size_t i = 0, len = ARR_LEN(env->conds); i < len; i++) {
650     jt_cond *cond = env->conds[i];
```

```
651       if (cond->node != NULL && get_nodes_block(cond->node) == succ_block_dupl->orig_block) {
652         if (cond->preds[pred_no] != &env->cond_always_true &&
653             cond->preds[pred_no] != COND_UNKNOWN &&
654             rbitset_is_set(pred_conds, cond->preds[pred_no]->index)) {
655           rbitset_set(succ_conds_expected, cond->index);
656         }
657       }
658     }
659 
660     return rbitsets_equal(succ_block_dupl->conds, succ_conds_expected, ARR_LEN(env->conds));
661   }
662 
663   /** Returns whether the given array contains a jt_block_dupl* with the given cond set. */
664   static bool contains_cond_set(jt_environment *env, jt_block_dupl **block_dupls, unsigned *cond_set)
665   {
666     for (size_t i = 0, len = ARR_LEN(block_dupls); i < len; i++) {
667       if (rbitsets_equal(block_dupls[i]->conds, cond_set, ARR_LEN(env->conds))) {
668         return true;
669       }
670     }
671     return false;
672   }
673 
674   /** Computes the duplicates needed for the successors of `block_dupl`. */
675   static void get_succ_duplicates(jt_environment *env, jt_block_dupl *block_dupl)
676   {
677     ir_node *block = block_dupl->orig_block;
678     assert(is_Block(block));
679 
680     unsigned *cond_set = block_dupl->conds;
681     assert(cond_set != NULL);
682 
683     DB((dbg, LEVEL_1, "get_succ_duplicates %+F '", block));
684     debug_cond_set(env, cond_set);
685     DB((dbg, LEVEL_1, "'\n"));
686 
687     unsigned *cond_set2 = rbitset_alloca(ARR_LEN(env->conds));
688 
689     foreach_block_succ(block, edge) {
690       rbitset_copy(cond_set2, cond_set, ARR_LEN(env->conds));
691 
692       ir_node *succ_block = get_edge_src_irn(edge);
693       if (get_irn_idx(succ_block) >= env->orig_last_idx) {
694         continue;
695       }
696       int succ_pos = get_edge_src_pos(edge);
697 
698       jt_edge_info *edge_info = get_edge_info(env, get_irn_n(succ_block, succ_pos));
699       if (rbitsets_have_common(cond_set2, edge_info->forbidden_conds, ARR_LEN(env->conds))) {
700         continue;
701       }
702 
703       rbitset_and(cond_set2, edge_info->relevant_conds, ARR_LEN(env->conds));
704       rbitset_or(cond_set2, edge_info->always_true_conds, ARR_LEN(env->conds));
705 
706       for (size_t i = 0, len = ARR_LEN(env->conds); i < len; i++) {
707         jt_cond *cond = env->conds[i];
708         if (cond->node != NULL && get_nodes_block(cond->node) == succ_block
709             && cond->preds[succ_pos] != &env->cond_always_true
710             && cond->preds[succ_pos] != COND_UNKNOWN
711             && rbitset_is_set(cond_set, cond->preds[succ_pos]->index)) {
712           rbitset_set(cond_set2, cond->index);
713         }
714       }
715 
716       ir_node *dupl_block = succ_block;
```

```
717       jt_block_dupl **succ_duplicates = pmap_get(jt_block_dupl *, env->block_dupls, succ_block);
718       if (succ_duplicates == NULL) {
719         succ_duplicates = NEW_ARR_F(jt_block_dupl *, 0);
720       } else {
721         if (contains_cond_set(env, succ_duplicates, cond_set2)) {
722           continue;
723         }
724         dupl_block = exact_copy(succ_block);
725         DB((dbg, LEVEL_1, ">>> Duplicating %+F for '", succ_block));
726         debug_cond_set(env, cond_set2);
727         DB((dbg, LEVEL_1, "' as %+F\n", dupl_block));
728       }
729
730       jt_block_dupl *entry = OALLOC(&env->obst, jt_block_dupl);
731       entry->orig_block = succ_block;
732       entry->dupl_block = dupl_block;
733       entry->conds = rbitset_duplicate_obstack_alloc(&env->obst, cond_set2, ARR_LEN(env->conds));
734       set_irn_link(dupl_block, entry);
735       ARR_APP1(jt_block_dupl *, succ_duplicates, entry);
736
737       pmap_insert(env->block_dupls, succ_block, succ_duplicates); // ARR_APP1 might modify the array pointer
738
739       get_succ_duplicates(env, succ_duplicates[ARR_LEN(succ_duplicates) - 1]);
740     }
741   }
742
743   static ir_node *get_original_node(ir_node *node)
744   {
745     return irn_visited(node) ? get_irn_link(node) : node;
746   }
747
748   static bool is_original_node(jt_environment *env, ir_node *node)
749   {
750     return get_irn_idx(node) < env->orig_last_idx;
751   }
752
753   /**
754    * Adjusts the control flow predecessors for the given @block_dupl. It can gain additional predecessors
755    * if it is reachable from more than one duplicate of an original predecessor; it can lose predecessors
756    * if one of its predecessors now jumps to another duplicate of the same block.
757    *
758    * If @block_dupl is an original block, we do not remove any predecessors, but instead replace them
759    * by Bad nodes so that the predecessors of Phi nodes can be determined correctly.
760    *
761    * This operation need only be performed once per block and is therefore guarded by the visited flag.
762    * If the block has not been visited yet, it has the same predecessors as its original block.
763    */
764   static void get_predecessors(jt_environment *env, jt_block_dupl *block_dupl)
765   {
766     assert(block_dupl != NULL);
767     assert(is_Block(block_dupl->dupl_block));
768     assert(is_Block(block_dupl->orig_block));
769
770     if (irn_visited_else_mark(block_dupl->dupl_block)) {
771       return;
772     }
773
774     ir_node **preds = NEW_ARR_F(ir_node *, get_Block_n_cfgpreds(block_dupl->dupl_block));
775     for (size_t i = 0, len = get_Block_n_cfgpreds(block_dupl->dupl_block); i < len; i++) {
776       ir_node *pred_node = get_Block_cfgpred(block_dupl->dupl_block, i);
777       ir_node *pred_block = get_nodes_block(pred_node);
778       if (pred_block == NULL) {
779         preds[i] = pred_node;
780         mark_irn_visited(pred_node);
781         continue;
782       }
```

```
783
784     jt_block_dupl **pred_duplicates = pmap_get(jt_block_dupl *, env->block_dupls, pred_block);
785     if (pred_duplicates == NULL) {
786       // pred_block is not reachable from Start block
787       preds[i] = new_r_Bad(env->irg, mode_X);
788       mark_irn_visited(pred_node);
789       add_End_keepalive(get_irg_end(env->irg), pred_block);
790       continue;
791     }
792
793     bool found = false;
794     for (size_t j = 0, len = ARR_LEN(pred_duplicates); j < len; j++) {
795       assert(is_Block(pred_duplicates[j]->orig_block));
796       assert(is_Block(pred_duplicates[j]->dupl_block));
797       if (is_predecessor(env, block_dupl, i, pred_duplicates[j]->conds)) {
798         jt_block_dupl *new_pred_block = pred_duplicates[j];
799         ir_node *new_pred;
800
801         jt_cond *fulfilled_cond = pmap_get(jt_cond, env->thread_successors, pred_node);
802         if (fulfilled_cond != NULL && (fulfilled_cond == &env->cond_always_true ||
803                                       rbitset_is_set(new_pred_block->conds, fulfilled_cond->index))) {
804           new_pred = new_r_Jmp(new_pred_block->dupl_block);
805         } else if (get_nodes_block(pred_node) == new_pred_block->dupl_block) {
806           new_pred = pred_node;
807         } else {
808           new_pred = exact_copy(pred_node);
809           set_nodes_block(new_pred, new_pred_block->dupl_block);
810           env->num_node_dupls_exact++;
811         }
812         set_irn_link(new_pred, pred_node);
813         mark_irn_visited(new_pred);
814
815         // The first predecessor can be inserted at the original position, all others are appended at the end.
816         if (!found) {
817           preds[i] = new_pred;
818           found = true;
819         } else {
820           ARR_APP1(ir_node *, preds, new_pred);
821         }
822       }
823     }
824     if (!found) {
825       preds[i] = new_r_Bad(env->irg, mode_X);
826     }
827   }
828
829   set_irn_in(block_dupl->dupl_block, ARR_LEN(preds), preds);
830 }
831
832 static ir_node *get_dominating_def(jt_environment *env, ir_node *node, ir_node *block,
833                                    pmap *dominating_def_cache);
834
835 static void get_pred_dominating_defs(jt_environment *env, ir_node *node)
836 {
837   assert(!is_Phi(node));
838   for (int i = 0, len = get_irn_arity(node); i < len; i++) {
839     ir_node *pred_node = get_original_node(get_irn_n(node, i));
840     if (!is_Bad(pred_node) && !is_Dummy(pred_node) && !is_Const(pred_node)) {
841       set_irn_n(node, i, get_dominating_def(env, pred_node, get_nodes_block(node), NULL));
842     }
843   }
844 }
845
846 /** Returns a definition of @node (either @node itself, or a duplicate, or a newly created Phi node)
847  * whose block dominates @block. */
848 static ir_node *get_dominating_def(jt_environment *env, ir_node *node, ir_node *block,
```

```
849                                    pmap *dominating_def_cache)
850 {
851   assert(is_Block(block));
852   jt_block_dupl *block_dupl = get_irn_link(block);
853 
854   if (block_dupl == NULL) {
855     // If no duplicates for this block have been created, it is not reachable
856     // from the Start node and can therefore be removed.
857     return new_r_Bad(env->irg, get_irn_mode(node));
858   }
859 
860   assert(block_dupl->dupl_block == block);
861   if (is_original_node(env, block)) {
862     assert(block_dupl->dupl_block == block_dupl->orig_block);
863   } else {
864     assert(block_dupl->dupl_block != block_dupl->orig_block);
865   }
866 
867   assert(is_original_node(env, node));
868 
869   // Try to fetch a cached result.
870   ir_node *result = NULL;
871   if (dominating_def_cache == NULL) {
872     dominating_def_cache = pmap_get(pmap, env->dominating_defs, node);
873   }
874 
875   if (dominating_def_cache == NULL) {
876     // Shortcut: if the original node's block has not been duplicated, there is only one definition.
877     // We still need to fix this node's predecessors though.
878     if (ARR_LEN(pmap_get(jt_block_dupl *, env->block_dupls, get_nodes_block(node))) == 1) {
879       result = node;
880       block = get_nodes_block(node);
881       block_dupl = get_irn_link(block);
882     } else {
883       dominating_def_cache = pmap_create();
884       pmap_insert(env->dominating_defs, node, dominating_def_cache);
885     }
886   } else {
887     result = pmap_get(ir_node, dominating_def_cache, block);
888   }
889 
890   if (result == NULL) {
891     // No cached result found
892     if (get_nodes_block(node) == block) {
893       // Same block -- return the original node
894       result = node;
895     } else if (get_nodes_block(node) == block_dupl->orig_block) {
896       // Duplicate of the definition block -- we need to duplicate the definition
897       result = exact_copy(node);
898       set_nodes_block(result, block);
899       env->num_node_dupls_exact++;
900     } else {
901       // None of the above -- we might need to insert a Phi node here, but we know that only after
902       // recursing. Insert a pending Phi for now, except if we have only one control flow predecessor.
903       get_predecessors(env, block_dupl);
904       size_t n_cfgpreds = get_Block_n_cfgpreds(block);
905 
906       if (n_cfgpreds == 1) {
907         ir_node *pred_block = get_Block_cfgpred_block(block, 0);
908         result = pred_block == NULL ? new_r_Bad(env->irg, get_irn_mode(node)) :
909                  get_dominating_def(env, node, pred_block, dominating_def_cache);
910         block = get_nodes_block(result);
911         if (block == NULL) {
912           return result;
913         }
914         block_dupl = get_irn_link(block);
```

```
915         } else {
916           ir_node *dummy = new_r_Dummy(env->irg, get_irn_mode(node));
917           ir_node **phi_ins = ALLOCAN(ir_node *, n_cfgpreds);
918           for (size_t i = 0; i < n_cfgpreds; i++) {
919             phi_ins[i] = dummy;
920           }
921           result = new_r_Phi(block, n_cfgpreds, phi_ins, get_irn_mode(node));
922         }
923       }
924     }
925
926     assert(result != NULL);
927
928     // Recursively fix the arguments of the result. If it is a Phi, we might additionally need to adjust
929     // the number of arguments to match the number of the block's cfpreds. We needd to do this only once.
930     if (!irn_visited_else_mark(result)) {
931       set_irn_link(result, node);
932       if (dominating_def_cache != NULL) {
933         // Insert result into the cache before making any recursive calls.
934         pmap_insert(dominating_def_cache, block, result);
935       }
936
937       get_predecessors(env, block_dupl);
938       size_t n_cfgpreds = get_Block_n_cfgpreds(block);
939
940       if (!is_Phi(result)) {
941         get_pred_dominating_defs(env, result);
942       } else if (block_dupl->orig_block == get_nodes_block(node)) {
943         // result is equal to node or a duplicate thereof
944
945         // All predecessor blocks are duplicates of the first n blocks, where n is the number of
946         // predecessors of the original block before Jump Threading. We do not know the value of n, but we know:
947         //  (1) If result == node, then n == get_irn_arity(result), because at this point, result still has the
948         //      original number of arguments
949         size_t known_n_cfgpreds = result == node ? get_irn_arity(result) : 0;
950         assert(n_cfgpreds >= known_n_cfgpreds);
951
952         //  (2) From the definition of get_predecessors, Bad predecessors can only occur within the first n
953         //      predecessors of the original or duplicate block, so when encountering a Bad at position i,
954         //      we know that 0 <= i <= n-1.
955
956         // Adjust the number of Phi preds
957         if ((size_t)get_irn_arity(result) != n_cfgpreds) {
958           ir_node **phi_ins = ALLOCAN(ir_node *, n_cfgpreds);
959           ir_node *dummy = new_r_Dummy(env->irg, get_irn_mode(result));
960           for (size_t i = 0; i < n_cfgpreds; i++) {
961             phi_ins[i] = i < known_n_cfgpreds ? get_irn_n(result, i) : dummy;
962           }
963           set_irn_in(result, n_cfgpreds, phi_ins);
964         }
965
966         // Adjust the first "known_n_cfgpreds" predecessors
967         for (size_t i = 0; i < known_n_cfgpreds; i++) {
968           assert(is_original_node(env, get_irn_n(result, i)));
969
970           if (get_Block_cfgpred_block(block, i) != NULL) {
971             set_irn_n(result, i, get_dominating_def(env, get_irn_n(result, i),
972                                                     get_Block_cfgpred_block(block, i), NULL));
973           } else {
974             // Even though this duplicate block's cfpred has been replaced by a Bad,
975             // the Phi pred is still relevant in block duplicates where this is not the case.
976             // Still have to call get_dominating_def on the pred to trigger control flow changes.
977             get_dominating_def(env, get_irn_n(result, i), get_nodes_block(get_irn_n(result, i)), NULL);
978           }
979         }
980
```

```
981         // Adjust the rest of the predecessors. For every i, find a j in 0..n-1 so that
982         // cfgpred i is a duplicate of cfgpred j
983         size_t orig_n_cfgpreds = get_Block_n_cfgpreds(block_dupl->orig_block);
984         for (size_t i = known_n_cfgpreds; i < n_cfgpreds; i++) {
985           ir_node *cfgpred = get_original_node(get_Block_cfgpred(block, i));
986           if (is_Bad(cfgpred)) { // see (2) above
987             set_irn_n(result, i, new_r_Bad(env->irg, get_irn_mode(result)));
988           } else if (i < orig_n_cfgpreds && is_Bad(get_Block_cfgpred(block_dupl->orig_block, i))) {
989             set_irn_n(result, i, get_dominating_def(env,
990                                                      get_original_node(get_irn_n(node, i)),
991                                                      get_Block_cfgpred_block(block, i), NULL));
992           } else {
993             bool found = false;
994             for (size_t j = 0; j < n_cfgpreds && j < orig_n_cfgpreds; j++) {
995               ir_node *orig_pred_j = get_Block_cfgpred(block_dupl->orig_block, j);
996               ir_node *dupl_pred_j = get_Block_cfgpred(block_dupl->dupl_block, j);
997
998               if ((!is_Bad(orig_pred_j) && get_original_node(orig_pred_j) == cfgpred)
999                   || (!is_Bad(dupl_pred_j) && get_original_node(dupl_pred_j) == cfgpred)) {
1000                found = true;
1001
1002                set_irn_n(result, i, get_dominating_def(env,
1003                                                         get_original_node(get_irn_n(node, j)),
1004                                                         get_Block_cfgpred_block(block, i), NULL));
1005                break;
1006              }
1007            }
1008            assert(found);
1009          }
1010        }
1011      } else {
1012        // We have a pending Phi. Eliminate it if all its non-Bad, non-self predecessors are the same.
1013        ir_node *first_nonbad_pred = NULL;
1014        bool need_phi = false;
1015        for (size_t i = 0; i < n_cfgpreds; i++) {
1016          ir_node *pred_block = get_Block_cfgpred_block(block, i);
1017          ir_node *pred_dominating_def = pred_block == NULL ?
1018                                     new_r_Bad(env->irg, get_irn_mode(node)) :
1019                                     get_dominating_def(env, node, pred_block, dominating_def_cache);
1020          set_irn_n(result, i, pred_dominating_def);
1021          if (!is_Bad(pred_dominating_def)) {
1022            if (first_nonbad_pred == NULL) {
1023              first_nonbad_pred = pred_dominating_def;
1024            } else {
1025              need_phi |= (i > 0 && pred_dominating_def != result && pred_dominating_def != first_nonbad_pred);
1026            }
1027          }
1028        }
1029        if (!need_phi) {
1030          ir_node *result_old = result;
1031          result = first_nonbad_pred;
1032          assert(irn_visited(result));
1033          assert(get_irn_link(result) == node);
1034          exchange(result_old, result);
1035          // The dummy Phi might linger around somewhere in the cache from recursive calls.
1036          // Exchange it there as well.
1037          foreach_pmap(dominating_def_cache, entry) {
1038            if (entry->value == result_old) {
1039              entry->value = result;
1040            }
1041          }
1042        }
1043      }
1044
1045      // We might have created a new loop block.
1046      if (is_Phi(result) && get_irn_mode(node) == mode_M) {
```

```
1047        set_Phi_loop(result, 1);
1048        add_End_keepalive(get_irg_end(env->irg), result);
1049      }
1050    }
1051
1052    assert(result != NULL);
1053    return result;
1054  }
1055
1056  void opt_jumpthreading(ir_graph *irg)
1057  {
1058    FIRM_DBG_REGISTER(dbg, "firm.opt.jumpthreading");
1059    DB((dbg, LEVEL_1, "===> Performing jumpthreading on %+F\n", irg));
1060
1061    assure_irg_properties(irg,
1062                          IR_GRAPH_PROPERTY_NO_UNREACHABLE_CODE
1063                          | IR_GRAPH_PROPERTY_CONSISTENT_OUT_EDGES
1064  #ifdef JT_NO_CRITICAL_EDGES
1065                          | IR_GRAPH_PROPERTY_NO_CRITICAL_EDGES
1066  #endif
1067                          | IR_GRAPH_PROPERTY_CONSISTENT_LOOPINFO
1068                         );
1069
1070    ir_reserve_resources(irg, IR_RESOURCE_IRN_LINK | IR_RESOURCE_IRN_VISITED);
1071
1072    jt_environment env;
1073    env.irg = irg;
1074    env.orig_last_idx = get_irg_last_idx(irg);
1075    obstack_init(&env.obst);
1076    env.cond_always_true.node = NULL;
1077    env.conds = NEW_ARR_F(jt_cond *, 0);
1078    env.cf_edge_info = pmap_create();
1079    env.thread_successors = pmap_create();
1080    env.block_dupls = pmap_create();
1081    env.dominating_defs = pmap_create();
1082    env.num_node_dupls_exact = 0;
1083
1084  #ifdef JT_DUMP_COND_GRAPH
1085    obstack_printf(&env.obst, "%s-%02u-jt-conds.vcg", get_irg_dump_name(irg), irg->dump_nr++);
1086    obstack_1grow(&env.obst, '\0');
1087    char *file_name = (char *)obstack_finish(&env.obst);
1088    env.cond_graph = fopen(file_name, "wb");
1089    obstack_free(&env.obst, file_name);
1090  #endif
1091
1092    // Find threading opportunities
1093    inc_irg_visited(irg);
1094  #ifdef JT_DUMP_COND_GRAPH
1095    fprintf(env.cond_graph, "graph: { title: \"cond graph of %s\"\n", get_entity_name(get_irg_entity(irg)));
1096    fprintf(env.cond_graph, "node: { title: \"n%p\" label: \"COND_ALWAYS_TRUE\" }\n", &env.cond_always_true);
1097  #endif
1098    irg_block_walk_graph(irg, init_and_find_cmps, NULL, &env);
1099  #ifdef JT_DUMP_COND_GRAPH
1100    fprintf(env.cond_graph, "}\n");
1101    fclose(env.cond_graph);
1102  #endif
1103
1104    // Exit early if no threading opportunities found
1105    if (ARR_LEN(env.conds) == 0) {
1106      ir_free_resources(irg, IR_RESOURCE_IRN_LINK | IR_RESOURCE_IRN_VISITED);
1107      obstack_free(&env.obst, NULL);
1108      return;
1109    }
1110
1111    inc_irg_visited(irg);
1112    annotate_edges(&env);
```

```
1113
1114    // Compute the needed duplicates for each block. The start and end block are never duplicated.
1115    jt_block_dupl **start_dupl = NEW_ARR_F(jt_block_dupl *, 1);
1116    start_dupl[0] = OALLOC(&env.obst, jt_block_dupl);
1117    start_dupl[0]->orig_block = get_irg_start_block(irg);
1118    start_dupl[0]->dupl_block = get_irg_start_block(irg);
1119    start_dupl[0]->conds = rbitset_obstack_alloc(&env.obst, ARR_LEN(env.conds));
1120    pmap_insert(env.block_dupls, get_irg_start_block(irg), start_dupl);
1121    set_irn_link(get_irg_start_block(irg), start_dupl[0]);
1122
1123    jt_block_dupl **end_dupl = NEW_ARR_F(jt_block_dupl *, 1);
1124    end_dupl[0] = OALLOC(&env.obst, jt_block_dupl);
1125    end_dupl[0]->orig_block = get_irg_end_block(irg);
1126    end_dupl[0]->dupl_block = get_irg_end_block(irg);
1127    end_dupl[0]->conds = rbitset_obstack_alloc(&env.obst, ARR_LEN(env.conds));
1128    pmap_insert(env.block_dupls, get_irg_end_block(irg), end_dupl);
1129    set_irn_link(get_irg_end_block(irg), end_dupl[0]);
1130
1131    get_succ_duplicates(&env, start_dupl[0]);
1132    inc_irg_visited(irg);
1133
1134    // Fix control flow predecessors for each block duplicate
1135    inc_irg_visited(irg);
1136    foreach_pmap(env.block_dupls, entry) {
1137      jt_block_dupl **dupls = entry->value;
1138      for (size_t i = 0, len = ARR_LEN(dupls); i < len; i++) {
1139        get_predecessors(&env, dupls[i]);
1140
1141        for (size_t j = 0, len = get_Block_n_cfgpreds(dupls[i]->dupl_block); j < len; j++) {
1142          ir_node *cfpred = get_Block_cfgpred(dupls[i]->dupl_block, j);
1143          ir_node *pred_block = get_nodes_block(cfpred);
1144          if (pred_block != NULL) {
1145            // Recursively get the dominating definitions of nodes reachable via regular control flow
1146            get_pred_dominating_defs(&env, cfpred);
1147          }
1148        }
1149      }
1150    }
1151
1152    // We need to keep the duplicate block alive if the original was kept alive.
1153    // Also we need to get the predecessor definitions of kept alive non-Block nodes.
1154    for (int i = 0, len = get_End_n_keepalives(get_irg_end(irg)); i < len; i++) {
1155      ir_node *keepalive = get_End_keepalive(get_irg_end(irg), i);
1156      if (!is_original_node(&env, keepalive)) {
1157        continue;
1158      }
1159      if (is_Block(keepalive)) {
1160        jt_block_dupl **block_dupls = pmap_get(jt_block_dupl *, env.block_dupls, keepalive);
1161        if (block_dupls != NULL) {
1162          for (size_t j = 0, len = ARR_LEN(block_dupls); j < len; j++) {
1163            if (block_dupls[j]->dupl_block != keepalive) {
1164              add_End_keepalive(get_irg_end(irg), block_dupls[j]->dupl_block);
1165            }
1166          }
1167        }
1168      } else if (get_nodes_block(keepalive) != NULL) {
1169        jt_block_dupl **block_dupls = pmap_get(jt_block_dupl *, env.block_dupls, get_nodes_block(keepalive));
1170        if (block_dupls != NULL) {
1171          for (size_t j = 0, len = ARR_LEN(block_dupls); j < len; j++) {
1172            // might need to adjust Phi inputs
1173            ir_node *dom_def = get_dominating_def(&env, keepalive, block_dupls[j]->dupl_block, NULL);
1174            if (block_dupls[j]->dupl_block != get_nodes_block(keepalive)) {
1175              add_End_keepalive(get_irg_end(irg), dom_def);
1176            }
1177          }
1178        }
```

```
1179      }
1180    }
1181    inc_irg_visited(irg);
1182
1183    ir_free_resources(irg, IR_RESOURCE_IRN_LINK | IR_RESOURCE_IRN_VISITED);
1184    obstack_free(&env.obst, NULL);
1185    confirm_irg_properties(irg, IR_GRAPH_PROPERTIES_NONE);
1186  }
```