Early evaluation of branches via decoupled access-execute to enable super-block optimizations

Andreas Scherman
Abstract

Early evaluation of branches via decoupled access-execute to enable super-block optimizations

Andreas Scherman

Modern CPUs rely on expensive branch predictors to speed up execution. Predictions nevertheless imply speculation, which is inherently costly, as mispredictions and re-execution of instructions can not only slow down execution but require extra energy expenditure.

From the compiler perspective, the presence of branches complicates static analysis and hinders compile time optimizations.

This work evaluates a software-only technique to remove branches and build super-blocks, thus, enabling more powerful compile-time optimizations, without the hardware support for dynamic branch prediction. Our approach eliminates branches and builds larger basic blocks using the Decoupled Access-Execute approach. Selected branches are hoisted and evaluated early in a so-called Access phase. If all branches are taken (or not-taken respectively), a simplified version of the code is run where these branches have been safely removed. Otherwise, the original version of the code is run.

The end goal of this transformation is to enable optimizations on the simplified version. In the frame of this thesis, we have evaluated the benchmarks without enabling any additional optimizations and observed performance improvements in two out of eight benchmarks and performance penalties ranging between 4% to 27% on the remaining six. Based on these promising results, we expect that the optimizations triggered on the super-blocks to hide the small overhead and lead to significant performance improvements.
1 Introduction

To deliver a high performance, modern hardware has an increasing amount of complexity adding features spanning from the basic branch predictor, hardware prefetcher to support out-of-order execution. In modern CPUs, one of the most power hungry components is the out-of-order execution engine[1], making it a prime target of improving energy efficiency.

The out-of-order execution engine relies on predictions, speculation and patterns over which instructions to run, and as such there exists the case of mispredictions happening which can be very costly, as this requires re-execution of instructions. This work aims to reduce the mispredictions and re-execution of instructions using compiler support, which in turn will lead to an improved run-time of the program as well as a more energy efficient execution.

1.1 Goals

The purpose of this work is to build compiler passes that analyze the code and use the outcome of the analysis to transform conditional branches to unconditional branches, allowing the pass the merge the two basic blocks to a single, larger basic block. This enables a more advanced instruction scheduling that exposes memory level parallelism and instruction level parallelism, although this will not be taken advantage of in this work.
1.2 Basic idea

As mentioned in Section 1.1, the idea is to build a compiler pass which transforms a code piece into another code piece. For this thesis, the code piece before the transformation will be referred to as CAE (Coupled Access Execute), while the code after the transformation will be referred to as DAE (Decoupled Access Execute).

In the most basic form, we can imagine the idea as being:

1. Access phase: check if we can run the optimized version of our code
2. If possible, run the optimized version of our code, which is referred to as the optimized phase. If we cannot run the optimized version of our code, run the original version, which is referred to as the execute phase.

For this thesis, the optimized version of our code means that we have taken a conditional branch and transformed it to an unconditional branch. This can lead to basic blocks being merged into larger basic blocks, which in turn enables more advanced instruction scheduling. The larger basic blocks enable more advanced instruction scheduling as there are more instructions to do reordering over, and having more instructions to reorder means we can have a greater instruction level parallelism. However, this work does not perform any additional instruction scheduling over the larger basic block, but rather relies on hardware and other optimizations through the -O3 compilation flag.

As the semantics of the code change when transforming a conditional branch to an unconditional branch, we need the access phase to make sure that the branch edge reduced is actually taken in order to make sure we do not alter the meaning of the code. In order to make meaningful and useful merges, static branch prediction will be used.
2 Background information

2.1 Out-of-order execution

As most modern-day CPUs use pipelines for instruction-level parallelism, a paradigm for maximizing the efficiency of the parallelism is to have out-of-order execution. Out-of-order execution is when a CPU executes instructions by their input data and requirements, rather than by their order. This means that the CPU can process instructions which are independent of each other at the same time. The out-of-order execution engine is responsible for finding instructions that can be processed simultaneously. However, an issue that can occur with out-of-order execution is hazards. As the instructions are executed in parallel, very little in regards to dependencies are taken, and there’s nothing guaranteeing that the instructions do not conflict. It’s expensive to resolve these conflicts between instructions, and it’s as such favorable to work to avoid them happening.

2.2 Modern compiler design

Most modern compilers are designed in three phases, the frontend, the optimization step, and the backend. The idea is that the frontend parses the source code, checks for errors and builds an abstract syntax tree of the source code. The optimization step analyzes the code and does transformations of it to make it more efficient - such as removing redundant code, simplifying algebraic expressions et cetera. The backend then generates the actual code for the target instruction set and does the architecture-dependent optimizations as well.

Looking at LLVM[2], we can see an advantage to this three-phase design. LLVM works in the optimization and backend step, which means if the frontend gives the same intermediate representation to the LLVM optimizer - we can do the same optimizations independent of the source code language. The optimization step is divided into compiler passes, where each pass does a specific type of transformation on the code. A pass can be visualized as a function which takes the intermediate representation of the code as input, and does a transformation on it, and then passes the transformed code as input to the next pass. After sufficient passes have been run through, the code is then passed on to the backend for machine code generation.

2.3 Control Flow Graph

Each program can be broken down into a control flow graph, where each node of the graph is a basic block, and the edges out from a node represent the branch. Each basic block can be seen as a piece of code which does calculations but does not have any change in control flow. This means that a basic block has one entry point and one exit point.

The pseudo-code in Listing 1 can be represented in terms of basic blocks as Figure 1, where the if-statement is located in the for.body block, and where T represents the branch where the if-statement is true, and the F represents false. The y++ instruction is located in the if.then block. The x++ instruction is located in the for.inc block. The check for every loop iteration, if $x < 5$, is located at the for.cond block.
Listing 1: Code snippet for control flow graph example

```c
for(x=0; x < 5; x++) {
    x = y;
    if (y == 1) {
        y++;
    }
}
```

Figure 1: Resulting CFG of the code snippet in Listing 1

2.4 CFG Skeleton

Listing 2: Code snippet for CFG Skeleton example

```c
for (x = 0; x < 5; x++) {
    t = t*x;
    if (x == 3) {
        x++;
    }
}
```

The CFG skeleton is the set of instructions which can modify which branch is taken. For example, looking at the code in Listing 2, the instruction \( t = t \times x \) has no impact of which branch is taken inside the loop (under the assumption \( t \neq x \)), and is as such not part of the CFG skeleton, and can as such safely be removed with the exact same branches being taken. Generating this CFG skeleton set is not trivial on bigger applications, as it requires alias analysis over the pointers. In the LLVM implementation of this thesis, the code to generate the CFG skeleton set was developed by UART (Uppsala Architecture Research Team).
2.5 Decoupled Access-Execute Model

The decoupled access-execute model means that the data access instructions have been decoupled from the computational instructions. This means that the access phase is memory-bounded, while the execute phase is compute-bounded. This is advantageous as we can run the access phase to prefetch data at a lower frequency to preserve energy without sacrificing run-time.

In practice, this means that we split a targeted loop into two phases: the access phase and the execute phase, as can be visualized in Figure 2. The access phase will as such be free from side-effects by removing all computational instructions and only keep the prefetch instructions, while the execute phase does the computational work and is kept in its original form.

![Figure 2: Transformation of the target function from the coupled access-execute model to the decoupled access-execute model](image-url)
3 Extending the decoupled-access execute model for eliminating branches

3.1 Static Branch Prediction

Static branch prediction is a form of branch prediction which does not consider any information about the execution of the code – but rather it predicts the result of the branch instruction through its predicate and destination block. This work uses static branch prediction for finding targeted branches, which in this work means that the branch has a greater probability than a certain threshold of being taken. These targeted branches will then be the basis for transforming the conditional branch to an unconditional branch.

3.2 Annotation of branches

Annotation is a way of attaching data to instructions from inside a compiler pass. This means that we can carry information through from one compiler pass to another. In this work, we annotate each branch with the corresponding branch probability, which means that we can then get the branch probability for each edge in a future pass without recalculations.

3.3 Loop Chunking

This work targets branches which are located inside of loops (referred to as a targeted loop). This means that a prediction for a branch is done over \( n \) iterations rather than a single one. In order to do this, we have a loop inside the access phase as well, which checks whether our branch prediction was correct for \( n \) loop iterations. If a single of these \( n \) loop iterations turn out to be a misprediction, we need to run the execute function for \( n \) iterations. Likewise, if the \( n \) loop iterations turns out to be correct, we can run the optimized function for \( n \) iterations.

This \( n \) value will be referred to as the chunked loops granularity. By default, the granularity value would be equivalent to the number of loop iterations, but this can in most cases be meaningless. It is improbable to have a branch which always takes the same control-flow path for each and every loop iteration. Likewise, if the branch which controls whether to exit the loop or not is a targeted branch, it will always fail in the case of the last iteration - and we will always have to run the execute function rather than the optimized.

The way of getting around this issue of predicting the entire loop is the chunking mechanism. The chunking mechanism means that the loop body is executed in chunks, such that we run it for iterations 0 to 9, then 10 to 19, etc. This allows us to check the loop for a granularity amount of times in the access phase to see whether we predicted the branches correctly - if we did, run the optimized function for granularity iterations, otherwise, run the execute function for granularity iterations. This control-flow is visualized in Figure 3.

As can probably be thought of, deciding this granularity value is very difficult, as the optimal granularity value can be different for every loop and every branch.
3.4 Generating a new access function

As mentioned in Section 2.5, the access function in the original version of the decoupled access-execute model worked as a prefetcher for the execute phase. It was generated through removing all arithmetical instructions.

The access function for this work has a new responsibility: keeping track of whether our branch prediction was correct for granularity iterations. In fact, the access function can be thought of a selector for either the optimized or execute function by making sure that the prediction for the targeted branches was correct in order to safely run the optimized code, as in the case of a misprediction the execute function has to be run in order to not change the semantics of the program. This usage, on the other hand, does require arithmetical instructions, although only those part of the CFG Skeleton (Section 2.4), as we need to calculate the path through the control-flow graph for the program.

3.5 Adding the optimized version

The optimized function will be a clone of the target loop, with the difference that each conditional branch with a high probability of being taken (referred to as targeted branch) will be reduced to an unconditional branch to the basic block with the highest probability. This reduction can then be further carried out by then merging the unconditional jump into a single, larger basic block if there are no other incoming branches to the destination block. This new basic block will then be the basis for enabling the possibility of having a greater instruction level parallelism, as there are more instructions for the out-of-order execution engine to pick and instruction scheduling can be done to a further extent.

3.6 Selecting between the execute and optimized function

In order to check whether the prediction is correct, we introduce a boolean variable called the branch_cond which is the flag for keeping track whether our prediction has been correct so far. This flag will then be the basis for deciding whether to run the execute or optimized phase. The flag will be initialized to true in the beginning, and then for every targeted branch in the access phase we will check whether the predicate is correctly predicted and update the flag accordingly – set it to false in case of a misprediction, or keep it true if correctly
predicted. The `branch_cond` flag will then be used after exiting the access phase to select whether to run the execute or optimized phase, as illustrated in Listing 3.

Listing 3: An example of the selection mechanism, where function `original` refers to the original loop and function `selection` is after the selection mechanism has been added. Assume that `if a[i] != 0` is a targeted branch.

```plaintext
fun original:
    for i in 0..1000:
        if a[i] != 0:
            a[i] = b[i] + ptr->field

fun selection:
    branch_cond = true
    for i in 0..1000:
        branch_cond = and(branch_cond, a[i] != 0)

    # selection mechanism
    if branch_cond == true:
        # optimized phase
        run optimized_function()
    else:
        # execute phase
        run execute_function()
```

3.7 Compilation flow

Knowing each of these mechanisms, we can start imagining how to piece them together in a compilation flow. The three different pieces we have are the annotation of branches, the chunking of loops and the transformation from a regular loop into a decoupled access-execute loop with an optimized phase.

It makes very little sense to annotate the branches after having chunked the loops, as that would entail that the branches which occur from having chunked the loop could be targeted, which is something we want to refrain from doing.

As the transformation to the decoupled access-execute loop with an optimized phase depends on having both the chunked loop ready, as well as the annotation of branches, it is the last step of the compilation flow, giving us the following final order:

1. Annotation of branches
2. Loop chunking
3. Transformation by decoupling the original loop

3.8 Putting the pieces together

Knowing in which order to apply each mechanism, we can start listing the transformation each pass does. Given the original function in Listing 4, we can incrementally add each of our compiler passes to show the difference for each step.
Listing 4: An unmodified function original for which to transform in to a de-coupled access-execute function

```haskell
fun original:
  for i in 0..1000:
    if a[i] > 0:
      a[i] = 0
    if a[i] != 0:
      a[i] = b[i] + ptr->field
  return
```

3.8.1 Annotation of branches

Annotating each branch with the corresponding branch probability does not alter the semantics or syntax of the source code in any way, apart from adding the meta-data to the instruction. For Listing 5, assume that the if $a[i] > 0$ statement is not targeted, as that is assumed to be a 50%-50% branch, while the if $a[i] != 0$ is targeted with the probability 90%-10%. Likewise, the exit-branch of the loop is targeted.

Listing 5: Adding annotation of each branch to function original

```haskell
fun original:
  for i in 0..1000: #90%-10%
    if a[i] > 0: #50%-50%
      a[i] = 0
    if a[i] != 0: #90%-10%
      a[i] = b[i] + ptr->field
  return
```

3.8.2 Applying Loop chunking

Further adding the loop chunking pass does require additional transformation as can be viewed in Listing 6. In particular, we need to introduce a new variable $v_i$ which works as a counter for how far we have gotten for each chunk, while we let the variable $i$ work as before. That is, each chunk can be viewed as $v_i$ counting up to $sup$, and then $sup$ is incremented by granularity, and the counting repeats.
Listing 6: Function *original* after both the annotation pass and loop chunking mechanisms have been applied.

```kotlin
fun original:
    sup = granularity
    vi = 0
    i = 0
    while true:
        if (vi < sup):
            if (i < 1000): #90%-10%
                if a[i] > 0: #50%-50%
                    a[i] = 0
                if a[i] != 0: #90%-10%
                    a[i] = b[i] + ptr->field
                    i++
                    vi++
            else:
                return
        else:
            vi = sup
            sup = sup + granularity
```

3.8.3 Transformation by decoupling the original loop

After having applied both the annotation and the loop chunking, the source code of adding the decoupling of the loop can be seen in Listing 7. The first thing to notice is that we have created copies of iterator variables \(i\) and \(vi\) (\(vi_c\) and \(i_c\)) (Line 5-8 in Listing 7). This stems from an important concept of the access phase: we require the access phase to be *side-effect free*. This can be thought of as that the access phase executes a chunk, then either the execute or optimized phase *repeats* the same iteration and does the actual calculations.

From a top-down perspective, we can see that the access phase stretches from line 10 to 25 in Listing 7 where line 11 contains the initialization of the flag for checking whether our prediction has been correct so far. Line 14-17 contains the check and update of the flag for whether the branch was correctly predicted. Note that the branch which was not targeted (\(a[i_c] > 0\)) is removed as it is not targeted, nor does it alter the flow of the control-flow graph (no instruction part of the CFG Skeleton).

In the optimized phase (line 26-32 in Listing 7), note that there is no check if \(i < 1000\), as the optimized phase executes *granularity* iterations regardless. Note that this means that the last iteration of a targeted loop is almost always run in the execute phase. That is, if a loop has 10 iterations, and granularity is 8, the last two iterations will always be run in the execute phase.
Listing 7: Function original after the branches have been annotated, the loop has been chunked and decoupled.

granularity = 10
fun original:
  sup = granularity
  vi = 0
  vi_c = 0
  i = 0
  i_c = 0
  while true:
    # Access phase
    branch_flag = true
    while (vi < sup):
      if (i < 1000):
        # Branch for (a[i] != 0) > 50%
        branch_flag = and(branch_flag, a[i] != 0)
      # Branch for (i < 1000) > 50%
      branch_flag = and(branch_flag, i < 1000)
      i++
      vi++
    else:
      # If we can't run granularity steps, we have to run the
      # unoptimized code (as the optimized runs granularity steps
      # regardless)
      branch_flag = false
      vi = sup
    # Check if we predicted correct
    if (branch_flag):
      # Run optimized code
      while (vi_c < sup):
        if a[i_c] > 0: #50%-50%
          a[i_c] = 0
          a[i_c] = b[i_c] + ptr->field
          i_c++
          vi_c++
        else:
          # Run original function
          while (vi_c < sup):
            if (i_c < 1000): #90%-10%
              if a[i_c] > 0: #50%-50%
                a[i_c] = 0
              if a[i_c] != 0: #90%-10%
                a[i_c] = b[i_c] + ptr->field
                i_c++
                vi_c++
            else:
              return
      vi_c = sup
  sup = sup + granularity
A visualization in the form of a control-flow graph of how the final transformation of a simple loop will look can be seen in Figure 4. We have that the subscript $f$ represents a function, $B$ represents a block of code and $L$ represents a loop. $\text{Access}_L$ will serve for making sure that our prediction was correct and that the optimized phase can be safely run. The execute function is the same as $\text{Target}_L$. The optimized loop $\text{Optimized}_L$ is the same function as the $\text{Target}_L$, except with conditional branches reduced.

![Diagram](image)

Figure 4: Transformation of the target function from the coupled access-execute model to the decoupled access-execute model
4 Methodology

The implementation has been done in C++ using the compiler infrastructure LLVM [2]. The work will be evaluated by comparing the performance in terms of run-time of the transformed code with the original code.

There will be three functions which will need to be generated, the execute function, the optimized function, as well as the access function.

4.1 Structure of the compiler passes

If we let each node represent a pass, we can design the compiler pass to have the same formation as Figure 5. Note that in the implementation we structure the annotation of branches after the loop chunking – this works due to the annotation leaving the outer loop (the chunking loop) untargeted.

Annotate every branch instruction with the probability

Create the access, optimized and execute function.
Structure the control flow and change the functions accordingly.

Figure 5: Structure of the compiler passes

4.1.1 Loop Chunking

Each targeted loop in the targeted function will be chunked in order to create the granularized loop.

The LLVM implementation for the loop chunking pass was developed in UART (Uppsala Architecture Research Team).

4.1.2 Annotating the branches

LLVM features a module for attaching metadata to the intermediate representation. This allows static data which have been attached to instructions to be carried through passes. This is used in the branch annotation pass as a way of attaching the branch instruction with the corresponding probability for each of the two edges. Annotating the probability is done such that the probability can later be used by another pass to generate the optimized function. In order to get the branch probability, static branch prediction was used, as described in Section 4.2. Note that the annotation needs only to be done on the targeted function, as that is the only function which uses the branch probability information.

4.1.3 Creating the access function

In order to generate the access function, we make a clone of the target loop and reduce the cloned loop to only contain instructions which are part of the CFG skeleton, as described in Section 2.4.
4.1.4 Creating the execute function

The execute function is generated by cloning the target loop. No other transformation is required, as the execute function should be identical to the targeted loop.

4.1.5 Creating the optimized function

Pseudo-code for generating the optimized function can be seen in Listing 8, where the \textit{prob0} member of branch variable refers to the probability value of the first edge leading to another basic block. Likewise, the \textit{prob1} member refers to the second outgoing edge.

In order to generate the optimized function, we first make a clone of the target loop (Line 7 in Listing 8) and iterate over each basic block in the cloned targeted loop and use the metadata attached to the branches to see if the terminator branch has two successors, where neither branch has a 100% of being taken, nor both 50% (line 11 in Listing 8). If one of them have a higher probability than a certain threshold parameter, find which one has the higher probability (line 12-19 in Listing 8). If neither branch probability matches the threshold for reducing the branch, continue iterating (line 12-13 in Listing 8). Knowing that we have a branch with a probability above the probability threshold which is not trivial, we can then remove the edge with the lower probability of being taken (line 22 in Listing 8). We can then replace the conditional jump with an unconditional jump to the branch with the highest probability (line 20 and line 23 in Listing 8).

As we are guaranteed to have checked whether this branch will be taken in the access phase, no change in semantics of the running code will happen. This means we can let the compiler merge the two blocks through \texttt{-O3} if there are no other basic blocks pointing to the successor. This larger basic block will then be the basis for more advanced optimizations.

As LLVM works with \textit{static single assignment} form, the \texttt{phi} instruction in the basic block for which the branch with the least probability points to has to remove the in-value from that branch as the branch no longer exists.

The algorithm for generating the optimized function can be seen in Listing 8.
**Listing 8: Algorithm for generating the optimized function**

```python
THRESHOLD = 0.5

for each function f:
    # Check if function is a targeted unoptimized function
    if (not shouldOptimize(f)):
        return;
    cloned_f = clone(f)
    for each block in cloned_f:
        branch = get_terminator(block)
        # Unequality to avoid 50%-50%. Larger than 0 to avoid 100%-0%.
        if (branch.prob0 != branch.prob1 && branch.prob0 > 0 && branch.prob1 > 0):
            if (branch.prob0 <= THRESHOLD && branch.prob1 <= THRESHOLD):
                continue
            if (branch.prob0 > THRESHOLD):
                dst = branch.successor(0)
                comp = branch.successor(1)
            else:
                dst = branch.successor(1)
                comp = branch.successor(0)
            uncond_branch = init_uncond_branch(dst)
            # note that remove_predecessor needs to clean up the phi-instruction
            remove_predecessor(complement, block);
            swapInstruction(branch, uncond_branch)
```

### 4.1.6 Structure of the generated code

The generated code can be grouped into two steps:

1. **Access phase (line 18-33 in Listing 9):**  Prefetch the branch conditions which have a probability >50% and check if they’re equal to true for the next granularity loop steps

2. **If the flag for our predictions is true (line 34 in Listing 9), i.e. we predicted correctly, run the optimized code (line 35-39 in Listing 9), otherwise run the standard loop (line 42-49 in Listing 9).**

After having chunked the loop, we further want to include the `branch_cond` flag to keep track of our prediction (line 24-26 in Listing 9), as well as the branch instruction to either the optimized or non-optimized function (line 34 in Listing 9). When looking at this through a high-level language view, the loop variable, as well as the virtual iterator, needs to be cloned in order to have a separate one for the access phase, as well as a separate one which is shared between the execute and optimized function. In practice, this was done through simply removing the store instructions, and as such letting the data reset through every iteration. Note that this still requires the `branch_cond` flag to be stored, as otherwise that one would have been lost as well.

For the example in Listing 9, the `vi` variable is the virtual iterator which is used as a counter for the chunking mechanism. Variable `sup` is the upper limit of that chunk, that which `vi` counts up to.
Listing 9: Code snippet of DAE transformation of function *original*, where the transformed code is function *dae*

```python
# original

granularity = 10

fun original:
    # Assume that the true branch for (i < 1000) > 50%
    for i in 0..1000:
        # Assume that the true branch for (a[i] != 0) > 50%
        if a[i] != 0:
            a[i] = b[i] + ptr->field

# dae

fun dae:
    sup = granularity
    vi = 0
    # Cloned virtual iterator
    vi_c = 0
    i = 0
    # Cloned loop counter
    i_c = 0
    while true:
        # Access phase
        branch_flag = true
        while (vi < sup):
            # Branch for (a[i] != 0) > 50%
            branch_flag = and(branch_flag, a[i] != 0)
            # Branch for (i < 1000) > 50%
            branch_flag = and(branch_flag, i < 1000)
            i++
            vi++
            else:
                # If we can't run granularity steps, we have to run the unoptimized code (as the optimized runs granularity steps regardless)
                branch_flag = false
                vi = sup
        # Check if we predicted correct
        if (branch_flag):
            # Run optimized code
            while (vi_c < sup):
                a[i_c] = b[i_c] + ptr->field
                i_c++
                vi_c++
            else:
                # Run original function
                while (vi_c < sup):
                    if (i_c < 1000):
                        if a[i_c] != 0:
                            a[i_c] = b[i_c] + ptr->field
                            i_c++
                            vi_c++
                    else:
                        return
                vi_c = sup
        sup = sup + granularity
```
4.1.7 Algorithm for generating the code

The following listed steps can be followed in order to transform the function $\text{original}$ to function $\text{dae}$ in Listing 9.

1. Preparing the access, execute and optimized functions

   (a) Find targeted function $\text{Target}_L$ and the targeted loop $\text{Access}_L$. Create two copies of $\text{Access}_L$ and let them have the name $\text{Execute}_L$ and $\text{Optimized}_L$.

   (b) Take loop $\text{Optimized}_L$ and apply the optimization transformation mentioned in Section 4.1.5.

   (c) Take the instruction set $S$ which contains all instructions which are not part of the CFG skeleton for loop $\text{Access}_L$ and for each instruction in the set replace all uses with NULL-like values and then remove the instruction from the basic block.

   (d) Create an allocation to a bool variable, with value $\text{true}$, which represents the branch cond and place it in the entry block of loop $\text{Access}_L$.

   (e) Iterate through each branch instruction $bi$ in loop $\text{Access}_L$. If $bi$ is a conditional branch with one edge having meta-data of above 50% chance of being taken, then the following two cases occur:

   - If the true-branch has $>50\%$ of being taken, insert an AND statement before the branch instruction with arguments $\text{branch\_cond}$ and the branch condition. Insert a store instruction before the branch instruction to store the result of the AND statement at $\text{branch\_cond}$.
   - If the false-branch has $>50\%$ of being taken, insert an NOT statement before the branch instruction with the branch condition as an argument, followed by an AND statement before the branch instruction with arguments $\text{branch\_cond}$ and the negated branch condition. Insert a store instruction to store the result of the AND statement at $\text{branch\_cond}$ before the branch instruction.

2. Updating the control flow graph

   (a) Let $\text{Chunking}_B$ be the exit block of loop $\text{Access}$ which contains the increment instructions to the virtual iterator used for the chunking mechanism.

   (b) Let $\text{Select}_B$ be an empty basic block.

   (c) Insert the empty basic block $\text{Select}_B$ before $\text{Chunking}_B$, that is, have all branches which point to $\text{Chunking}_B$ point to $\text{Select}_B$.

   (d) Insert a load instruction in $\text{Select}_B$ to the branch cond variable and a conditional branch instruction over the loaded value. If the value is true, change the control flow to basic block $\text{Optimized}_B$ and if it is false, change the control flow to basic block $\text{Execute}_B$.

   (e) Have each exit branch of the loop $\text{Optimized}_L$ point to $\text{Chunking}_B$.

   (f) Have each exit branch of the loop $\text{Execute}_L$ point to $\text{Chunking}_B$. 

23
Figure 6: Change in control-flow of transforming a chunked loop to the decoupled access-execute model

An intuitive way of understanding the updating of the control flow graph (2. in the algorithm), is that we transform the chunked targeted loop into the access phase, followed by inserting the selection block, optimized and execute phase between the new access phase and the chunking block. For further intuition, a visualization of how the control-flow graph changes can be observed in Figure 6. *Chunking* $B$ is the basic block which contains the instructions for incrementing the virtual iterator for the chunking mechanism to work. Inserting the block *Select* $B$, the loops *Optimized* $L$ and *Execute* $L$ between the access phase and the block for incrementing the virtual iterator means that we repeat the same iterations for both the access and computation phase as the access phase is free from side-effects.
4.2 Static Branch Prediction

In order to find the branch probability statically, the analysis module BranchProbabilityInfo\textsuperscript{3} from LLVM is used. This module is loosely based on the heuristics described by T. Ball and R. J. Larus\textsuperscript{4}.

This is a function analysis which provides information on the relative probabilities of each "edge" in the function's CFG where such an edge is defined by a pair (PredBlock and an index in the successors). The probability of an edge from one block is always relative to the probabilities of other edges from the block. The probabilities of all edges from a block sum to exactly one (100%). We use a pair (PredBlock and an index in the successors) to uniquely identify an edge, since we can have multiple edges from Src to Dst. As an example, we can have a switch which jumps to Dst with value 0 and value 10.\textsuperscript{3}
5 Evaluation environment

We evaluated our proposal on a selection of benchmarks from the SPEC CPU2006 Benchmark Suite, namely: milc, mcf, libQ, lbm, soplex, astar, h264ref, and bzip. The measurements were performed using the PAPI framework, which reports execution time. The benchmarks were run on a machine with the following specifications:

Intel® Core™ i7-2600K CPU with a frequency range from 1.6GHz to 3.4GHz. The machine has 16GB DDR3 RAM running at 1333MHz, and a cache hierarchy of 32 KB private (per core) L1 instruction and data caches, 256 KB private (per code) L2, and a shared 8192 KB L3.
6 Results

6.1 Static Branch Prediction

For each branch which has two successors where one of the edges has a probability above 50%, the probability of the more probable is plotted in Figure 7. Note that only the bigger probability is plotted - as the chart would become *inversely symmetric* if the lower ones were included as well - this can be visualized as thinking that for every branch in the 90% to 100%, there’s a corresponding one in the 0% to 10%. Note that this is for each and every branch in the entire benchmark.

The transformation was applied only to loops inside of functions which the program spent a majority of its execution time in. This profiling information was acquired from Tsinghua University[7]. In Table 1 we can see how many branches was targeted - in this case, this refers to the total count of branches which had an edge with a probability above 50% of being taken. That is, the branches which are considered *not targeted* in Table 1, each of its edges have a 50% probability of being taken. We can see the count of branches which was in each respective 10% interval in Table[1].

6.2 Timings

The timings in Figure 6.2 show the specific run-time for each benchmark for a specific granularity, for a specific branch probability threshold. Figure 9 shows how the best run-time for each bench normalized compares to its respective CAE counterpart. Note that for 462.libquantum, 464.h264ref and 433.milc no branch probability threshold higher than 50% was attempted, which can be explained by looking in Table 1 at the targeted branches in each interval - the >50% is equivalent to the >90% targeting.

Looking at the result, particularly in Figure 9 we can see that two out of eight benchmarks performed in terms of run-time better than their respective CAE counterpart.
Figure 7: Distribution of branch probabilities for benchmarks in 10% intervals

Table 1: Branch probability distribution for each benchmark, where the lower probability is non-inclusive and upper inclusive

<table>
<thead>
<tr>
<th>Bench</th>
<th>Loops</th>
<th>Total Branches</th>
<th>Targeted Branches</th>
<th>50%-60%</th>
<th>60%-70%</th>
<th>70%-80%</th>
<th>80%-90%</th>
<th>90%-100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>401 bzip2</td>
<td>11</td>
<td>77</td>
<td>20</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>420 mcf</td>
<td>1</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>421 ardr</td>
<td>2</td>
<td>56</td>
<td>18</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>451asha</td>
<td>6</td>
<td>66</td>
<td>15</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>470 lbm</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>464 libquantum</td>
<td>1</td>
<td>25</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>433 milc</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

28
Figure 8: Timings for each benchmark
Figure 9: The fastest time for each bench normalized to the respective CAE time.
6.3 Inspecting each benchmark

6.3.1 bzip2

The bzip2 benchmark was one of the benchmarks that performed better than the corresponding CAE code. It also had the most targeted branches (20), while also having the greatest granularity (8192). Note that out of these 20 targeted branches, 15 were located in the 90%-100% probability interval. This means the probability of entering the optimized phase was high, and with a high count of branches removed we get large basic blocks suitable for important -O3 optimizations as well.

6.3.2 mcf

The mcf benchmark performed better than the corresponding CAE code. The benchmark had two targeted branches - one in the >90% area, while the other below. We can see we hit a sweet spot when targeting two branches for the granularity 8. We can also see that for the >90% interval, we did not get the same speed-up in execution time compared to the >50% interval. This means that a single branch in the 50%-60% area made a big difference. In this case, when we find a granularity for which a hard to predict branch holds, the benefits are high: as not only are we helping the compiler with advanced optimizations, we are also helping the out-of-order execution engine to avoid mispredictions. Likewise, looking at >90% interval, as this branch is already very easy to predict, the out-of-order execution engine already predicts it very well, and not much benefit is added.

6.3.3 astar

When targeting all 18 branches, we can see that we spent very little time in the optimized phase. When targeting the ten branches with probability above 90%, we got a lot of more hits in the optimized phase - yet not a run-time performance improvement compared to CAE. For >50% branch threshold, we can see we get a steady faster execution time, although we do not enter the optimized phase. This is due to the natural overhead of the chunking mechanism which is lowered with a larger granularity.

6.3.4 soplex

The soplex benchmark had 15 targeted branches, with 7 branches having the probability >50% and ≤90%, and 8 branches in the >90% to ≤100% span. Little time was spent in the optimized phase and the performance was degraded for the soplex benchmark compared to the CAE version. Through inspection, it was observed that the targeted loops had very few iterations, which explains why the optimized phase was barely entered.

6.3.5 lbm

The lbm benchmark had only two targeted branches, and with both branches targeted the optimized phase was barely entered. With only one branch (>90%), the time spent in the optimized phase was greater than 90%. We can liken lbm to mcf, in the sense that the biggest difference between them was that for mcf
and >50%, we got a run-time improvement due to predicting correctly for an important branch. For the low probability branch in lbm, we could not find a prediction which held.

6.3.6 libquantum
The libquantum benchmark had one branch targeted. It spent the majority of time in the access phase, and spent rest of the time in the optimized phase, while barely any time at all in the execute phase. Through inspection, we have that the loop body is very small, and as such the access phase together with the chunking mechanism will be too big of an overhead to get a better execution time.

6.3.7 h264ref
The benchmark h264ref had 7 targeted branches, while each of these had a probability greater than 90%. However, even with the granularity as low as 2, the optimized phase was not entered. This is due to conflicting branches being targeted.

6.3.8 milc
The benchmark milc had one targeted branch, and it only entered the optimized phase when the granularity was set to 2. The targeted loop of milc has only three iterations, and the targeted branch was the exit branch to the loop, which is why the granularity of two worked, while three did not.

6.4 Further Analysis
With each benchmark analyzed, we can start drawing the main conclusions. The benchmarks libquantum and milc had only one branch targeted, which is the branch to exit the loop, which is already very simple to predict, and as such did not give us a speed up, even though the majority of time was spent in the optimized phase. Likewise, the benchmarks lbm and mcf had only two targeted branches, and while a lot of time was spent in the optimized phase, the performance in terms of run-time was still degraded. The idea of having few targeted branches and spending a lot of time in the optimized phase can be thought of as while we did spend a lot of time in the optimized phase, the optimized phase was very similar to the execute phase and will as such not yield a very good improvement.

The benchmarks with greater than two targeted branches are h264ref, soplex, astar and bzip2. Benchmark astar does show the greatest potential, as we have ten targeted branches with the majority of time spent in optimized phase even for higher granularities. However, even for astar the runtime performance did not exceed the CAE counterpart. Looking at the percentual part for each benchmarks access phase and astar: the access phase for astar is the second greatest, with only benchmark libquantum having a greater time spent in the access phase. Benchmark h264ref can be dismissed as the optimized phase is not entered. Benchmark soplex spends very little time in the optimized phase as well.
7 Discussion

When observing how well the DAE version performs compared to the CAE counterpart, we can observe that in terms of run-time and utilization of the optimized phase, there is a fine balance between having lots of targeted branches and the entering of the optimized phase. Intuitively, we can think of this as the more targeted branches we have, the less probable we are of entering the optimized phase. An example of this phenomenon where we had zero branches below 90% probability, and seven branches with a probability >90%, and yet the optimization phase was never entered, can be observed in Graph 464.h264ref (>50%). Likewise, if there are few targeted branches and a lot of the run-time is spent in the optimized phase the performance will still not be particularly impressive as the optimized phase is simply resembling the execute phase. This can be thought of as that the optimized phase is simply not very optimized. For an example of this phenomenon, see Graph 470.lbm (>90%).

Another observation to make is the importance of the access phase. When the access phase is small, running the execution phase barely brings a performance degradation. This can be seen in Graph 464.h264ref (>50%) and Graph 470.lbm (>50%), as while we did not enter the optimized phase at all or spent very little time there, the performance decrease in terms of run-time is very little. This means, in terms of potential, had the branch predictions for h264 and lbm been better and had we entered the optimized phase with a greater granularity value, those would have had the greatest potential of getting a lower run-time than the corresponding CAE counterpart.

One of the benchmarks which performed better than its corresponding CAE counterpart was the mcf benchmark, but only for when the granularity was 8. This showcases the potential advantage of this work in a fairly simple way - when we manage to find a granularity which holds for a branch which is difficult to predict, there is a great run-time improvement. We can think of this as not only did we allow more advanced optimizations through the larger basic block, but we also completely relieved the out-of-order execution engine of having to predict this branch. For more or less trivial branches such as the ones with a >90% probability of being taken, the out-of-order execution engine almost always predicts correctly, and as such we do not get as great run-time improvement from a hardware perspective.

Worth noting is as well how a couple of benchmarks, such as astar, performed greater as the granularity increased, even though the optimization phase was entered less and less. This has to do with how the loop chunking mechanism can cause a performance increase in and by itself as a low granularity value will bring overhead.
8 Related Work

8.1 Static Branch Prediction

A great deal of algorithms and methods for predicting which branch is taken statically uses heuristics - and is as such not very precise, but in most cases give a fairly good result. In Branch Prediction for Free [4], the heuristic for loop branches is if either of the outgoing edges is a back-edge, it is predicted. Intuitively, we can think of this is that a loop iterates many times and only exit once. For the non-loop branches, six simple heuristics are combined and chained: the opcode, loop, call, return, guard, and store heuristic.

- The opcode heuristic predicts that comparisons of an integer is less than or equal to zero is false.
- The loop heuristic predicts the successor that does not postdominate the branch, and the successor is a loop-header or pre-header.
- The call heuristic predicts the successor as "non-taken" that contains a call or unconditionally passes control to a block with a call that it dominates, and the successor does not postdominate the branch.
- The return heuristic predicts the successor which does not contain a return statement.
- The guard heuristic predicts the successor in which the register used in the branch instruction is used without being defined, and the successor block does not postdominate the branch.
- The store heuristic predicts the successor as "non-taken" if it contains a store instruction and the successor block does not postdominate the branch.
- The pointer heuristic predicts that pointer comparisons to NULL are false and that a comparison between two pointers is false.

The paper "Static Branch Frequency and Program Profile Analysis" [8] builds on these heuristics, and extends them to work with probabilities rather than a strict boolean - taken or not taken. It does so by using the Dempster-Shafer theory of evidence [9]. The main idea is that it takes into account what each of the chained heuristics results in, rather than just grouping them together. There have been implementations of these heuristics in earlier versions of LLVM [10].

A method which does not work with heuristics is the Value Range Propagation method [11]. The value range propagation method works by calculating the value range of each variable as it traverses each instruction, which essentially gives you the probability that the variable takes one of the values in the range. When finding a branch you can then easily calculate the probability that the condition is true. In general, this gives a better result than the heuristics - but as for loops, you have to do a backtrack in order to find the value ranges, which is more costly than the simple heuristics. In terms of code complexity, it is also a lot harder to implement.
8.2 Control-flow Decoupling

The paper Control-flow Decoupling [12] touches upon eliminating branch mispredictions as well. Much like this work, they split a single loop into two separate loops: the first containing only the branch’s predicate computation while the second one contains both the branch and its control-dependent instructions. One of the major differences is that the paper Control-flow Decoupling uses hardware support to have the first loop communicate the branch predicate outcomes to the second loop through an architectural queue. As the queue controls which path to take through the program, we can liken this queue to this work’s $branch_{cond}$ flag. The hardware becomes an important step in order to exploit the information gained in the first loop, as we can then pop the queue to resolve the branch during the fetch stage.

8.3 Inspector-executor model

To avoid having to make predictions statically, an approach to using run-time information would be through the inspector-executor model [13]. The inspector-executor model splits the code into two parts – one consisting of the inspector, running a lesser version of the code to find memory dependencies. When the executor then executes the code, information gained through the inspection phase can be used. In this work, this would entail using the information gained from evaluating the branch predicates.

8.4 Decoupled Access-Execute Model

As described in Section 2.5, the decoupled access-execute model [14] separates the data access from the arithmetic and computational work. The original decoupled access-execute model primarily focused on reducing energy while maintaining high performance through using prefetching. This work uses the decoupling to evaluate branches early and selecting whether to run the optimized or the original code.
9 Future Work

9.1 Memory dependencies of instructions in one basic block

An optimization for improving the execution time is to use more advanced instruction scheduling inside of the merged superblock. That is, intuitively we can think that there are more instructions to reorder in the larger basic block than the smaller one, allowing for greater instruction scheduling.

This reordering can be done through finding each load instruction, and categorizing them into the following sets:

- Requirements: instructions that must be executed before the load, i.e. that are required for computing the address of the load.

- Aliases: Other memory accesses that target the same memory address as the load. Since aliasing is not entirely solved at compile-time, the compiler can only mark the level of confidence - the likelihood of aliasing. This set can as such be broken down further into three subsets:
  - MustAlias: When the two memory objects are guaranteed to always start at exactly the same location, i.e. the pointers are equal.
  - MayAlias: When the two pointers might refer to the same object.
  - PartialAlias: When the two memory objects are known to be overlapping in some way, but do not start at the same address.

- Independent: Instructions that are not part of any of the previous set, and can as such be reordered independently.

From these sets we can then reorder the instructions without changing the semantics of the code.
9.2 Multiple optimized functions

The current implementations checks that if we predicted correctly for each branch for a certain amount of times (the number of times is referred to as the granularity). Ideally, we would like to be able to mispredict, and still run an optimized version. This is useful as having a greater amount of branches and a higher granularity value in the access phase implies greater speed up when running the optimized loop - but we also have a higher risk of not running the optimized function.

In order to run the optimized function although we mispredicted, could be done through creating a truth table for the predictions, and matching optimizations for each cell.

That is, given the function:

```plaintext
fun org:
  for i in 0..1000:
    if a[i] != 0:
      a[i] = b[i] + ptr->field
    if a[i] < 0:
      a[i] = 0
```

Assume that both if-statement branches are targeted. Note that the granularity is 2 - we check if the branches are correct for 2 iterations, and run the optimized version the same amount of times. The new idea would be to create a truth table on the form:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the first $B_1$ represent the first branch ($a[i] \neq 0$) and $B_2$ represent the second branch ($a[i] < 0$). As the granularity is 2, we get them repeated twice. Using this approach would require us to generate 64 optimized code snippets, which would increase exponentially depending on granularity and the number of branches. However, with the help of the static branch prediction and further analysis, the number of possibilities could be limited with a fair amount. Then we have an optimized version for each and every possible branch option. Essentially – this would completely remove the case of prediction from
the equation.

9.3 Loop unrolling

An idea would be to have a non-fixed granularity value, and prepare optimized versions of the loop unrolled. What this means is that we can prepare a granularity amount of loop clones, unrolled 0..granularity amount of times. This means if our prediction was only correct for 5 loop iterations, and our granularity value was 10 - our work was not lost, as we can run the optimized version for 5 loop iterations.

Likewise, the targeted loop in the optimized function could be unrolled as well for further instruction level parallelism.

References


