Predication and Speculation

Compilers for High Performance Computers

Stefano Petrilli
stefano.petrilli@upc.edu
Jakob Eberhardt
jakob.eberhardt@estudiantat.upc.edu

November 30, 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 units which allows them to fetch, decode, and execute multiple instructions in parallel through separate pipeline stages within a single clock cycle. This increases the instruction throughput and allows for an *Instructions Per Cycle* (IPC) count greater than one. The latest main vendor's architectures, AMD Zen5 and Intel Alder Lake, feature cores that are respectively eight and six 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 designs 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 nineteen 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 elevating ILP are control flows which are hard to predict, 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 committed. 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, the speculatively executed instructions have to be discarded and the correct branch target needs 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 instructions might have been retired. In addition to this, as the pipelines are deep, several cycles have to pass before the pipeline is filled again. Based on the work of Kwan Lin et al.[10], branch misprediction accounts for 20% of the IPC in modern processors and represents 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*[9] reported in Table 1.

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 weights used in *LLVM* for both branch mispredictions and optimistic load costs, we can conclude that a branch misprediction, in modern high-performance architecture, is three times more expensive than a load operation that hits in 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` has 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. The `Cortex M4` is also an embedded processor, but has a three-stage pipeline and executes 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 subsection 2.3 and in subsection 2.4.

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**: 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 prediction, which is part of the microarchitectural level.

- **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 will discuss how Predication and Speculation may help deliver higher performances respectively in Section 2 and in Section 3.

2 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 [10]. 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 1: Replacing a conditional branch with arithmetic operations.

An example of this is presented in Figure 1 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 is 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 that attaches conditions directly to instructions, allowing them to either execute or skip based on the condition's outcome.

```
MOV R0, R1
```

Listing 1: Standard MOV in ARMv7 assembly

```
MOVGT R0, R1
```

Listing 2: Predicated MOV in ARMv7 assembly

The `ARMv7` architecture supports several predicated instructions. For example, the standard `MOV` instructions in Listing 1 takes the value in `R1` and moves it into `R0` while, the predicated version, `MOVGT`, showed in Listing 2 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 2: Branching versus predication in `ARM` assembly

In Figure 2 two assembly translations of the code in Figure 1 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 2, the benefits of predication are staggering. The code size is reduced from five instructions to three 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 forms 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 the software needs to have predictable behavior and respond within strict timing constraints. This is necessary for instance in flight control systems, medical devices, and other mission-critical software. Section 2.1 focuses on a more in-depth analysis of how `ARMv7` and `IA64` architectures introduce support for predication. It is 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 [3]. 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 necessarily 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 happens. Production compilers like *LLVM* and *GCC* use heuristics to decide at compile time whether predication is beneficial or not. In Subsection 2.3 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 2.2 while Subsection 2.5 talks about compilers' limitation and shows an example where *LLVM* fails to identify *if-conversion.* Lastly, in Subsection 2.4, a series of micro-benchmarks will be proposed to showcase in which cases and what impact *if-conversion* has.

2.1 How is Predication Achieved in the Architecture

Various ways to support predication at the 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 that 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.

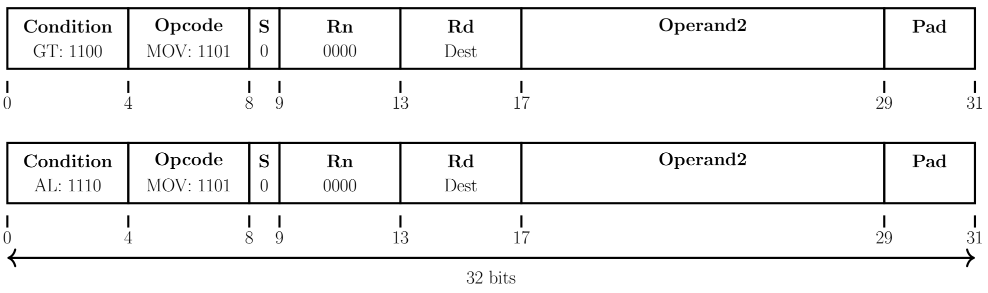


Figure 3: Bit-level encoding of `ARMv7`'s `MOVGT` (on top) and of `MOV` (on the bottom) instructions, illustrating fields for condition codes, opcode, status bit, register operands, and operand values.

In Figure 3 we can observe how `MOV` and `MOVG` bit encoding only differ for the first four 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 offers 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. Using this, approach, entire sections of code can be totally predicated, minimizing the dependency on branch predictors.

Listing 3: Predicated MOV in Itanium

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

In Listing 3, it is possible to see the `IA-64` equivalent of the assembly code showed in Listing 2.

It is important to mention that no modern *ISA* has a predication support that is as central as `ARMv7`. If we look at `x86`, `RISC-V` and even the more recent version of `ARM` such as `ARMv8` and `ARMv9`, none of them decided to give predication such a central role. This may prove that there are better architectural choices that grant higher performances than such extensive predication support. In the specific case of `ARM`, the direction that has been taken is towards a more lean architecture with less complex instructions. This allows for a design that is smaller in terms of area and allows using the encoding space that was before used by the predication to double the number of available registers [6]. Nevertheless, all the modern *ISA* surveyed still present support for a small set of predicated instructions.

2.2 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).

```cpp
/// 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()];

    // Skip blocks already analyzed.
    if (BBI.IsAnalyzed)
      continue;

    // 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 4: 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 4.

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 patterns are searched in the sub-CFG graph.

```cpp
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
}
```

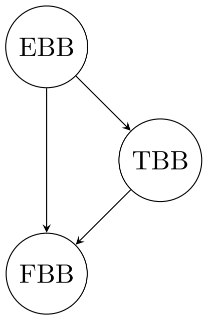


Figure 5: The C++ code on the left, has a EBB, a TBB and a fall through FBB. When compiled it gives origin to the triangular branch on the right.

The most simple pattern identified in the CFG is a *Triangular Branch*, an example is shown in Figure 5. 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 6.

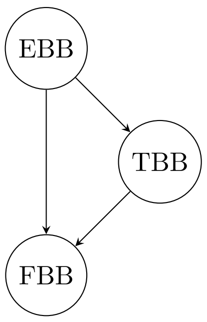
```cpp
unsigned NumPredicatedInstructions = 0;

// 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 6: 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 instructions 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 7, the before and after in machine code is shown for the code provided in Figure 5

```
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
```

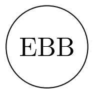


Figure 7: Comparison of assembly code and control flow graphs (CFGs) for `triangleBranch`: original code with branching versus optimized code with predication, demonstrating CFG simplification.

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

2.3 Compiler's Heuristics

As demonstrated in Section 2 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 2.2 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 additional cost. It's important to note that compared to other architectures, predication in `ARMv7` is a central part of the architecture and therefore using it does not present relevant overheads.

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

Figure 8: 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 8

```
 1 bool ARMBaseInstrInfo::
 2 isProfitableToIfCvt(MachineBasicBlock &TBB,
 3                     unsigned TCycles, unsigned TExtra,
 4                     MachineBasicBlock &FBB,
 5                     unsigned FCycles, unsigned FExtra,
 6                     BranchProbability Probability) const {
 7   if (!TCycles)
 8     return false;
 9 
10   // Attempt to estimate the relative costs of predication versus branching.
11   // Here we scale up each component of UnpredCost to avoid precision issue
       when
12   // scaling TCycles/FCycles by Probability.
13   const unsigned ScalingUpFactor = 1024;
14 
15   unsigned PredCost = (TCycles + FCycles + TExtra + FExtra) * ScalingUpFactor;
16   unsigned UnpredCost;
17   if (!Subtarget.hasBranchPredictor()) {
18     // This code block contains the case where a pranch predictor
19     // is not available. For the sake of brevity is not reported
20   } else {
21     unsigned TUnpredCost = Probability.scale(TCycles * ScalingUpFactor);
22     unsigned FUnpredCost =
23       Probability.getCompl().scale(FCycles * ScalingUpFactor);
24     UnpredCost = TUnpredCost + FUnpredCost;
25     UnpredCost += 1 * ScalingUpFactor; // The branch itself
26     UnpredCost += Subtarget.getMispredictionPenalty() * ScalingUpFactor / 10;
27   }
28 
29   return PredCost <= UnpredCost;
30 }
```

Figure 9: *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 Listing 9:

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

The cost of omitting the predication is more complex as it needs to take into account the *Cost 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 sides of the branch is very expensive to execute but it does not get often executed, the overall cost of the branch might still be low. The Cost of Misprediction, on the other hand, is a static cost that is dependent on the target architecture. Some Cost of Misprediction are reported in Table 1.

Section 2.4 shows concrete examples of how these heuristics condition the output of the compiler.

2.4 Predication Benchmarks

Computing the cycle cost of a 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 others are calculated using further heuristics that the compiler leverages 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 choices that have been taken.

In this section, we will also benchmark the examples we are going to provide. Performing micro benchmarks on functions that are so small, it's not trivial. The changes we want to measure between the two versions are only a few assembly instructions, making it very sensitive to external perturbation. These factors include but are not limited to environmental influences, such as OS scheduling, background processes, and thermal effects, as well as microarchitectural noise like CPU caching, branch prediction, and speculative execution. Finally, for changes that are so small, the measure resolution that we can use is likely going to be too coarse. Even if the other limitations were to not exist, no `ARMv7` hardware was at our disposal. For all the previous reasons, a static analyzer tool appeared as the most appropriate tool for our purpose. The tool used for the benchmark is `llvm-mca` [8]. "`llvm-mca` is a performance analysis tool that uses information available in *LLVM* (e.g. scheduling models) to statically measure the performance of machine code in a specific CPU. Given an assembly code sequence, `llvm-mca` estimates the Instructions Per Cycle (IPC), as well as hardware resource pressure" [14].

```cpp
#include <cstdint>

uint32_t
computeLoan(bool isHouseLoan,
                int principal) {
  //EBB
  uint32_t
  baseRate = principal * 2 / 100;

  if (isHouseLoan) {
    // TBB
    baseRate = principal * 5 / 100;
  } else {
    // FBB
    baseRate = principal * 7 / 100;
  }

  // TailBB
  return (baseRate + 5) / 10 * 10;
}
```

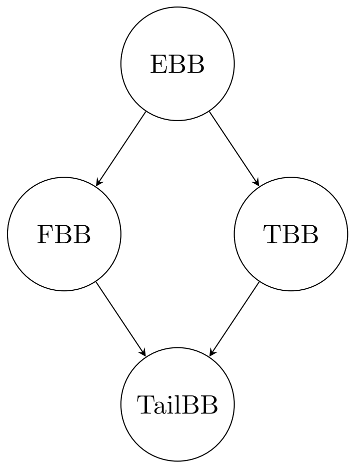


Figure 10: Code for `computeLoan` and the corresponding *Diamond* Control Flow Graph.

Considering the limitation of static analysis [15][17] and the nature of predication, the examples presented in this chapter are all highly computation-bounded rather than memory-bounded. As a consequence, the metrics that are going to be used to measure the performance of the various codes are *IPC* and *Total Execution Cycles* when executing the functions for 100 iterations. It's worth mentioning that these are also the cases where predication tends to perform the best.

The first example presented in this section is the function `computeLoan`. This function and the corresponding Control Flow Graph are visible in Figure 10. This particular CFG shape is referred to as *Diamond*.

When compiled with `-O0`, the resulting assembly code is visible in Figure 11, side by side with the corresponding *if-converted* code.

```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    tst     r0, #1
    beq     .LBB0_2
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_3
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    b       .LBB0_3
.LBB0_2:
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_2
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
.LBB0_3:
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}

.LCPI0_0:
    .long   100
.LCPI0_2:
    .long   7
.LCPI0_3:
    .long   5
.LCPI0_4:
    .long   10
```

```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    ldr     r3, .LCPI0_3
    ldr     r4, .LCPI0_2
    cmp     r0, #1
    mov     r1, r3
    movne   r1, r4
    ldr     r0, [sp, #8]
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}

.LCPI0_0:
    .long   100
.LCPI0_2:
    .long   7
.LCPI0_3:
    .long   5
.LCPI0_4:
    .long   10
```

Figure 11: The `ARMv7` assembly code of *computeLoan* when compiled using `-O0` code (on the left) and the `ARMv7` assembly code of *computeLoan* when compiling using *if-conversion* (on the right).

As in the ARMv7 architecture predication plays a central role, predicated instructions are used on both sides of the assembly translation shown in Figure 11. On the left side, tst, beq and b implement the branch at the end of the *EBB* block. The instruction tst checks the least significant bit of r0 (which contains the boolean value). The result of this test determines whether the branch will be taken or not by the instructions beq.

In the *if-converted* code, the branches are totally removed, and the code becomes branchless. This is, as expected, done using predicated instructions. In particular, the region where this is done is highlighted on the right part of Figure 11. The instruction cmp is used to perform a comparison and set the special registers. The instruction mov will be executed in any case, but, movne will only be executed if the result of cmp is not equal. When this is the case, the changes made by mov will be overwritten.

The second example is the function processOrder which is displayed with the corresponding CFG in Figure 12. This CFG shape takes the name of *Forked Diamond*.

```
#include <cstdint>

enum class CustomerType {
    REGULAR, PREMIUM
};

uint32_t
processOrder(CustomerType customerType,
             uint32_t orderAmount,
             uint32_t loyaltyPoints,
             uint32_t specialEvent)
{
  if (customerType ==
        CustomerType::REGULAR) {
    if (orderAmount > 100) {
      return orderAmount - 10;
    } else {
      return orderAmount;
    }
  else
    if (loyaltyPoints > 500) {
      return orderAmount - 10;
    } else {
      ++loyaltyPoints;
      return orderAmount;
    }
  }
}
```

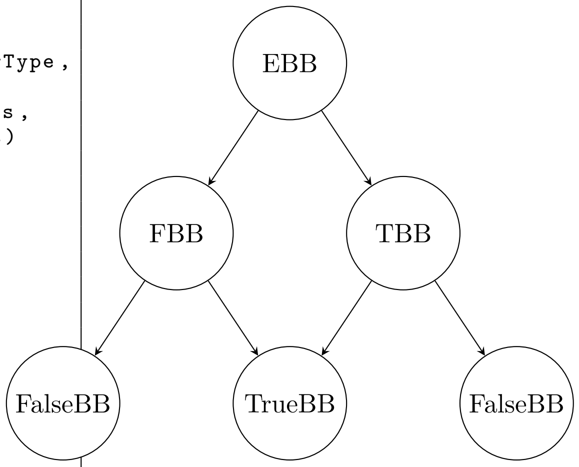


Figure 12: The C++ code for the function processOrder. The corresponding *Forked Diamond* Control Flow Graph.

```
    processOrder
        sub     sp, sp, #20
        str     r0, [sp, #12]
        str     r1, [sp, #8]
        str     r2, [sp, #4]
        str     r3, [sp]
        ldr     r0, [sp, #12]
        cmp     r0, #0
        bne     .LBB0_4
        ldr     r0, [sp, #8]
        cmp     r0, #100
        bls     .LBB0_3
        ldr     r0, [sp, #8]
        sub     r0, r0, #10
        str     r0, [sp, #16]
        b       .LBB0_7
    .LBB0_3:
        ldr     r0, [sp, #8]
        str     r0, [sp, #16]
        b       .LBB0_7
    .LBB0_4:
        ldr     r0, [sp, #4]
        cmp     r0, #500
        bls     .LBB0_6
        ldr     r0, [sp, #8]
        sub     r0, r0, #10
        str     r0, [sp, #16]
        b       .LBB0_7
    .LBB0_6:
        ldr     r0, [sp, #4]
        add     r0, r0, #1
        str     r0, [sp, #4]
        ldr     r0, [sp, #8]
        str     r0, [sp, #16]
    .LBB0_7:
        ldr     r0, [sp, #16]
        add     sp, sp, #20
        bx      lr
```

```
processOrder
    sub     sp, sp, #20
    str     r0, [sp, #12]
    str     r1, [sp, #8]
    str     r2, [sp, #4]
    str     r3, [sp]
    ldr     r0, [sp, #12]
    cmp     r0, #0
    ldrne   r0, [sp, #4]
    addle   r0, r0, #1
    strle   r0, [sp, #4]
    ldr     r0, [sp, #8]
    cmp     r0, #100
    subhi   r0, r0, #10
    str     r0, [sp, #16]
    ldr     r0, [sp, #12]
    cmp     r0, #0
    ldrne   r0, [sp, #4]
    cmpne   r0, #500
    subgt   r0, r0, #10
    strgt   r0, [sp, #16]
    ldr     r0, [sp, #16]
    add     sp, sp, #20
    bx      lr
```

Figure 13: The ARMv7 assembly code of *processOrder* when compiled using -O0 code (on the left) and the ARMv7 assembly code of *processOrder* when compiling using *if-conversion* (on the right).

In Figure 13 it's possible to see the basic and the *if-converted* versions of processOrder. The *Forked Diamond CFG* enables more substantial code reduction if compared to the other *if-conversion* seen so far. In this case, the number of predicated instructions is also much higher. For every three instructions in the *if-converted* version, one is a predicated instruction.

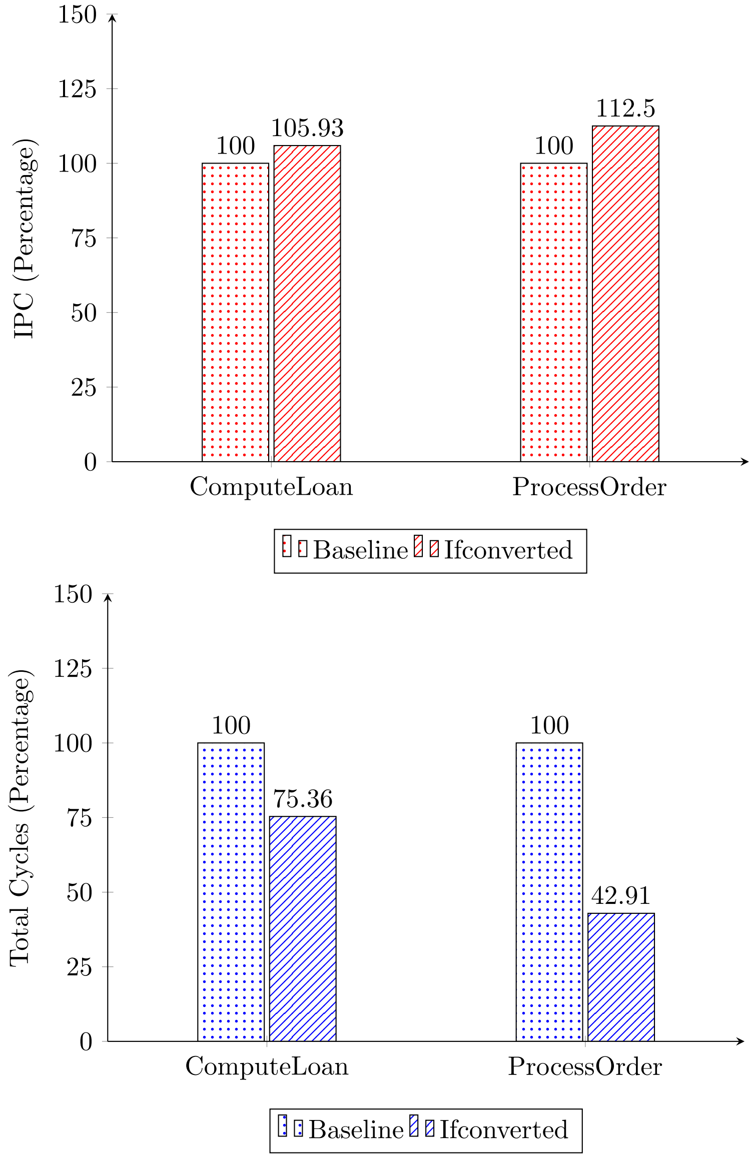


Figure 14: Percentage variations of *IPC* and *Totaly Cycles* for the functions `computeLoan` (*Diamon*) and `processOrder` (*Forked Diamond.*)

The *IPC* and *total cycle* comparison of the two examples provided are showcased in Figure 14. From this, we can conclude that not only do the *if-converted* versions have a higher *IPC*, they are also shorter and therefore use a much lower number of *Total Cycles*.

The compiler heuristics mentioned in Section 2.3 and the benchmarks reported in this section point out that whenever we face a *CFG* with the shape of *Diamond*, *Forked Diamond* and *Triangle*, applying *if-conversion* results in improved performances.

2.5 Compilers Limitations

In the examples reported in Section 2.4, *LLVM* always produced a great improvement when using *if-conversion*. But, this is not always the case. Compilers are far from perfect, and it's not difficult to identify examples where a human ends up creating better code than a compiler. The main reason for which performances are left on the table by compilers is because they fail to identify cases where *if-conversion* could be applied.

An example of this can be crafted starting from the `computeLoan` example shown in Section 2.4. As shown in the previous section, this is a perfect case to apply *if-conversion* and this optimization results in a noticeable performance improvement. Still, when the number of instructions and the complexity grows, compilers stop applying predication as the predicated block exceeds the size limit discussed in Section 2.3. The new version of `computeLoan` is referenced as `expandedComputeLoan` and is reported in Figure 15, the performance measured for the `-O0` version is compared with a version where the *if-conversion* optimization is applied manually. The corresponding assembly code is provided in appendix A. From the Figure, we can see that even though the compiler decided not to apply this optimization, applying still results in a great performance improvement as it is possible to see in Figure 16.

```
#include <cstdint>

uint32_t computeLoanPayment(bool isHouseLoan, uint32_t principal)
{
    uint32_t baseRate = 0;
    uint32_t tax = principal * 2 / 100;

    if (isHouseLoan) {
         baseRate = principal * 5 / 100;
       baseRate += 50;
    } else {
        baseRate = principal * 7 / 100;
      baseRate += 30;
    }

    uint32_t totalPayment = baseRate + tax;
    totalPayment = (totalPayment + 5) / 10 * 10;
    return totalPayment;
}
```

Figure 15: The expanded `computeLoan` function. The added lines are highlighted in red.

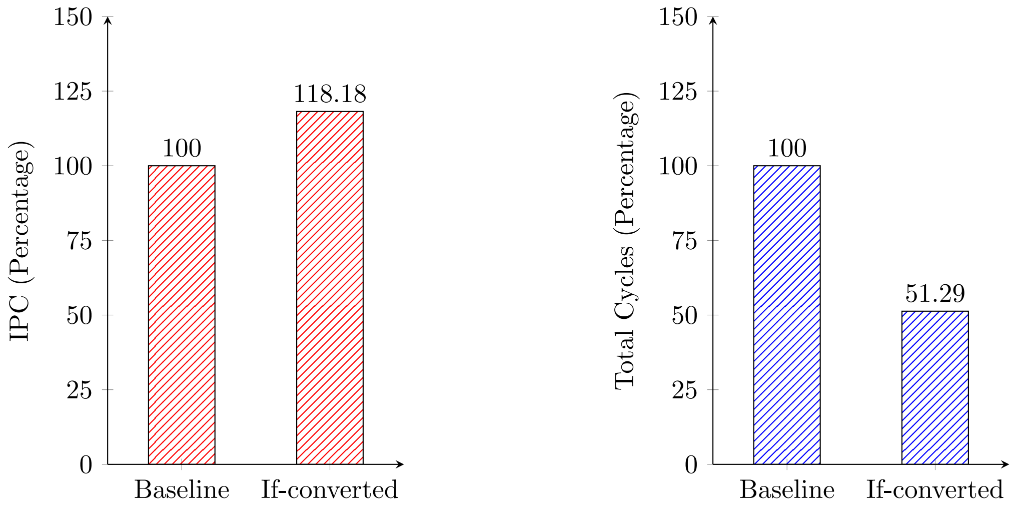


Figure 16: *IPC* and *Total Cycles* reported for `expandedComputeLoan` by *LLVM-MCA*.

The heuristics used by the compilers, and described in Section 2.3, are designed to be general and provide good results in most scenarios, the compiler's knowledge at compile time is inherently limited, and applying more complex optimization would require more compilation time.

3 Speculation

We consider an instruction to be *speculatively* executed [4] whenever we move it above a branch which determines its execution. When speculating during compilation, we typically try to tailor the control flow of the program toward 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.

3.1 Superblock Structure

If we consider the control flow of a program, a superblock [12] is a consecutive sequence of instructions that is always entered at the top-most basic block but may be left at arbitrary and multiple points. When forming a 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 [5]. An example of how the profile of the program seen in Listing 4 may look can be seen in the weighted control flow graph in Figure 17 [4]. The numbers on the edges correspond to the frequency at which the respective control flow path was taken during execution.

```c
avg = 0;
weight = 0;
count = 0;

while (ptr != NULL) {
    count++;
    if(ptr->wt < 0) {
        weight -= ptr->wt;
    } else {
        weight += ptr->wt;
    }
    ptr = ptr->next;
}

if(count !=0) {
    avg = weight / count;
}
```

Listing 4: C Code for Example CFG.

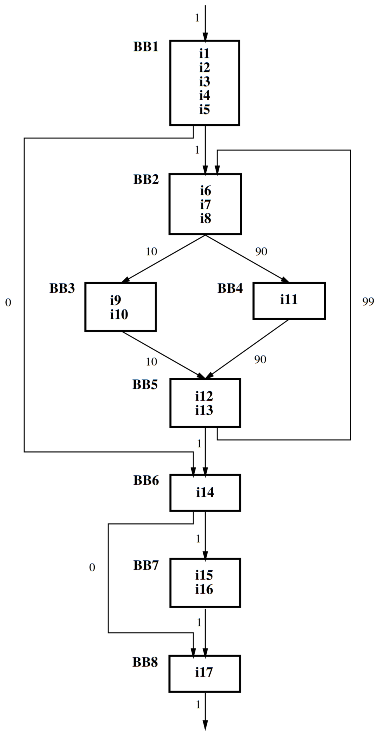


Figure 17: An example control flow graph with profiling data.
Courtesy of Chang et al. [4].

```
    ldr r1, _ptr    ; I1
    mov r7, 0       ; I2
    mov r2, 0       ; I3
    mov r3, 0       ; I4
    beq L3, r1, 0   ; I5
L0:
    add r2, r2, 1   ; I6
    ldr r4, 0[r1]   ; I7
    bge L1, r4, 0   ; I8
    sub r3, r3, r4  ; I9
    br L2           ; I10
L1:
   add r3, r3, r4   ; I11
L2:
   ldr r1, 4[r1]    ; I12
   bne L0, r1, 0    ; I13
L3:
    beq L4, r2, 0   ; I14
    div r7, r3, r2  ; I15
    str _avg r7     ; I16
L4:
```

Listing 5: `ARM` Assembly Code of Example CFG.

Figure 18 shows the loop part of the program which consists of BB2, BB3, BB4, and BB5. In 90% of the cases, we jump from BB2 to BB4 and BB5. Groups of blocks like this which frequently execute together are also called *trace*. Our goal is to create a superblock from this trace to reduce bookkeeping complexity which typically arises during trace scheduling when we move instructions across side entrances. However, the edge from BB3 to BB5 is a side entrance into the trace. To remove it, we essentially tail duplicate BB5 by copying `I12` and `I13` seen in Listing 5 to BB3' which now also forms a superblock by itself. We can omit `I10` since it was the branch instruction into BB5. Superblocks are not exclusively tools of speculation, rather they are also important for enabling other optimizations in compilers.

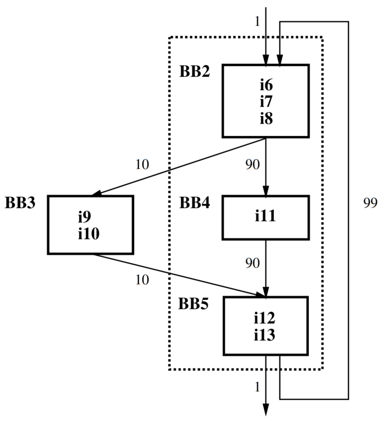


Figure 18: Loop Part with Side Entrance. Courtesy of Chang et al. [4].

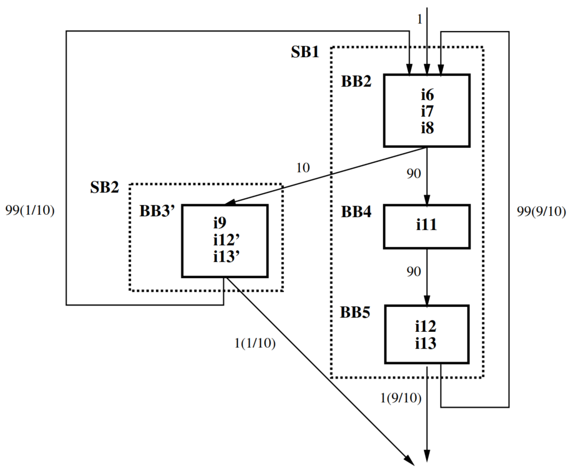


Figure 19: The loop part of the program after Superblock creation. Courtesy of Chang et al. [4].

3.2 Superblock Scheduling

After the trace selection and the formation of superblocks, the compiler has to schedule the instructions to enable as much ILP as possible and the constraint of producing a save and correct program. This is done by building a graph that represents the dependencies among the instructions. The dependency graph is used during and subsequent list scheduling. The compiler may use heuristics and given instruction latencies to assign priorities to instructions which are *ready* to produce a schedule that will use the available resources as efficiently as possible [4]. To enhance this further, the compiler needs to be able to move instructions above previous branches which is considered a speculation and will be discussed in Section 3.3.

3.3 Speculative Code Motion during Superblock Scheduling

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 branch instructions, we can define `live-out(Br)` as the set of variables that may be used before being defined in case `Br` will be taken.

3.3.1 Downward Motion

Downward code motion, meaning moving an 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 ensure correctness, as can be seen in Listing 6

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

Listing 6: 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 7: 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.

3.3.2 Restrictions for Upward Motion

Moving an instruction from a point after a branch to a point before the branch is referred 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 an example of such a long-latency instruction. As can be seen in Listing 8, depending on the input data, we may have to execute `SDIV`. Therefore it would be preferable to speculatively execute the division as can be seen in Listing 9. 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 8: 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 9: 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 an unnecessary a divide-by-zero exception.

In general, we can formulate two restrictions one has to consider when applying speculation:

Restriction 1: The destination register of `I` is not in `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` is taken.

In combination with hardware support of the target, different speculation models typically result in dependence graphs which may allow the compiler to perform certain speculations during superblock scheduling. The code in Listing 10 which corresponds to the program in Listing 4 which is courtesy of Chang et al. will serve as an example in the following sections to showcase increasingly aggressive speculation models. We assume a processing unit with universal functional units with a one-cycle latency for ALU instructions and a two-cycle load delay. The instruction labeled `I1` to `I12` belong to the superblock previously discussed in Section 3.1.

```
    ldr r1, _ptr       ; Load initial pointer, Initialize avg, count and weight
    mov r7, 0
    mov r2, 0
    mov r3, 0
    beq L3, r1, 0
L0:
    add r2, r2, 1      ; I1: Increment count
    ldr r4, 0[r1]      ; I2: Load ptr->wt into r4
    btl L1, r4, 0      ; I3: If wt < 0, branch to L1
    add r3, r3, r4     ; I4: Add wt to weight
    ldr r5, 4[r1]      ; I5: Load ptr->next into r5
    beq L3, r5, 0      ; I6: If next is NULL, jump to L3
    add r2, r2, 1      ; I7: Increment count
    ldr r6, 0[r5]      ; I8: Load next->wt into r6
    btl L1X, r6, 0     ; I9: If wt < 0, branch to L1X
    add r3, r3, r6     ; I10: Add wt to weight
    ldr r1, 4[r5]      ; I11: Move ptr to ptr->next->next
    bne L0, r1, 0      ; I12: Loop back to L0 if ptr != NULL
L3:
    beq L4, r2, 0      ; If count == 0, skip division
    div r7, r3, r2
    str _avg, r7
L4:

; ...

L1X:
    mov r1, r5         ; Adjust ptr to second node (ptr = ptr->next)
    mov r4, r6         ; Move wt to r4 for subtraction
L1:
    sub r3, r3, r4     ; Subtract wt from weight (negative wt)
    ldr r1, 4[r1]      ; Move ptr to ptr->next
    bne L0, r1, 0      ; Loop back to L0 if ptr != NULL
```

Listing 10: The assembly code resulting from the code in Listing 4 after superblock formation. Additionally, the loop iteration was unrolled once to allow more code motion [4]. The instructions within the SB1 are labeled from one to twelve. In case the pointer in the unrolled iteration was not null but the weight is less than zero, L1X acts as a recovery code to prepare register `r1` and `r4` for the subtraction case. Note that the superblock formation can also be considered a speculation since the code for our expected control flow (`ptr->wt > 0`) will be located in series in memory which likely increases locality.

3.3.3 Restricted Code Percolation

Restricted Percolation is a conservative code motion technique where the compiler moves instructions across basic blocks or control-flow boundaries under strict safety conditions. Restricted Code Percolation falls into the *avoid errors* class [2], meaning the compiler will never move an instruction that can cause an exception above a branch. A none-excepting instruction may be moved above a branch if it does not violate the formally defined Restriction 1. Although this model limits the degrees of freedom for code motion, it for example does not add additional pressure on the Dcache as no unneeded memory instructions will be executed. Additional instruction memory page faults may occur due to speculative execution. However, these are handled as they occur and do not require additional hardware. In the following Figure 21 we see a superblock schedule under the restricted model which executes correctly without the need for any additional hardware support. To this end, we have to respect both restrictions which results in the dependence graph seen in Figure 20 where flow (f) and output (o) data dependencies within the superblock are indicated by solid lines. Control dependencies, meaning the case where the execution of a given instruction depends on the branch direction of a previous one are indicated by dashed lines. For example, `I4` is control dependent on instruction `I3` since it may branch to L1. Additionally, `I4` requires the value loaded by `I2` which is a data flow dependency.

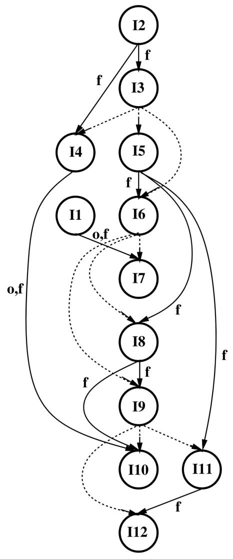
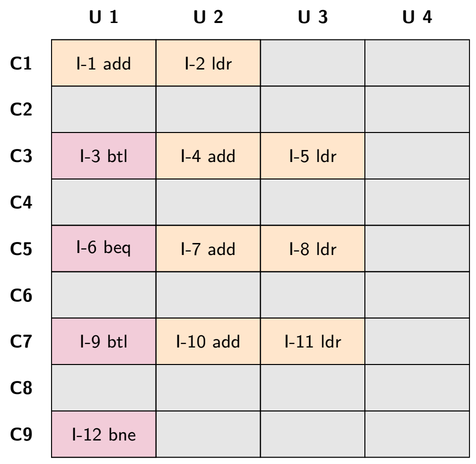


Figure 20: Dependence Graph for Restricted Code Percolation. Courtesy of Chang et al. [4].



	U 1	U 2	U 3	U 4
C1	I-1 add	I-2 ldr		
C2				
C3	I-3 btl	I-4 add	I-5 ldr	
C4				
C5	I-6 beq	I-7 add	I-8 ldr	
C6				
C7	I-9 btl	I-10 add	I-11 ldr	
C8				
C9	I-12 bne			

Figure 21: Under Restricted Code Percolation, we take nice cycles to execute an unrolled loop iteration. Due to the strict restrictions, the schedule tableau is sparse.

Under Restricted Code Percolation, the compiler is not able to schedule long latency instructions such as the load instruction `I5` earlier since it may cause an exception. Additionally, we are limited when moving for example `I7`. Although its data dependencies are satisfied by the previously scheduled `I1`, its destination register `r2` is part of `live-out(I6)` on which `I7` is control dependent on.

3.3.4 Safe Speculation Model

Safe Speculation [2] is an extension to Restricted Code Percolation which relaxes Restriction 2 in certain cases. Safe speculation preserves the *avoid erros* property by employing compile-time program analysis to formally exclude the occurrence of exceptions during run time. Typically, this may include speculative loads in cases where the compiler can prove that the load will never raise an exception, for example by checking the boundaries of an allocated memory array. Other examples are division operations where there is formal proof that we may never attempt to divide by zero. Although it comes with an additional compile-time cost, it does not require additional hardware nor does it introduce the risk of false results which are inherent to for example *ignore errors* speculations.

3.3.5 General Code Percolation

General Code Percolation belongs to the category of *ignore errors* [2] which lifts Restriction 2 by ignoring exceptions. To this end, it requires a non-excepting counterpart for any excepting instruction that we want to execute speculatively. This may include memory instructions, integer divide, and all floating-point arithmetic. These special instructions can be moved above control-dependent branches a long as they respect Restriction 1. However, as we ignore the occurring exception conditions, the program is valid to produce an incorrect under General Code Percolation. These errors can arise from different conditions. If a non-excepting load instruction tries to access an invalid address, the result loaded into the destination register will be garbage. If this value of the load is used, it will likely cause errors or actual exceptions. In the same manner, non-excepting integer division and floating-point instructions can cause unpredictable behavior. The same applies to store operations, yet, compilers are typically more conservative with stores since it requires complex memory disambiguation analysis which oftentimes cannot be proven. The effects of General Code Percolation on the dependence graph can be seen in Figure 22.

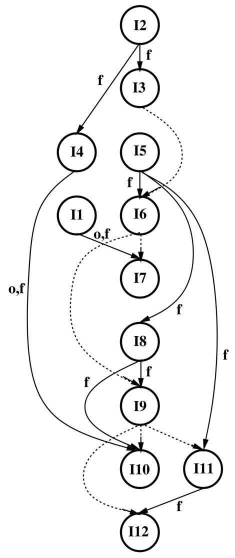


Figure 22: Dependence Graph for General Code Percolation. Courtesy of Chang et al. [4].

	U 1	U 2	U 3	U 4
C1	I-1 add	I-2 ldr	I-5 ldr	
C2				
C3	I-3 btl	I-4 add	I-8 ldr	I-11 ldr
C4	I-6 beq			
C5	I-7 add	I-9 btl	I-10 add	I-12 bne

Figure 23: A valid superblock schedule under General Code Percolation. By lifting restriction 2, we are able to schedule long-latency instructions such as `I5`, `I8`, and `I11` earlier at the cost of input-dependent output errors.

Control dependencies can now be removed if the destination register of an instruction if not in `live-out()` of the above control flow instruction. For example, this applies to the load instruction `I8` which is control dependent on branch instruction `I6`, yet, the destination register `r6` of `I8` is not live in the taken-section of `I6`. Hence, the load can be scheduled before the branch, but it may cause an exception as the pointer in register `r5` may have been null. However, ignoring eventual exceptions allows us to reduce the number of control dependencies to six compared to the nine in the restricted example in Section 3.3.3. A resulting schedule can be seen in Figure 23. An unrolled iteration is now scheduled to take five cycles instead of the previous nine.

3.3.6 Boosting Code Percolation

When employing Boosting Code Percolation, the compiler can speculatively schedule instructions that violate both Restriction 1 and Restriction 2. The model belongs to the so-called *resolve errors* [2] category, meaning the result of the program will ultimately be the same as a non-speculatively scheduled version. To this end, we require hardware support of the target system which allows to delay the commitment of a speculatively executed instruction until the branch it is dependent on commits. This includes a *shadow register file* [16] which will hold boosted instructions until the result of the respective branch is known. For speculative store instructions, a *shadow store buffer* is required. The hardware will keep track of boosted instructions. This includes for example one or multiple bits which indicate how many branches the instructions were moved upwards. The previously described superblock structure is vital to the following parts, as we can assume that boosted instructions are part of the not-taken path of the branch. For example, in case the branch commits while boosted instructions are still in the pipeline, they are somewhat not executed speculatively anymore and we can remove their boosted label, hence converting them to normal instructions. In case the branch was taken, the boosted instructions in the pipeline will be squashed. However, boosted instructions will also finish before their branch commits. In this case, the result of the instruction will be held in a shadow register. In a superblock, we can simply copy the shadow value into the real register or the write buffer in case of stores if the branch was not taken. In the taken case, we destroy the shadow values and have to start executing the branch target code. We overcome Restriction 2 by delaying the handling of exceptions caused by boosted instructions until their branch commits. If the branch is taken, we can ignore all exceptions of boosted instructions, since they were never supposed to be executed anyway. In case the branch was not taken and we encounter one more exception, we have to roll back and squash the pipeline, and start re-executing the not-taken code sequentially until the first exception occurs.

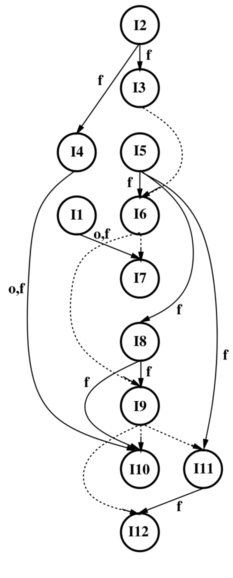
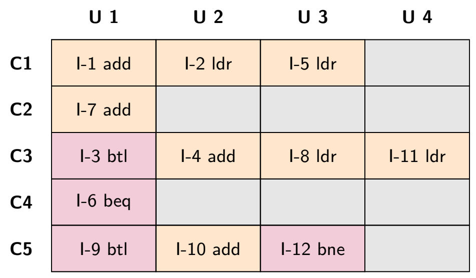


Figure 24: Dependence Graph for Boosting Code Percolation. Courtesy of Chang et al. [4].



	U 1	U 2	U 3	U 4
C1	I-1 add	I-2 ldr	I-5 ldr	
C2	I-7 add			
C3	I-3 btl	I-4 add	I-8 ldr	I-11 ldr
C4	I-6 beq			
C5	I-9 btl	I-10 add	I-12 bne	

Figure 25: Schedule for Boosting Code Percolation

The dependence graph resulting from Boosting Code Percolation can be seen in Figure 24. When assuming hardware support for boosting instructions above one branch, the number of control dependencies in the superblock can be reduced to three. The scheduling freedom provided by the hardware support can be seen in Figure 25. For example, we can now schedule the increment instruction `I7` for the previously idle cycle two without the need for additional recovery code since the hardware will hold the value back until the branch of `I6` is committed and not taken.

3.3.7 Sentinel Scheduling

Sentinel Scheduling [12] is another model belonging to the *resolve errors* [2] category. Sentinel scheduling exploits certain architectural features to detect and resolve exceptions caused by speculatively scheduled instructions. To his end, excepting instructions are split into two parts; the instruction itself which performs the operation, and a sentinel which will handle potentially occurring exceptions. The operation part can be moved above a branch as long as the sentinel remains in the *home block* of the non-speculative control flow graph. Typically, we are able to remove sentinels recursively by exploiting the sentinel of an output-dependent instruction in the same home block, because this sentinel will report the exception of both instructions which makes it a so-called *shared sentinel.* To enable sentinel scheduling, the hardware needs to provide additional support for the compiler to mark speculative instructions using a bit in the opcode. Additionally, the registers need to be extended with an exception tag to indicate any exception that occurred during execution. If a speculative instruction causes an exception, it is not immediately reported. In this case, the exception bit of the destination register is set and the program counter obtained from a history queue is copied into the data part of the register. In the following, if a given instruction uses one or more registers as a source that have their exception tag set, the first exception that has occurred in program order will be handled. For the case where a speculatively and potentially excepting instruction has no consumer in its home block, we need a dedicated sentinel instruction that takes a given register as a source operand and checks the exception tag. Essentially, this can be an arbitrary move instruction that takes the result of the speculative instruction as a source.

3.4 Profiling and Speculation

In order to enable the compiler to enhance the schedule by speculating, we need to provide information about what input the programmer expects during run time and hence the most likely control flow behavior of the program. In this section, we will use the *LLVM* profiling infrastructure [13] to explore how profiling data can affect the program performance both in positive and negative ways, e.g. if the sampling data diverges from the actual input during run time. In our study, we want to compare a conservatively and aggressively speculative scheduled program [7]. The compilation and profiling steps can be seen in Appendix A in Listing 14. In *LLVM*, it is hard to restrict the compiler to one optimization, however, we try to minimize external influence by disabling common optimizations manually. We cannot use `-O0`, because it will also disable the `-enable-misched=true` which we have to use to enable machine-dependent scheduling decisions which likely will include speculations. We can instrument the program seen in Listing 4 to profile it using the `-fprofile-instr-generate`. Afterward, we can produce the profiling data by running the program. The code of Listing 11 is used to generate input data. Most importantly, we can specify the percentage of weight values that are positive, hence how frequently the branch deciding to subtract or add will be taken. For the analysis, we profile the application under the assumption that 99% of the weights are positive.

```c
struct Node* build_list(int64_t N, double percent_positive) {
    struct Node* head = NULL;
    struct Node* current = NULL;
    int64_t i;

    for (i = 0; i < N; i++) {
        struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
        if (new_node == NULL) {
            fprintf(stderr, "Memory allocation failed %" PRId64 "\n", i);
            exit(EXIT_FAILURE);
        }

        if ((rand() / (double)RAND_MAX) < percent_positive) {
            new_node->wt = rand() % 100 + 1;
        } else {
            new_node->wt = -(rand() % 100 + 1);
        }
        new_node->next = NULL;

        if (head == NULL) {
            head = new_node;
            current = new_node;
        } else {
            current->next = new_node;
            current = new_node;
        }
    }
    return head;
}
```

Listing 11: C Code for Input Data Generation. We can specify the percentage of weight which are positive.

In the following, we will compare the program compiled in three different ways while disabling most optimization as seen in Listing 14. Initially, the program is compiled with no additional profiling data. Next, we compile the program again and pass the compiler profiling data which assumes 99% positive weights using the `-fprofile-instr-use` option. For one program, we enable `-enable-misched` which enables target-specific scheduling decisions. Lastly, we compile one version without these additional scheduling optimizations. In the following experiment, we execute each version five times and take the average execution time measured as can be seen in Listing 12.

```c
clock_gettime(CLOCK_MONOTONIC, &start_time);
    while (ptr != NULL) {
        count++;
        if(ptr->wt < 0) {
            weight -= ptr->wt;
        } else {
            weight += ptr->wt;
        }
        struct Node* temp = ptr;
        ptr = ptr->next;
        free(temp);
    }
clock_gettime(CLOCK_MONOTONIC, &end_time);
```

Listing 12: Instrumented C Code for processing the nodes.

The following plot in Figure 26 shows the execution time for processing 6×10^8 nodes of the three versions. While we always profile and hence compile under the assumption that 99% of the weights are positive, we increase the percentage of negative weights on the x-axis.

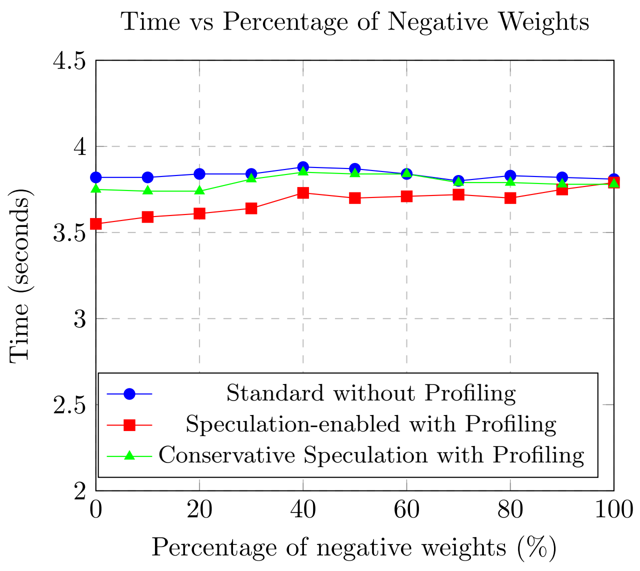


Figure 26: Execution Time with increased input data divergence during run time. The differences are marginal, yet, we can observe how the execution time of the speculative program is rising constantly while the standard program can recover after surpassing the 50% negative weight mark.

We can observe how the differences are marginal. One possible explanation is the execution environment which is a superscalar x86 processor that features branch prediction which will certainly affect the outcome. In fact, in the conservative and standard plots, we can observe a small bump in execution time around the 50% mark which may indicate that this is a configuration that is hard to predict. On the other hand, we can see how the speculation-enabled can not recover the original execution time. The experiment would be more meaningful on a VLIW processor where we are dealing with less noise coming from the hardware.

3.5 Attributes `likely` and `unlikely`

LLVM and the respective `clang` compiler support numerous attributes that programmers can use to give hints to the compiler. In this case, `likely` and `unlikely` [11] can be used to indicate if a branch will likely be taken or not. Similar to providing profiling data, this may enable the compiler to make better decisions during scheduling using the domain knowledge of the programmer. In our case, we may annotate the case where we have to process a negative weight as unlikely, as can be seen in Listing 13. However, in its current state, we could not observe any significant difference in execution time in our benchmarking setup.

```
while (ptr != NULL) {
    count++;
    if (ptr->wt < 0) [[unlikely]] {
        weight -= ptr->wt;
    } else {
        weight += ptr->wt;
    }
    ptr = ptr->next;
}
```

Listing 13: We can annotate the negative-case branch with `[[ unlikely ]]` to support the compiler during scheduling.

4 Conclusion

In this project, we have identified and analyzed the performance impact imposed by control flow limitations in modern processors. We introduced the superblock structure to avoid overly complex bookkeeping when applying speculative optimizations to code scheduling. We described several speculation models of different error handling categories, which each assume different levels of hardware support. This includes Restricted and Safe Percolation, which require no additional hardware but may be limited to sparse schedules. General Code Percolation can lead to higher levels of instruction-level parallelism at the cost of ignoring errors. Boosting Code Percolation and Sentinel Scheduling employ sophisticated hardware structures to enable aggressive speculations while resolving errors as they occur to enable both a tight schedule and correct results.
The theoretical analysis has been integrated with benchmarks that investigated the speculation capabilities of a modern compiler in combination with profiling data.
In Section 2, predication has been analyzed as a second means to overcome control flow limitations. The analysis of predication took into account different aspects such as higher register pressure and additional code size. We unfolded the architectural requirements for predication and undertook a deep analysis of the heuristics that the *LLVM* compiler uses to predicate code regions. Also in the case of predication, the initial theoretical analysis has been completed by production-like benchmarks that provided examples of how prediction may increase performance significantly. We also showed how human programmers can still beat compilers by bringing an example case where compiler's heuristic fail to recognize a region that, when *if-converted* results in better performances.

A Appendix

```
expandedComputeLoanPayment:
  push     {r11, lr}
  mov      r11, sp
  sub      sp, sp, #24
  and      r0, r0, #1
  strb     r0, [r11, #-1]
  str      r1, [r11, #-8]
  mov      r0, #0
  str      r0, [sp, #12]
  ldr      r0, [r11, #-8]
  lsl      r0, r0, #1
  ldr      r1, .LCPI0_0
  bl       __aeabi_idiv
  str      r0, [sp, #8]
  ldrb     r0, [r11, #-1]
  tst      r0, #1
  beq      .LBB0_2
  ldr      r0, [r11, #-8]
  ldr      r1, .LCPI0_3
  mul      r0, r0, r1
  ldr      r1, .LCPI0_0
  bl       __aeabi_idiv
  str      r0, [sp, #12]
  ldr      r0, [sp, #12]
  add      r0, r0, #50
  str      r0, [sp, #12]
  b        .LBB0_3
.LBB0_2:
  ldr      r0, [r11, #-8]
  ldr      r1, .LCPI0_2
  mul      r0, r0, r1
  ldr      r1, .LCPI0_0
  bl       __aeabi_idiv
  str      r0, [sp, #12]
  ldr      r0, [sp, #12]
  add      r0, r0, #30
  str      r0, [sp, #12]
.LBB0_3:
  ldr      r0, [sp, #12]
  ldr      r1, [sp, #8]
  add      r0, r0, r1
  str      r0, [sp, #4]
  ldr      r0, [sp, #4]
  add      r0, r0, #5
  ldr      r1, .LCPI0_4
  bl       __aeabi_idiv
  ldr      r1, .LCPI0_4
  mul      r0, r0, r1
  str      r0, [sp, #4]
  ldr      r0, [sp, #4]
  mov      sp, r11
  pop      {r11, pc}
```

```
expandedComputeLoanPayment:
  push     {r11, lr}
  mov      r11, sp
  sub      sp, sp, #24
  and      r0, r0, #1
  strb     r0, [r11, #-1]
  str      r1, [r11, #-8]
  mov      r0, #0
  str      r0, [sp, #12]
  ldr      r0, [r11, #-8]
  lsl      r0, r0, #1
  ldr      r1, .LCPI0_0
  bl       __aeabi_idiv
  str      r0, [sp, #8]

  ldrb     r0, [r11, #-1]
  cmp      r0, #0

  ldrne    r0, [r11, #-8]
  ldrne    r1, .LCPI0_3
  mulne    r0, r0, r1
  ldrne    r1, .LCPI0_0
  blne     __aeabi_idiv
  addne    r0, r0, #50
  strne    r0, [sp, #12]

  ldreq    r0, [r11, #-8]
  ldreq    r1, .LCPI0_2
  muleq    r0, r0, r1
  ldreq    r1, .LCPI0_0
  bleq     __aeabi_idiv
  addeq    r0, r0, #30
  streq    r0, [sp, #12]

  ldr      r0, [sp, #12]
  ldr      r1, [sp, #8]
  add      r0, r0, r1
  str      r0, [sp, #4]
  ldr      r0, [sp, #4]
  add      r0, r0, #5
  ldr      r1, .LCPI0_4
  bl       __aeabi_idiv
  ldr      r1, .LCPI0_4
  mul      r0, r0, r1
  str      r0, [sp, #4]
  ldr      r0, [sp, #4]
  mov      sp, r11
  pop      {r11, pc}
```

Figure 27: On the left, the `expandedComputeLoan` version as created by the compiler. On the right, the same function with the manually applied *if-conversion*.

```
CC = clang
CFLAGS = -O1 \
    -fno-inline \
    -fno-unroll-loops \
    -fno-vectorize \
    -fno-slp-vectorize \
    -fno-builtin \
    -fno-omit-frame-pointer \
    -fno-merge-all-constants

SRCS = main.c
OBJS = main.o

all: main
main: main.o
    $(CC) $(CFLAGS) -o main main.o
main.o: main.c
    $(CC) $(CFLAGS) -c main.c
main_restricted: main.c
    $(CC) $(CFLAGS) -o main_restricted main.c -mllvm -enable-misched=false -mllvm -avoid-speculation=true
main_boosting: main.c
    $(CC) $(CFLAGS) -o main_boosting main.c -mllvm -enable-misched=true -mllvm -avoid-speculation=false

main.prof: main.c
    $(CC) $(CFLAGS) -fprofile-instr-generate -o main.prof main.c
default.profraw: main.prof
    ./main.prof 1000000 0.99
profile.profdata: default.profraw
    llvm-profdata merge -output=profile.profdata default.profraw
main_restricted_pgo: main.c profile.profdata
    $(CC) $(CFLAGS) -fprofile-instr-use=profile.profdata -o main_restricted_pgo main.c -mllvm -enable-misched=false -mllvm -avoid-speculation=true
main_boosting_pgo: main.c profile.profdata
    $(CC) $(CFLAGS) -fprofile-instr-use=profile.profdata -o main_boosting_pgo main.c -mllvm -enable-misched=true -mllvm -avoid-speculation=false

benchmark_restricted: main_restricted_pgo
    count=5; \
    for i in `seq 1 $$count`; do \
        time=`./main_restricted_pgo 600000000 1.0`; \
        echo "Run $$i: $$time seconds"; \
    done;

benchmark_boosting: main_boosting_pgo
    count=5; \
    for i in `seq 1 $$count`; do \
        time=`./main_boosting_pgo 600000000 1.0`; \
        echo "Run $$i: $$time seconds"; \
    done;

benchmark_unprofiled: main_restricted
    count=5; \
    for i in `seq 1 $$count`; do \
        time=`./main_restricted 600000000 1.0`; \
        echo "Run $$i: $$time seconds"; \
    done;
clean:
    rm -f main main.o main_restricted main_boosting main_restricted_pgo main_boosting_pgo main.prof *.profdata *.profraw
```

Listing 14: Compilation and Profiling Steps. Benchmarks can be recreated e.g. with `make benchmark_restricted && make benchmark_boosting && make benchmark_unprofiled`

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] Roger A. Bringmann, Scott A. Mahlke, and Wen-mei W. Hwu. “A study of the effects of compiler-controlled speculation on instruction and data caches”. In: *HICSS (1).* 1995, pp. 211–220. URL: https://doi.org/10.1109/HICSS.1995.375392.

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

[4] 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.

[5] 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.

[6] ARM Community. *Performance Effect Because of Removing Some Instructions from ARMv8.* Accessed: 2024-11-23. URL: https://community.arm.com/support-forums/f/architectures-and-processors-forum/7125/performance-effect-because-of-removing-some-instructions-from-armv8.

[7] Dave Estes. *SchedMachineModel: Adding and Optimizing a Subtarget.* https://llvm.org/devmtg/2014-10/Slides/Estes-MISchedulerTutorial.pdf. Accessed: 27-Nov-2024. 2014.

[8] Giuseppe A. Di Guglielmo. *llvm-mca: A Static Performance Analysis Tool.* https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html. LLVM Developer Mailing List RFC. 2018.

[9] 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.

[10] 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).

[11] LLVM Project. *Clang Supported Syntaxes Documentation.* https://clang.llvm.org/docs/AttributeReference.html. Accessed: November 27, 2024. 2024.

[12] 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.

[13] LLVM Project. *LLVM Project - Profile Data Tool.* https://llvm.org/docs/CommandGuide/llvm-profdata.html. Accessed: 27-Nov-2024.

[14] LLVM Project. *llvm-mca: Machine Code Analyzer.* Accessed: 2024-11-22. 2023.

[15] Fabian Ritter and Sebastian Hack. “AnICA: analyzing inconsistencies in microarchitectural code analyzers”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022), pp. 1–29.

[16] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. “Boosting beyond static scheduling in a superscalar processor”. In: *SIGARCH Comput. Archit. News* 18.2SI (May 1990), pp. 344–354. ISSN: 0163-5964. DOI: `10.1145/325096.325160`. URL: `https://doi.org/10.1145/325096.325160`.

[17] Shaojie Tan et al. “Uncovering the performance bottleneck of modern HPC processor with static code analyzer: a case study on Kunpeng 920”. In: *CCF Transactions on High Performance Computing* 6.3 (2024), pp. 343–364.