Impact of code compression on estimated worst-case execution times

Haluk OZAKTAS[b]
Karine HEYDEMANN[b]
Christine ROCHANGE[a]
Hugues CASSÉ[a]

[a] IRIT - University of Toulouse
[b] LIP6 - University Pierre and Marie Curie

26 October 2009

- Embedded systems are often constrained by
  - Code size
  - Execution time
  - Energy consumption
  - ...

- Embedded systems are often constrained by
  - Code size
  - Execution time
  - Energy consumption
  - ...
- Various techniques have been proposed to improve each criteria
  - Code compression
  - Compiler optimizations
  - Code and data placement strategies
  - ...

- Embedded systems are often constrained by
  - Code size
  - Execution time
  - Energy consumption
  - ...
- Various techniques have been proposed to improve each criteria
  - Code compression
  - Compiler optimizations
  - Code and data placement strategies
  - ...
- But interactions of optimizations targeting different objectives are rarely analyzed
  - Project MORE: **M**ulti-criteria **O**ptimizations for **R**eal-time **E**mbedded Systems

- Embedded systems are often constrained by
  - Code size
  - Execution time
  - Energy consumption
  - ...
- Various techniques have been proposed to improve each criteria
  - Code compression
  - Compiler optimizations
  - Code and data placement strategies
  - ...
- But interactions of optimizations targeting different objectives are rarely analyzed
  - Project MORE: **M**ulti-criteria **O**ptimizations for **R**eal-time **E**mbedded Systems

- This works aims to analyze the impact of code compression on the worst case execution time (WCET)

Current Section

Principles

- Representing the code in a more compact form
  - Reduces storage needs
  - Modifies the effective memory bandwidth
    - Improved/decreased system performance
  - Decreases memory accesses
    - May reduce energy consumption

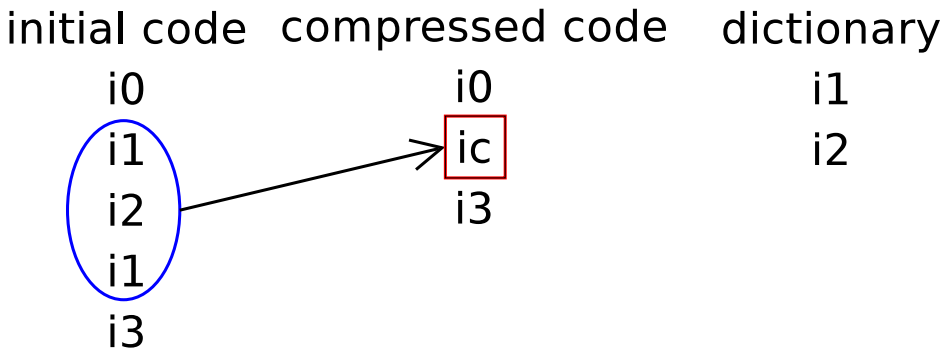
Principles

- Representing the code in a more compact form
  - Reduces storage needs
  - Modifies the effective memory bandwidth
    - Improved/decreased system performance
  - Decreases memory accesses
    - May reduce energy consumption
- Off the line compression
  - During or after the compilation process
  - Compression time is not critical

Principles

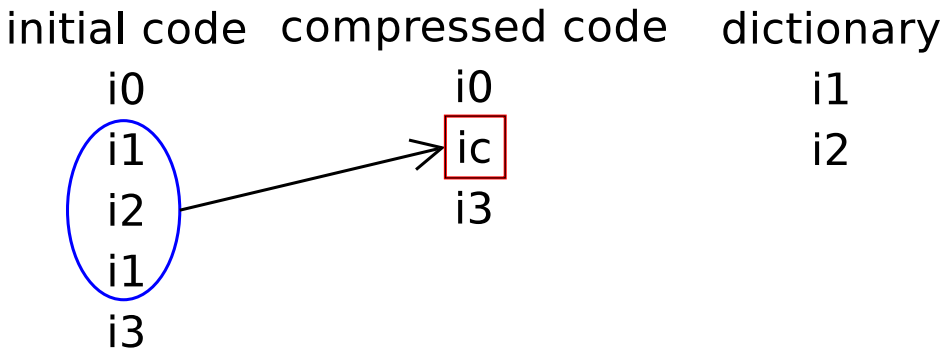
- Representing the code in a more compact form
  - Reduces storage needs
  - Modifies the effective memory bandwidth
    - Improved/decreased system performance
  - Decreases memory accesses
    - May reduce energy consumption
- Off the line compression
  - During or after the compilation process
  - Compression time is not critical
- Decompression during execution
  - Decompression time is critical!
  - How? software/hardware/mix
  - Where? pre-cache/post-cache/in-pipeline

Dictionary based code compression



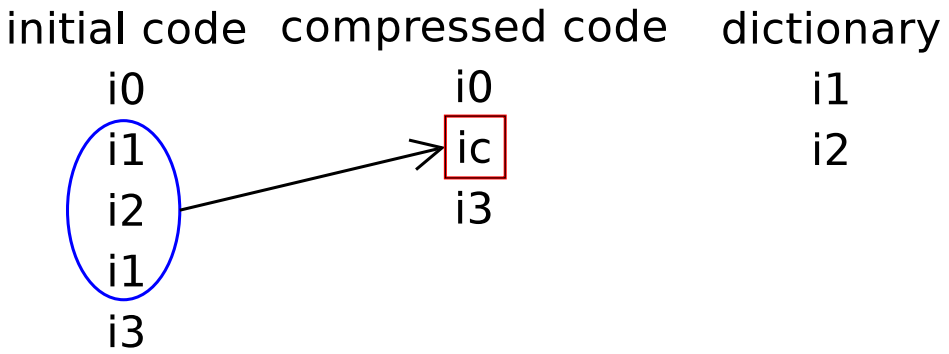
- Build a dictionary with words from the initial code
- Replace these words with corresponding dictionary indexes

Dictionary based code compression



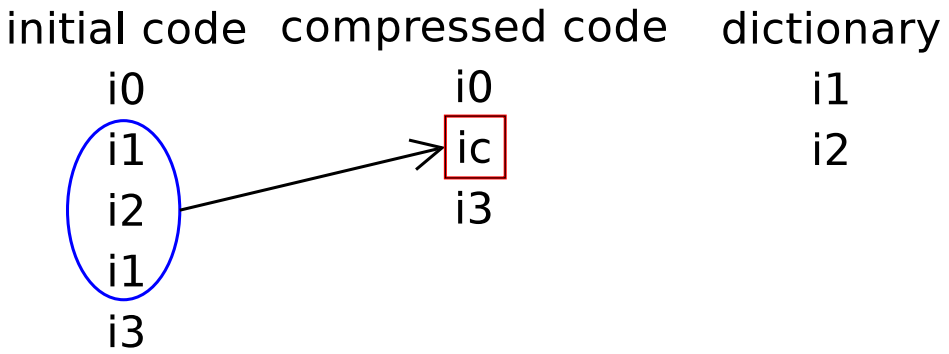
- Build a dictionary with words from the initial code
- Replace these words with corresponding dictionary indexes
- Advantage: fast decompression with simple hardware

Dictionary based code compression



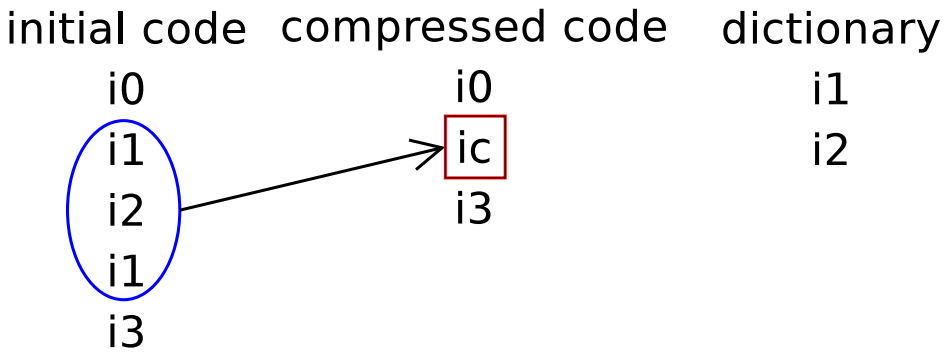
- Build a dictionary with words from the initial code
- Replace these words with corresponding dictionary indexes
- Advantage: fast decompression with simple hardware
- How to choose the words to include in the dictionary?

Dictionary based code compression



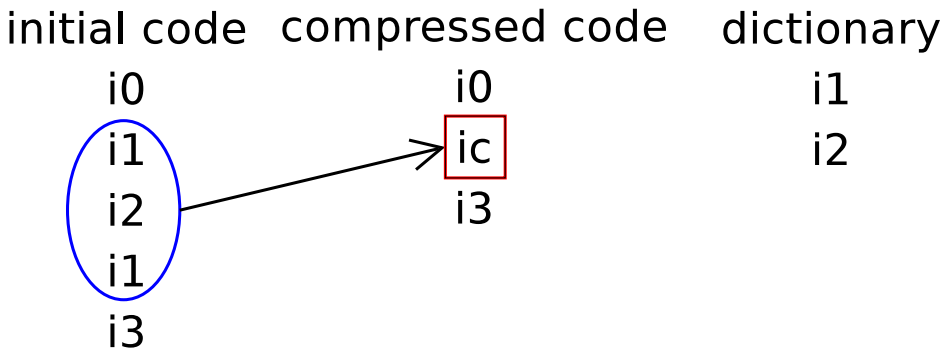
- Build a dictionary with words from the initial code
- Replace these words with corresponding dictionary indexes
- Advantage: fast decompression with simple hardware
- How to choose the words to include in the dictionary?
  - Most statically repeated words to improve compression rate

Dictionary based code compression



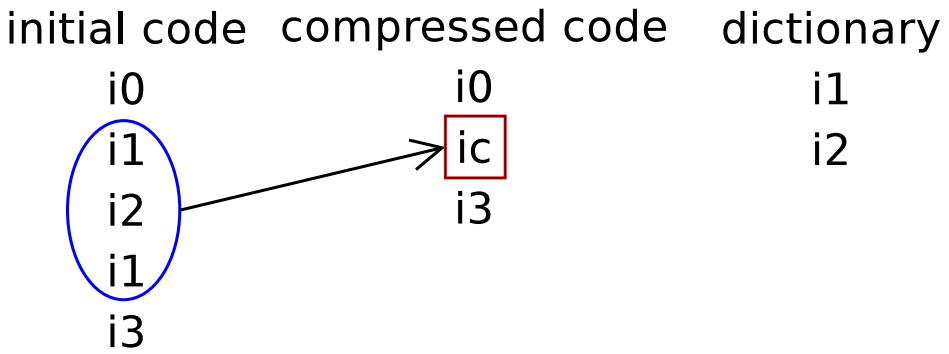
- Build a dictionary with words from the initial code
- Replace these words with corresponding dictionary indexes
- Advantage: fast decompression with simple hardware
- How to choose the words to include in the dictionary?
  - Most statically repeated words to improve compression rate
  - Most executed words to improve performance and energy consumption

Dictionary based code compression



- Build a dictionary with words from the initial code
- Replace these words with corresponding dictionary indexes
- Advantage: fast decompression with simple hardware
- How to choose the words to include in the dictionary?
  - Most statically repeated words to improve compression rate
  - Most executed words to improve performance and energy consumption
  - A mixture of both

Dictionary based code compression



- Build a dictionary with words from the initial code
- Replace these words with corresponding dictionary indexes
- Advantage: fast decompression with simple hardware
- How to choose the words to include in the dictionary?
  - Most statically repeated words to improve compression rate
  - Most executed words to improve performance and energy consumption
  - A mixture of both
  - ...

Implementation of dictionary based code compression

- Full instructions are words to put in the dictionary

Implementation of dictionary based code compression

- Full instructions are words to put in the dictionary
- 2 or 3 consecutive instructions are replaced with an encoding instruction

Implementation of dictionary based code compression

- Full instructions are words to put in the dictionary
- 2 or 3 consecutive instructions are replaced with an encoding instruction
- An encoding instruction contains only instructions from the same basic block to avoid that:

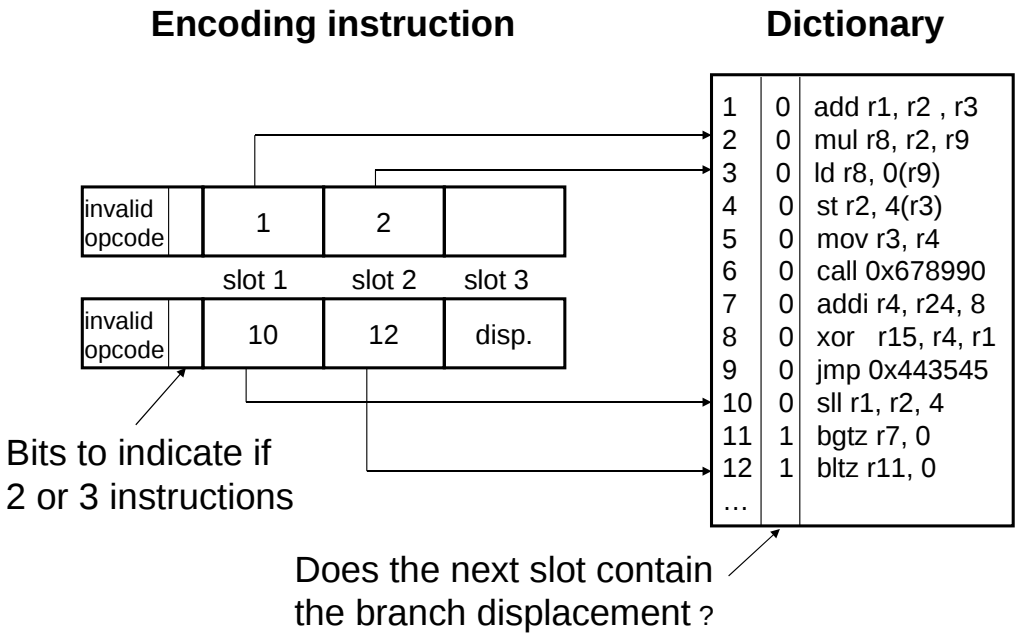
Implementation of dictionary based code compression

- Full instructions are words to put in the dictionary
- 2 or 3 consecutive instructions are replaced with an encoding instruction
- An encoding instruction contains only instructions from the same basic block to avoid that:
  - Multiple branches share the same address
    - Branch prediction is affected

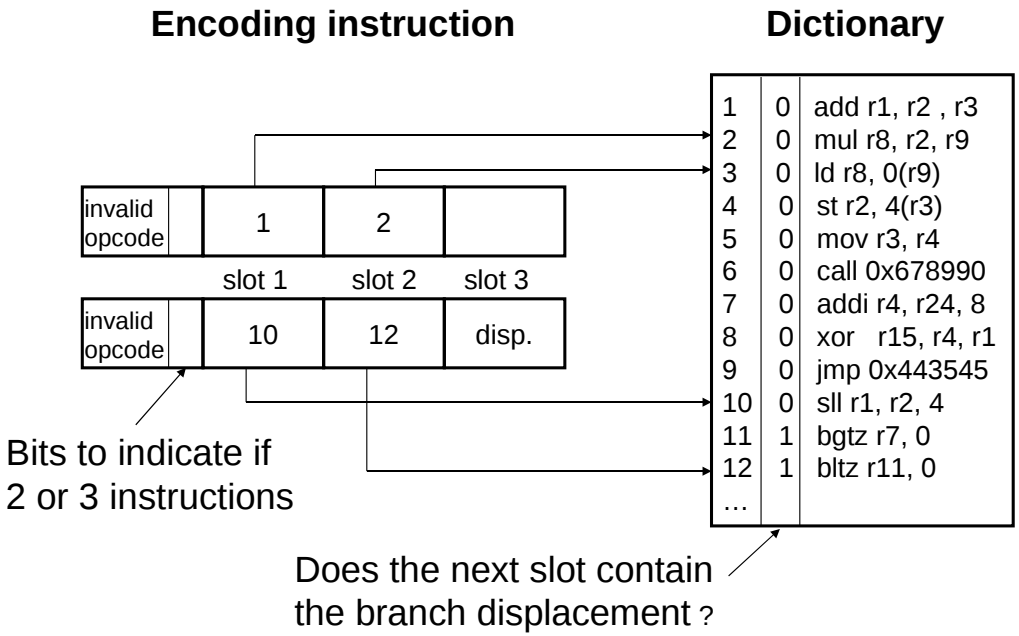
Implementation of dictionary based code compression

- Full instructions are words to put in the dictionary
- 2 or 3 consecutive instructions are replaced with an encoding instruction
- An encoding instruction contains only instructions from the same basic block to avoid that:
  - Multiple branches share the same address
    - Branch prediction is affected
  - Multiple branch targets share the same address
    - Which instruction is the real target?

Encoding instruction and dictionary

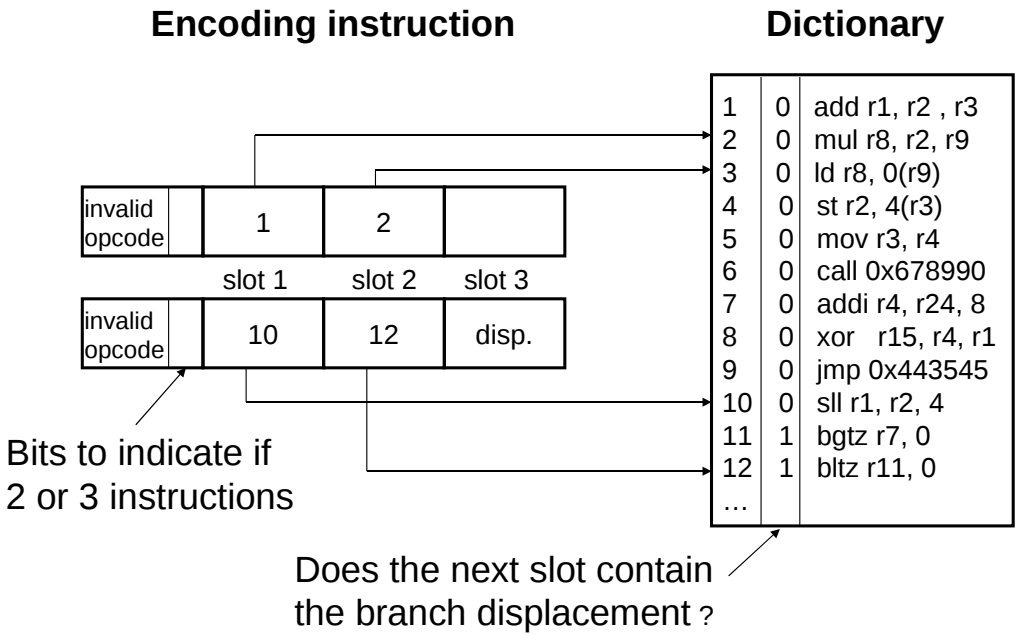


Encoding instruction and dictionary



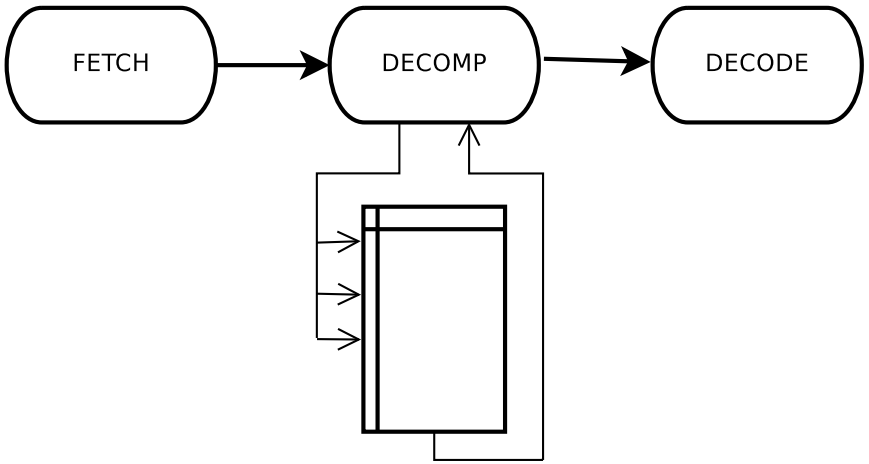
- Encoding instruction
  - 32 bits length: resolves code alignment problems
  - 3 slots of 8 bits: up to 256 addressable entries in the dictionary

Encoding instruction and dictionary



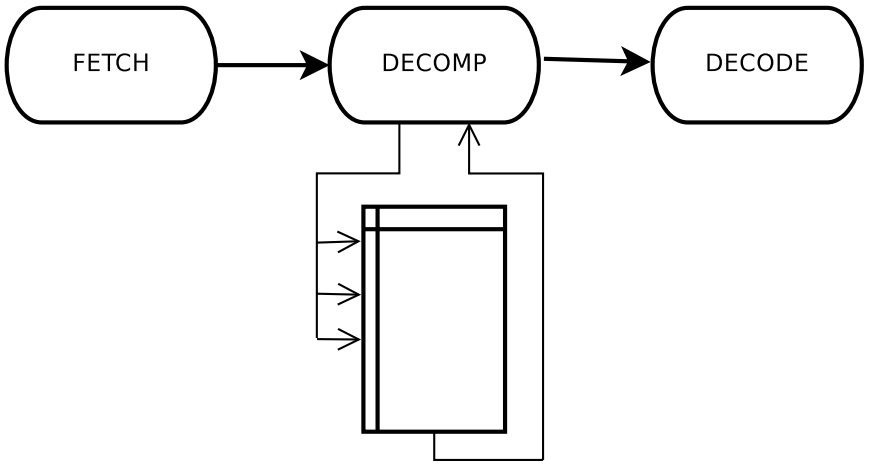
- Encoding instruction
  - 32 bits length: resolves code alignment problems
  - 3 slots of 8 bits: up to 256 addressable entries in the dictionary
- Dictionary
  - P% is filled with most executed and the remaining is filled with most repeated instructions

In-pipeline decompression



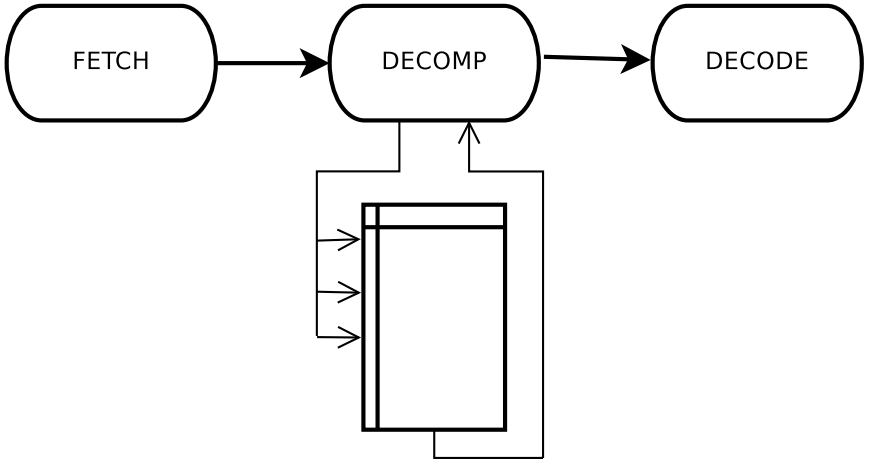
- An additional decompression stage between fetch and decode
  - Resembles to extra decode stages of i686 (and so forth) that dynamically translate IA-32 instructions into micro-operations

In-pipeline decompression



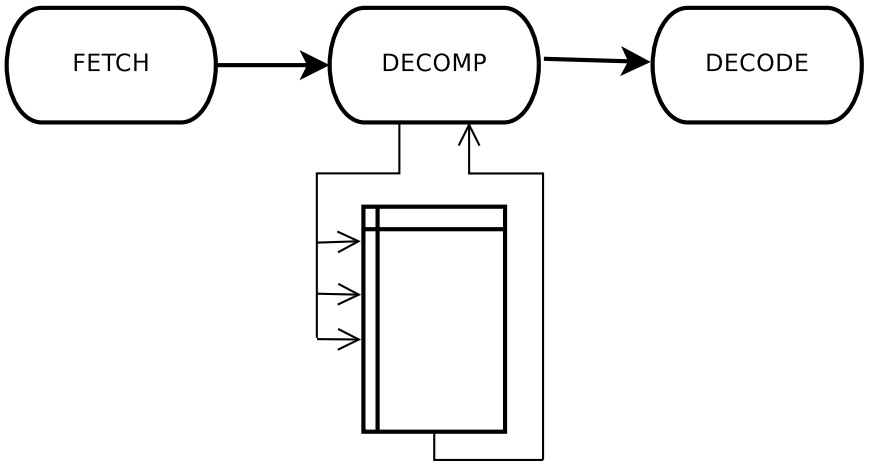
- An additional decompression stage between fetch and decode
  - Resembles to extra decode stages of i686 (and so forth) that dynamically translate IA-32 instructions into micro-operations
- Cache becomes virtually larger $\Rightarrow$ fewer cache misses and fewer memory accesses
  - Can improve performance and decrease energy consumption

In-pipeline decompression



- An additional decompression stage between fetch and decode
  - Resembles to extra decode stages of i686 (and so forth) that dynamically translate IA-32 instructions into micro-operations
- Cache becomes virtually larger $\Rightarrow$ fewer cache misses and fewer memory accesses
  - Can improve performance and decrease energy consumption
- Implementable regardless of the complexity of the processor

In-pipeline decompression



- An additional decompression stage between fetch and decode
  - Resembles to extra decode stages of i686 (and so forth) that dynamically translate IA-32 instructions into micro-operations
- Cache becomes virtually larger $\Rightarrow$ fewer cache misses and fewer memory accesses
  - Can improve performance and decrease energy consumption
- Implementable regardless of the complexity of the processor
- Requires modification of the processor core!

Current Section

Estimated WCETs computation

- The estimation of Worst-Case Execution Times (WCETs) usually consists of 3 steps

Estimated WCETs computation

- The estimation of Worst-Case Execution Times (WCETs) usually consists of 3 steps
  - **Flow analysis** determines flow facts like loop bounds and infeasible paths

Estimated WCETs computation

- The estimation of Worst-Case Execution Times (WCETs) usually consists of 3 steps
  - **Flow analysis** determines flow facts like loop bounds and infeasible paths
  - **Low level analysis** computes the worst case execution cost of each basic block taking into account target hardware

Estimated WCETs computation

- The estimation of Worst-Case Execution Times (WCETs) usually consists of 3 steps
  - **Flow analysis** determines flow facts like loop bounds and infeasible paths
  - **Low level analysis** computes the worst case execution cost of each basic block taking into account target hardware
    - Examination of history based components (instruction/data caches)

Estimated WCETs computation

- The estimation of Worst-Case Execution Times (WCETs) usually consists of 3 steps
  - **Flow analysis** determines flow facts like loop bounds and infeasible paths
  - **Low level analysis** computes the worst case execution cost of each basic block taking into account target hardware
    - Examination of history based components (instruction/data caches)
    - Computation of execution cost of each basic block

Estimated WCETs computation

- The estimation of Worst-Case Execution Times (WCETs) usually consists of 3 steps
  - **Flow analysis** determines flow facts like loop bounds and infeasible paths
  - **Low level analysis** computes the worst case execution cost of each basic block taking into account target hardware
    - Examination of history based components (instruction/data caches)
    - Computation of execution cost of each basic block
  - **WCET computation** combines flow facts and execution costs

Estimated WCETs computation

- The estimation of Worst-Case Execution Times (WCETs) usually consists of 3 steps
  - **Flow analysis** determines flow facts like loop bounds and infeasible paths
  - **Low level analysis** computes the worst case execution cost of each basic block taking into account target hardware
    - Examination of history based components (instruction/data caches)
    - Computation of execution cost of each basic block
  - **WCET computation** combines flow facts and execution costs
- We focus on low level analysis since code compression doesn't change flow facts

Instruction cache analysis

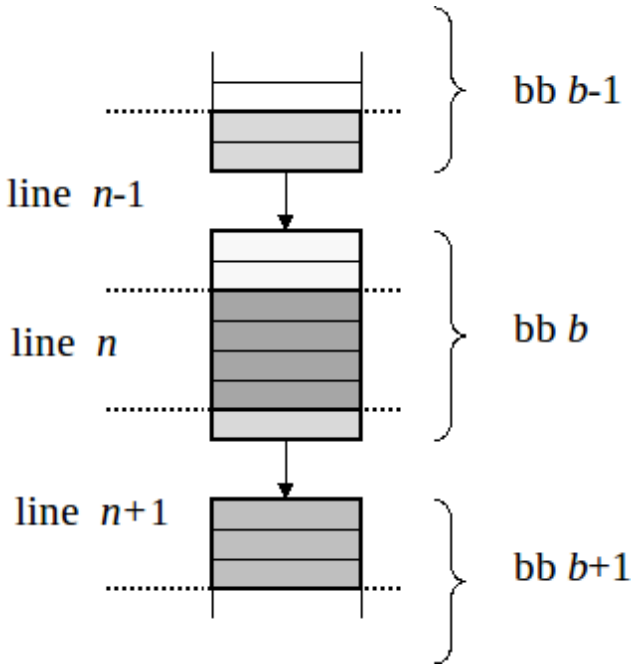
- First part of the low level analysis

Instruction cache analysis

- First part of the low level analysis
- Based on the determination of abstract cache states (ACS)
  - ACS: a set of concrete cache states possible at a given point in control flow graph (CFG)

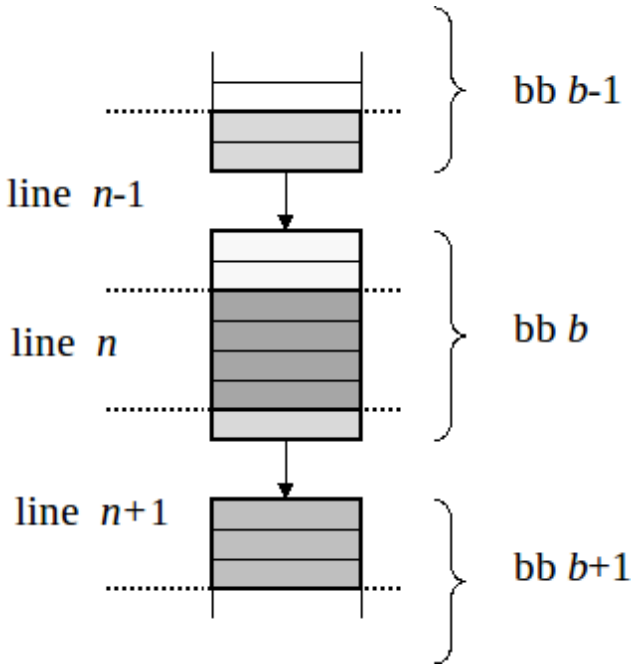
Instruction cache analysis

- First part of the low level analysis
- Based on the determination of abstract cache states (ACS)
  - ACS: a set of concrete cache states possible at a given point in control flow graph (CFG)
  - To each cache line, a set of l-blocks is associated
    - L-block: instructions of a basic block that share the same cache line
    - 2 types of l-block: *full* and *partial*



Instruction cache analysis

- First part of the low level analysis
- Based on the determination of abstract cache states (ACS)
  - ACS: a set of concrete cache states possible at a given point in control flow graph (CFG)
  - To each cache line, a set of l-blocks is associated
    - L-block: instructions of a basic block that share the same cache line
    - 2 types of l-block: *full* and *partial*



- As a result, each l-block is classified as *Always Hit*, *Always Miss*, *Persistent* or *Not Classified*

Basic block execution cost computation

- A technique based on the execution graphs that express data, control and structural dependencies between instructions

Basic block execution cost computation

- A technique based on the execution graphs that express data, control and structural dependencies between instructions
  - Much faster than the one that uses abstract interpretation, at the cost of a limited loss of accuracy

Basic block execution cost computation

- A technique based on the execution graphs that express data, control and structural dependencies between instructions
  - Much faster than the one that uses abstract interpretation, at the cost of a limited loss of accuracy
  - Possible instruction schedules are computed as a function of the state of the pipeline

Basic block execution cost computation

- A technique based on the execution graphs that express data, control and structural dependencies between instructions
  - Much faster than the one that uses abstract interpretation, at the cost of a limited loss of accuracy
  - Possible instruction schedules are computed as a function of the state of the pipeline
  - An upper bound of the execution cost is then derived from all possible schedules

Basic block execution cost computation

- A technique based on the execution graphs that express data, control and structural dependencies between instructions
  - Much faster than the one that uses abstract interpretation, at the cost of a limited loss of accuracy
  - Possible instruction schedules are computed as a function of the state of the pipeline
  - An upper bound of the execution cost is then derived from all possible schedules
- Decoupled analysis of instruction cache and pipeline is not safe if the processor isn't proved *timing-anomaly-free*

Basic block execution cost computation

- A technique based on the execution graphs that express data, control and structural dependencies between instructions
  - Much faster than the one that uses abstract interpretation, at the cost of a limited loss of accuracy
  - Possible instruction schedules are computed as a function of the state of the pipeline
  - An upper bound of the execution cost is then derived from all possible schedules
- Decoupled analysis of instruction cache and pipeline is not safe if the processor isn't proved *timing-anomaly-free*
  - A safe approach: consider all possible cache behaviors for computing the execution cost of a basic block
    - Execution graph takes into account I-block classification in fetch stage latency

Basic block execution cost computation

- A technique based on the execution graphs that express data, control and structural dependencies between instructions
  - Much faster than the one that uses abstract interpretation, at the cost of a limited loss of accuracy
  - Possible instruction schedules are computed as a function of the state of the pipeline
  - An upper bound of the execution cost is then derived from all possible schedules
- Decoupled analysis of instruction cache and pipeline is not safe if the processor isn't proved *timing-anomaly-free*
  - A safe approach: consider all possible cache behaviors for computing the execution cost of a basic block
    - Execution graph takes into account I-block classification in fetch stage latency
  - Keep the maximum value as the basic block cost

How code compression can affect estimated WCETs?

- Decompression penalty can increase execution costs

How code compression can affect estimated WCETs?

- Decompression penalty can increase execution costs
  - WCET is likely to increase but the effect is predictable and can be included in the execution graph

How code compression can affect estimated WCETs?

- Decompression penalty can increase execution costs
  - WCET is likely to increase but the effect is predictable and can be included in the execution graph
- Code compression compacts several instructions into one

How code compression can affect estimated WCETs?

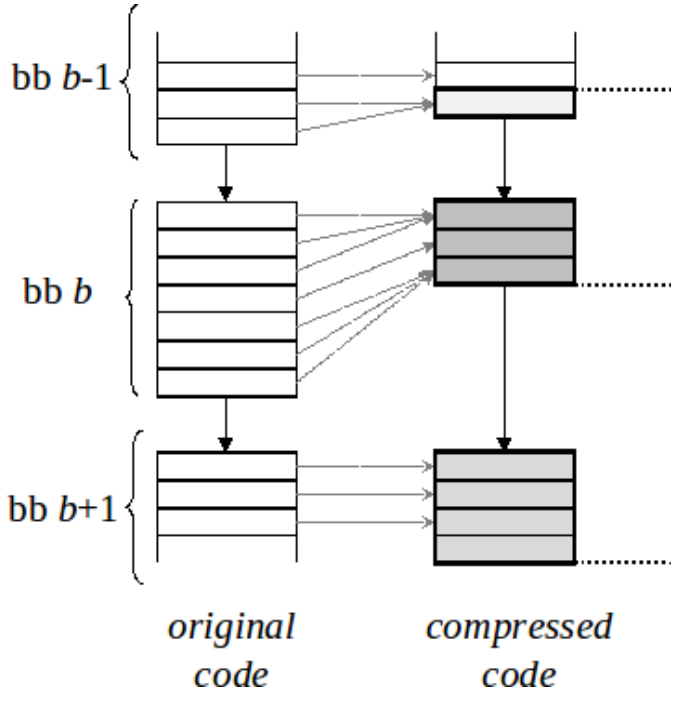
- Decompression penalty can increase execution costs
  - WCET is likely to increase but the effect is predictable and can be included in the execution graph
- Code compression compacts several instructions into one
  - Code is smaller and its placement in the cache is modified

How code compression can affect estimated WCETs?

- Decompression penalty can increase execution costs
  - WCET is likely to increase but the effect is predictable and can be included in the execution graph
- Code compression compacts several instructions into one
  - Code is smaller and its placement in the cache is modified
  - Number of full I-blocks depends on the basic block length
    - It is likely to decrease

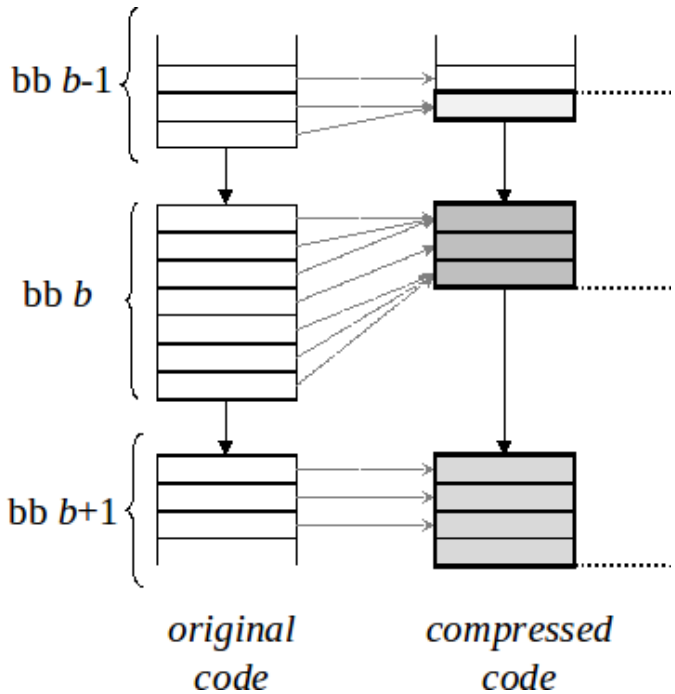
How code compression can affect estimated WCETs?

- Decompression penalty can increase execution costs
  - WCET is likely to increase but the effect is predictable and can be included in the execution graph
- Code compression compacts several instructions into one
  - Code is smaller and its placement in the cache is modified
  - Number of full I-blocks depends on the basic block length
    - It is likely to decrease
  - Number of partial I-blocks depends on the alignment of the code with respect to cache line boundaries
    - Effects on the number of partial I-blocks is unpredictable



How code compression can affect estimated WCETs?

- Decompression penalty can increase execution costs
  - WCET is likely to increase but the effect is predictable and can be included in the execution graph
- Code compression compacts several instructions into one
  - Code is smaller and its placement in the cache is modified
  - Number of full I-blocks depends on the basic block length
    - It is likely to decrease
  - Number of partial I-blocks depends on the alignment of the code with respect to cache line boundaries
    - Effects on the number of partial I-blocks is unpredictable

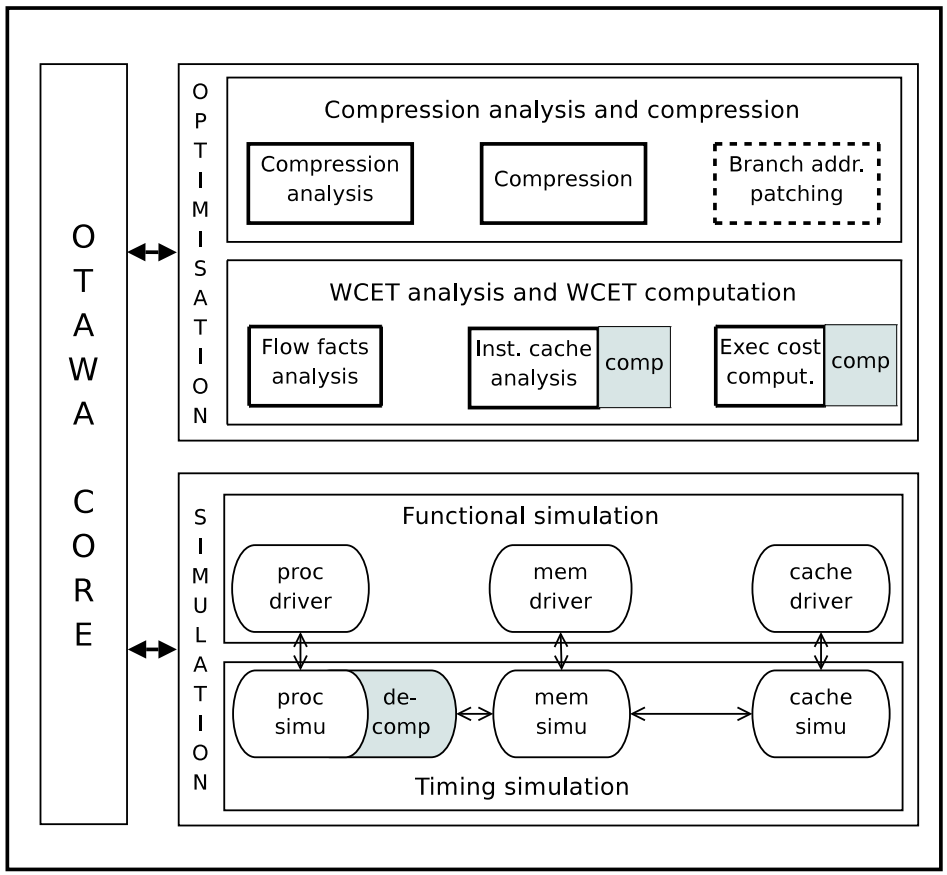


- Effects on the estimated WCETs are hard to predict

Current Section

OTAWA

- OTAWA: Framework for code analysis and simulation (IRIT)
  - Library providing a series of tools for WCET analysis
  - Cycle-level simulator built on the SystemC library



Implementation of WCET analysis

- Since code compression has no impact on flow facts, our goal is to analyze the accuracy of the cache and pipeline analysis

Implementation of WCET analysis

- Since code compression has no impact on flow facts, our goal is to analyze the accuracy of the cache and pipeline analysis
  - $\Rightarrow$ Estimated WCETs are computed from flow information *determined by profiling*

Implementation of WCET analysis

- Since code compression has no impact on flow facts, our goal is to analyze the accuracy of the cache and pipeline analysis
  - $\Rightarrow$ Estimated WCETs are computed from flow information *determined by profiling*
    - Better suited to analyze the effects of code compression
    - Input data may **not** drive the execution on the longest path

Implementation of WCET analysis

- Since code compression has no impact on flow facts, our goal is to analyze the accuracy of the cache and pipeline analysis
  - $\Rightarrow$ Estimated WCETs are computed from flow information *determined by profiling*
    - Better suited to analyze the effects of code compression
    - Input data may **not** drive the execution on the longest path
- WCET is computed considering observed execution counts and estimated execution costs:

Implementation of WCET analysis

- Since code compression has no impact on flow facts, our goal is to analyze the accuracy of the cache and pipeline analysis
  - $\Rightarrow$ Estimated WCETs are computed from flow information *determined by profiling*
    - Better suited to analyze the effects of code compression
    - Input data may **not** drive the execution on the longest path
- WCET is computed considering observed execution counts and estimated execution costs:
  - $WCET = \sum x_{i,j}.c_{i,j}$
    - $x_{i,j}$ is the execution count of two-block sequence $b_i - b_j$
    - $c_{i,j}$ is the maximum cost of block bj in sequence $b_i - b_j$
    - For sequences that include *Persistent* l-blocks, the accuracy is improved by distinguishing between the cost at the first loop iteration and the cost at other iterations

Base configuration

- Processor
  - Two way super scalar
  - In order execution
  - No branch prediction
  - Perfect data cache
  - 10-cycle instruction cache miss penalty
- Compression
  - 75% of the dictionary is filled with most executed instructions
    - $\Rightarrow$ Code compression improves not only code size but also execution time and energy consumption

Benchmarks

- From Mälardalen University website
  - `adpcm`: adaptive differential pulse code modulation
  - `crc`: cyclic redundancy check
  - `compress`: data compression
  - `matmul`: matrix multiplication
  - `nsichneu`: simulation of a Petri net

Benchmarks

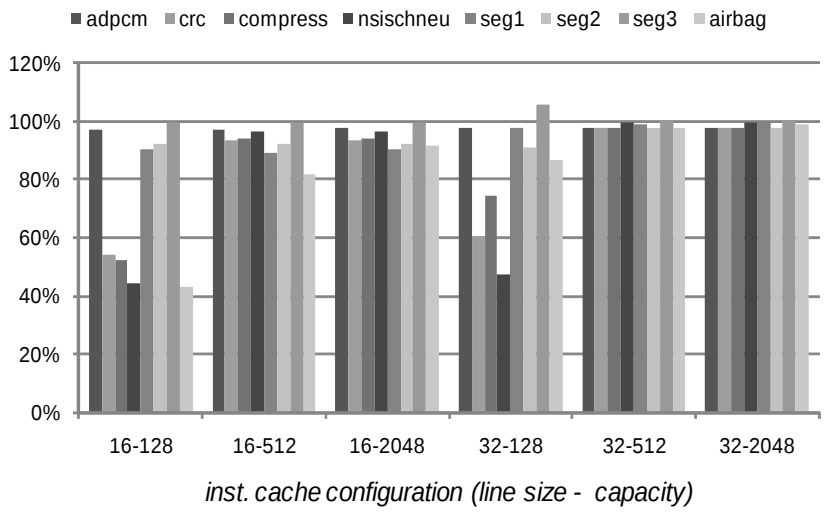
- From Mälardalen University website
  - `adpcm`: adaptive differential pulse code modulation
  - `crc`: cyclic redundancy check
  - `compress`: data compression
  - `matmul`: matrix multiplication
  - `nsichneu`: simulation of a Petri net
- Developed by IRIT
  - `seg`: image segmentation; includes 3 tasks that are considered as 3 benchmarks
    - `seg1`: finds regions of adjacent similar pixels in the image
    - `seg2`: fuses adjacent regions
    - `seg3`: fuses pixels that belong to fused regions
  - `airbag`: airbag control software

Current Section

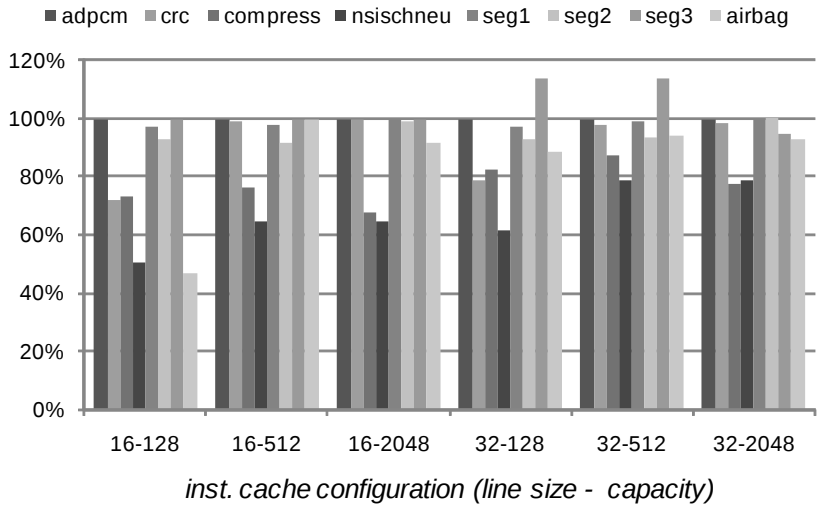
Code size

Benchmark	Compression rate	Compression rate incl. dictionary	Compression rate of the main function
adpcm	19.2%	9.5%	21.7%
crc	24.5%	10.4%	53.0%
compress	22.1%	10.1%	33.4%
nsichneu	29.1%	18.4%	43.6%
seg	18.5%	11.3%	n/a
seg1	n/a	n/a	3.0%
seg2	n/a	n/a	2.3%
seg3	n/a	n/a	4.1%
airbag	31.8%	25.1%	42.8%

Observed and estimated worst case execution time

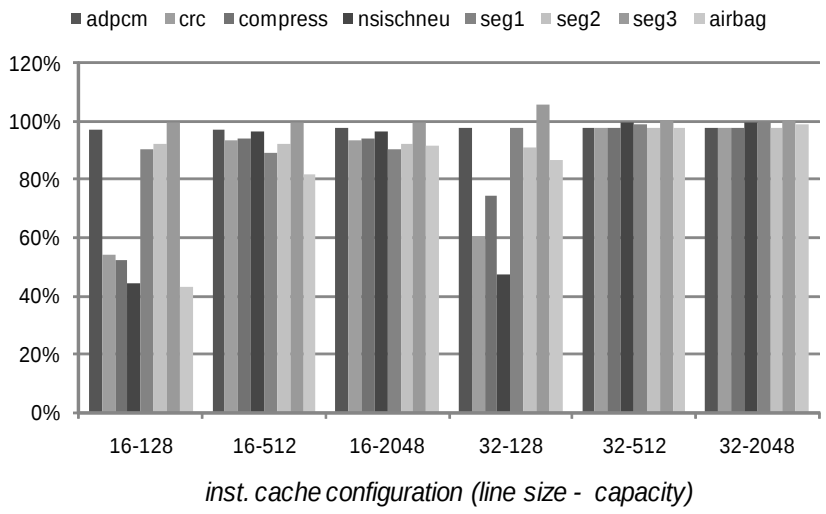


- Observed execution time

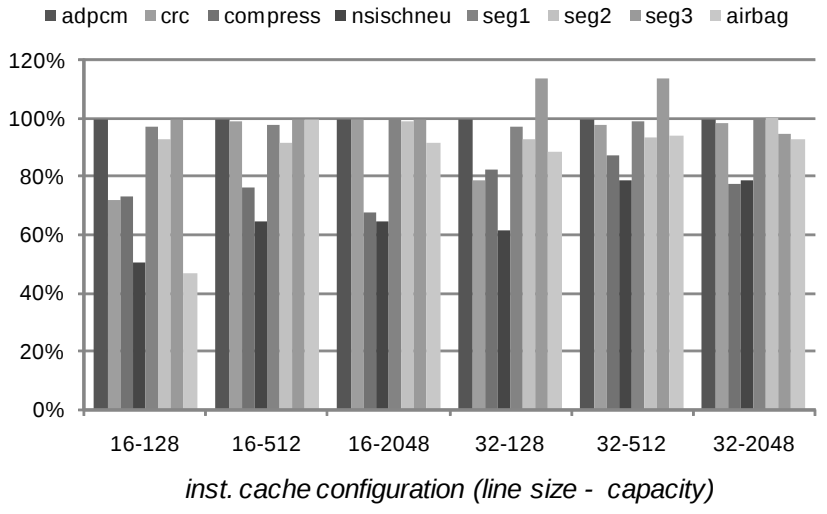


- Worst-case execution time

Observed and estimated worst case execution time

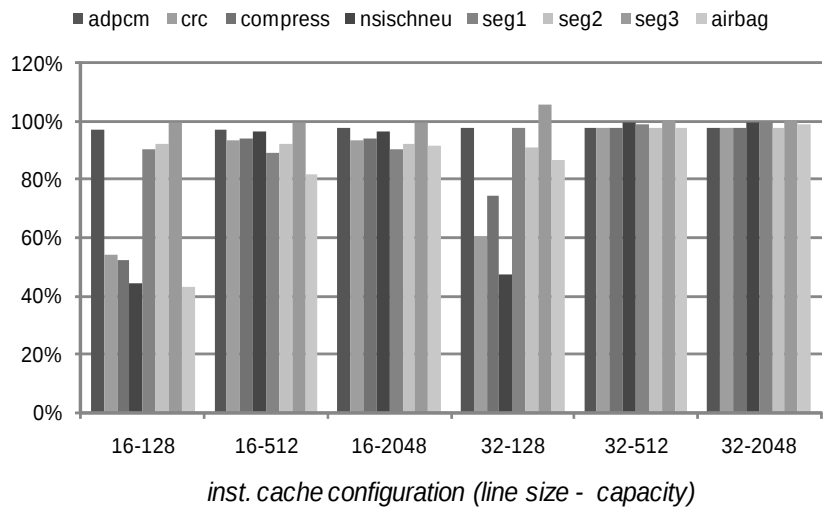


- Observed execution time
  - Decreased particularly for small caches
    - $\Rightarrow$ Reduced cache miss

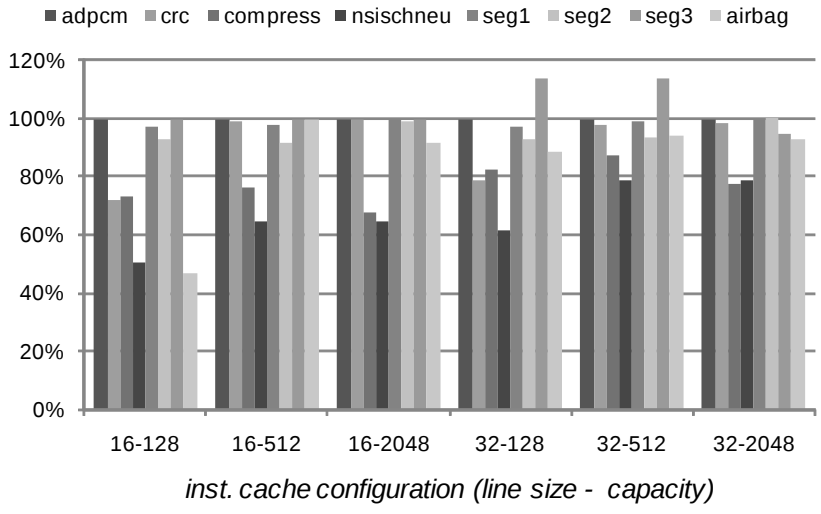


- Worst-case execution time

Observed and estimated worst case execution time

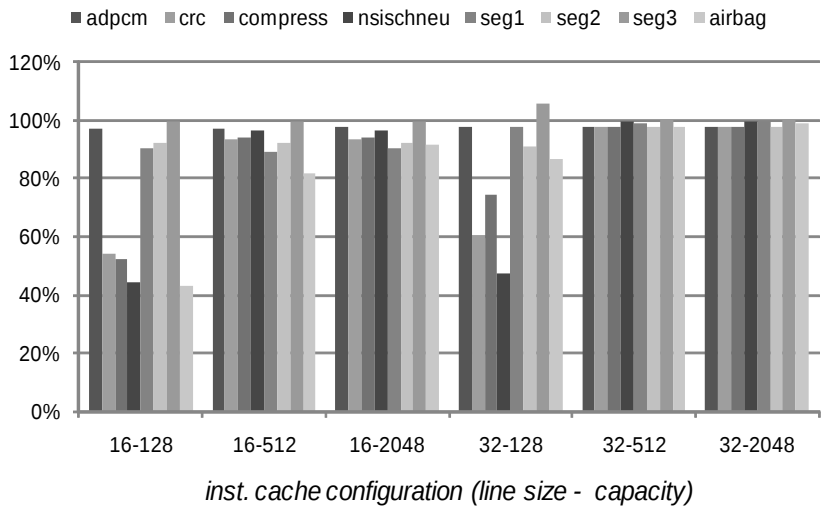


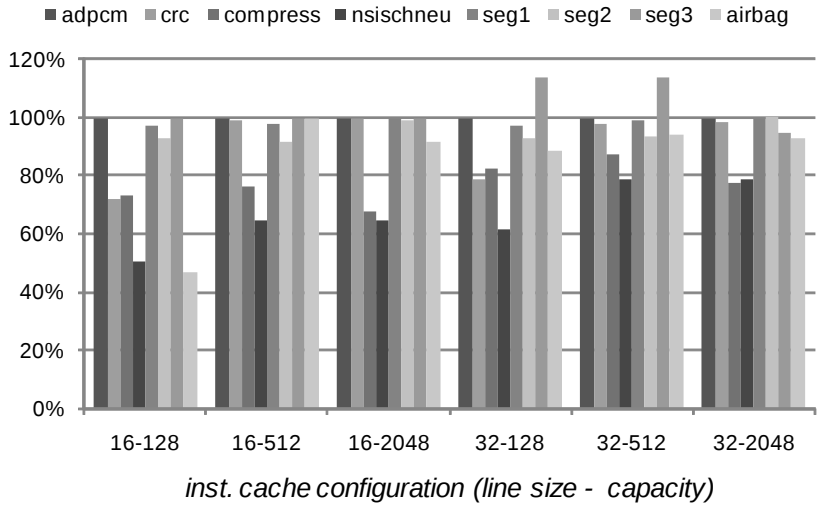
- Observed execution time
  - Decreased particularly for small caches
    - ⇒ Reduced cache miss
  - Increased in one case
    - ⇒ Modified code alignment increased cache accesses



- Worst-case execution time

Observed and estimated worst case execution time

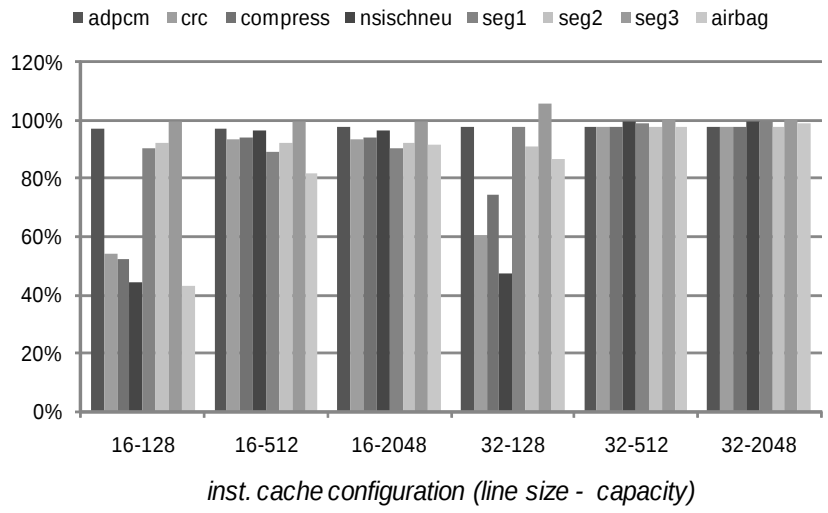


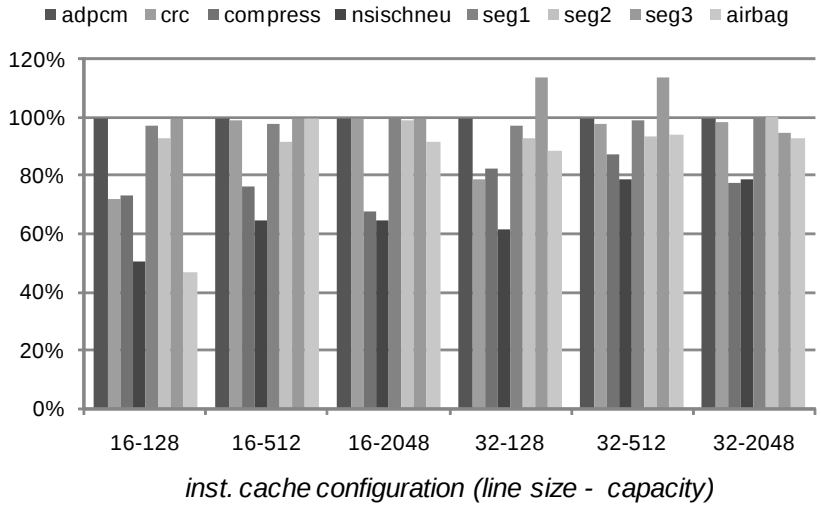


- Observed execution time
  - Decreased particularly for small caches
    - ⇒ Reduced cache miss
  - Increased in one case
    - ⇒ Modified code alignment increased cache accesses

- Worst-case execution time
  - Improved less (more) than observed execution time for small (large) caches

Observed and estimated worst case execution time





- Observed execution time
  - Decreased particularly for small caches
    - $\Rightarrow$ Reduced cache miss
  - Increased in one case
    - $\Rightarrow$ Modified code alignment increased cache accesses

- Worst-case execution time
  - Improved less (more) than observed execution time for small (large) caches
    - $\Rightarrow$ Corresponds to a decrease (increase) of the WCET estimation accuracy

Observed and estimated worst case execution time

	cache size (bytes)					
	line = 16 bytes			line = 32 bytes		
	128	512	2048	128	512	2048
observed	26.0%	7.0%	5.6%	16.0%	1.5%	1.2%
WCET	19.2%	8.6%	8.7%	10.0%	4.4%	6.4%

$\Rightarrow$ WCET estimation accuracy is increased except for very small caches

Observed and estimated worst case execution time

	cache size (bytes)					
	line = 16 bytes			line = 32 bytes		
	128	512	2048	128	512	2048
observed	26.0%	7.0%	5.6%	16.0%	1.5%	1.2%
WCET	19.2%	8.6%	8.7%	10.0%	4.4%	6.4%

$\Rightarrow$ WCET estimation accuracy is increased except for very small caches

- With a perfect instruction cache, estimated WCET of the compressed code is almost the same as the original code

Observed and estimated worst case execution time

	cache size (bytes)					
	line = 16 bytes			line = 32 bytes		
	128	512	2048	128	512	2048
observed	26.0%	7.0%	5.6%	16.0%	1.5%	1.2%
WCET	19.2%	8.6%	8.7%	10.0%	4.4%	6.4%

⇒ WCET estimation accuracy is increased except for very small caches

- With a perfect instruction cache, estimated WCET of the compressed code is almost the same as the original code
  ⇒ Accuracy of the instruction cache analysis is increased

Observed and estimated worst case execution time

	cache size (bytes)					
	line = 16 bytes			line = 32 bytes		
	128	512	2048	128	512	2048
observed	26.0%	7.0%	5.6%	16.0%	1.5%	1.2%
WCET	19.2%	8.6%	8.7%	10.0%	4.4%	6.4%

$\Rightarrow$ WCET estimation accuracy is increased except for very small caches

- With a perfect instruction cache, estimated WCET of the compressed code is almost the same as the original code
  - $\Rightarrow$ Accuracy of the instruction cache analysis is increased
- For `nsichneu` whose estimated WCET is improved while observed time isn't improved for larges caches
  - Number of *Always Miss* I-blocks is cut by about 45%
  - Number of *Not Classified* I-blocks is cut by 50% to 70%

Current Section

Conclusion and future work

- Code compression reduces instruction cache misses while in-pipeline decompression hides decompression penalty by pipelined execution
  - Improvement on observed and worst case execution times

Conclusion and future work

- Code compression reduces instruction cache misses while in-pipeline decompression hides decompression penalty by pipelined execution
  - Improvement on observed and worst case execution times
- Code compression changes repartition and classification of I-blocks which impacts instruction cache analysis.

Conclusion and future work

- Code compression reduces instruction cache misses while in-pipeline decompression hides decompression penalty by pipelined execution
  - Improvement on observed and worst case execution times
- Code compression changes repartition and classification of I-blocks which impacts instruction cache analysis.
  - In most of the cases, improvement on the estimated WCET is more significant than on the observed time

  $\Rightarrow$ Code compression results to a more accurate cache analysis in most of the cases

Conclusion and future work

- Code compression reduces instruction cache misses while in-pipeline decompression hides decompression penalty by pipelined execution
  - Improvement on observed and worst case execution times
- Code compression changes repartition and classification of I-blocks which impacts instruction cache analysis.
  - In most of the cases, improvement on the estimated WCET is more significant than on the observed time

  $\Rightarrow$ Code compression results to a more accurate cache analysis in most of the cases

- Future work
  - Take into account WCET related information during compression in order to further increase accuracy of the cache analysis

FIN

THANK YOU !!!

Questions?