Code Generation for Branch Prediction: a review

Quinn Pham

You

What are the seminal academic papers that should be included in a literature review about code generation for branch prediction?

You

What are the seminal academic papers that should be included in a literature review about code generation for branch prediction?

ChatGPT

As of my last knowledge update in January 2022, there might be new research papers and developments in the field of code generation for branch prediction. However, I can provide you with a list of seminal academic papers up to that point that you might consider including in a literature review on this topic:

1. **"Two-Level Adaptive Training Branch Prediction"**

You

What about the paper: "Improving the accuracy of static branch prediction using branch correlation"?

You

What about the paper: "Improving the accuracy of static branch prediction using branch correlation"?

ChatGPT

As of my last knowledge update in January 2022, I don't have specific information about a paper titled "Improving the accuracy of static branch prediction using branch correlation." It's possible that the paper was published after my last update or that it may not be widely recognized in the academic community.

You

What about the paper: "Improving the accuracy of static branch prediction using branch correlation"?

Published in 1994

ChatGPT

As of my last knowledge update in January 2022, I don't have specific information about a paper titled "Improving the accuracy of static branch prediction using branch correlation." It's possible that the paper was published after my last update or that it may not be widely recognized in the academic community.

You

What about the paper: "Improving the accuracy of static branch prediction using branch correlation"?

Published in 1994

"Free Access" ACM article

ChatGPT

As of my last knowledge update in January 2022, I don't have specific information about a paper titled "Improving the accuracy of static branch prediction using branch correlation." It's possible that the paper was published after my last update or that it may not be widely recognized in the academic community.

Static Branch Prediction

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
  - Always predict taken

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
  - Always predict taken
  - Always predict not taken

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
  - Always predict taken
  - Always predict not taken
  - Predict taken for backward branches & not taken for forward branches

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
  - Always predict taken
  - Always predict not taken
  - Predict taken for backward branches & not taken for forward branches
  - Predict based on opcode of branch instructions

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
    - Always predict taken
    - Always predict not taken
    - Predict taken for backward branches & not taken for forward branches
    - Predict based on opcode of branch instructions
    - …

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
  - Always predict taken
  - Always predict not taken
  - Predict taken for backward branches & not taken for forward branches
  - Predict based on opcode of branch instructions
  - …
- Profile-based

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
  - Always predict taken
  - Always predict not taken
  - Predict taken for backward branches & not taken for forward branches
  - Predict based on opcode of branch instructions
  - …
- Profile-based
  - Prediction made based on profiling information

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
    - Always predict taken
    - Always predict not taken
    - Predict taken for backward branches & not taken for forward branches
    - Predict based on opcode of branch instructions
    - …
- Profile-based
    - Prediction made based on profiling information
- Program-based

Static Branch Prediction

- Predict the outcome of each branch in the program at compile-time
  - Always predict taken
  - Always predict not taken
  - Predict taken for backward branches & not taken for forward branches
  - Predict based on opcode of branch instructions
  - …
- Profile-based
  - Prediction made based on profiling information
- Program-based
  - Prediction made based on static analysis

Predicting Conditional Branch Directions From Previous Runs of a Program

Joseph A. Fisher and Stefan M. Freudenberger

Hewlett-Packard Laboratories
1501 Page Mill Rd. 3U-5
Palo Alto, CA 94304
jfisher@hpl.hp.com freuden@hpl.hp.com

Predicting Conditional Branch Directions From Previous Runs of a Program

Joseph A. Fisher and Stefan M. Freudenberger

Hewlett-Packard Laboratories
1501 Page Mill Rd. 3U-5
Palo Alto, CA 94304
jfisher@hpl.hp.com freuden@hpl.hp.com

Can we get accurate static branch prediction by feeding information about previous runs of a program to the compiler?

Fisher & Freudenberger (1992)

Fisher & Freudenberger (1992)

- First attempt at profile-based static branch prediction

Fisher & Freudenberger (1992)

- First attempt at profile-based static branch prediction
- Perform several profile runs of a program and record branch direction counts

Fisher & Freudenberger (1992)

- First attempt at profile-based static branch prediction
- Perform several profile runs of a program and record branch direction counts
    - For each branch, sum direction counts from profiles

Fisher & Freudenberger (1992)

- First attempt at profile-based static branch prediction
- Perform several profile runs of a program and record branch direction counts
  - For each branch, sum direction counts from profiles
  - Statically predict the most frequent direction for each branch

Fisher & Freudenberger (1992)

- First attempt at profile-based static branch prediction
- Perform several profile runs of a program and record branch direction counts
  - For each branch, sum direction counts from profiles
  - Statically predict the most frequent direction for each branch
- Empirically found that **programs are dominated by branches that go in one direction** with little variation between input workloads

Fisher & Freudenberger (1992)

- First attempt at profile-based static branch prediction
- Perform several profile runs of a program and record branch direction counts
  - For each branch, sum direction counts from profiles
  - Statically predict the most frequent direction for each branch
- Empirically found that **programs are dominated by branches that go in one direction** with little variation between input workloads
  - ie. branches that can be effectively predicted at compile-time

Improving the Accuracy of Static Branch Prediction Using Branch Correlation

Cliff Young and Michael D. Smith
Division of Applied Sciences
Harvard University, Cambridge, MA 02138

{cyoung, smith}@das.harvard.edu

Improving the Accuracy of Static Branch Prediction Using Branch Correlation

Cliff Young and Michael D. Smith
Division of Applied Sciences
Harvard University, Cambridge, MA 02138

{cyoung, smith}@das.harvard.edu

Improving the Accuracy of Static Branch Prediction Using Branch Correlation

Cliff Young and Michael D. Smith
Division of Applied Sciences
Harvard University, Cambridge, MA 02138

{cyoung, smith}@das.harvard.edu

Create a compiler transformation based on profile information that encodes the branch history information in the program counter via basic block duplication

Improving the Accuracy of Static Branch Prediction Using Branch Correlation

Cliff Young and Michael D. Smith
Division of Applied Sciences
Harvard University, Cambridge, MA 02138

{cyoung, smith}@das.harvard.edu

Create a compiler transformation based on profile information that encodes the branch history information in the program counter via basic block duplication

Exploit branch correlation without additional hardware like the global register and pattern history table

Improving the Accuracy of Static Branch Prediction Using Branch Correlation

Cliff Young and Michael D. Smith
Division of Applied Sciences
Harvard University, Cambridge, MA 02138

{cyoung, smith}@das.harvard.edu

Improving Semi-static Branch Prediction by Code Replication

Andreas Krall
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
andi@mips.complang.tuwien.ac.at

Create a compiler transformation based on profile information that encodes the branch history information in the program counter via basic block duplication

Exploit branch correlation without additional hardware like the global register and pattern history table

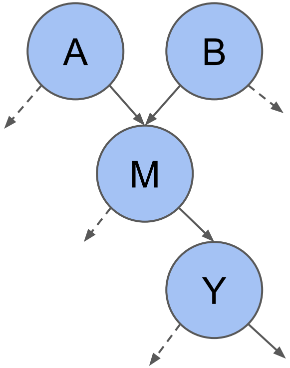
Young & Smith (1994)

Young & Smith (1994)

- Collect profile information with branch directions based on the basic block path to the branch

Young & Smith (1994)

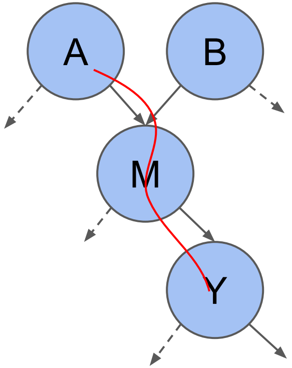
- Collect profile information with branch directions based on the basic block path to the branch



Young & Smith (1994)

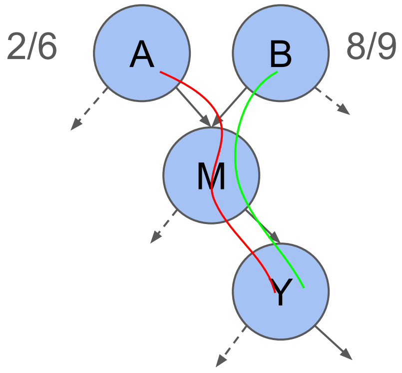
- Collect profile information with branch directions based on the basic block path to the branch

2/6



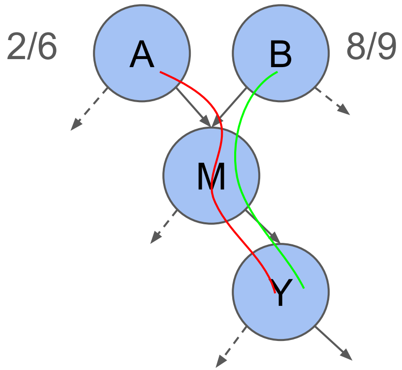
Young & Smith (1994)

- Collect profile information with branch directions based on the basic block path to the branch



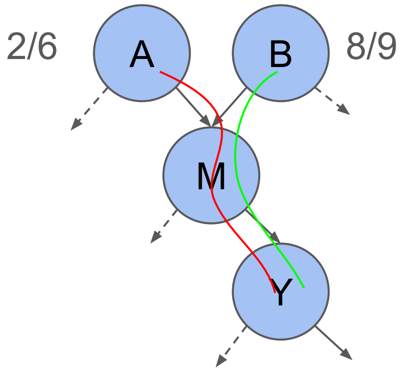
Young & Smith (1994)

- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history



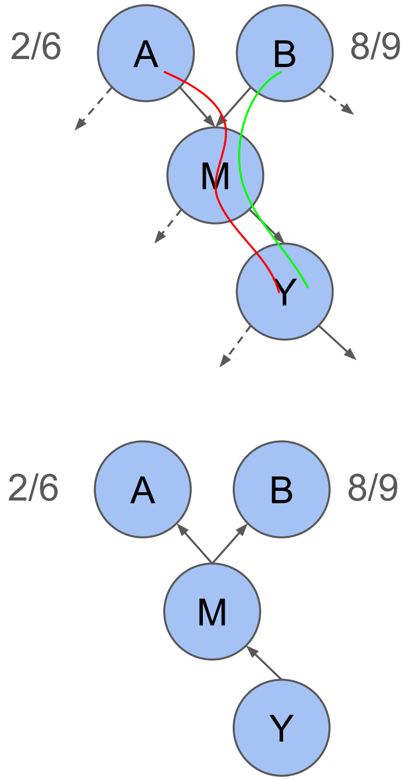
Young & Smith (1994)

- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch



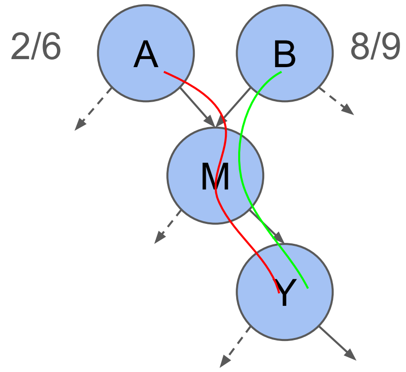
Young & Smith (1994)

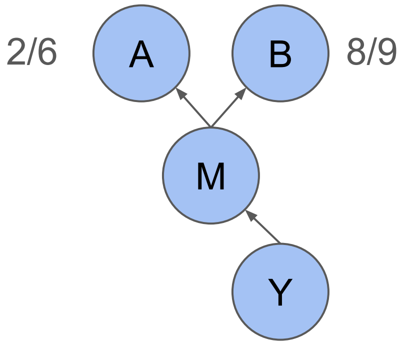
- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch



Young & Smith (1994)

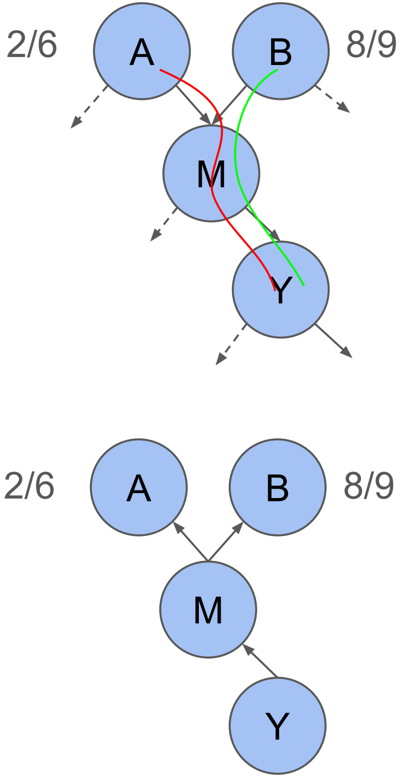
- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch
- Minimize number of paths in each branch history tree





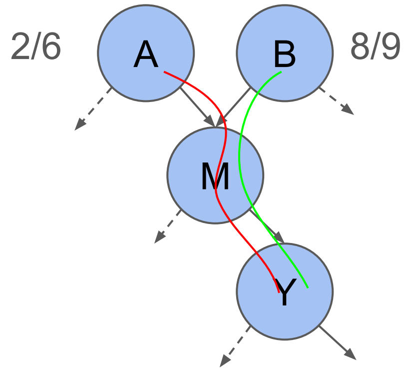
Young & Smith (1994)

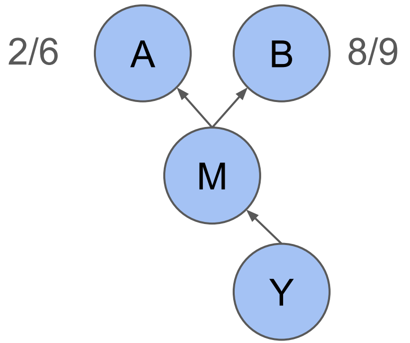
- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch
- Minimize number of paths in each branch history tree
  - Prune nodes while maximizing prediction accuracy



Young & Smith (1994)

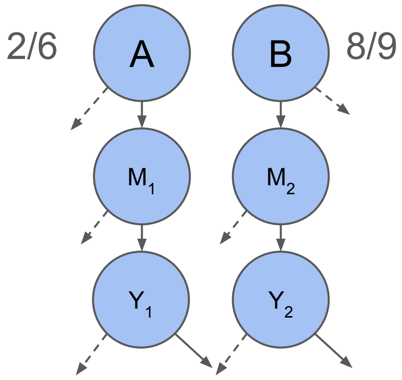
- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch
- Minimize number of paths in each branch history tree
  - Prune nodes while maximizing prediction accuracy
- Perform global reconciliation across all branch history trees to determine the basic block duplication required

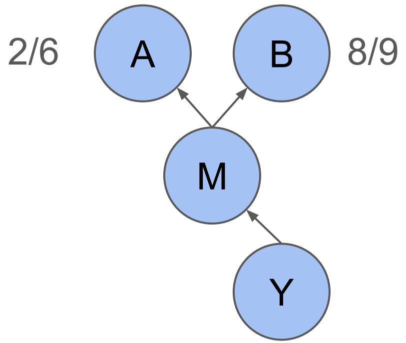




Young & Smith (1994)

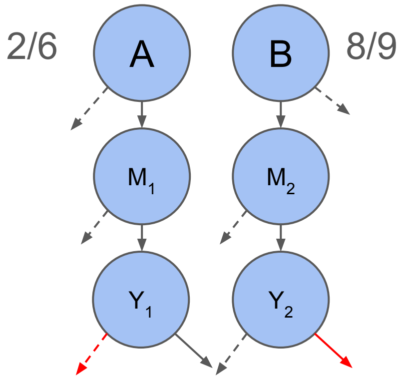
- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch
- Minimize number of paths in each branch history tree
  - Prune nodes while maximizing prediction accuracy
- Perform global reconciliation across all branch history trees to determine the basic block duplication required

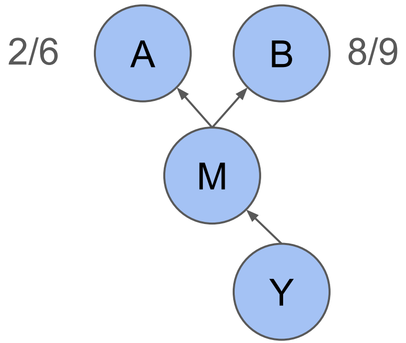




Young & Smith (1994)

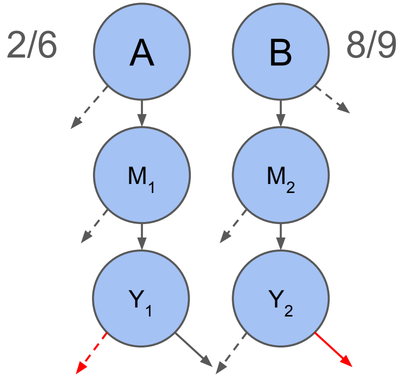
- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch
- Minimize number of paths in each branch history tree
  - Prune nodes while maximizing prediction accuracy
- Perform global reconciliation across all branch history trees to determine the basic block duplication required

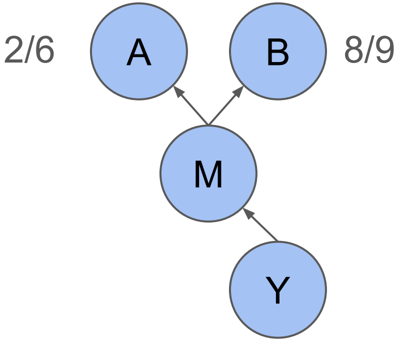




Young & Smith (1994)

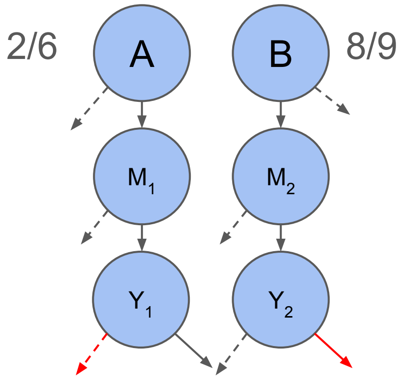
- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch
- Minimize number of paths in each branch history tree
  - Prune nodes while maximizing prediction accuracy
- Perform global reconciliation across all branch history trees to determine the basic block duplication required
  - Avoid exponential branching paths using compile-time heuristics

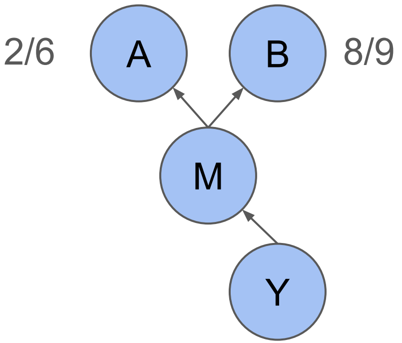




Young & Smith (1994)

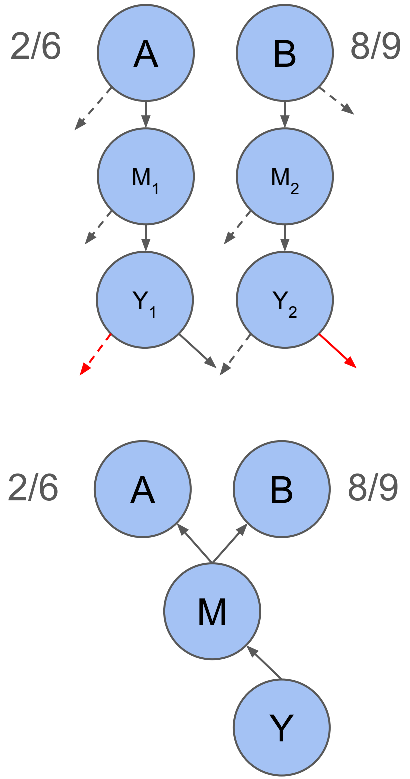
- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch
- Minimize number of paths in each branch history tree
  - Prune nodes while maximizing prediction accuracy
- Perform global reconciliation across all branch history trees to determine the basic block duplication required
  - Avoid exponential branching paths using compile-time heuristics
- Layout the transformed code





Young & Smith (1994)

- Collect profile information with branch directions based on the basic block path to the branch
  - Branch path history is more powerful than branch pattern history
  - Maintain a branch history tree for each branch
- Minimize number of paths in each branch history tree
  - Prune nodes while maximizing prediction accuracy
- Perform global reconciliation across all branch history trees to determine the basic block duplication required
  - Avoid exponential branching paths using compile-time heuristics
- Layout the transformed code
  - Avoid increasing the dynamic instruction count



Profile Guided Code Positioning

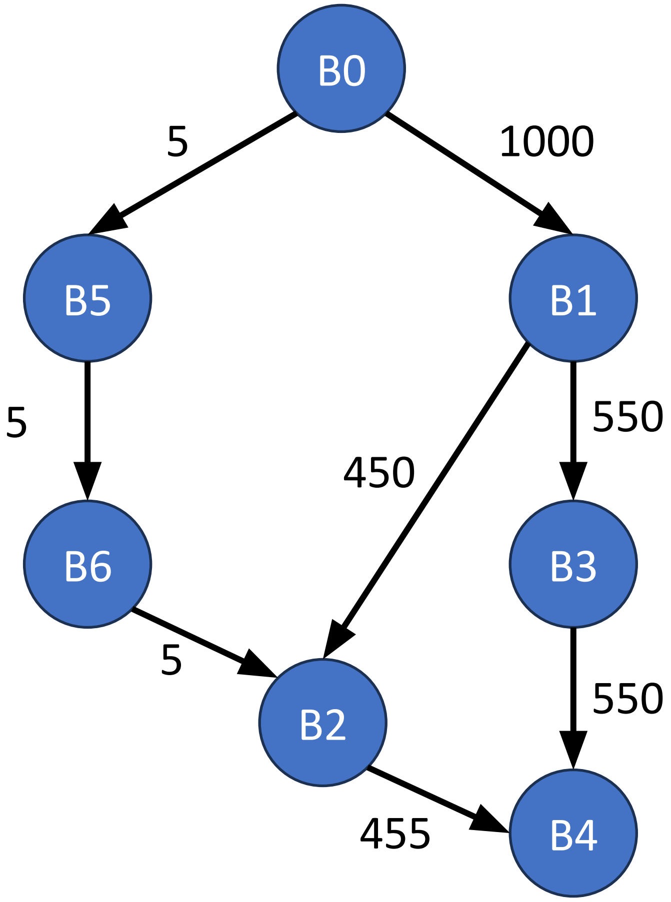
Karl Pettis and Robert C. Hansen

Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014

Profile Guided Code Positioning

Karl Pettis and Robert C. Hansen

Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014

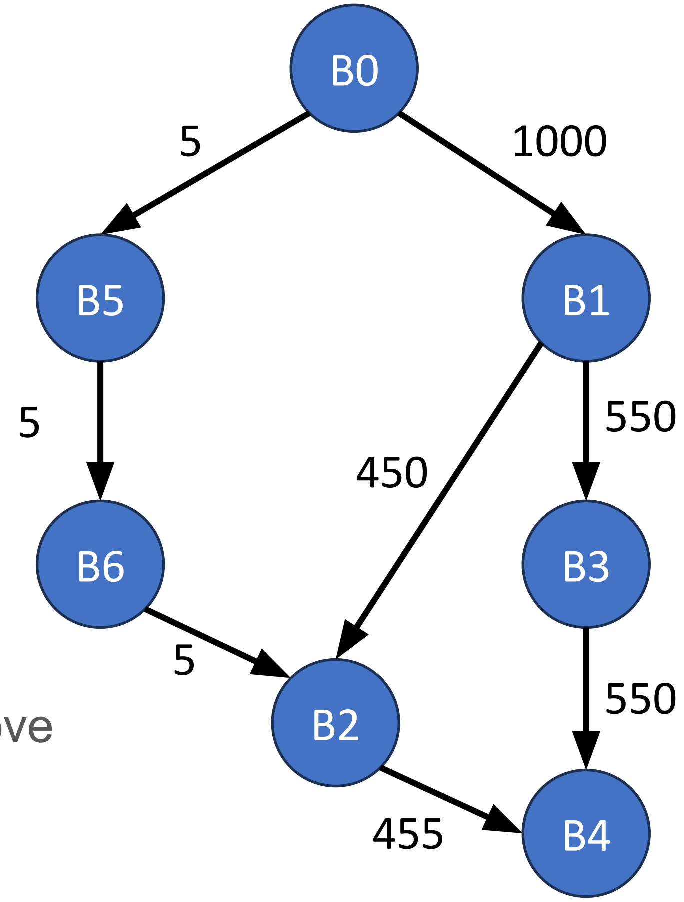


Profile Guided Code Positioning

Karl Pettis and Robert C. Hansen

Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014

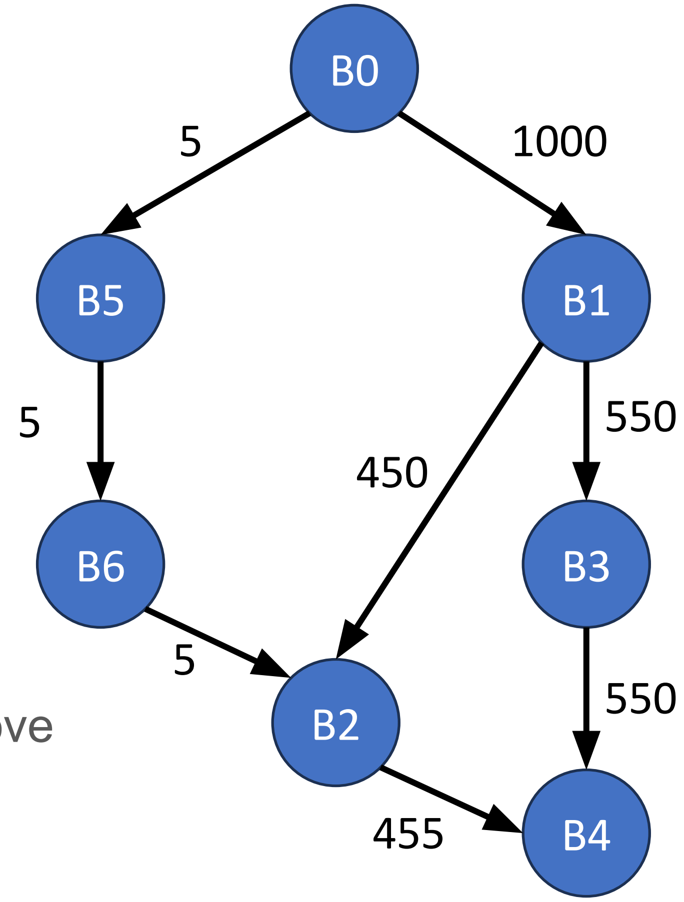
Reposition basic blocks using profile data to improve static branch prediction for a PA-RISC CPU



Profile Guided Code Positioning

Karl Pettis and Robert C. Hansen

Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014



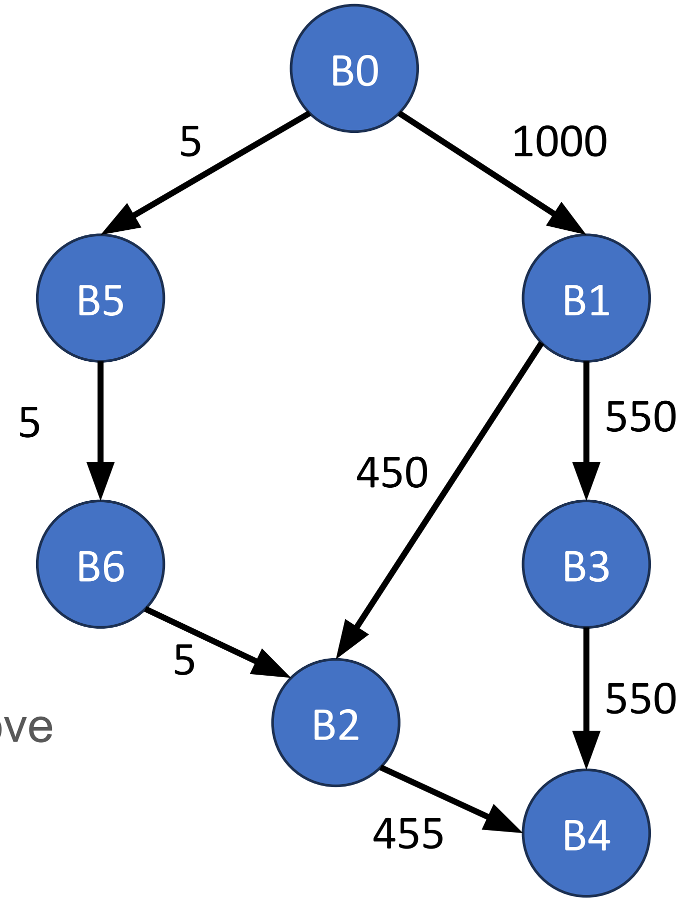
Reposition basic blocks using profile data to improve static branch prediction for a PA-RISC CPU

- Backward branches taken

Profile Guided Code Positioning

Karl Pettis and Robert C. Hansen

Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014



Reposition basic blocks using profile data to improve static branch prediction for a PA-RISC CPU

- Backward branches taken
- Forward branches not taken

Reducing the Cost of Branches

Scott McFarling and John Hennessy
Computer Systems Laboratory
Stanford University

Reducing the Cost of Branches

Scott McFarling and John Hennessy
Computer Systems Laboratory
Stanford University

Introduces the concept of the delayed branch

McFarling & Hennessey (1986)

Delayed branch

McFarling & Hennessey (1986)

Delayed branch

- Machine continues executing instructions after the branch until the condition is determined

McFarling & Hennessey (1986)

Delayed branch

- Machine continues executing instructions after the branch until the condition is determined
  - Not speculative

McFarling & Hennessey (1986)

Delayed branch

- Machine continues executing instructions after the branch until the condition is determined
    - Not speculative
- Compiler tries to schedule useful instructions into the “delay” slots following a branch instruction

McFarling & Hennessey (1986)

Delayed branch

- Machine continues executing instructions after the branch until the condition is determined
  - Not speculative
- Compiler tries to schedule useful instructions into the “delay” slots following a branch instruction
  - Before branch: branch condition must not depend on the instruction

McFarling & Hennessey (1986)

Delayed branch

- Machine continues executing instructions after the branch until the condition is determined
  - Not speculative
- Compiler tries to schedule useful instructions into the “delay” slots following a branch instruction
  - Before branch: branch condition must not depend on the instruction
    - Always beneficial

McFarling & Hennessey (1986)

Delayed branch

- Machine continues executing instructions after the branch until the condition is determined
  - Not speculative
- Compiler tries to schedule useful instructions into the “delay” slots following a branch instruction
  - Before branch: branch condition must not depend on the instruction
    - Always beneficial
  - After branch / From target: must be safe to execute the instruction whether the branch is taken or not

McFarling & Hennessey (1986)

Delayed branch

- Machine continues executing instructions after the branch until the condition is determined
  - Not speculative
- Compiler tries to schedule useful instructions into the “delay” slots following a branch instruction
  - Before branch: branch condition must not depend on the instruction
    - Always beneficial
  - After branch / From target: must be safe to execute the instruction whether the branch is taken or not
    - Beneficial if the instruction is along the correct path

McFarling & Hennessey (1986)

Delayed branch

- Machine continues executing instructions after the branch until the condition is determined
  - Not speculative
- Compiler tries to schedule useful instructions into the “delay” slots following a branch instruction
  - Before branch: branch condition must not depend on the instruction
    - Always beneficial
  - After branch / From target: must be safe to execute the instruction whether the branch is taken or not
    - Beneficial if the instruction is along the correct path
- Difficult to find safe instructions to fill the “delay” slots

McFarling & Hennessey (1986)

Delayed branch with squashing

McFarling & Hennessey (1986)

Delayed branch with squashing

- Use a bit to indicate the predicted direction

McFarling & Hennessey (1986)

Delayed branch with squashing

- Use a bit to indicate the predicted direction
  - Delay slots filled by instructions from that direction

McFarling & Hennessey (1986)

Delayed branch with squashing

- Use a bit to indicate the predicted direction
  - Delay slots filled by instructions from that direction
    - Squash when incorrect

McFarling & Hennessey (1986)

Delayed branch with squashing

- Use a bit to indicate the predicted direction
  - Delay slots filled by instructions from that direction
    - Squash when incorrect
- Use a bit to specify if squashing is needed on a misprediction

McFarling & Hennessey (1986)

Delayed branch with squashing

- Use a bit to indicate the predicted direction
  - Delay slots filled by instructions from that direction
    - Squash when incorrect
- Use a bit to specify if squashing is needed on a misprediction
  - Machine does not need to squash if the compiler can fill the delay slots with safe instructions

Branch Prediction For Free

THOMAS BALL
tom@cs.wisc.edu

JAMES R. LARUS
larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin – Madison

Branch Prediction For Free

THOMAS BALL
tom@cs.wisc.edu

JAMES R. LARUS
larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin – Madison

- Program-based static branch prediction

Branch Prediction For Free

THOMAS BALL
tom@cs.wisc.edu

JAMES R. LARUS
larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin – Madison

- Program-based static branch prediction
- Uses natural loop analysis to predict loop branches

Branch Prediction For Free

THOMAS BALL
tom@cs.wisc.edu

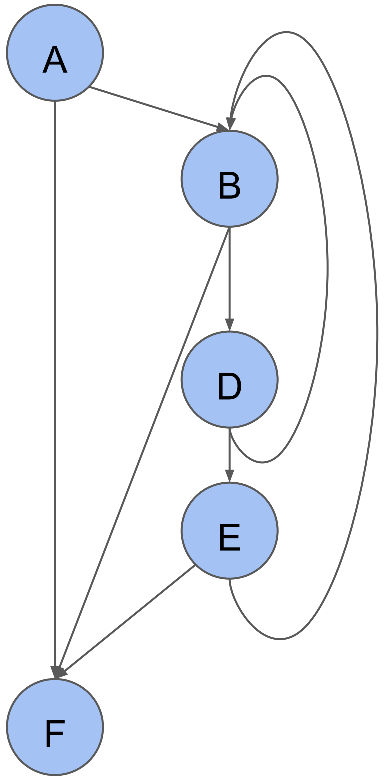
JAMES R. LARUS
larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin – Madison

- Program-based static branch prediction
- Uses natural loop analysis to predict loop branches
- Uses heuristics to predict non-loop branches

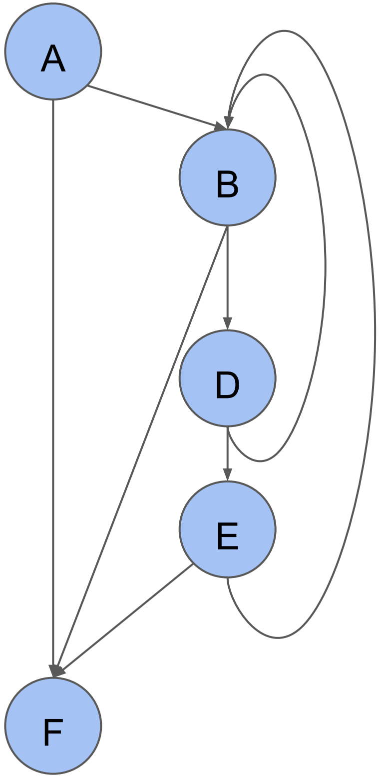
Ball & Larus (1993): Predicting loop branches

Ball & Larus (1993): Predicting loop branches



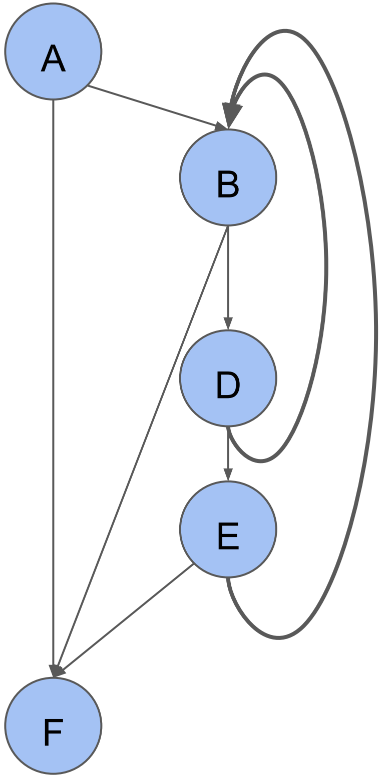
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG



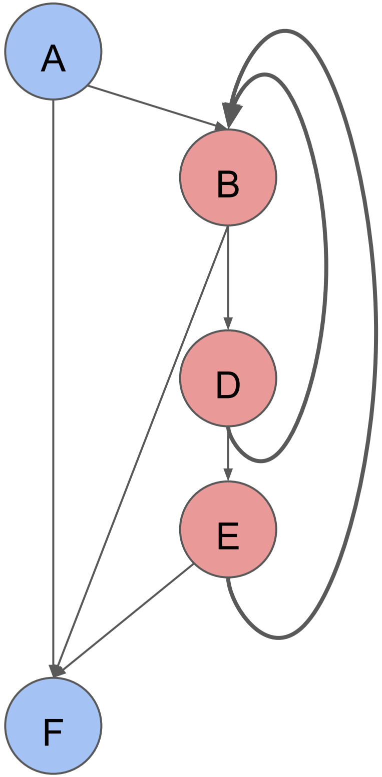
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG



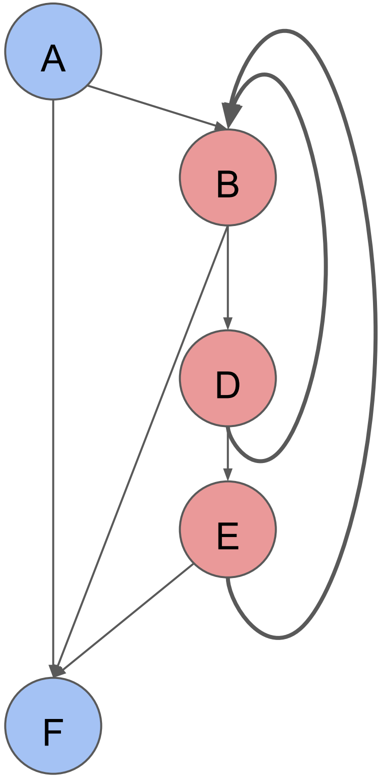
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG



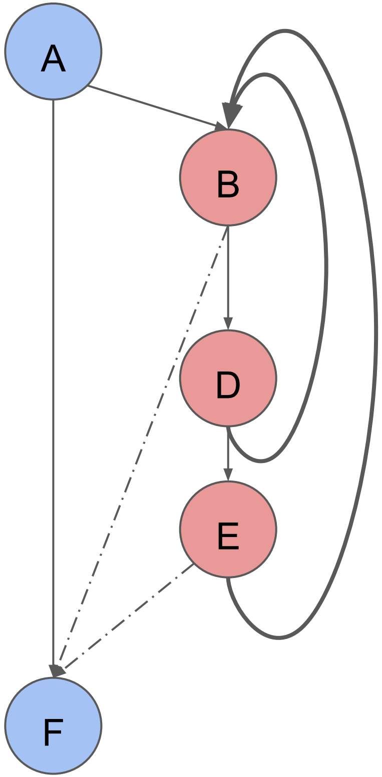
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop



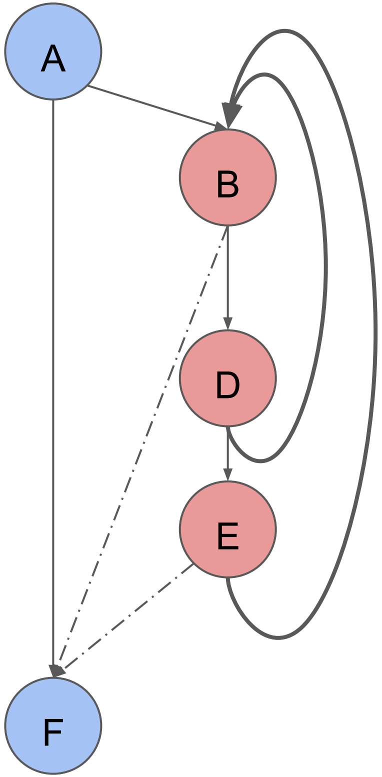
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop



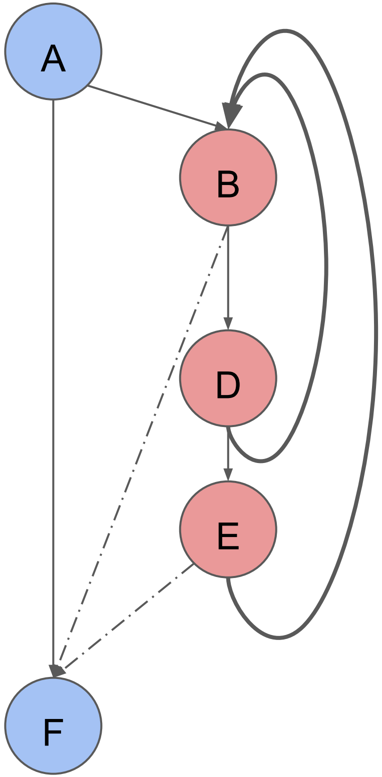
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop
- Loop Branch



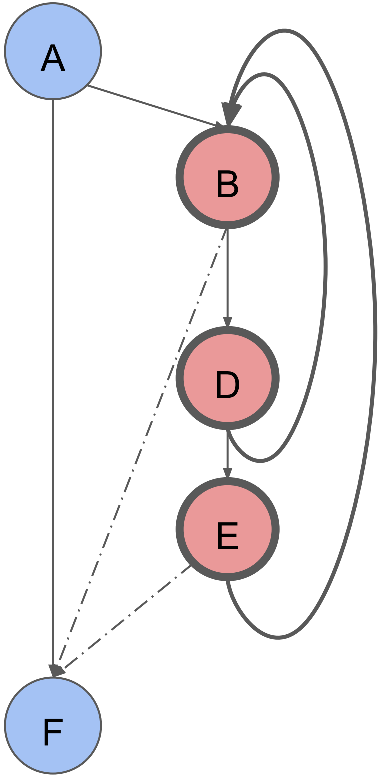
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop
- Loop Branch
  - Either of its outgoing edges is an exit-edge or back-edge



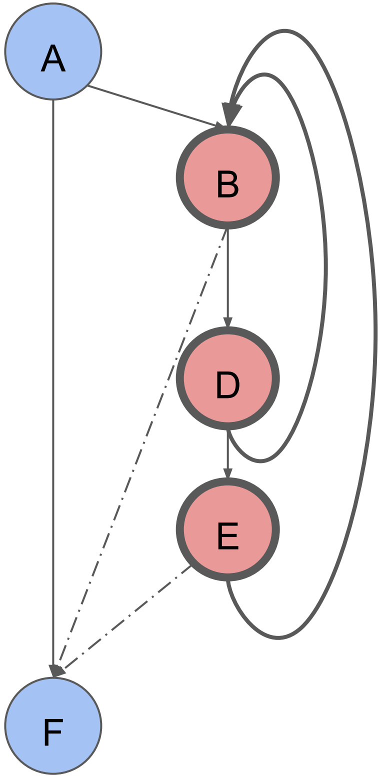
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop
- Loop Branch
  - Either of its outgoing edges is an exit-edge or back-edge



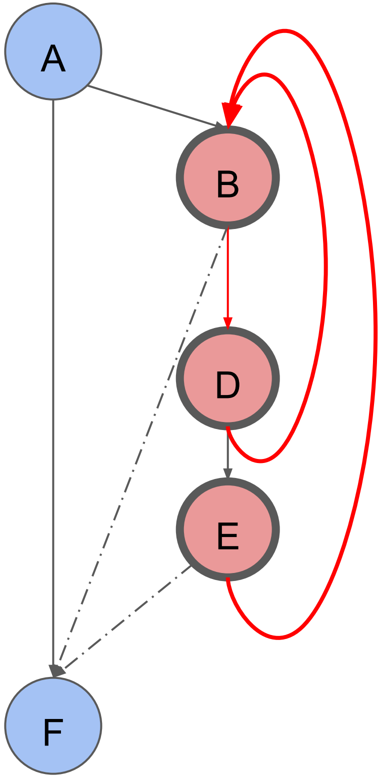
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop
- Loop Branch
  - Either of its outgoing edges is an exit-edge or back-edge
    - Predict back-edge if possible, else predict non exit edge



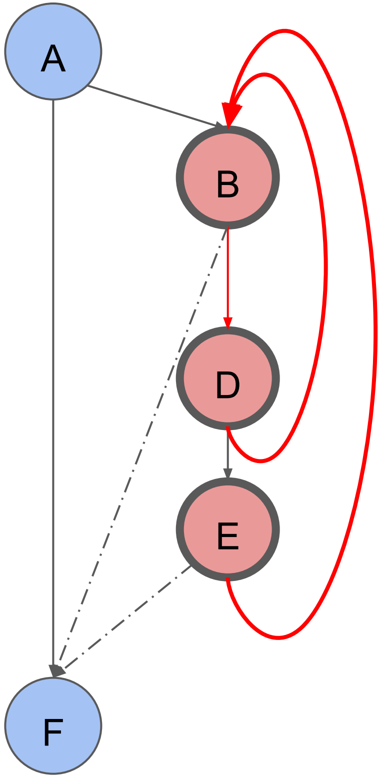
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
    - Exit-edges: edges leaving a natural loop
- Loop Branch
    - Either of its outgoing edges is an exit-edge or back-edge
        - Predict back-edge if possible, else predict non exit edge



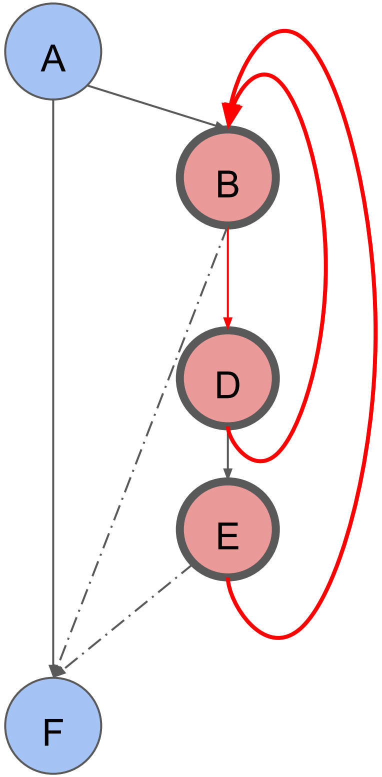
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop
- Loop Branch
  - Either of its outgoing edges is an exit-edge or back-edge
    - Predict back-edge if possible, else predict non exit edge
      - Loops iterate many times and only exit once



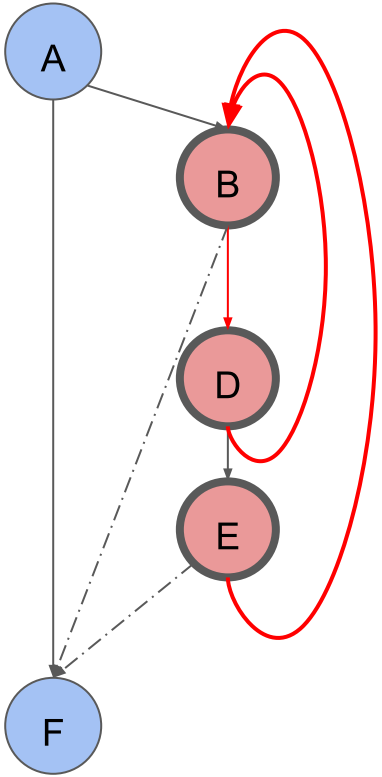
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop
- Loop Branch
  - Either of its outgoing edges is an exit-edge or back-edge
    - Predict back-edge if possible, else predict non exit edge
      - Loops iterate many times and only exit once
- Non-loop Branch



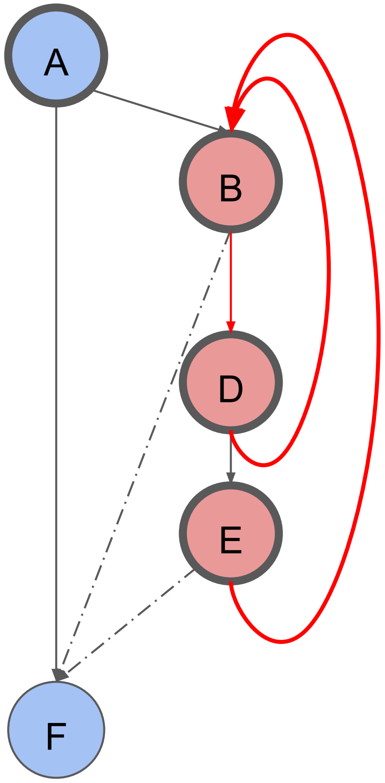
Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
  - Exit-edges: edges leaving a natural loop
- Loop Branch
  - Either of its outgoing edges is an exit-edge or back-edge
    - Predict back-edge if possible, else predict non exit edge
      - Loops iterate many times and only exit once
- Non-loop Branch
  - Neither of its outgoing edges are an exit-edge or a back-edge



Ball & Larus (1993): Predicting loop branches

- Find back-edges and natural loops in the CFG
    - Exit-edges: edges leaving a natural loop
- Loop Branch
    - Either of its outgoing edges is an exit-edge or back-edge
        - Predict back-edge if possible, else predict non exit edge
            - Loops iterate many times and only exit once
- Non-loop Branch
    - Neither of its outgoing edges are an exit-edge or a back-edge



Ball & Larus (1993): Predicting non-loop branches

Ball & Larus (1993): Predicting non-loop branches

1. Opcode Heuristic

Ball & Larus (1993): Predicting non-loop branches

1. Opcode Heuristic

- `BGTZ` & `BGEZ`: taken
- `BLTZ` & `BLEZ`: not taken

Ball & Larus (1993): Predicting non-loop branches

1. Opcode Heuristic

- `BGTZ` & `BGEZ`: taken
- `BLTZ` & `BLEZ`: not taken

Many programs use negative integers to denote error values

Ball & Larus (1993): Predicting non-loop branches

1. Opcode Heuristic

- `BGTZ` & `BGEZ`: taken
- `BLTZ` & `BLEZ`: not taken
- `BNE` between floats: taken
- `BEQ` between floats: not taken

Many programs use negative integers to denote error values

Ball & Larus (1993): Predicting non-loop branches

1. Opcode Heuristic

- `BGTZ` & `BGEZ`: taken
- `BLTZ` & `BLEZ`: not taken
- `BNE` between floats: taken
- `BEQ` between floats: not taken

Many programs use negative integers to denote error values

Floating point numbers are rarely equal

Ball & Larus (1993): Predicting non-loop branches

2. Loop Heuristic

Ball & Larus (1993): Predicting non-loop branches

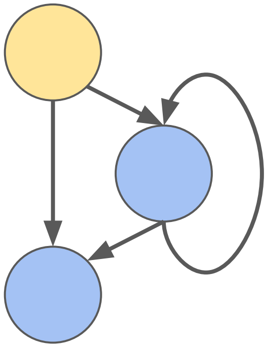
2. Loop Heuristic

- Prefer branch direction to successor block that is a loop header

Ball & Larus (1993): Predicting non-loop branches

2. Loop Heuristic

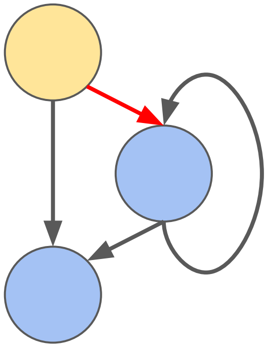
- Prefer branch direction to successor block that is a loop header



Ball & Larus (1993): Predicting non-loop branches

2. Loop Heuristic

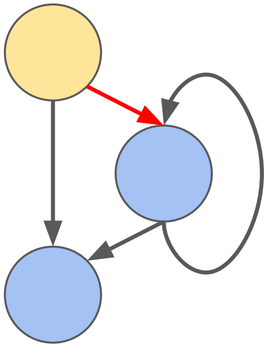
- Prefer branch direction to successor block that is a loop header



Ball & Larus (1993): Predicting non-loop branches

2. Loop Heuristic

- Prefer branch direction to successor block that is a loop header

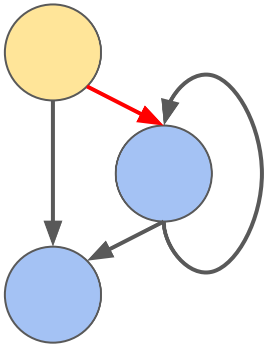


```
while(i < n) {
  …
}
```

Ball & Larus (1993): Predicting non-loop branches

2. Loop Heuristic

- Prefer branch direction to successor block that is a loop header



```
while(i < n) {
  …
}
```

```
  if (i >= n) goto skip
body:
  …
  if (i < n) goto body
skip:
```

Ball & Larus (1993): Predicting non-loop branches

3. Call Heuristic

Ball & Larus (1993): Predicting non-loop branches

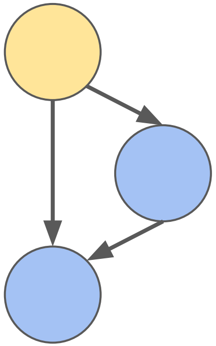
3. Call Heuristic

- Prefer branch direction to successor block that does **not** contain a call

Ball & Larus (1993): Predicting non-loop branches

3. Call Heuristic

- Prefer branch direction to successor block that does **not** contain a call

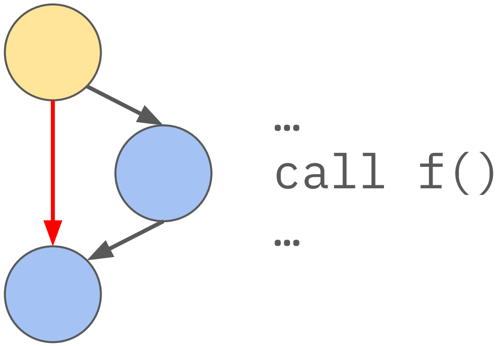


```
…
call f()
…
```

Ball & Larus (1993): Predicting non-loop branches

3. Call Heuristic

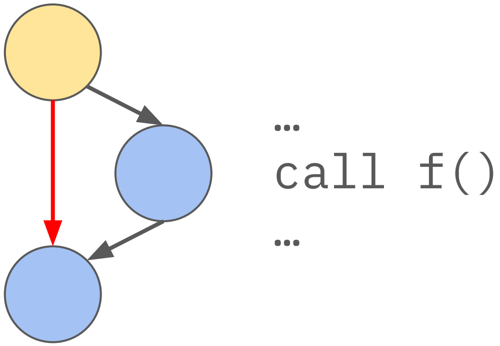
- Prefer branch direction to successor block that does **not** contain a call



Ball & Larus (1993): Predicting non-loop branches

3. Call Heuristic

- Prefer branch direction to successor block that does **not** contain a call

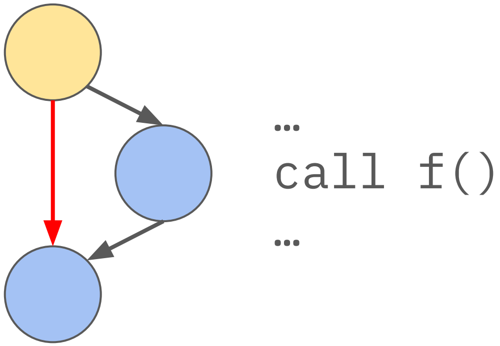


Many conditional calls are to handle rare situations

Ball & Larus (1993): Predicting non-loop branches

3. Call Heuristic

- Prefer branch direction to successor block that does **not** contain a call



Many conditional calls are to handle rare situations

eg: printing output

Ball & Larus (1993): Predicting non-loop branches

4. Return Heuristic

Ball & Larus (1993): Predicting non-loop branches

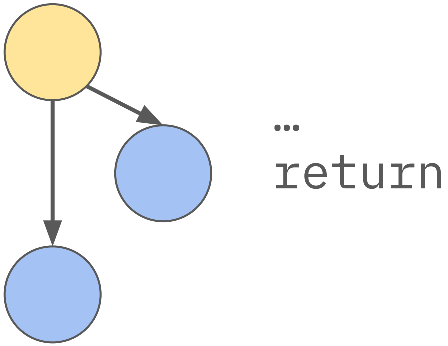
4. Return Heuristic

- Prefer branch direction to successor block that does **not** contain a return

Ball & Larus (1993): Predicting non-loop branches

4. Return Heuristic

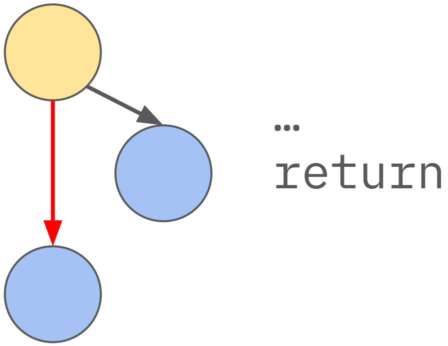
- Prefer branch direction to successor block that does **not** contain a return



Ball & Larus (1993): Predicting non-loop branches

4. Return Heuristic

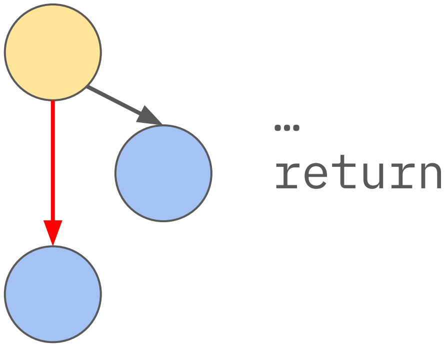
- Prefer branch direction to successor block that does **not** contain a return



Ball & Larus (1993): Predicting non-loop branches

4. Return Heuristic

- Prefer branch direction to successor block that does **not** contain a return

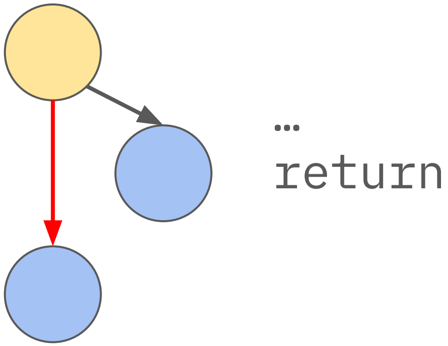


Many conditional returns handle rare situations

Ball & Larus (1993): Predicting non-loop branches

4. Return Heuristic

- Prefer branch direction to successor block that does **not** contain a return



Many conditional returns handle rare situations

eg: base case in recursion

Ball & Larus (1993): Predicting non-loop branches

5. Guard Heuristic

Ball & Larus (1993): Predicting non-loop branches

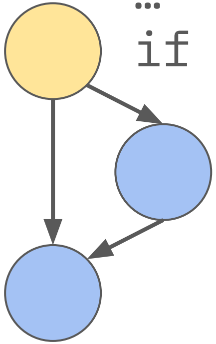
5. Guard Heuristic

- Prefer branch direction to successor block that uses an operand of the branch instruction

Ball & Larus (1993): Predicting non-loop branches

5. Guard Heuristic

- Prefer branch direction to successor block that uses an operand of the branch instruction

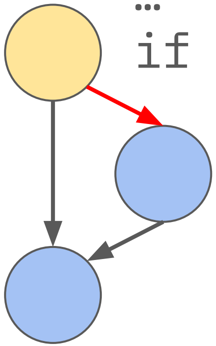


```
…
if x != 0 goto
    …
    use x
    …
```

Ball & Larus (1993): Predicting non-loop branches

5. Guard Heuristic

- Prefer branch direction to successor block that uses an operand of the branch instruction



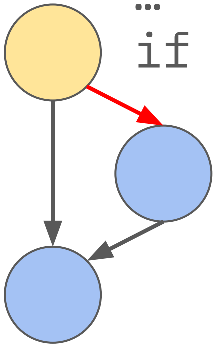
```
…
if x != 0 goto
    …
    use x
    …
```

Ball & Larus (1993): Predicting non-loop branches

5. Guard Heuristic

- Prefer branch direction to successor block that uses an operand of the branch instruction

Many guard conditionals usually allow the guarded value to flow to its use

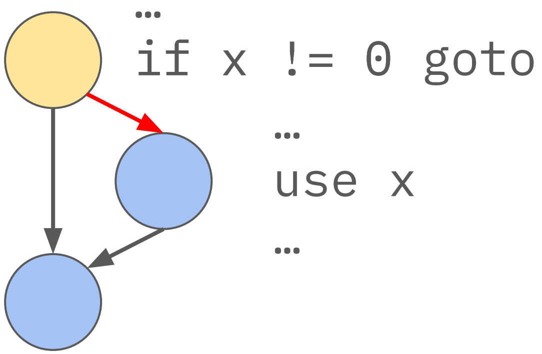


```
…
if x != 0 goto
     …
     use x
     …
```

Ball & Larus (1993): Predicting non-loop branches

5. Guard Heuristic

- Prefer branch direction to successor block that uses an operand of the branch instruction



Many guard conditionals usually allow the guarded value to flow to its use

eg: guarding for existence

Ball & Larus (1993): Predicting non-loop branches

6. Store Heuristic

Ball & Larus (1993): Predicting non-loop branches

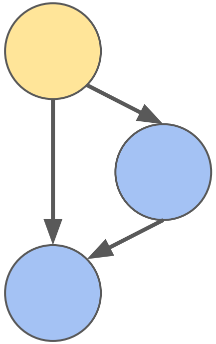
6. Store Heuristic

- Prefer branch direction to successor block that does **not** contain a store instruction

Ball & Larus (1993): Predicting non-loop branches

6. Store Heuristic

- Prefer branch direction to successor block that does **not** contain a store instruction

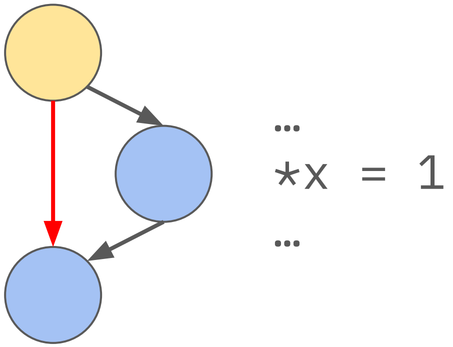


```
…
*x = 1
…
```

Ball & Larus (1993): Predicting non-loop branches

6. Store Heuristic

- Prefer branch direction to successor block that does **not** contain a store instruction

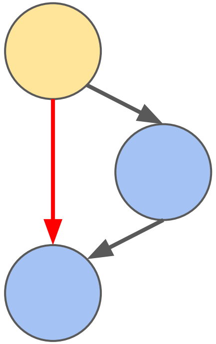


Ball & Larus (1993): Predicting non-loop branches

6. Store Heuristic

- Prefer branch direction to successor block that does **not** contain a store instruction

Many conditional stores are to handle rare situations



```
…
*x = 1
…
```

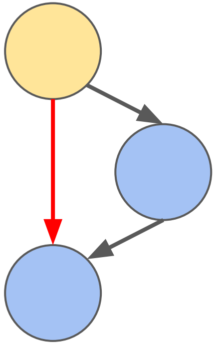
Ball & Larus (1993): Predicting non-loop branches

6. Store Heuristic

- Prefer branch direction to successor block that does **not** contain a store instruction

Many conditional stores are to handle rare situations

eg: updating a maximum



```
...
*x = 1
...
```

Ball & Larus (1993): Predicting non-loop branches

7. Point Heuristic

Ball & Larus (1993): Predicting non-loop branches

7. Point Heuristic

- Predict pointer comparisons as false

Ball & Larus (1993): Predicting non-loop branches

7. Point Heuristic

- Predict pointer comparisons as false
    - Comparing 2 pointers

Ball & Larus (1993): Predicting non-loop branches

7. Point Heuristic

- Predict pointer comparisons as false
  - Comparing 2 pointers
  - Comparing a pointer to a null

Ball & Larus (1993): Predicting non-loop branches

7. Point Heuristic

- Predict pointer comparisons as false
  - Comparing 2 pointers
  - Comparing a pointer to a null

Both cases are rarely true

Ball & Larus (1993): Predicting non-loop branches

1. **Opcode**
2. **Loop**
3. **Call**
4. **Return**
5. **Guard**
6. **Store**
7. **Point**

Ball & Larus (1993): Predicting non-loop branches

1. **Opcode**
2. **Loop**
3. **Call**
4. **Return**
5. **Guard**
6. **Store**
7. **Point**

- More than 1 heuristic can apply to a branch

Ball & Larus (1993): Predicting non-loop branches

1. **Opcode**
2. **Loop**
3. **Call**
4. **Return**
5. **Guard**
6. **Store**
7. **Point**

- More than 1 heuristic can apply to a branch
  - Voting

Ball & Larus (1993): Predicting non-loop branches

1. **Opcode**
2. **Loop**
3. **Call**
4. **Return**
5. **Guard**
6. **Store**
7. **Point**

- More than 1 heuristic can apply to a branch
  - Voting
  - Order

Ball & Larus (1993): Predicting non-loop branches

1. Opcode

2. Loop

3. Call

4. Return

5. Guard

6. Store

7. Point

- More than 1 heuristic can apply to a branch
  - Voting
  - Order
- How to prioritize?

Ball & Larus (1993): Predicting non-loop branches

1. **Opcode**
2. **Loop**
3. **Call**
4. **Return**
5. **Guard**
6. **Store**
7. **Point**

- More than 1 heuristic can apply to a branch
  - Voting
  - Order
- How to prioritize?
  - **Point**, **Call**, **Opcode**, **Return**, **Store**, **Loop**, **Guard**

Dynamic Branch Prediction

Dynamic Branch Prediction

- Prediction of branches change during the execution of the program

Dynamic Branch Prediction

- Prediction of branches change during the execution of the program
- Requires special hardware

Dynamic Branch Prediction

- Prediction of branches change during the execution of the program
- Requires special hardware
  - PHT

Dynamic Branch Prediction

- Prediction of branches change during the execution of the program
- Requires special hardware
    - PHT
    - Global shift register

Dynamic Branch Prediction

- Prediction of branches change during the execution of the program
- Requires special hardware
  - PHT
  - Global shift register
  - Table of perceptrons

Dynamic Branch Prediction

- Prediction of branches change during the execution of the program
- Requires special hardware
  - PHT
  - Global shift register
  - Table of perceptrons
  - …

Reducing Branch Costs via Branch Alignment

Brad Calder and Dirk Grunwald

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA
{calder,grunwald}@cs.colorado.edu

Reducing Branch Costs via Branch Alignment

Brad Calder and Dirk Grunwald

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA
{calder,grunwald}@cs.colorado.edu

- Use PH’s algorithm to minimize the number of taken branches by placing the hot path in a straight line of fall through execution

Reducing Branch Costs via Branch Alignment

Brad Calder and Dirk Grunwald

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA
{calder,grunwald}@cs.colorado.edu

- Use PH's algorithm to minimize the number of taken branches by placing the hot path in a straight line of fall through execution
- Show that branch alignment improves dynamic branch prediction accuracy

Code Placement for Improving Dynamic Branch Prediction Accuracy

Daniel A. Jiménez

Deptartment of Computer Science
Rutgers University
Piscataway, New Jersey, USA

and

Departamento de Arquitectura de Computadores
Universidad Politécnica de Cataluña
Barcelona, Cataluña, Spain

djimenez@cs.rutgers.edu

Jiménez (2005)

- Introduces PHT Partitioning

Jiménez (2005)

- Introduces PHT Partitioning
    - Partitions branches based on their bias so that they are less likely to interfere destructively with each other

Jiménez (2005)

- Introduces PHT Partitioning
  - Partitions branches based on their bias so that they are less likely to interfere destructively with each other
  - Inserts no-op instructions to change the bits of branch instructions so that they map to the appropriate partition

Jiménez (2005)

- Introduces PHT Partitioning
  - Partitions branches based on their bias so that they are less likely to interfere destructively with each other
  - Inserts no-op instructions to change the bits of branch instructions so that they map to the appropriate partition

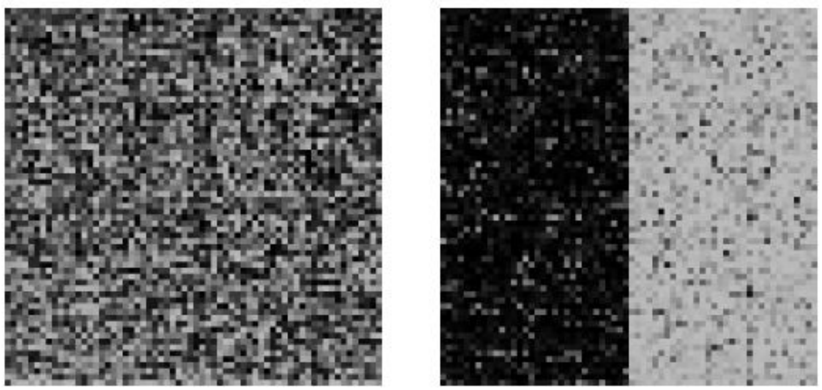


Figure 1. Average PHT entries before and after compiler-based alignment.

Branchless Code Generation for Modern Processor Architectures

Alexandros Angelou
University of Thessaly, Department of Computer Science and Telecommunications
analexandros@uth.gr

Antonios Dadaliaris
University of Thessaly, Department of Computer Science and Telecommunications
dadaliaris@uth.gr

Georgios Dimitriou
University of Thessaly, Department of Computer Science and Telecommunications
dimitriu@uth.gr

Michael Dossis
University of Western Macedonia, Department of Informatics
mdossis@uowm.gr

Branchless Code Generation for Modern Processor Architectures

Alexandros Angelou
University of Thessaly, Department of Computer Science and Telecommunications
analexandros@uth.gr

Antonios Dadaliaris
University of Thessaly, Department of Computer Science and Telecommunications
dadaliaris@uth.gr

Georgios Dimitriou
University of Thessaly, Department of Computer Science and Telecommunications
dimitriu@uth.gr

Michael Dossis
University of Western Macedonia, Department of Informatics
mdossis@uowm.gr

No need to predict a branch that isn’t there

Angelou et al. (2021)

Angelou et al. (2021)

- Remove branches in a program using code transformations

Angelou et al. (2021)

- Remove branches in a program using code transformations
    - Use the condition as the data or as a mask to generate the desired data

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition $result = condition \times case\_true + \neg condition \times case\_false$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow

$$result = condition \times case\_true + \neg condition \times case\_false$$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow
  - BitMasking

$$result = condition \times case\_true + \neg condition \times case\_false$$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow
  - BitMasking

$$result = condition \times case\_true + \neg condition \times case\_false$$

$$mask(c) = integer(\neg c) - 1$$

$$result = (mask(c) \land value1) \lor (mask(\neg c) \land value2)$$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow
  - BitMasking
    - Slow

$$result = condition \times case\_true + \neg condition \times case\_false$$

$$mask(c) = integer(\neg c) - 1$$

$$result = (mask(c) \land value1) \lor (mask(\neg c) \land value2)$$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow
  - BitMasking
    - Slow
  - Conditional move and predication

$$result = condition \times case\_true + \neg condition \times case\_false$$

$$mask(c) = integer(\neg c) - 1$$

$$result = (mask(c) \wedge value1) \vee (mask(\neg c) \wedge value2)$$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow
  - BitMasking
    - Slow
  - Conditional move and predication
    - Some ISAs have support for complex branch operations

$$result = condition \times case\_true + \neg condition \times case\_false$$

$$mask(c) = integer(\neg c) - 1$$

$$result = (mask(c) \wedge value1) \vee (mask(\neg c) \wedge value2)$$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow
  - BitMasking
    - Slow
  - Conditional move and predication
    - Some ISAs have support for complex branch operations
      - `MOVN` & `MOVZ` in MIPS

$$result = condition \times case\_true + \neg condition \times case\_false$$

$$mask(c) = integer(\neg c) - 1$$

$$result = (mask(c) \wedge value1) \vee (mask(\neg c) \wedge value2)$$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow
  - BitMasking
    - Slow
  - Conditional move and predication
    - Some ISAs have support for complex branch operations
      - `MOVN` & `MOVZ` in MIPS
      - Instruction predication in ARM A32

$$result = condition \times case\_true + \neg condition \times case\_false$$

$$mask(c) = integer(\neg c) - 1$$

$$result = (mask(c) \land value1) \lor (mask(\neg c) \land value2)$$

Angelou et al. (2021)

- Remove branches in a program using code transformations
  - Use the condition as the data or as a mask to generate the desired data
  - May be able to remove unpredictable branches
- 3 methods
  - Multiplication and addition
    - Very slow

$$result = condition \times case\_true + \neg condition \times case\_false$$

  - BitMasking
    - Slow

$$mask(c) = integer(\neg c) - 1$$

$$result = (mask(c) \wedge value1) \vee (mask(\neg c) \wedge value2)$$

  - Conditional move and predication
    - Some ISAs have support for complex branch operations
      - `MOVN` & `MOVZ` in MIPS
      - Instruction predication in ARM A32
- Implemented using the LLVM Framework and MIPS

Thank you :)

References

Cliff Young and Michael D. Smith. 1994. Improving the accuracy of static branch prediction using branch correlation. SIGOPS Oper. Syst. Rev. 28, 5 (Dec. 1994), 232–241. https://doi.org/10.1145/381792.195549

Daniel A. Jiménez. 2005. Code placement for improving dynamic branch prediction accuracy. SIGPLAN Not. 40, 6 (June 2005), 107–116. https://doi.org/10.1145/1064978.1065025

Jason Mccandless and David Gregg. 2012. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. ACM Trans. Archit. Code Optim. 8, 4, Article 24 (January 2012), 20 pages. https://doi.org/10.1145/2086696.2086703

Thomas Ball and James R. Larus. 1993. Branch prediction for free. In Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (PLDI '93). Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/155090.155119

Joseph A. Fisher and Stefan M. Freudenberger. 1992. Predicting conditional branch directions from previous runs of a program. In Proceedings of the fifth international conference on Architectural support for programming languages and operating systems (ASPLOS V). Association for Computing Machinery, New York, NY, USA, 85–95. https://doi.org/10.1145/143365.143493

Andreas Krall. 1994. Improving semi-static branch prediction by code replication. In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94). Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/178243.178252

S. McFarling and J. Hennesey. 1986. Reducing the cost of branches. SIGARCH Comput. Archit. News 14, 2 (May 1986), 396–403. https://doi.org/10.1145/17356.17402

James E. Smith. 1981. A study of branch prediction strategies. In Proceedings of the 8th annual symposium on Computer Architecture (ISCA '81). IEEE Computer Society Press, Washington, DC, USA, 135–148.

Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. SIGPLAN Not. 25, 6 (Jun. 1990), 16–27. https://doi.org/10.1145/93548.93550

Alexandros Angelou, Antonios Dadaliaris, Michael Dossis, and Georgios Dimitriou. 2022. Branchless Code Generation for Modern Processor Architectures. In Proceedings of the 25th Pan-Hellenic Conference on Informatics (PCI '21). Association for Computing Machinery, New York, NY, USA, 300–305. https://doi.org/10.1145/3503823.3503879

Brad Calder and Dirk Grunwald. 1994. Reducing branch costs via branch alignment. In Proceedings of the sixth international conference on Architectural support for programming languages and operating systems (ASPLOS VI). Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/195473.195553

- Profile vs no profile
- Static vs dynamic
- Binary vs IR
- Need hardware?

- Fisher empirically showed that many applications are dominated by statically-predictable branches
    - Thus static history-based schemes are effective across the different data sets of most applications.

What/Why branch prediction?

Branch Correlation

Profile-based static branch prediction

- Without branch correlation
    - For each branch in a program, a profiler collects dynamic branch statistics
        - Taken
        - Not taken
    - The statistics determine the static prediction of each branch
        - Does not consider branch correlation
- With branch correlation
    - Collect taken vs not taken statistics for each execution path that reaches a branch. Execution branch contains k conditional branches, == shift register with k bits.
    - Considers branch path histories instead of branch pattern histories
        - Records basic block numbers

Why Compilers

Ball & Larus (1993)

- Branch prediction for free
- Program-based (not profile-based)
    - Performs well for a large and diverse set of C and Fortran programs
- Use natural loop analysis to predict branches that control iterations of loops
- Heuristics for predicting non-loop branches
    - Dominate dynamic branch count of many programs
    - Simple, require little program analysis effective for coverage and missrate
- Not as good as profiling
    - Still useful, surprisingly high accuracy
    - Average miss rate of 20%
        - Average miss rate of 26% for non-loop branches
- Ignores indirect branches
- Looks only at 2 way conditional branches
- Classify branches as loop or non-loop
    - Backward -> usually control loop iterations
    - Non-backwards -> may also control loop iterations via exiting/continuing the iteration
- Find natural loops in a CFG
    - For any vertex v in nat-loop(y), at least one of v’s successors must be in nat(y). For any vertex in v, either none of its outgoing edges are exit edges or exactly one is an exit edge

Fisher & Freudenberger (1992)

- How predictable branches are when previous runs of a program are used to feed information back to the compiler?
  - Even code with a complex flow of control are dominated by branches which go in one way and that this direction usually varies little when one changes the data used as the predictor and target.
- Interested in static branch prediction
  - Find out whether one will get acceptable results using the branch behaviour of the program during prev runs to predict branch directions for subseq runs.
- Sum branch direction counts of a variety of runs
  - Use that to statically predict
- Static branch prediction
  - User inserted directives: which direction is likely
  - Compiler analysis using heuristics
  - Profiling, get statistics and feed back to the program and recompile

Non-loop branch heuristics

- Opcode heuristic
    - Many programs use negative integers to denote error values,
        - Bltz and blez are not taken, and bgtz and bgez are taken
        - Beq FLOAT FLOAT, not taken (usually false)
    - Low coverage, high accuracy
- Loop Heuristic
    - Successor does not postdominate the branch and is either a loop head or a loop preheader
        - Ie. passes control unconditionally to a loop head which it dominates
        - Predict property
    - Predictbranches to execute loops over avoiding loops
        - Many compilers generate code for while loops and for loops by generating an if-the around a do-until loop.
- Call Heuristic
    - Successor block contains a call or unconditionally passes control to a block with a call that it dominates and the successor block does not postdominate the branch.
        - Predict without property
    - Expected that you should do the opposite action, but found this empirically through testing
        - Many conditional calls are to handle exceptional situations ex: printing output
- Return Heuristic
    - Successor block contains a return or unconditionally passes control to a block that contains a return
        - Predict without property
    - Recursion: return is base case.
    - Good performance
- Guard Heurstic
    - Register r is an operand of the branch instruction, register r is used in the successor block before it is defined, and the successor block does not postdominate the branch.
        - Predict with property
    - Attempts to find instances in which a branch on a value guards a later use of that value
        - Many guard contiionals are used to catc exception condition and the common case is for a guard to allure value to flow to use.
    - Also think of pointer guards, is pointer null?
    - Uh oh updating the maximum

Young:94

- Profile-based
- Code transformation
- Exploits branch correlation
- Static branch prediction
- Encode branch history information in the program counter
- Duplication and placement of basic blocks
- Do not transform branches with exponential branching paths and no branch correlation using compile-time heuristics
- Do not need special hardware (global register)
- Conditional branches include a static prediction bit that encodes the predicted branch direction

Pointer comparisons

Predict not true

If they had type information, could do this easily

If not, need to detect code sequences that are pointer comparisons

- Other failed heuristics:
    - Number of insts between branch and target
    - Domination and post domination relations between branch and successors
- More than 1 heuristic can apply to a branch
    - How to priotitize heuristics?
        - Total order
        - Voting
        - No heuristic? Random
- Point,call,Opcode,return,store,loop,guard
- High acc on non-loop branches is crucial to get long sequences of correct predictions

Perfect static predictor

Predicts the more frequently outgoing edge of each branch in a program.

- Superpipelined machines (deep) require accurate branch prediction to avoid costly branch mispredictions
- Also see many branches
- Many different branch prediction schemes
    - Dynamic
        - hardware
    - Static
        - profiling

Krall (1994)

- Defines semi-static branch prediction: relies on profiling
- Static branch prediction: relies on static analysis
- Constructs branch prediction state machines for loop branches and correlated branches
  - Divide loop branches into intra (within) and exit (leave)

Young & Smith (1994)

- Static branch prediction using branch correlation
- Profiling
  - Keep a FIFO with length k (basic block number and direction)
  - Record outcome of branch in data structure for the block (counter pairs indexed by the current path in FIFO)
    - # Taken & # total
  - Flush queue on proc call / return
    - Include call/return as first cond branch in fifo
- Minimizing each branch history tree
  - Find minimum number of unique paths necessary to maximize branch prediction accuracy
    - TAKE, FALLTHRU, BOTH
- Global Reconciliation
  - Will terminate
    - Has potential to generate exponential block copies, unlikely, use heuristics to detect and limit expansion
- Layout
  - Naive algorithm
    - Never add branches where there was none. Does not increase the dynamic instruction count
    - Duplicates blocks when needed
  - Heuristics to help: ignore small total counts, ignore indecisive stats (less duplication and less sensitive to training data set)
- Uses code duplication to improve branch prediction for compile-time optimization and instructions scheduling techniques
- Duplicated code
  - More I cache misses
  - More page faults
- No hardware required
  - Save space on chip (better use for transistors?)
- History paths correlate to bp in 3 ways:
  - logically (always correlate) can be found without profiling, statistically (statistically correlate), and falsely (based on data set, not a generally characteristic of program) <- not useful (harmful)

Branch Prediction

Static Branch Prediction

- Prediction made at compile-time for each branch

Dynamic Branch Prediction

- Prediction of a branch changes during the execution of the program
- Requires special hardware

Reducing cost of branches

- Delayed branch
  - Simple: machine continues executing instructions after the branch until cond is determined
  - Compiler tries to schedule useful instructions into slots after branch from 1 of 3 locations
    - Before: branch must not depend
    - after/target: must be safe to execute whether the branch is taken or not
  - No live state may be destroyed and no illegal ops can be done (ex: loading from null addr)
  - Use before rather than after/target if possible since it is always beneficial, not just when correct path
- With squashing
  - Major limitation of delay is the difficulty of filling the branch delay slots with safe instructions
  - Squashing direction
    - Predict branch at compile time (always guess taken) and hint likelihood of branch taken, use bit to specify
    - Don’t need to squash if can fill delay slots with instructions from before the branch. Use bit to specify
- Could use profiling to improve static prediction
- If machine always predicts taken/not taken
  - Can restructure if then else statements to benefit based on profile information
  - If branch is usually not taken, insert branch above to help for taken guessing machines