If-Convert as Early as You Must

Dorit Nuzman
Mobileye
Israel
dorit.nuzman@mobileye.com

Ayal Zaks
Mobileye
Israel
ayal.zaks@mobileye.com

Ziv Ben-Zion
Mobileye
Israel
ziv.ben-zion@mobileye.com

Abstract

Optimizing compilers employ a rich set of transformations that generate highly efficient code for a variety of source languages and target architectures. These transformations typically operate on general control flow constructs which trigger a range of optimization opportunities, such as moving code to less frequently executed paths, and more. Regular loop nests are specifically relevant for accelerating certain domains, leveraging architectural features including vector instructions, hardware-controlled loops and data flows, provided their internal control-flow is eliminated. Compilers typically apply predicating if-conversion late, in their backend, to remove control-flow undesired by the target. Until then, transformations triggered by control-flow constructs that are destined to be removed may end up doing more harm than good.

We present an approach that leverages the existing powerful and general optimization flow of LLVM when compiling for targets without control-flow in loops. Rather than trying to teach various transformations how to avoid misoptimizing for such targets, we propose to introduce an aggressive if-conversion pass **as early as possible**, along with carefully addressing pass-ordering implications. This solution outperforms the traditional compilation flow with only a modest tuning effort, thereby offering a robust and promising compilation approach for branch-restricted targets.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → *Single instruction, multiple data*; • **Hardware** → *Digital signal processing*.

Keywords: If-Conversion, Phase-Ordering, Decoupled Access Execute, DAE, Predication, DSP, Zero Overhead Loop

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CC '24, March 2–3, 2024, Edinburgh, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641562

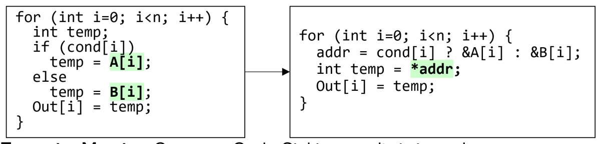
ACM Reference Format:
Dorit Nuzman, Ayal Zaks, and Ziv Ben-Zion. 2024. If-Convert as Early as You Must. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3640537.3641562

1 Introduction

General-purpose compilers optimize for arbitrary targets using an Intermediate Representation (IR) which models the Control Flow Graphs (CFG) of functions. Control flow branches are a primary subject for optimization — traditionally preferring to move unused code away from frequently executed paths, replacing conditional code with unconditional code, and more.

Unfortunately, moving code around branches is liable to hamper certain optimizations that are sensitive to memory accesses. Figure 1 depicts an example of two loads inside an `if-then-else` construct in a loop, progressing along *regular* affine access patterns — advancing by an invariant increment at each loop iteration. Such accesses are amenable to a wide range of optimizations including prefetching [22, 35], preloading and vectorization [9], polyhedral transformations [33], and more. The compiler however may decide to hoist such conditional loads and replace them by a single unconditional load, as shown in the figure, thereby reducing code size and potentially enabling subsequent optimizations. However, such load-merging trades two regular accesses for one irregular access to memory, which is less amenable to further optimizations listed above, and even more-so for the domain of our focus.

We focus on the growing domain of applications characterized by three key features: (1) they process regular memory-accesses residing in (2) "regular loop-nests", defined as a nest of countable-loops, and whose (3) internal control-flow is largely balanced, thereby amenable to full predication. We term this application-domain *Branch Evasive with Early-configuration of Regular Iterators*, or *BE'ERI* in short. This set of features traditionally characterized the DSP domain, and has become even more prevalent with the resurgence of sensing and machine-learning applications, with their demand for efficient linear-algebra processing in embedded low-power devices. Architectures in this domain have support (1) for regular memory accesses in the form of preconfiguration [21, 27], (2) for loop (backedge) branches in the form of Zero-Overhead Loops [31, 32], and (3) for other (non-loop) branches in the form of predication.

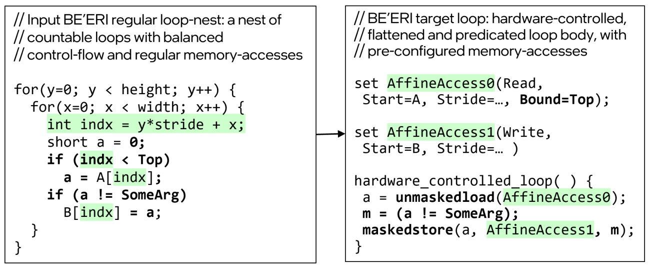


```
for (int i=0; i<n; i++) {
  int temp;
  if (cond[i])
    temp = A[i];
  else
    temp = B[i];
  Out[i] = temp;
}
```

```
for (int i=0; i<n; i++) {
  addr = cond[i] ? &A[i] : &B[i];
  int temp = *addr;
  Out[i] = temp;
}
```

Excessive Merging: Common-Code-Sinking results in irregular memory-access

Figure 1. De-optimization around control-flow.



```
// Input BE'ERI regular loop-nest: a nest of
// countable loops with balanced
// control-flow and regular memory-accesses

for(y=0; y < height; y++) {
  for(x=0; x < width; x++) {
    int indx = y*stride + x;
    short a = 0;
    if (indx < Top)
      a = A[indx];
    if (a != SomeArg)
      B[indx] = a;
  }
}
```

```
// BE'ERI target loop: hardware-controlled,
// flattened and predicated loop body, with
// pre-configured memory-accesses

set AffineAccess0(Read,
 Start=A, Stride=…, Bound=Top);

set AffineAccess1(Write,
 Start=B, Stride=… )

hardware_controlled_loop( ) {
 a = unmaskedload(AffineAccess0);
 m = (a != SomeArg);
 maskedstore(a, AffineAccess1, m);
}
```

Figure 2. Hardware-controlled loop formation: loop-nests with control-flow (1) are flattened into a straight-line block of code (boldface text), with (2) loop iterations and regular memory-accesses preconfigured and controlled by hardware (highlighted text).

Note that irregular accesses are supported, albeit less efficiently. Such architectures tend to have wide SIMD and VLIW or data-flow capabilities to fulfill high-performance and power-efficiency requirements, and benefit from mechanisms that can efficiently feed the processing engine with data via preloading, prefetching, or stream preconfiguration. Evading all branches coupled with preconfiguring regular memory accesses helps reduce the resources consumed in each iteration on such architectures, following the principle of Decoupled Access Execute [29] and akin to Stream Specialized Processors [35]. TI's C7x DSP with its streaming address generator [30], and RISC-V's Stream Semantic Registers (SSR) Extension [27] are examples of such targets.

Figure 2 depicts an example of a loop compiled in our domain. The two regular memory accesses in the loop are preconfigured in advance, instead of being computed inside the loop based on induction-variables. The two control-flow `if` statements are removed by *predicating if-conversion* and replaced with data flow. The conditional load and store may have side-effects and are therefore predicated using masking [1]. The first condition guarding the address bound of the load is folded into its preconfiguration, allowing to the load to be left unmasked. The second condition guarding the stored value is computed and fed into a masked-store instruction each iteration. The result is a flat code sequence whose memory-accesses are governed by the address-generation unit.

[1]Unlike if-conversion in LLVM middle-end, which is not predicating; see Section 3.1.

A dilemma arises when trying to compile regular loop nests to a BE'ERI architecture using a general-purpose compiler: on one hand, such compilers employ advanced optimizations including vectorization and software-pipelining that are critical for exploiting the SIMD and VLIW or dataflow nature of BE'ERI targets. On the other hand, other control-flow triggered transformations of general-purpose compilers may be tempted to exploit opportunities that are irrelevant in this domain due to its branch-evasive nature, and as shown in Figure 1, break the ability to preconfigure memory-accesses.

This paper describes an innovative approach for leveraging the powerful pipeline of a general-purpose compiler, including in particular its vectorization and software-pipelining capabilities, to optimize for targets that are Branch-Evasive with Early-configuration of Regular Iterators. This is achieved by eliminating branches very early, thereby preventing opportunities to tamper with regular accesses in the first place. Specifically, an if-conversion pass called *Early CFG-flattening* is introduced to produce a branchless version of the IR. Phase ordering concerns are considered carefully to ensure that performance is preserved.

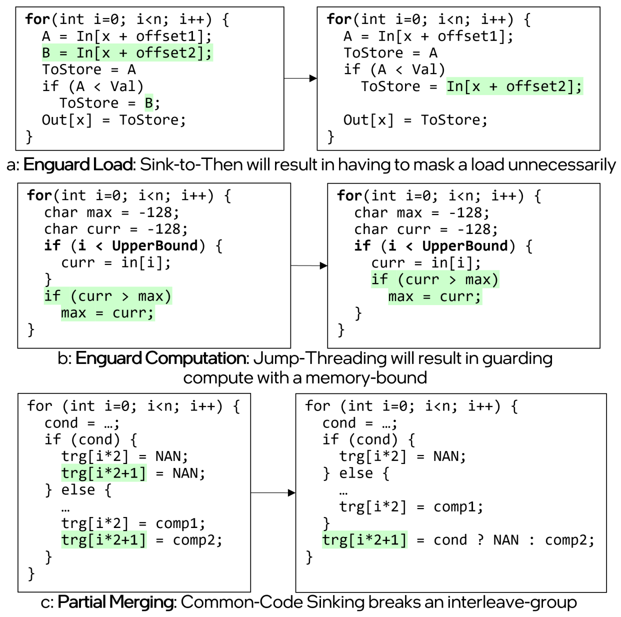
Our approach is implemented in LLVM and experimented with extensively, demonstrating that a drastically different flow of early-as-possible aggressive if-conversion is able to preserve regular accesses and outperform the traditional compilation-flow approach for the BE'ERI domain. The contributions of this paper include:

- an extensive survey of branch-associated vs. memory-minded optimization;
- an innovative compilation approach based on early-as-possible aggressive if-conversion, that best leverages a general-purpose compiler for targets that are Branch-Evasive with Early-configuration of Regular Iterators;
- a detailed description of a concrete practical implementation in LLVM, including a new optimization for merging masked loads and stores;
- an extensive experimental analysis evaluating the proposed approach, proving it is feasible and more suitable for target architectures of our focus.

In the rest of the paper we first present concrete examples of problems and challenges that motivate this study, based on traditional compilation flows (§2). Then specific compilers employing if-conversion late and early are described (§3), followed by their experimental evaluation which demonstrates the feasibility of early if-conversion (§4). A discussion covering key tradeoffs, design decisions and possible alternatives follows (§5). Finally, we compare this work with related prior art (§6) and conclude (§7).

2 Optimization Pitfalls

Compilers usually schedule target-specialization passes late, as part of their target-dependent backend. It would therefore



```
for(int i=0; i<n; i++) {
  A = In[x + offset1];
  B = In[x + offset2];
  ToStore = A
  if (A < Val)
    ToStore = B;
  Out[x] = ToStore;
}
```

```
for(int i=0; i<n; i++) {
  A = In[x + offset1];
  ToStore = A
  if (A < Val)
    ToStore = In[x + offset2];

  Out[x] = ToStore;
}
```

a: **Enguard Load**: Sink-to-Then will result in having to mask a load unnecessarily

```
for(int i=0; i<n; i++) {
  char max = -128;
  char curr = -128;
  if (i < UpperBound) {
    curr = in[i];
  }
  if (curr > max)
    max = curr;
}
```

```
for(int i=0; i<n; i++) {
  char max = -128;
  char curr = -128;
  if (i < UpperBound) {
    curr = in[i];
    if (curr > max)
      max = curr;
  }
}
```

b: **Enguard Computation**: Jump-Threading will result in guarding compute with a memory-bound

```
for (int i=0; i<n; i++) {
  cond = …;
  if (cond) {
    trg[i*2] = NAN;
    trg[i*2+1] = NAN;
  } else {
    …
    trg[i*2] = comp1;
    trg[i*2+1] = comp2;
  }
}
```

```
for (int i=0; i<n; i++) {
  cond = …;
  if (cond) {
    trg[i*2] = NAN;
  } else {
    …
    trg[i*2] = comp1;
  }
  trg[i*2+1] = cond ? NAN : comp2;
}
```

c: **Partial Merging**: Common-Code Sinking breaks an interleave-group

Figure 3. De-optimizations around control-flow.

seem natural, when compiling for branch-evasive architectures, to apply control-flow flattening (as in Figure 2) late. This means that until late in the pipeline, the compiler operates on exposed control-flow, which leads to inadvertently performing undesired transformations, as surveyed next.

2.1 Enguard Loads

Figure 3a shows a load instruction which the compiler places under a condition, noticing that the loaded value `B` is used only there, where it will hopefully execute less frequently. On a branch-evasive target however, this optimization has no benefit: subsequent if-conversion will turn this load into a masked load, restricted with a mask dependence. Later passes may not be able to undo this de-optimization as the information that the load can execute unconditionally is lost.

Guarding the load of `B` in this example has another disadvantage, associated with the load of `A`: the two neighboring unmasked loads can potentially be optimized together into a single "Superword" vector load by SLP [13] or into a single preconfigured access, but separating the two loads where one is masked but the other is not, hampers such optimization.

2.2 Enguard Computations

Figure 3b shows a computation conditionally updating `max` which the compiler places under the first `if` statement, noticing it has effect only there, where it will hopefully execute less frequently. Again, on a branch-evasive target this has no benefit: subsequent if-conversion will remove both `if` statements turning their code into a masked load and a select between the current and updated `max`. Being a memory-access upper-bound mask, it can later be removed from the load by preconfiguration, producing an unmasked load (as in Figure 2). However, the selective updating of `max` will remain tied to the bound-mask (as it is not governed by the preconfigured memory-unit), resulting in having to compute the mask in each iteration of the loop.

The above examples deal with unfortunate code moved under conditions, needlessly inflicting it with additional masks, only to be reversed later, if possible, at increased compile-time and compilation complexity. The next examples deal with unfortunate code moved out of conditions.

2.3 Excessive Merging

Figure 1 described in Section 1 shows two conditional regular loads which the compiler replaces with a single unconditional yet irregular load. Such code motion relieves subsequent if-conversion from masking the loads, thereby facilitating vectorization for targets supporting SIMD without masks, for instance. This relief comes at the price of turning the memory accesses into an irregular one, leading to less efficient vector gather or scalar operations. Worse, BE'ERI targets cannot preconfigure irregular accesses.

Restricting this optimization to merge only loads that are irregular to begin with, or that access the same address, in order to avoid de-optimization, is difficult and requires compile-time expensive analyses (aliasing, scalar-evolution). Such analyses are not suitable within lightweight passes that are liable to perform such opportunistic code motion. This is the case in LLVM's *InstCombine* and *SimplifyCFG* passes, that are invoked many times to cleanse the IR.

2.4 Partial Merging

Figure 3c shows one pair of conditional stores to the same address which the compiler replaces with a single unconditional store. In contrast to the previous scenario where such motion was overly aggressive and should be avoided if it replaces regular accesses with an irregular one, here merging was not aggressive enough — both pairs of conditional stores should be merged, or none. This is because accesses to adjacent addresses can be optimized together by SLP as described in 2.1, or preconfigured together using the same address-generation unit, or vectorized together as an interleave-group [23] using a single wide store. Separating these accesses by partial code motion hampers these optimizations, resulting in three nonconsecutive stores that lead to less efficient code; e.g. vectorization would have to use scatters or masking which may not be supported at all, or as efficiently, as a consecutive unmasked vector store.

3 Compilers for BE'ERI Targets

Compiling code containing control-flow and memory constructs for BE'ERI targets requires that all branches be evaded by predication or hardware-control of loops, and that regular memory accesses be optimized by preconfiguration. It

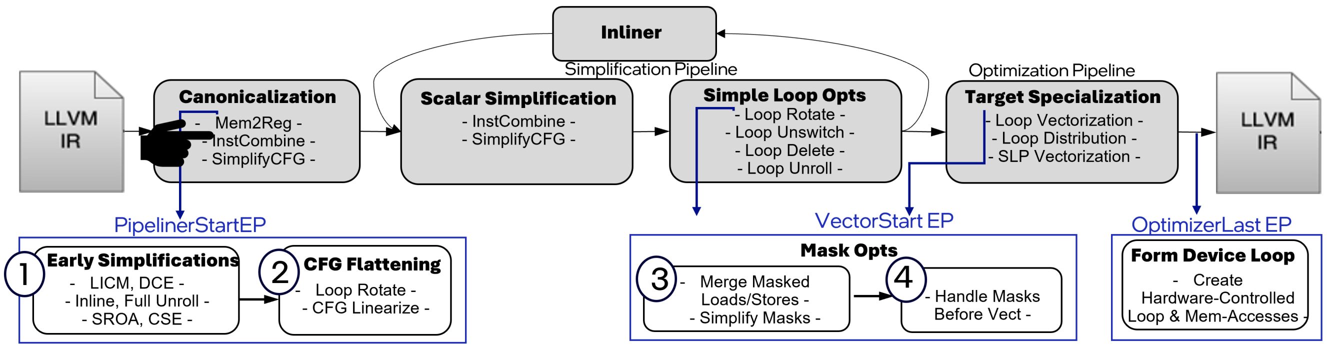


Figure 4. Pass-sequence diagram. Gray blocks on top row denote existing LLVM passes (from https://polly.llvm.org/docs/Architecture.html) with Extension-Points for target customization (EPs). White blocks on bottom row denote passes introduced for BE'ERI targets. All 3 BE'ERI compilers use *Form Device Loop* stage, where: ***base*** compiler employs *(2)CFG Flattening* late as part of Form Device Loop stage at *OptimizerLastEP*; ***early*** compiler employs *(1)* and *(2)CFG Flattening* early, with *(4)HandleMasksBeforeVect* at *VectorStartEP*; ***early-tuned*** compiler employs all white blocks with improvements to *CFG Linearize* and *HandleMasksBeforeVect*.

also requires attending to the de-optimization of memory accesses caused by optimizations around branches, surveyed in the previous section. This Section outlines three compilers for such targets, distinct in how they try to prevent LLVM passes from performing these undesired transformations:

- ***base:*** a compiler that tries to evade each de-optimization while using the traditional approach of employing aggressive if-conversion late;
- ***early:*** a compiler that prevents de-optimizations by invoking aggressive if-conversion early, and
- ***early-tuned:*** a compiler that adds improvements to the *early* compiler.

The unconventional phase-ordering employed by the latter two compilers is the main subject of our experimental evaluation, aiming to examine if, in the context of branch evasive targets, their early-flattening based approach is feasible and competitive with the former — a robust and optimized compiler based on a standard compilation flow.

3.1 Optimized Late If-Conversion: *base*

An optimized LLVM-based compiler targeting a BE'ERI architecture serves as a baseline for evaluation, referred to as *base*. The compiler leverages LLVM's standard optimization pipeline, shown schematically in Figure 4, customized by plugging additional passes in *Extension Points* (EPs).

The *base* compiler introduces passes into the *OptimizerLast* EP. These passes, denoted *Form Device Loop* in the Figure, convert loop-nests into hardware-controlled blocks, along with preconfigured memory accesses as shown in Figure 2, similar to mapping nested-loops into a Stream-Semantics Region (SSR) [27]. These passes operate on a loop-nest that consists of up to four levels (parametric on the number of dimensions the streaming unit supports), where each level can contain a single SESE loop whose iteration-count is invariant in all enclosing loops. [2] Each loop-level can potentially also contain non-loop code and control-flow. The passes assume a target that has a streaming unit capable of modelling a multi-dimensional access-pattern with the ability to predicate side-effects. They consist of *loop-nest collapsing*, *streaming-unit configuration*, and *CFG-flattening*, and result in one loop of a single BB, whose termination and strided accesses are controlled by the streaming unit. Such target-dependent transformations effectively lower the IR restricting subsequent passes, so are naturally applied late.

Loop-nest collapsing converts the loop-nest into a perfectly nested-loop; Starting from the outermost loop level, it sinks all code from outer-levels into the innermost loop with proper predication to ensure side-effects and updates carried across-iterations preserve order of execution. ***Streaming-unit configuration*** uses LLVM's Scalar-evolution analysis (SCEV) to configure min/max/stride of each dimension in the streaming unit according to the access-pattern (start/end/advancement) of a memory-access. Addresses of respective loads/stores are replaced with those produced by the streaming unit.

CFG-flattening is a predicating if-conversion pass. The if-converting passes that exist in LLVM either treat side-effect-free code only using Selects, or predicate the side-effects of vector loads and stores using their masked variants as part of loop vectorization. The vectorized code produced by loop vectorization may, however, contain branches whose conditions are known to be uniform [20] or if the instructions they guard remain scalar. Furthermore, code vectorized by

[2]Future improvements can relax these limitations by extending the passes to support loop-levels with multiple loops, conditional loops, and by employing loop-transformation (distribution, collapsing) to address excessive dimensions and other target limitations.

```
CFG_flatten(loop-nest L) {
  // Stage1: Calculate edge and basic-block predicates.
  for each basic-block BB in loop-nest L, in top-down order {
    if BB is a loop-header
      block_predicate[BB] = true; // Loops are unconditional.
    else
      block_predicate[BB] = OR {edge_predicate[P_i, BB]}
                              over all predecessors P_i of BB;
    for each successor S_i of BB
      edge_predicate[BB,S_i] =
        AND {block_predicate[BB], condition(BB->S_i)};
  }

  // Stage2: Bruteforce linearization of the code.
  for each basic-block BB, in the desired linearized order {
    // 2.1: replace each phi with selects using edge_predicates.
    for each phi in non-header BB {
      val = phi_arg(P_0); // value coming from predecessor P_0.
      for every predecessor P_i, i>0
        val = select edge_predicate[P_i,BB] ? phi_arg(P_i) : val;
    }
    // 2.2: mask side-effects.
    for each load/store instruction in non-header BB
      replace with masked load/store using mask=block_predicate[BB];
    // 2.3: linearize branches.
    if BB ends with a non-loop conditional branch
      replace branch with an unconditional branch to the next BB;
  }
}
```

Figure 5. CFG Flattening algorithm. Input: a loop-nest with internal control-flow, consisting of SESE countable loops in simplified form - with pre-header and single back-edge. Output: a loop-nest without internal control-flow.

other means such as supported by OpenCL may also contain branches. Therefore a new if-conversion pass called *CFG-Flattening* was introduced, whose simple algorithm follows that found in LLVM's loop vectorizer, extended to handle loop nests, as listed in Figure 5. This pass eliminates **all** non-loop branches by predicating vector code **as well as scalar** code — using masked single-element vector loads and stores. The *base* compiler places CFG-flattening also at *OptimizerLast* in order to eliminate all branches unsupported by the target.

Many passes that exist in LLVM and appear before *OptimizerLast* EP apply control-flow transformations. These passes include *InstCombine* and *SimplifyCFG*, which appear in Figure 4 as part of the Canonicalization and Scalar Simplification stages, but repeat many more times in later stages as well. In order to try and prevent these passes from performing undesired transformations, outlined in Section 2, the baseline compiler includes numerous "bandages" as superficial forms of mitigation, following duct-taping programming approach: these local fixes are introduced in response to specific incidents as they arise, rather than an architected systematic solution. Limitations of this approach are further discussed in Section 5. Lastly, note that the *OptForSize* compilation mode is used, in order to steer optimizations away from versioning code across multiple alternatives, only to be folded by if-conversion. The next sections propose alternatives aiming to trade these bandages for a more robust solution based on moving CFG-flattening earlier.

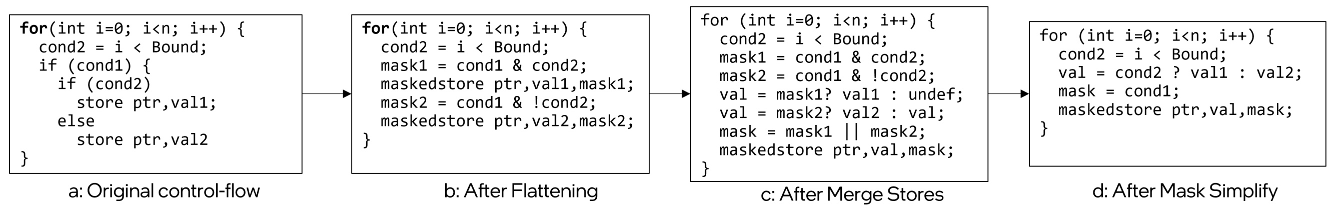
3.2 Basic Early If-Conversion: *early*

Applying bandages to block various compilation passes from making detrimental branch-related decision, in the context of late if-conversion, is inherently cumbersome and arguably unscalable. A preventive alternative eliminates all potentially misleading branches before any such potentially harmful pass is invoked. In the pass diagram of Figure 4 this immunization point corresponds to the first *PipelineStart* extension point, next to the pointer icon, prior to any *InstCombine* and *SimplifyCFG* pass. Such an alternative compiler is called *early* — it invokes if-conversion as early as possible, as the 2nd *CFG Flattening* stage in Figure 4, compared to the *base* compiler which places it as late as possible — in the *OptimizerLast* extension point at *Form Device Loop* stage.

Few additional passes are invoked early along with CFG-flattening due to their ability to promote or simplify memory accesses before they are masked, without having harmful effects on other accesses. These passes include Scalar Replacement Of Aggregates (SROA) and Common Subexpression Elimination (CSE), which in turn call for earlier invocations of Loop Invariant Code Motion (LICM), Dead Code Eliminate (DCE), function inlining, and complete loop unrolling. These passes constitute the 1st *Early Simplifications* stage denoted in Figure 4, which *early* invokes in *PipelineStartEP* prior to the 2nd *CFG Flattening* stage.

One case drastically affected by early if-conversion in LLVM is vectorization. The auto-vectorization passes of LLVM [15], both its Loop Vectorizer (LV) and Superword-Level-Parallelism vectorizer (SLP [13, 25]), process scalar control-flow code and **refrain from vectorizing masked operations** — as produced by CFG-flattening. Scheduling the Vectorization passes before CFG-flattening at *OptimizerStartEP* is not suitable for such cost-based sensitive optimizers. They are best applied late on optimized IR, and extended to process masked operations. This aligns with LV's roadmap [16], but involves extensive refactoring to separate predication from vectorization. Extending SLP to process masked operations is also a noteworthy endeavor, which may provide additional advantages [6, 24].

Until these efforts mature, a simple Reverse-If-Conversion pass was introduced and placed in the *VectorStart* extension point to enable the vectorizers (and can be removed once they are taught to operate on masked code). Given a flattened IR with masked loads and stores, the Reverse-If-Conversion pass transforms them back into regular unmasked loads and stores guarded by branches [36]. The simple implementation of this pass in *early* scans the input IR top-down, replacing each masked load/store with a (unmasked) load/store in a new BB guarded by a conditional branch, with the mask as its condition. Applying reverse if-conversion before vectorization is followed by an invocation of CFG-flattening afterwards, in case vectorization generated branches or did not take place.



```
for(int i=0; i<n; i++) {
  cond2 = i < Bound;
  if (cond1) {
    if (cond2)
      store ptr,val1;
    else
      store ptr,val2
}
```

```
for(int i=0; i<n; i++) {
  cond2 = i < Bound;
  mask1 = cond1 & cond2;
  maskedstore ptr,val1,mask1;
  mask2 = cond1 & !cond2;
  maskedstore ptr,val2,mask2;
}
```

```
for (int i=0; i<n; i++) {
  cond2 = i < Bound;
  mask1 = cond1 & cond2;
  mask2 = cond1 & !cond2;
  val = mask1? val1 : undef;
  val = mask2? val2 : val;
  mask = mask1 || mask2;
  maskedstore ptr,val,mask;
}
```

```
for (int i=0; i<n; i++) {
  cond2 = i < Bound;
  val = cond2 ? val1 : val2;
  mask = cond1;
  maskedstore ptr,val,mask;
}
```

Figure 6. Merge Loads/Stores optimization on masked flattened IR.

Lastly, note that the *OptForSize* compilation mode applied by *base* helps *early* ensure that no new control-flow will be introduced after it applies if-conversion early.

3.3 Optimized Early If-Conversion: *early-tuned*

Invoking CFG-flattening as early as possible by *early* revealed performance deficiencies that may be attributed to few root causes, including (1) missed mergings of masked loads and stores, (2) convoluted masks due to overly simplified if-conversion, and (3) convoluted control-flow due to overly simplified reverse-if-conversion. Several fixes were devised and applied on top of *early* to produce *early-tuned*:

Improved load/store merging: The common-code transformations of LLVM are able to optimize multiple loads or stores that access a common address, as in Figure 6a, but they depend on control-flow. Applying CFG-flattening early eliminates control-flow by masking such loads and stores separately, as in Figure 6b. To compensate for this missed optimization, a new *MergeMaskedLoadsStores* pass was introduced into *early-tuned* at the 3rd *Mask Opts* stage in Figure 4 in order to merge such masked loads or stores into a single masked load or store using combined mask and value, as in Figure 6c. The pass scans the code recording groups of loads (stores) that access the same location. For each group, if the loads (stores) can be placed together (no dependences intervene), they are replaced by a single masked load (store) to that memory location, whose mask is produced by OR-ing the masks of the original loads (stores). The value feeding the store is selected from the original values of the original stores based on their masks.

Merging masked loads is generally profitable, but merging masked stores incurs the cost of combining both mask and value, which are subject to further optimization, as in Figure 6d. The *InstCombine* pass of LLVM is often able to perform this simplification, but a complementary *Simplify-Masks* pass was added to better optimize cases where multiple masks share common parts, such as *cond1* in Figure 6. Despite this effort there are still cases where store merging best be avoided, as shown in Section 4.4.

Improved flattening: *early-tuned* improves the phi-to-select linearization of the CFG-flattening algorithm (Stage 2.1 in Figure 5) by noticing that when all predecessors `P_i` of `BB` have a common single predecessor, there is no need to include the predicate of the common-predecessor; Simplified selection predicates can be produced using only the branch-condition from that common-predecessor to its successors, instead of the entire edge_predicates. Such improvement already exists in LLVM's SimplifyCFG Pass and has also been described more generally [6].

Improved Vectorization of Masked Code: The reverse-if-conversion implemented in *early* is rather straight-forward: it un-masks each memory access separately. This fragments the code into many small basic-blocks, leading to poor SLP-vectorization and suboptimal loop-vectorization of interleave-groups. Multiple loads and stores that share a common mask are grouped and un-masked together in an improved version of reverse-if-conversion included in *early-tuned.*

4 Experimental Evaluation

Applying early-flattening to the scenarios of Section 2 confirms that it can achieve our main goal: this flow prevents de-optimizations while preserving regular accesses throughout the compilation process up to vectorization and Device-Loop formation, when applied to these scenarios.

Next we evaluate how a large testsuite consisting of a wide range of tests responds to the radically different flow of early-flattening. We use the full OpenCL benchmark suite of our compiler, without changes, which contains hundreds of Deep Learning and Computer Vision tests covering many computational kernels and few full applications. Many tests leave most of the optimization to the compiler, which helps evaluate how well optimization capabilities are preserved.

Performance is measured in number of cycles using a simulator modelling a realistic BE'ERI target, featuring wide VLIW and SIMD capabilities, with ZOL support, and regular masked loads and stores governed by a streaming address-generation unit, complemented with a gather-scatter unit. We measure the total number of cycles executed by each loop offloaded to the target. All operations, including memory accesses, take a fixed number of cycles. We measured and present the performance achieved by each of the three compilers described in Section 3: *base*, *early* and *early-tuned.* Note that the improvements added to *early* to produce *early-tuned* also apply to *base* but are ineffective there, due to its traditional and highly tuned flow.

Figure 7 shows the cycle-counts of *early* and *early-tuned*, both normalized to the cycle-count of *base*, where lower is

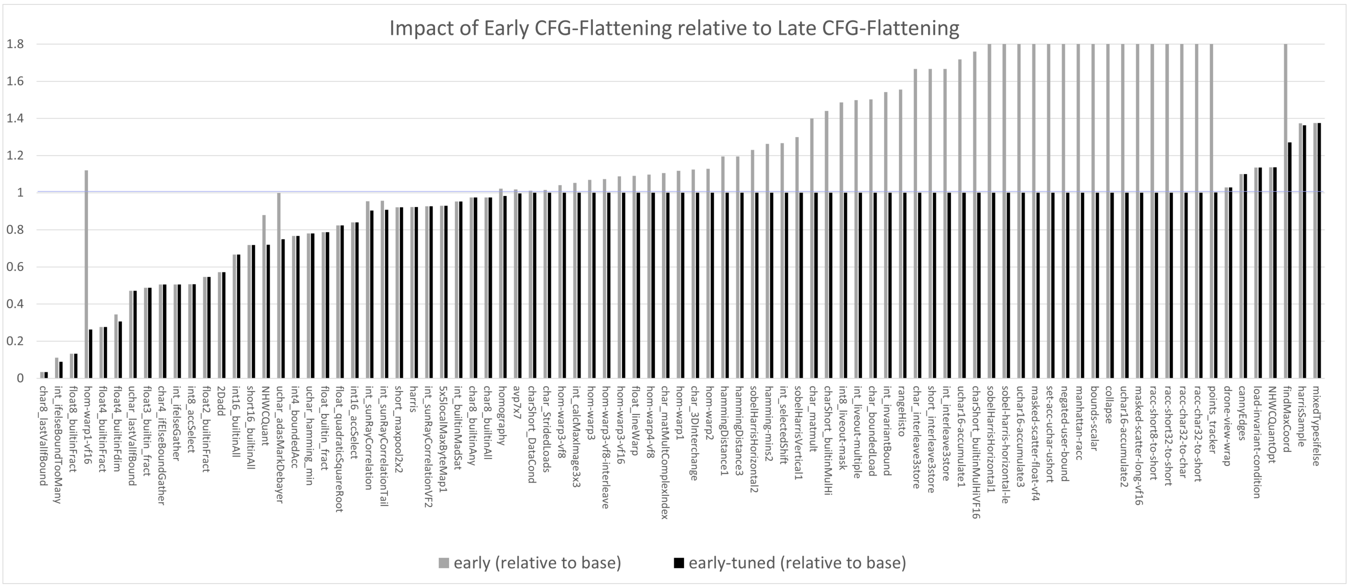


Figure 7. Impact of early CFG-flattening: *early* and *early-tuned* cycle counts are shown relative to *base*, lower is better. Gray bars are cut off at 1.8x, but reach up to 177x.

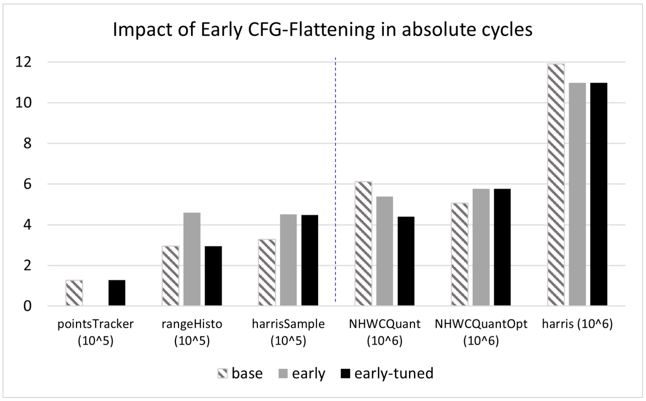


Figure 8. Impact of early CFG-flattening: bars are in number of cycles, in units of 10^5 and 10^6 cycles for the first and last three tests, respectively.

better. The former reflects the effect of basic phase-reordering while the latter illustrates the relative performance of additional improvements.

Many tests of the benchmark were agnostic to changes in the compiler and are thus omitted. All tests that were affected are shown, filtering only repetitions of identical behavior, i.e., due to slight variations of the same basic test. The results are sorted according to the relative performance of *early-tuned*, from largest speedup on the left to largest slowdown on the right. Bars (of *early*) are cut-out at 80% degradation for brevity.

Figure 8 brings a "magnified view" for a few of the larger tests of Figure 7, comparing the same three compilers but showing their absolute cycle-counts, for long running full applications that consist of a large loop or multiple loops: *pointsTracker* tracks movement of points between frames; *rangeHisto* is a utility that computes a histogram of ranges used for tone mapping, *harrisSample* and *harris* compute Harris Corner Detection, and *NHWCQuantizeOpt* and *NHWC-Quantize* are two versions of a Deep-Learning test computing convolutional layer NHWC quantization — the former moves loads and stores manually whereas the latter leaves this optimization to the compiler. The performance of these tests is analyzed below along with all tests of Figure 7.

4.1 Overall Behavior

On average, *early-tuned* is on-par with and even better than *base* — reaching 9% performance improvement across all 87 affected tests. This required modest effort of about 6 person-months, to implement improvements outlined in Section 3.3, eliminating an 8x average degradation of *early* compared to *base*. This demonstrates that while the initial impact of the new flow is significant, it suffices to address few key weaknesses to quickly close the gap with the regular flow.

Note that the overall performance improvement, while being an encouraging outcome, was not the primary goal of the *early* compilers; The primary goal was to demonstrate that the more robust measures employed by the *early* compilers do not incur a performance penalty despite the drastically different compilation-flow that these measures involve. This improved robustness manifested itself in both *early* and *early-tuned* successfully compiling 17 tests that *base* failed to compile. Most failures were due to excessive merging (§2.3) resulting in missed SROA opportunities and loop-vectorized random gathers exceeding resources. The remaining couple of failures were due to control-flow blocking SLP vectorization, resulting in excessive register pressure. These pitfalls were avoided in the *early* compilers.

The rest of this section dives into detailed analysis of specific cases, which provides opportunities for further improvements of both *base* and *early-tuned*.

4.2 Gains from Phase-Ordering

The first group of tests on the left of Figure 7 are tests that *early-tuned* improves compared to *base*. It consists of 33 tests (38% of all tests depicted), almost all of which are improved by *early* as well, implying that nearly all improvements stem from early flattening and pre-flattening phases alone. Surprisingly, only few improvements resulted directly from preserving regular memory accesses — in a couple of tests that exhibit the issues illustrated in Figure 1 and Figure 3c. These issues were addressed by *early-tuned*, leading to 50% and 17% improvements over *base*.

Most improvements are attributed indirectly to later passes working better after flattening. Some improvements are more casual than causal, resulting from changes in early canonicalization decisions that happen to better suit later passes; but in other tests, as we show later, these changes resulted in slowdowns. The main improvements observed fall into the following four cases.

The largest improvement originates from loop invariant code motion (LICM) optimizing a loop provided its body had been flattened. This opened opportunities for further optimization, including LoopDeletion which determined that an innermost loop in one test needed its last iteration only, leading to 97% (30x) improvement.

Another large improvement comes from loop vectorization (LV) optimizing a loop provided it is free of switch statements, which are potentially created from branches by CFG-Simplification unless they are flattened first. For targets that support neither branches nor switch statements, this if-to-switch transformation is largely redundant if not detrimental. Note that *base* does not intercept this transformation because doing so would involve excessive refactoring. This led to 91% (11x) improvement in the test at hand.

Several tests improved because SLP vectorization is more effective in the absence of control-flow [6, 28]. This led to a 45%–87% (1.8x–7.5x) gain in a handful of tests, depending on the size of the resulting vectors, between 2–8 elements.

Various tests improved because of different decisions made by *InstCombine*. In one case folding of type-conversions into loads, performed by *base*, was prohibited in *early* because it masks the loads. This however resulted in 49% improvement due to subsequent improved code generation. In two cases different **reassociation** decisions improved the ability of the later *SLP* pass to vectorize operations, leading to roughly 30% improvement. Early-flattening improved the canonicalization of compare-select logic in one test resulting in 27% improvement.

4.3 Resolved Degradations

The second group of tests at the center of Figure 7 consists of 47 tests (54% of all tests depicted) having a performance gap between *early* and *base* which *early-tuned* manages to close. These include several tests which *early* failed to compile due to excessive resources, depicted alongside severe degradations exceeding 1.8x slowdown. Four main improvements were applied in order for *early-tuned* to close this gap.

Improved Flattening. The CFG-flattening implemented in *early* is rather straight-forward, producing predicates in a straight-forward manner, similar to that employed by LV. A dozen tests were more efficiently if-converted in *base* by LLVM's CFG-Simplication pass, before reaching its late CFG-flattening pass. The simple improvement added to *early-tuned* eliminated the 2%–26% degradations of these tests compared to *base*. Future work can further improve CFG flattening such as presented in recent work [26].

Repercussions of a Different Flow. Changing the flow exposed a few latent shortcomings of later passes unprepared to optimize different patterns. Two tests revealed suboptimal instruction-selection in our backend, contributing 30% and 47% improvements over *early* when fixed. One case involved an *InstCombine* phi optimization — folding constants into phi-nodes — which created a pattern that scalar-evolution analysis (SCEV) failed to handle, directly effecting memory access computations and resulting in degradations of five tests. Fixing this issue improved two tests by 42% and 70%, turning three tests from failure-to-compile by *early* to success on par with *base*. One of these three tests is *pointsTracker*, shown also in Figure 8. Lastly, seven tests failed to vectorize a loop due to an unexpected type conversion within an accumulation pattern, exposing a known latent deficiency of LV. The early invocation of Loop Rotation in preparation for flattening caused *InstCombine* to chose a different canonicalization of type conversion. These tests suffered huge performance loss — fixing the issue brought 9x–177x speedups, even though their loops are free of control-flow, i.e., no flattening actually took place.

Vectorizing Masked Code. The simple enhancement to reverse-if-conversion in *early-tuned* provided 36%–91% improvements by enabling improved SLP/loop-vectorization thanks to reduced fragmentation into small basic-blocks. In *rangeHisto* (shown also in Figure 8), this fixed an initial 55% degradation.

A secondary issue involves an internal optimization that operates before LV and lacked support for masked operations. Fixing it improved ten tests by 5%–86%. This demonstrates how a permuted flow can help stress-test a compiler, uncover latent unsupported cases, and improve the compiler's overall robustness.

Merging Masked Loads and Stores. This optimization fuses groups of masked loads and stores together. It improves the performance of *early* for a few tests by 8%–35%, thereby closing their gaps with *base*. It also helps further accelerate a few other tests which *early* improves over *base*, reaching 18%–25%, with *NHWCQuant* of Figure 8 among them. The control-flow merging of loads and stores employed by *base* may thus have opportunities for improvement, but so does the merging of masked loads and stores employed by *early-tuned*, as we show next.

4.4 Remaining Degradations

The remaining few degradations, on the right of Figure 7, fall into two main categories: under-optimized memory-accesses and sub-optimal handling of new IR patterns.

Merging masked loads and stores employed by *early-tuned* produces inferior results to control-flow merging employed by *base*, on two tests. Early CFG-flattening of another test obstructs a LICM opportunity involving an invariant selection of base addresses, leading to two masked loads instead of one, which *early-tuned* is unable to merge. The patterns exposed by these tests provide opportunities for further tuning and improvement of *early-tuned*.

The remaining performance losses are associated with poorer canonicalization of *early-tuned* compared to *base*, due to transformations described earlier (§4.2), that resulted in **gains for other tests**: missed folding of type-conversions into loads due to masking and lack of if-to-switch conversion. For example, *HarrisSample* (shown also in Figure 8) suffered a 36% degradation due to a blocked canonicalization of data-types around loads, and *NHWCQuantizeOpt* suffered a 14% degradation due to improved optimization of mask-logic in *base* following if-to-switch simplification.

5 Discussion

Several alternatives to applying if-conversion early are surveyed next, along with tradeoffs and design-decisions that motivated and guided our proposed approach.

5.1 Alternative Mitigations and their Drawbacks

Section 2 examines four scenarios where memory accesses became more difficult to optimize because of optimizations around branches. Several alternative approaches to early if-conversion can be applied to try and remedy this predicament, but have significant drawbacks:

Fix potentially offending Passes: treat the root cause — go after passes liable to "pessimize" memory-accesses and teach them to be "smarter" by equipping them with cost-models that can guide them towards more informed decisions. Such cost-models, however, involve compile-time-consuming analyses including Scalar Evolution (SCEV), memory aliasing, and fusion of strided accesses into interleave-groups, which are not suitable to apply within cleanup Passes that are invoked many times. Furthermore, these cost-models may depend on opportunities and preferences of later passes that may consider a wider context, so fit better within them, rather than within local opportunistic simplifications.

A more modest approach could try to block such occurrences categorically, without expensive cost models. However even such "bandages" require the ability to separate the profitable transforms from the destructive ones. In practice, code-motion scenarios around control-flow are not encapsulated within separate Passes that can be controlled by target-specific switches in LLVM, but rather dispersed among many Passes. For example, the scenarios of Section 2 are caused by *Jump Threading*, various *InstCombines*, several *CFG-Simplifications*, as well as other dedicated sink/hoist and merge loads/store Passes. These occurrences are unbounded and inseparable from the overall optimization Passes they participate in. This lesson is learned from extensive work with the *base* compiler and its "bandages".

Fix potentially offended Passes: treat the effect — teach later Passes to deal with patterns that earlier code-motion transformations had complicated, possibly by undoing them first. This often involves reverse-engineering, a pattern-based approach which is not systematic nor robust; the selection patterns that can result from scenarios depicted in Figures 1 and 3c for example are boundlessly diverse, and also depend on information being preserved. In the example of Figure 3a however, semantic information on whether the load can be executed speculatively is lost. Lastly, this special handling needs to be added to each Pass that may be effected: Vectorization, SROA, and more, each would need to be extended to deal with the implications of control-flow optimizations.

Fix the source programming language: prevent harmful situations before their inception by forbidding the use of control-flow in the source language level, e.g., using a Domain Specific Language. Note that the front-end should refrain from expanding source-level constructs such as ternary operators into branches. This approach effectively restricts the programmer or burdens her with the branch elimination task. Control-flow is however a most natural way to express programs with a dynamic nature, and the burden of predication best be automated. Our proposed approach to eliminate branches as early as possible after the front-end, achieves the same goal but without affecting the programmer.

5.2 Benefits of Early-Flattening

Removing the trigger for the potential detrimental optimizations by early flattening creates a "safe-zone" for optimization: the flattened IR allows a general-purpose optimizing compiler to avoid branch-related pitfalls that can complicate things for subsequent optimizers, and offers in addition several important advantages.

Robustness and Scalability: a major advantage is to have a single concise architectural solution in one place rather than devising multiple different fixes across various parts of the compiler for each individual branch-related scenario that arises, or that may arise in the future. Early flattening makes for a much more comprehensive, robust and scalable solution than the aforementioned alternatives.

Compilation efficiency: compile-time is saved from being spent on irrelevant optimization paths or attempts to undo them later. Compile-time is better served by focusing on optimizing a more relevant representation of the program, than employing time-consuming analyses through the rest of the compiler, trying to predict if certain code motion opportunities across branches are potentially helpful or harmful, or trying to reverse-engineer transformations after the fact. Indeed while a few more passes were added to early and early-tuned, compile time hasn't increased [3], which may attest to the fact that flattening early has made analyses of subsequent passes simpler and avoided redundant optimization paths.

Practical: the alternatives entail a long-term refactoring effort, because there isn't always a clear separation between canonicalizations and optimizations, nor is there always a good way for targets to control these canonicalizations.

Empowering Other Optimizations: straight-line code is more amenable to optimization because many analyses and transformations are confined to single-basic-block boundaries, and therefore limited by control-flow. This welcome side-effect was indeed responsible for most of the gains of the new phase-ordering (§4.2).

Aligned with other projects: the early-flattening based approach is aligned with ongoing projects that promote the use of a predicated IR for optimization, such as the Vector-Predicate project [18], VPlan [16] and advanced SLP vectorization [6], as discussed above, as well as the seminal work of August, Hwu and Mahlke [3].

5.3 Shortcomings of Early If-Conversion

Invoking if-conversion early has several shortcomings with potential mitigations.

Repercussions of a different flow: any drastically different flow may have miscellaneous effects caused by arbitrary passes encountering new IR patterns. These "fuzzing" side-effects usually indicate an overfitting of passes to specific patterns. Note that such effects go both ways — distinct optimization paths may lead to slowdowns (§4.3,§4.4) but may also lead to speedups(§4.2). In any case, such effects reveal latent opportunities to improve the robustness and quality of the compiler in general.

Resurrection of control-flow: early-flattening is based on the assumption that control-flow will remain flat throughout subsequent passes. This is in well aligned with current general compiler pipelines whose canonicalization efforts tend to simplify control-flow rather than complicate it. Specific optimization passes may however potentially introduce new control-flow constructs, most notably code-versioning employed by loop vectorization and loop unswitching. Turning on *OptForSize* optimization mode can prevent passes from introducing control-flow, because it typically increases the code size. In addition, rare occurrences of passes that introduce new control-flow tend to be controversial, off by default, and provide dedicated switches to control them.

Missed Optimizations: general purpose compilers including LLVM consider arbitrary targets that support branches rather than predication, especially of scalar operations, in their IR and passes. Therefore, applying if-conversion early risks missing fruitful optimizations. This is the case for several passes, including common-code sinking, SROA, CSE, LV and SLP. Optimizations inhibited by predication can be recovered by either invoking them earlier, extending them to operate on predicated code, complementing them with new passes, or preceding them with reverse if-conversion.

Passes inhibited by predication are good candidates for early invocation, prior to if-conversion, provided they refrain from irregulating memory accesses, and operate well on unoptimized IR. For example, some *InstCombine* canonicalizations are hampered by predications, but this pass is liable to sink code in uninformed and potentially harmful ways. Vectorization passes are disabled by predication but are cost-based and so best operate on optimized IR rather than be invoked early. On the other hand, simplification passes such as SROA and CSE as well as cleanups such as LICM and DCE, can be invoked early to leverage control-flow, and assist early if-conversion by simplifying its input.

Another important missed optimization is merging conditional accesses to the same location (§4.3). The profitability of merging depends on many factors, including simplification of masks (§3.3), and as evident by remaining opportunities (§4.4), it can be further tuned. Merging can be applied early, prior to flattening, leveraging dominance analysis to simplify masks. It can alternatively be applied late, on straight-line predicated code, potentially at *Form Device Loop* stage, where resource utilization can be more accurately estimated. E.g, the code in Figure 6d can be produced by processing the control-flow of Figure 6a or the predicated code of Figure 6b. Whether applied early or late, merging should be done only in dedicated, informed, passes; The key is to prohibit any

[3]Only 5 out of over 2000 tests exhibited measurable compilation time effects, 4 of which for better and one for worse.

opportunistic merging **elsewhere** in the compiler. This is what early CFG-flattening guarantees.

6 Related Work

If-conversion has been a topic of compiler research for over four decades [1]. There is, in general, an inherent tension between applying if-conversion early in the compilation process and applying it late [3, 10]. On one hand, by increasing the size of blocks, if-conversion potentially increases opportunities for all subsequent optimizations, motivating its early application. On the other hand, branches eliminated by if-conversion trade their target-specific cost with other costs including register pressure, a tradeoff best evaluated late. The common practice of contemporary general-purpose optimizing compilers is therefore to apply limited if-conversion early or apply aggressive if-conversion late, retaining control-flow in the IR through most of their compilation flow.

August, Hwu and Mahlke [3] proposed to apply aggressive if-conversion early for predicated superscalar architectures ignoring most scheduling concerns, coupled with partial reverse if-conversion [2] applied late during instruction scheduling to mitigate overly predicated results. Jordan, Kim and Krall [10] investigated the tradeoffs between applying if-conversion early in LLVM-IR versus late in machine-IR. They targeted a predicated VLIW architecture with SIMD and hardware loops. Applying if-conversion early provided them with greater optimization opportunities at the risk of increased register pressure. Their conclusion suggested that both may be desired along with better cost models. Our work also leverages LLVM for predicated architectures and highlights the potential of early if-conversion. In contrast to prior over-predication concerns, we focus on preserving the ability to optimize memory-accesses, vectorize, and software-pipeline, by aggressively if-converting as early as possible.

One major use of if-conversion has been to mitigate branch misprediction penalties, albeit the difficultly for compilers to identify which branches are destined to mispredict. Wish Branches [11], for instance, proposed to delay this decision to runtime, by encoding a branch along with its if-converted version. Our work focuses on full if-conversion for branch-evasive architectures, where such penalties are less relevant.

Full reverse if-conversion was introduced by Warter et al. [36]. We use a fairly straightforward implementation in order to facilitate LLVM's loop and SLP vectorization passes, which are currently control-flow-based. Chen, Mendis and Amarasinghe [6] applied if-conversion in order to facilitate (an enhanced version of) LLVM's SLP vectorization pass, followed by reverse if-conversion. Their work introduced a more sophisticated SSA IR which allows vectorizing instructions across loops, and their reverse if-conversion is capable of materializing new loops. Ding and Önder [7] presented an approach which may improve upon if-conversion using SSA with Future Values. Our work focused on leveraging LLVM's existing SSA IR and passes.

Zimmerman [40] used profiling information to drive if-conversion in LLVM for superscalar microprocessors, and Kong et al. [12] did so using GCC for the Sunway processor. Moll [18] proposed to predicate LLVM's IR to cover all vector instructions. We use a fairly straightforward implementation of full if-conversion which uses LLVM's existing select instruction along with its masked (single element) vector load and store intrinsics.

Vectorization can employ partial if-conversion where uniform branches are retained [19, 20]. Vectorizing for branch aversive targets require that all branches be if-converted. Software-pipelining and Modulo scheduling are strongly connected with if-conversion and reverse if-conversion of loop bodies, in the context of VLIW targets [8, 17, 34, 38, 39], and CGRA's [37]. We leverage LLVM's existing modulo scheduler which operates on loops with if-converted bodies [14].

Generalized Index-Set Splitting was proposed to optimize multi-basic-block loops into single-basic-block ones [4], but creates multiple loops and only moves the control-flow from inner to outer loops in the loop-nest. Recognizing affine memory accesses is vital for many compiler optimizations including polyhedral transformations [33], vectorization [23], utilization of local memories [5, 9], and more. We focus on preserving such accesses and optimizing them through early if-conversion.

7 Conclusion

Architectures averse to branches and irregular memory accesses include a wide range of accelerators having support for hardware-controlled loops, preconfigured streams, software prefetching, scratchpad memories, VLIW, SIMD, dataflow processing, and more. General-purpose compilers are capable of targeting such architectures but tend to suffer from premature optimization. Intermediate representations that expose control-flow constructs create a false impression of optimization opportunities for compilers of such targets, leading their transformations to make detrimental decisions.

Instead of teaching all relevant transformations how to best optimize for such targets, we propose to effectively modify the intermediate representation prior to any relevant transformation. Introducing an if-conversion pass as early as possible provides a robust and modular implementation of this approach, revealing several phase-ordering related deficiencies. A few additional preparatory and post processing passes suffice to preserve and even accelerate performance across a wide range of tests. This innovative compilation flow of LLVM is feasible and promising for BE'ERI targets, and can hopefully steer powerful general-purpose compilers in this direction for similar targets.

References

[1] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

[2] David I. August, Wen Mei W. Hwu, and Scott A. Mahlke. 1999. Partial reverse if-conversion framework for balancing control flow and predication. *International Journal of Parallel Programming* 27, 5 (1999), 381–423. https://doi.org/10.1023/A:1018787007582

[3] David I. August, Wen-mei W. Hwu, and Scott A. Mahlke. 1997. A Framework for Balancing Control Flow and Predication. In *Proceedings of 30th Annual International Symposium on Microarchitecture* (Research Triangle Park, NC) *(Micro '97)*. IEEE Computer Society, USA, 92–103. https://doi.org/10.1109/MICRO.1997.645801

[4] Christopher Barton, Arie Tal, Bob Blainey, and José Nelson Amaral. 2005. Generalized Index-Set Splitting. In *Compiler Construction*, Rastislav Bodik (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 106–120.

[5] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. A Compiler Framework for Optimization of Affine Loop Nests for Gpgpus. In *Proceedings of the 22nd Annual International Conference on Supercomputing* (Island of Kos, Greece) *(ICS '08)*. Association for Computing Machinery, New York, NY, USA, 225–234. https://doi.org/10.1145/1375527.1375562

[6] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All You Need is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3519939.3523701

[7] Shuhan Ding and Soner Önder. 2010. Unrestricted Code Motion: A Program Representation and Transformation Algorithms Based on Future Values. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 26–45.

[8] Kemal Ebcioğlu. 1987. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming* (Colorado Springs, Colorado, USA) *(Micro 20)*. Association for Computing Machinery, New York, NY, USA, 69–79. https://doi.org/10.1145/255305.255317

[9] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. 2005. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*. IEEE Computer Society, USA, 161–172. https://doi.org/10.1109/PACT.2005.33

[10] Alexander Jordan, Nikolai Kim, and Andreas Krall. 2013. IR-Level versus Machine-Level If-Conversion for Predicated Architectures. In *Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems* (Shenzhen, China) *(ODES '13)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/2443608.2443611

[11] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale Patt. 2006. Wish Branches: Enabling Adaptive and Aggressive Predicated Execution. *IEEE Micro* 26 (2006), 48–58. https://api.semanticscholar.org/CorpusID:6838785

[12] JinYing Kong, Lin Han, JinLong Xu, and Kai Nie. 2022. Research on control flow conversion technique based on Domestic Sunway compiler. In *2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP)*. IEEE Computer Society, Xi'an, China, 1340–1344. https://doi.org/10.1109/ICSP54964.2022.9778356

[13] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[14] Tanya M. Lattner. 2005. *An Implementation of Swing Modulo Scheduling with Extensions for Superblocks*. Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.cs.uiuc.edu.`.

[15] LLVM. 2023. *Auto-Vectorization in LLVM*. LLVM. Retrieved October 26, 2023 from https://llvm.org/docs/Vectorizers.html

[16] LLVM. 2023. *Vectorization Plan*. LLVM. Retrieved October 26, 2023 from https://llvm.org/docs/VectorizationPlan.html

[17] Dragan Milicev and Zoran Jovanovic. 2002. Control Flow Regeneration for Software Pipelined Loops with Conditions. *International Journal of Parallel Programming* 30 (06 2002), 149–179. https://doi.org/10.1023/A:1015453520790

[18] Simon Moll. 2020. *Vector Predication Roadmap*. LLVM. Retrieved October 26, 2023 from https://llvm.org/docs/Proposals/VectorPredication.html

[19] Simon Moll and Sebastian Hack. 2018. Partial Control-Flow Linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 543–556. https://doi.org/10.1145/3192366.3192413

[20] Simon Moll, Shrey Sharma, Matthias Kurtenacker, and Sebastian Hack. 2019. Multi-Dimensional Vectorization in LLVM. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing* (Washington, DC, USA) *(WPMVP'19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/3303117.3306172

[21] Jaime H. Moreno, Victor V. Zyuban, Uzi Shvadron, Fredy D. Neeser, Jeff H. Derby, Malcolm S. Ware, Krishnan Kailas, Ayal Zaks, Amir B. Geva, Shay Ben-David, Sameh W. Asaad, Thomas W. Fox, Daniel Littrell, Marina Biberstein, Dorit Naishlos, and Hillery C. Hunter. 2003. An innovative low-power high-performance programmable signal processor for digital communications. *IBM J. Res. Dev.* 47, 2-3 (2003), 299–326. https://doi.org/10.1147/RD.472.0299

[22] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS V)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/143365.143488

[23] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-Vectorization of Interleaved Data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/1133981.1133997

[24] Vasileios Porpodas and Pushkar Ratnalikar. 2021. PostSLP: Cross-Region Vectorization of Fully or Partially Vectorized Code. In *Languages and Compilers for Parallel Computing*, Santosh Pande and Vivek Sarkar (Eds.). Springer International Publishing, Cham, 15–31.

[25] Rodrigo C. O. Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís F. W. Góes, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Vectorization-Aware Loop Unrolling with Seed Forwarding. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3377555.3377890

[26] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. 2022. DARM: Control-Flow Melding for SIMT Thread Divergence

Reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–13. https://doi.org/10.1109/CGO53902.2022.9741285

[27] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. 2021. Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores. *IEEE Trans. Comput.* 70, 2 (feb 2021), 212–227. https://doi.org/10.1109/TC.2020.2987314

[28] Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, USA, 165–175. https://doi.org/10.1109/CGO.2005.33

[29] James E. Smith. 1982. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Austin, Texas, USA) *(ISCA '82)*. IEEE Computer Society Press, Washington, DC, USA, 112–119.

[30] TI. 2023. *C7000 C/C++ Optimization Guide*. TI. Retrieved January 2022 from www.ti.com

[31] Gang-Ryung Uh, Yuhong Wang, Sanjay Jinturkar, Chris Burns, and Vincent Cao. 2000. Techniques for Effectively Exploiting a Zero Overhead Loop Buffer. In *Proceedings of the 9th International Conference on Compiler Construction* (Berlin, Germany). 157–172. https://doi.org/10.1007/3-540-46423-9_11

[32] Janek van Oirschot. 2022. *Hardware Loops in the IPU Backend*. Graphcore. Retrieved May 2022 from https://llvm.org/devmtg/2022-05/slides/

[33] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral Code Generation in the Real World. In *Proceedings of the 15th International Conference on Compiler Construction* (Vienna, Austria) *(CC'06)*. Springer-Verlag, Berlin, Heidelberg, 185–201. https://doi.org/10.1007/11688839_16

[34] Miao Wang, Rongcai Zhao, Jianmin Pang, and Guoming Cai. 2008. Reconstructing Control Flow in Modulo Scheduled Loops. In *Seventh IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008)*. IEEE, Portland, OR, 539–544. https://doi.org/10.1109/ICIS.2008.16

[35] Zhengrong Wang and Tony Nowatzki. 2019. Stream-Based Memory Access Specialization for General Purpose Processors. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 736–749. https://doi.org/10.1145/3307650.3322229

[36] Nancy J. Warter, Scott A. Mahlke, Wen-Mei W. Hwu, and B. Ramakrishna Rau. 1993. Reverse If-Conversion. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 290–299. https://doi.org/10.1145/155090.155118

[37] Baofen Yuan, Jianfeng Zhu, Xingchen Man, Zijiao Ma, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2022. Dynamic-II Pipeline: Compiling Loops With Irregular Branches on Static-Scheduling CGRA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 9 (2022), 2929–2942. https://doi.org/10.1109/TCAD.2021.3121346

[38] Han-saem Yun, Jihong Kim, and Soo-mook Moon. 2001. A First Step Towards Time Optimal Software Pipelining of Loops with Control Flows. In *Proceedings of the 10th International Conference on Compiler Construction*. Springer-Verlag, Berlin, Heidelberg, Genove, Italy. https://doi.org/10.1007/3-540-45306-7_13

[39] Han-Saem Yun, Jihong Kim, and Soo-Mook Moon. 2002. Optimal Software Pipelining of Loops with Control Flows. In *Proceedings of the 16th International Conference on Supercomputing* (New York, New York, USA) *(ICS '02)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/514191.514210

[40] Eric Zimmerman. 2005. *Profile-directed If-Conversion in Superscalar Microprocessors*. Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign. https://llvm.org/pubs/2005-07-ZimmermanMSThesis.html

Received 13-NOV-2023; accepted 2023-12-23