Predication and Speculation

Compilers for High Performance Computers

Stefano Petrilli
stefano.petrilli@upc.edu

Jakob Eberhardt
jakob.eberhardt@estudiantat.upc.edu

November 19, 2024

Contents

Listings

List of Figures

List of Tables

1 Introduction

Superscalar processors with deep pipelines and out-of-order execution represent the state of the art in processor architecture.

Superscalar processors have multiple execution unit which allows them to fetch, decode, and execute multiple instructions in parallel through separate pipeline stages within a single clock cycle. This increases the instructions throughput and allows for an Instructions Per Cycle (IPC) count that is greater than 1. The latest main vendor's architectures, AMD Zen5 and Intel Alder Lake, feature cores that are respectively 8 and 6 instructions wide.

When talking about deep pipelines, we refer to the number of stages that the instructions have to go through from the fetch to the moment they are retired. High performance design employ different pipelines for different classes of instructions. For both the last generation of AMD and Intel desktop processors, the depth of the integer pipeline is estimated to be 19 stages.

When talking about Out of Order Execution, we refer to architecture design where instructions can be executed as soon as their input operands are ready, rather than strictly in the original program order. By executing instructions out of order, the processor can keep its execution units busy even when some instructions are stalled due to delays like cache misses or data dependencies.

These design choices proved to deliver unmatched performances when provided with enough Instruction Level Parallelism (ILP). Based on the work of August et al. [1], the main obstacles to high ILP are difficult to predict control flows and ambiguous memory dependencies.

1.1 The cost of branch misprediction

The reason why branch mispredictions represent a big obstacle for modern machines has to do with the design choices described so far.

Another central component of modern designs are branch predictors. Branch predictors allow for one side of conditional branches to be speculatively executed before the branch is evaluated. When the speculated side and the evaluated side coincide, we have the benefit of keeping the pipeline full and executing instructions ahead of time. When these speculations fail, tough, the instructions speculatively executed will need to be discarded and the right side will need to be executed. Due to out of order execution, between the time the branch is speculatively executed and the time it is evaluated, a high number of instruction might have been retired. In addition to this, as the pipelines are deep, several cycles have to pass before the pipeline is filled again and start retiring new instruction. Based on the work of Kwan Lin et al. [6], branch misprediction account for 20% of the IPC in modern processor and is representing the main limit to having deeper, more efficient pipelines. A good empirical average measure of the cost of a branch misprediction comes from the weight used in the heuristics of *llvm* [5].

Architecture	**Misprediction Penalty**	**Optimistic Load Cost**
Sapphire Rapids	14	5
Alder Lake-P	14	5
Ice Lake	14	5
Broadwell	16	5
Haswell	16	5
Cortex A57	14	4
Cortex R52	8	1
Cortex M4	2	2

Table 1: Branch Misprediction Penalty and Optimistic Load Cost used in *llvm*'s heuristics for various Intel and ARM architectures.

From the weight used in *llvm* for both branch misprediction and optimistic load cost reported in Table 1, we can conclude that a branch misprediction, in a modern high performance architecture, is three times more expensive than a load operation that hits the L1 cache. Certain design choices peculiar to the embedded field make the cost of branch misprediction higher or lower. The data regarding `Cortex R52` and `Cortex M4` have been cherry-picked precisely to describe this. The `Cortex R52` is a design that implements advanced safety features like lockstep redundancy which introduces further overhead in case of misprediction. Due to these characteristics, it has a misprediction penalty to load cost ratio of 8:1. `Cortex M4` is also an embedded processor but has a 3 stages pipeline and execute instructions in order. This results in a ratio of branch misprediction cost to load cost of 1:1. The way the weights reported are used by the compiler's heuristics are treated in more detail in Sections **??**, **??** and **??**.

1.2 Compilation techniques to improve IPC

Two techniques have been introduced into compilers to try to mitigate the obstacles to high IPC described so far:

- **Speculation**: Is a technique where the compiler makes assumptions about the program's behavior to generate optimized code paths, potentially executing certain operations early or avoiding them entirely. If the speculated outcomes are incorrect at runtime, mechanisms like rollback or patching are used to maintain correctness. This has not to be confused with branch speculation, which is executed at the architectural level with the aid of the branch predictor.
- **Predication**: Instead of using conditional branches to decide which instructions to execute, predicated instructions execute all possible paths but only commit the results of the path that meets a specific condition, known as the predicate. This is only possible when the Instruction Set Architecture (ISA) supports predicated instructions.

We discuss in more detail how Speculation and Predication help delivery higher performances respectively in Section 2 and in Section 3. Additionally, we discuss in Section 4 how these techniques can be combined to achieve even greater benefit in Section 3

2 Speculation

Whenever we move an instruction above a branch it is control-flow dependent on, we consider this instruction to be *speculatively* executed [3]. When speculating during compilation, we typically try to tailor the control-flow of the program towards the actual behavior we expect during runtime. The most common types of speculations include loads, computations and stores. While these may increase performance in certain cases, they each come with considerations and hazards one has to be aware of. In this section, we will study the concepts and heuristics behind speculative execution.

2.1 Example

```
if (x > 0) {
    y = array[x];
}
```

Listing 1: Setting the value of `y` depends on `x` being positive.

```
int temp = array[x];
if (x > 0) {
  y = temp;
}
```

Listing 2: We can speculatively load the value of `array[x]` into a additional variable.

```
LDR     r3, [r0]
CMP     r3, #0
BLE     skip_load_and_store
LDR     r1, [r2, r3, LSL #2]
STR     r1, [r4]
skip_load_and_store:
```

Listing 3: Once we know the branch is not taken, we issue a load of `data[x] into r1`

```
LDR     r3, [r0]
LDR     r1, [r2, r3, LSL #2]
CMP     r3, #0
BLE     skip_store
STR     r1, [r4]
skip_store:
```

Listing 4: We immediately issue a load for `data[x]`, even tough we do not know yet if it will be needed.

2.2 Superblock Structure

If we consider the control flow of a program, a superblock [7] is a consecutive sequence of instruction which is always entered at the top-most basic block but may be left at arbitrary and multiple points. When forming superblock, we try to anticipate the future control flow of the program during run time by either profiling it with synthetic data or other means [4]. An example of how the profile of a loop execution may look like can be seen in figure 1. Note that most iterations consist of BB2, BB3 and BB5. Blocks which are frequently executed together are also called traces. To create a superblock from this trace, we need to remove the side entrance from BB4 to BB5 and hence to the trace section. Typically, this done with tail duplication. In this case, we need to add BB5' to remove the side entrance, as can be seen in figure 2. By forming superblocks, we can avoid bookkeeping complexity which typically arises during trace scheduling when we move instructions across side entrances. Note that also BB4 and BB5' now form a superblock. Superblocks are not exclusively tools of speculation, rather they are also important for enabling other optimizations in compilers.

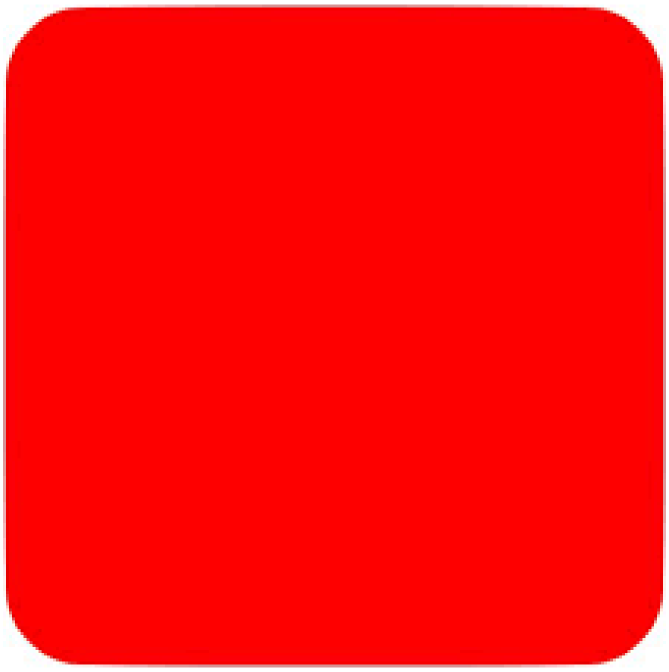


Figure 1: Trace with Side Entrance
The edge from BB4 to BB5 is a so-called side entrance into thew the most important trace consisting of BB2, BB3, and BB5.

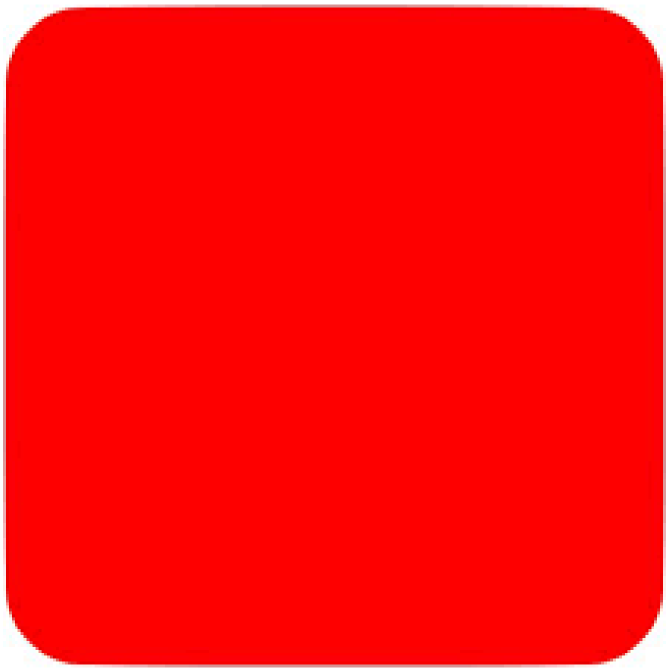


Figure 2: Superblock Example
In this example, we removed the side entrance into the trace by duplicating BB5 (BB5')
for the case where BB2 is branching to BB4.

2.3 Superblock Scheduling

After the trace selection and the formation of superblocks, the compiler has to schedule the instructions with the goal of enabling as much ILP as possible and the constraint of producing a save and correct program. This is done by building a graph which represents the dependencies among the instructions. The dependency graph is used during and a subsequent list scheduling. The compiler may use heuristics and given instruction latencies to assign priorities to instructions which are *ready* to produce a schedule which will use the available resources as efficient as possible [3]. To this end, the compiler needs to be able to move instructions above previous branches which is considered a speculation.

2.4 Speculative Code Motion Models

In many cases, higher performance can be achieved by moving certain instructions either down or upward. However, in both cases we have to retain the correctness of the original program. A variable or register is called *live* if it holds a value that will be needed by other instructions. For branching instructions, we can define `live-out(Br)` as the set of variables which may be used before being defined in case `Br` will be taken. Downward code motion, meaning moving a instruction `I` beneath a later instruction `Br` can always be applied if `Br` is not dependent on the result of `I`. In case the destination register of `I` is in `live-out(Br)`, we have to insert a copy of `I` between `Br` and its target to insure correctness, as can be seen in listing 5

```
    LDR     r1, [R2]
    CMP     r3, #0
    BEQ     Target
    ADD     r4, r1, r5
    ; ...
Target:
    SUB     r6, r1, r7
```

Listing 5: Since `r1` is needed in the branch target of the `BEQ` instruction, `live-out(BEQ)` contains `r1`.

```
    CMP     r3, #0
    BEQ     Target
    LDR     r1, [R2]
    ADD     r4, r1, r5
    ; ...
Target:
    LDR     r1, [r2]
    SUB     r6, r1, r7
```

Listing 6: We can delay the `LDR` instruction by moving it downward. However, in case the `BEQ`-branch is taken, we need to load the correct value using a copy of the `LDR` instruction.

Although downward motion can be easily done, the cases where we will gain performance are limited. Moving an instruction from a point after a branch to a point before the branch is refereed to as upward motion. Typically, one may try to schedule long-latency instructions early to reduce the critical path, for example in a superblock. Similar to other architectures, the division instruction `SDIV` is a example of such a long-latency instruction. As can be seen in listing 7, depending on the input data, we may have to execute `SDIV`. Therefore it would we preferable to speculatively execute the division as can be seen in listing 8. However, the destination of `SDIV` (`r1`) is in `live-out(BEQ)` if it is taken, hence the speculation would alter the result if the compiler does not introduce compensation code, e.g. a copy of `r1` for the `taken` case. Additionally, `SDIV` may cause an exception since it was moved above the branch which checks the loaded value in `r1` to be zero.

```
    LDR r1, [address]
    CMP r1, #0
    BEQ taken
    SDIV r1, #5, r1
    B end
taken:
    ADD r1, r1, #2
end:
```

Listing 7: Depending on the input data, we may have to execute a division which typically takes many cycles in most architectures.

```
    LDR r1, [address2]
    SDIV r1, #5, r1
    CMP r1, #0
    BEQ taken
    B end
taken:
    ADD r1, r1, #2
end:
```

Listing 8: The long-latency division was naively moved upwards. Depending on the input data, the program will now compute wrong results, because the destination register of the speculatively executed division `r1` is in `live-out(BEQ)`. In addition, `SDIV` may cause a unnecessary a divide-by-zero exception.

Restriction 1: The destination register of `I` is not part of `live-out(Br)`. Hence, the result of `I` is not used before it is redefined when the branch was taken.

Restriction 2: Instruction `I` will not cause an exception which will alter the program execution if `Br` was taken.

Restricted Percolation is a conservative code motion technique where the compiler moves instructions across basic blocks or control-flow boundaries under strict safety conditions. Instructions are only moved when it is guaranteed that such movement will not alter the program's observable behavior. Minimal or no speculation is involved, and the movement is based on provable safety. The technique strictly respects data dependencies (ensuring that an instruction's operands are ready) and control dependencies (ensuring the instruction is only executed when it should be).

3 Predication

For the reasons outlined in Section 1.1, we can conclude that branches are expensive. Even when using state of the art branch predictors, certain branches are systematically hard to predict or have such low occurrences that not enough data are collected to perform a good prediction [6]. Predication is used to replace conditional branches with other instructions. In *LLVM* and in *GCC*, this is usually done during the backend optimization phase where the compiler works on the machine level representation of the code. The name that is generally given to this code optimization pass is `if-conversion`.

```
if (x > 0) y = 10;    ⟶    y = (x > 0)  * 10 +
else       y = 20;             (x <= 0) * 20;
```

Figure 3: Replacing a conditional branch with arithmetic operations.

An example of this is presented in Figure 3 where code containing a conditional branch is replaced with code that performs exactly the same operation without the use of conditional branches. The example presented uses arithmetic properties to replace the branch and is usually referred to as **Arithmetic Predication** or **Branchless Programming**. Using Arithmetic Predication to replace branches presents limitations: multiplications are also expensive, and it's difficult to apply most of the time.

One way to overcome these limitations is by utilizing predicated instructions supported by the architecture, when these are available. A predicated instruction in `ARMv7` is an instruction which attaches conditions directly to instructions, allowing them to either execute or skip based on the condition's outcome.

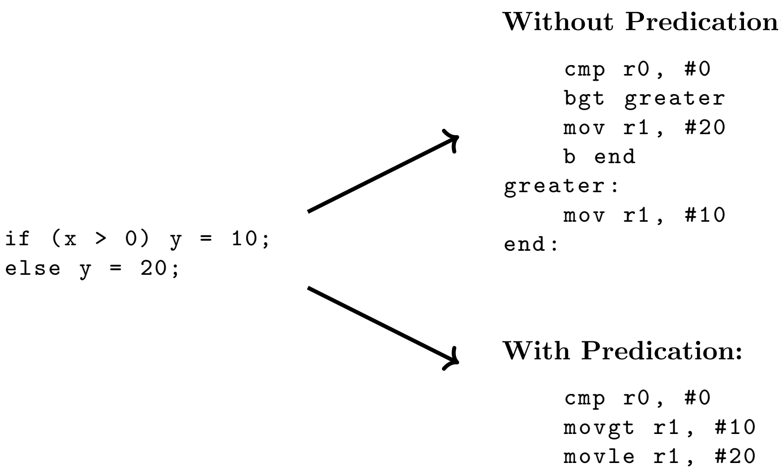
```
MOV R0, R1
```

Listing 9: Standard MOV in ARMv7 assembly

```
MOVGT R0, R1
```

Listing 10: Predicated MOV in ARMv7 assembly

The ARMv7 architecture supports several predicated instructions. For example, the standard MOV instructions in Listing 3 takes the value in R1 and moves it into R0 while, the predicated version, MOVGT, showed in Listing 3 moves the value in R1 to R0 only when the last comparison was "greater than".



```
if (x > 0) y = 10;
else y = 20;
```

Without Predication:

```
    cmp r0, #0
    bgt greater
    mov r1, #20
    b end
greater:
    mov r1, #10
end:
```

With Predication:

```
    cmp r0, #0
    movgt r1, #10
    movle r1, #20
```

Figure 4: Branching versus predication in ARM assembly

In Figure 4 two assembly translation of the code in Figure 3 are displayed. The top one is when no predication is used, while the bottom one uses predicated instructions. By comparing the two versions in Figure 4, the benefits of predication are staggering. The code size is reduced from 5 instructions to 3, while removing one conditional branch and one unconditional branch. Another benefit that is outlined by this example is **Deterministic Execution**. It's difficult to predict how many cycles an assembly code needs to be executed when branch predictions and other form of speculations are involved. As we removed all the branches and as we have no memory accesses in our snippet of code, we can determine exactly how long it's going to take to execute the operation. Deterministic Execution is crucial in certain fields where software needs to have a predictable behavior and respond within strict timing constraints. This is necessary for instance in flight control systems, in medical devices and other mission-critical software. Section 3.1 focuses on a more in depth analysis of how ARMv7 and IA64 architectures introduce support for predication.

It's also necessary to mention some limitations of predication.

- **Increased Instruction Executions**: predication can lead to unnecessary execution of both paths in a conditional statement. In certain cases this may lead to a very inefficient use of the CPU and to wasting several cycles. This is particularly true when the predicated code has long conditional chains.
- **Higher register pressure**: as we might need to hold intermediate results in registers, the register pressure increases. Registers are a highly contented resource, and allocating them efficiently is a difficult problem in compilers [2]. If the increased pressure introduced by predication results in spilling to memory, performance degradation is to be expected.

- **Code Size**: While the example presented, demonstrates how predication can result in code size reduction, this is not necessary the case. In fact, as a more complex control flow is taken into account, code size is more likely to increment.

For these reasons, if predication is applied indiscriminately, performance regression happen. Production compilers like *LLVM* and *GCC* use heuristics to decide at compile time whether predication is beneficial or not. In Subsection 3.3 we will treat this in more detail reporting and analyzing some of these heuristics.

Finding whether predication is applicable is per se a challenge. The logic that *LLVM* uses to find basic blocks where predication is applicable is reported in Subsection 3.2.

Finally, in Subsection 3.4, a series of micro benchmarks will be proposed to showcase in which cases and what impact predication has.

3.1 How is predication achieved in the architecture

Various ways to support predication at ISA level have been developed.

In the case of `ARMv7` , predication is achieved through condition flags in the *Program Status Register* (PSR) and the instruction's condition field, This is a special register used to set various bits which describe the execution state, among these bits, there are the Zero (Z), Negative (N), Carry (C), and Overflow (V) bits which are set whenever an arithmetic instruction is executed. When the CPU encounters a predicated instruction, it evaluates the condition field against the current PSR flags. If the condition is met, the instruction executes; otherwise, it is treated as a no-op.

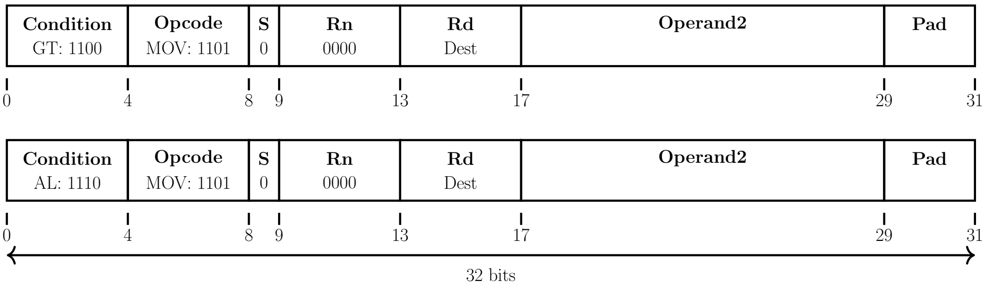


Figure 5: Bit-level encoding of `ARMv7`'s `MOVGT` (on top) and of `MOV` (on the bottom) instructions, illustrating fields for condition codes, opcode, status bit, register operands, and operand values.

In Figure 5 we can observe how `MOV` and `MOVG` bit encoding only differ for the first 4 bits. The condition field in `ARMv7` applies to almost all instructions. In ARM's architecture, most instructions have a 4-bit condition field. This makes non predicated instructions a particular case of predication where the condition field is set to `1110`.

Another architecture design that offer predication support is the Itanium `IA-64`. The `IA-64` has 64 predicate registers (`p0` to `63`) that allow each instruction to be conditionally executed based on specific predicate values. Each instruction can specify a predicate register, offering highly granular control over conditional execution. Thanks to this approach, entire sections of code can be totally predicated, minimizing the dependency on branch predictors.

Listing 11: Predicated MOV in Itanium

```
cmp.gt p1, p0 = r0, #0
(p1) mov r1 = 10
(p0) mov r1 = 20
```

In Listing 11, it's possible to see the IA-64 equivalent of the assembly code showed in Listing 4.

3.2 Detecting Predicable Regions

Some operations are difficult to predicate. Interruption, Exceptions and System Calls have side effects that expand beyond the local execution and might trigger events that are impossible to revert. Certain Control Flow Instructions like `jmp` and `call` alter the program's execution path, predicating this requires hardware to simulate both the taken and non-taken paths simultaneously. Additionally, predicating large blocks results in executing many unnecessary instructions when the predicate is false. The performance penalty of executing unnecessary instructions outweighs the benefit of avoiding a branch. A set of basic block is therefore considered not predicable if it exceeds certain sizes.

The discovery of predicable regions is the beginning of the `if-conversion` step. This optimization pass starts by iterating through all the basic blocks in a function's control flow graph (CFG).

```
/// Analyze the structure of the sub-CFG starting from the specified block.
/// Record its successors and whether it looks like an if-conversion candidate
void IfConverter::AnalyzeBlock(
    MachineBasicBlock &MBB, std::vector<std::unique_ptr<IfcvtToken>> &Tokens)
        {
  // Initialize stack with the starting block.
  SmallVector<MachineBasicBlock *, 16> BBStack = {&MBB};

  while (!BBStack.empty()) {
    MachineBasicBlock *BB = BBStack.pop_back_val();
    BBInfo &BBI = BBAnalysis[BB->getNumber()];

    // Analyze branches and instructions in the block.
    AnalyzeBranches(BBI);
    ScanInstructions(BBI, BB->begin(), BB->end());

    // Skip blocks unsuitable for if-conversion.
    if (!BBI.IsBrAnalyzable || BBI.BrCond.empty()) {
      BBI.IsAnalyzed = true;
      continue;
    }

    // Push successors for further analysis.
    if (BBI.TrueBB && BBI.FalseBB) {
      BBStack.push_back(BBI.TrueBB);
      BBStack.push_back(BBI.FalseBB);
    }

    // Check for if-conversion patterns (e.g., diamonds, triangles).
    if (ValidDiamond(BBI)) {
      Tokens.push_back(std::make_unique<IfcvtToken>(BBI, ICDiamond));
    } else if (ValidTriangle(BBI)) {
      Tokens.push_back(std::make_unique<IfcvtToken>(BBI, ICTriangle));
    } else if (ValidSimple(BBI)) {
      Tokens.push_back(std::make_unique<IfcvtToken>(BBI, ICSimple));
    }

    BBI.IsAnalyzed = true;
  }
}
```

Figure 6: Simplified version of *LLVM*'s `AnalyzeBlock` implementation for `ARM` architectures.

For each block, the function `AnalyzeBlock` is called. A simplified version of this function is shown in Listing 6.

A *MachineBasicBlock* is a basic block after being translated to machine instructions. This function analyzes the structure of a `sub-CFG` starting from a given `MachineBasicBlock`. It evaluates branches and records successors to determine if the block is suitable for `if-conversion`. The function `AnalyzeBranches`, determines if its branches can be analyzed or reversed, and checks if it has fall-through behavior. The data collected when this function runs are then stored in the `MachineBasicBlock` struct and used later. The function `ScanInstruction` scans all the instructions in the block to determine if the block is predicable. In most cases, a block is predicable if all the instructions in the block are predicable. If the sub-CFG is not predicable, no further analysis are performed. If it is, certain pattern are searched in the sub-CFG graph.

```
void triangleBranch(int condition) {
    // EBB: Entry Basic Block
    x_ ++;
    if (x == 0) {
        // TBB: True Block
        x_ = 42;
    }
    // FBB: Fall-through Block
    y_ = 100; // Common operation
}
```

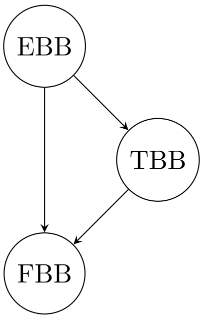


Figure 7: The C++ code on the left, has a EBB, a TBB and a fall through FBB. When compiled it gives origin to the triangular branch on the right.

The simplest pattern identified in the CFG is a *Triangular Branch*, an example is shown in Figure 7. This control flow pattern begins with an *Entry Basic Block* (EBB) which branches into a *True Basic Block* (TBB). Both the EBB and TBB subsequently converge into the *False Basic Block* (FBB).

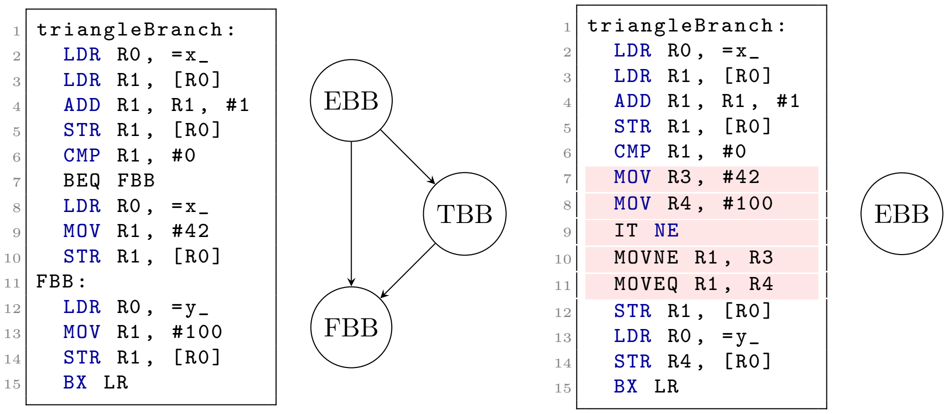
When a *Triangular Branch* is identified when using *ARMv7* as the target architecture, given the analysis already performed and the hardware support, we know that it can be predicated. The last step is therefore to verify if the number of instructions to be predicated is within the limit. This is performed in the function `MeetIfcvtSizeLimit`. The logic used in this function to calculate the number of instruction is reported in Listing 8.

```
unsigned NumPredicatedInstructions = 0;

// Count instructions to be predicated in the true block
for (auto &I : make_range(TIB, TIE)) {  // TIB and TIE mark the range
    NumPredicatedInstructions++;
}

// Count instructions to be predicated in the false block
for (auto &I : make_range(FIB, FIE)) {  // FIB and FIE mark the range
    NumPredicatedInstructions++;
}

if (NumPredicatedInstructions > 15)
  // Do not predicate
```

Figure 8: Logic used in *LLVM* to decide if a block is too big to be predicated.

The number of instructions to be predicated is determined by counting the number of instruction in the TIB and in the FIB. Each instruction in these ranges is counted, incrementing the NumPredicatedInstructions variable. If the total number of predicated instructions exceeds a predefined threshold of 15, the function decides not to proceed with predication, ensuring that excessively large predicated blocks are avoided to maintain efficiency.

In the case of the *Triangular Branch*, the FIB is common to both branches, therefore the size only depends on the number of instructions in the TIB. In Figure 9, the before and after in machine code is shown for the code provided in Listing 7



```
triangleBranch:
  LDR R0, =x_
  LDR R1, [R0]
  ADD R1, R1, #1
  STR R1, [R0]
  CMP R1, #0
  BEQ FBB
  LDR R0, =x_
  MOV R1, #42
  STR R1, [R0]
FBB:
  LDR R0, =y_
  MOV R1, #100
  STR R1, [R0]
  BX LR
```

```
triangleBranch:
  LDR R0, =x_
  LDR R1, [R0]
  ADD R1, R1, #1
  STR R1, [R0]
  CMP R1, #0
  MOV R3, #42
  MOV R4, #100
  IT NE
  MOVNE R1, R3
  MOVEQ R1, R4
  STR R1, [R0]
  LDR R0, =y_
  STR R4, [R0]
  BX LR
```

Figure 9: Comparison of assembly code and control flow graphs (CFGs) for triangleBranch: original code with branching versus optimized code with predication, demonstrating CFG simplification.

Three other patterns exists: *Diamond*, *Forked Diamond* and *Simple*. Each of these patterns requires a different size and feasibility analysis to determine if they can be *if-converted*.

3.3 Compiler's Heuristics

As demonstrated in Section 3 and highlighted by August et al. [1], indiscriminate use of predication leads to performance degradation. To determine whether to apply this technique, modern compilers rely on heuristics. The heuristics are highly architecture-dependent, as illustrated in Table 1. In the rest of this section, the heuristics used by the production state of the art compiler *LLVM* is described.

Section 3.2 already introduced that, when analyzing a block for *if-conversion* the possible cost is computed. The cost of predicating a Basic Block consists of two components: architecture-dependent predication cost and instruction latency. These costs vary based on the specific instruction and the architecture. Some architectures impose a fixed cost for predicating any instruction, while others either avoid this cost entirely or apply it selectively to certain instructions. Importantly, the cost of predication associated with each instruction is not arbitrarily defined by the compiler but instead reflects the actual cost of executing the instruction on the target architecture. Taking as reference the ARMv7 architecture, calls have an additional latency of 1 cycle to be predicated, while predicating any other instruction does not introduce any other cost. It's important to note that compared to other architectures, predication in ARMv7 is a central part of the architecture and therefore is cheap.

```
BBI.NonPredSize++;
unsigned ExtraPredCost = TII->getPredicationCost(MI);
unsigned NumCycles = SchedModel.computeInstrLatency(&MI, false);
if (NumCycles > 1)
    BBI.ExtraCost += NumCycles-1;
BBI.ExtraCost2 += ExtraPredCost;
```

Figure 10: Portion of code where the cost of predicating a Basic Block is calculated in *LLVM*.

The impact of predication can be both positive and negative on the number of cycles needed to execute a function. When the cost is negative, we always choose to apply the *if-conversion* transformation. On the other hand, having a positive cost does not automatically rule out predication. Instead, we apply a further heuristic that takes into account the cost of misprediction. The logic to perform the cost analysis of predication is in the function `ScanInstruction` which is partially reported in Listing 10

```
bool ARMBaseInstrInfo::
isProfitableToIfCvt(MachineBasicBlock &TBB,
                    unsigned TCycles, unsigned TExtra,
                    MachineBasicBlock &FBB,
                    unsigned FCycles, unsigned FExtra,
                    BranchProbability Probability) const {
  if (!TCycles)
    return false;

  // Attempt to estimate the relative costs of predication versus branching.
  // Here we scale up each component of UnpredCost to avoid precision issue when
  // scaling TCycles/FCycles by Probability.
  const unsigned ScalingUpFactor = 1024;

  unsigned PredCost = (TCycles + FCycles + TExtra + FExtra) * ScalingUpFactor;
  unsigned UnpredCost;
  if (!Subtarget.hasBranchPredictor()) {
    // This code block contains the case where a pranch predictor
    // is not available. For the sake of brevity is not reported
  } else {
    unsigned TUnpredCost = Probability.scale(TCycles * ScalingUpFactor);
    unsigned FUnpredCost =
      Probability.getCompl().scale(FCycles * ScalingUpFactor);
    UnpredCost = TUnpredCost + FUnpredCost;
    UnpredCost += 1 * ScalingUpFactor; // The branch itself
    UnpredCost += Subtarget.getMispredictionPenalty() * ScalingUpFactor / 10;
  }

  return PredCost <= UnpredCost;
}
```

Figure 11: *LLVM*'s `isProfitableToIfCvt` implementation for `ARM` architectures.

The cost used in `isProfitableToIfCvt` are:

- **TCycles**: The execution time (in cycles) of the ”True” block.
- **TExtra**: Additional cost (in cycles) for predicating the ”True” block.
- **FCycles**: The execution cost (in cycles) of the ”False” block.
- **FExtra**: Additional cost (in cycles) for predicating the ”False” block.

The actual cost of predication is calculated on line 15 of Image 11:

$$\text{PredCost} = (\text{TCycles} + \text{FCycles} + \text{TExtra} + \text{FExtra})$$

The cost of omitting the predication is more complex as it needs to take into account the *Costo of Misprediction* and the probability of each side of the branch has to be weighted based on how often the side is actually executed. This weighting is necessary because not all branches are equally likely to execute. The execution cost of the "True" or "False" path should reflect their respective probabilities, as this determines the expected cost of the branch in practice. Without weighting, the cost estimation would not accurately represent the real-world behavior of the code under typical execution conditions. For instance, if one of the two side of the branch is very expensive to execute but it does not get executed often, the overall cost of the branch might still be low. The Cost of Misprediction, on the other hand, is a static cost that is dependant on the target architecture. Some Cost of Misprediction are reported in Table 1.

Section 3.4 shows conrete examples of how these heuristic condition the output of the compiler.

3.4 Predication Benchmarks

Computing the cycle cost of basic block is not trivial. It depends on several variables and these variables are not always in plain sight: some of these variables depend on the target architecture while other are calculated using further heuristics that the compiler leverage to compute the probability of a given branch. As these internals are obscure, convoluted and buried in repositories that have millions of lines of code, it's impossible to reliably predict the compiler's output. The only option we are left with is to look at the compiler's output to try to understand the choice that have been taken.

As we have seen, the choice that the compiler takes, often results in the most efficient version of the code possible. However, this is not always the case. The heuristics used by the compiler and described in Section 3.3, are designed to be general and provide good results in most scenarios. Additionally, the compiler's knowledge at compile time is inherently limited. For these reasons, humans are often still able to produce better assembly code than compilers.

4 Predication and Speculation

References

[1] David I. August et al. “Integrated predicated and speculative execution in the IMPACT EPIC architecture”. In: *SIGARCH Comput. Archit. News* 26.3 (Apr. 1998), pp. 227–237. ISSN: 0163-5964. DOI: 10.1145/279361.279391. URL: https://doi.org/10.1145/279361.279391.

[2] Gregory J. Chaitin et al. “Register Allocation via Graph Coloring”. In: *Comput. Lang.* 6.1 (1981), pp. 47–57.

[3] P.P. Chang et al. “Three architectural models for compiler-controlled speculative execution”. In: *IEEE Transactions on Computers* 44.4 (1995), pp. 481–494. DOI: 10.1109/12.376164.

[4] Pohua P. Chang and Wen-mei W. Hwu. “Trace Selection For Compiling Large C Application Programs To Microcode”. In: *[1988] Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture - MICRO ’21.* 1988, pp. 21–29. DOI: 10.1109/MICRO.1988.639244.

[5] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO).* IEEE, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[6] Chit-Kwan Lin and Stephen J. Tarsa. “Branch Prediction Is Not a Solved Problem: Measurements, Opportunities, and Future Directions”. In: *arXiv preprint arXiv:1906.08170* (2019).

[7] Scott A. Mahlke et al. “Sentinel scheduling: a model for compiler-controlled speculative execution”. In: *ACM Trans. Comput. Syst.* 11.4 (Nov. 1993), pp. 376–408. ISSN: 0734-2071. DOI: 10.1145/161541.159765. URL: https://doi.org/10.1145/161541.159765.