Leveraging LLVM’s ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau
Jan Reineke

This work has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 101020415)
Cache Analysis is Important
Cache Analysis is Important

\[ x = a + b \]

LOAD r2, _a
LOAD r1, _b
ADD r3, r2, r1
Cache Analysis is Important

\[ x = a + b \]

LOAD r2, _a
LOAD r1, _b
ADD r3, r2, r1

Motorola PowerPC 755

<table>
<thead>
<tr>
<th>Exec. Time (Clock Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
</tr>
<tr>
<td>Worst Case</td>
</tr>
</tbody>
</table>

0
100
200
300
Instruction Cache Analysis

Source program

```java
while (x < 10) {
    x++; //a
    if (x < 5)
        x++; //b
    else
        y--; //c
}
```
Instruction Cache Analysis

Source program

```c
while (x < 10) {
    x++; //a
    if (x < 5)
        x++; //b
    else
        y--; //c
}
```

Binary program
**Instruction Cache Analysis**

**Source program**

```java
while (x < 10) {
    x++;
    //a
    if (x < 5)
    x++;
    //b
    else
    y--;  //c
}
```

**Control-flow graph**

```
 a
```

**LLVM**

**Binary program**

```
000000 0000 0000 0000 0001 0010 0100 0010 0001 0000 0010 0100
000000 0000 0000 0000 0011 0010 0100 0010 0010 0000 0010 0100
000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

**Abstraction**
Instruction Cache Analysis

Source program

```
while (x < 10) {
    x++;  //a
    if (x < 5)
        x++;  //b
    else
        y--;  //c
}
```

Control-flow graph

Classification:
- "always hit"
- "always miss"
- "unknown"

Abstraction

LLVM

Binary program
Challenges in Data Cache Analysis I

Source program

```c
int A[100];
for (int x = 0; x < 100; x++)
    sum += A[x]
for (int y = 99; y >= 0; y--)
    sum -= A[y]
```
Challenges in Data Cache Analysis I

Source program

```c
int A[100];
for (int x = 0; x < 100; x++)
    sum += A[x]
for (int y = 99; y >= 0; y--)
    sum -= A[y]
```

Control-flow graph

LLVM + Abstraction
Challenges in Data Cache Analysis I

Source program

```c
int A[100];
for (int x = 0; x < 100; x++)
    sum += A[x]
for (int y = 99; y >= 0; y--)
    sum -= A[y]
```

Addresses depend on loop iteration
Challenges in Data Cache Analysis I

Source program

```c
int A[100];
for (int x = 0; x < 100; x++)
    sum += A[x]
for (int y = 99; y >= 0; y--)
    sum -= A[y]
```

Addresses depend on loop iteration

Control-flow graph

LLVM + Abstraction

Cannot express dependence of addresses on iteration!
First Contribution: Symbolic Control-Flow Graphs

Source program

```c
int A[100];
for (int x = 0; x < 100; x++)
    sum += A[x]
for (int y = 99; y >= 0; y--)
    sum -= A[y]
```

Addresses depend on loop iteration
First Contribution: Symbolic Control-Flow Graphs

Source program

\begin{verbatim}
int A[100];
for (int x = 0; x < 100; x++)
    sum += A[x];
for (int y = 99; y >= 0; y--)
    sum -= A[y];
\end{verbatim}

Symbolic CFG

Addresses depend on loop iteration

LLVM + Abstraction

entry_i

\begin{itemize}
    \item backedge_i
    \item assume_{i,100}
\end{itemize}

A[i]

entry_j

\begin{itemize}
    \item backedge_j
    \item assume_{j,100}
\end{itemize}

A[99 - j]
First Contribution: Symbolic Control-Flow Graphs

Source program

```c
int A[100];
for (int x = 0; x < 100; x++)
    sum += A[x];
for (int y = 99; y >= 0; y--)
    sum -= A[y];
```

Symbolic CFG

Addresses depend on loop iteration

Captures dependence of addresses on loop iteration!
Symbolic CFG

Semantics

\[
\text{Symbolic CFG:}
\]

\[
\text{Semantics:}
\]

\[
A[i] 
\]

\[
A[99 - j]
\]

\[
\text{entry}_i
\]

\[
\text{entry}_j
\]

\[
\text{assume}_i,100
\]

\[
\text{assume}_j,100
\]

\[
\text{backedge}_i
\]

\[
\text{backedge}_j
\]
Symbolic CFG

Semantics

Loop variables capture iteration counts, here \( i \) and \( j \).
Symbolic CFG

Semantics

Loop variables capture iteration counts, here $i$ and $j$.

Three ways to manipulate variables:
Symbolic CFG

Semantics

Loop variables capture iteration counts, here $i$ and $j$.

Three ways to manipulate variables:

- $entry_i$ reset variable $i$ to 0
Symbolic CFG

Semantics

Loop variables capture iteration counts, here $i$ and $j$.

Three ways to manipulate variables:

- $\text{entry}_i$ reset variable $i$ to 0
- $\text{backedge}_i$ increment variable $i$
- $\text{entry}_j$ reset variable $j$ to 0
- $\text{backedge}_j$ increment variable $j$

Semantics

Loop variables capture iteration counts, here $i$ and $j$.

Three ways to manipulate variables:

- $\text{entry}_i$ reset variable $i$ to 0
- $\text{backedge}_i$ increment variable $i$
- $\text{entry}_j$ reset variable $j$ to 0
- $\text{backedge}_j$ increment variable $j$
Loop variables capture iteration counts, here $i$ and $j$.

Three ways to manipulate variables:

- **entry$_i$**: reset variable $i$ to 0
- **backedge$_i$**: increment variable $i$
- **assume$_i,e$**: can only take edge if variable $i$ is equal to expression $e$
Symbolic CFG

Semantics

Loop variables capture iteration counts, here $i$ and $j$.

Three ways to manipulate variables:

- **entry**$_i$: reset variable $i$ to 0
- **entry**$_j$: increment variable $i$
- **assume**$_{i,e}$: can only take edge if variable $i$ is equal to expression $e$

Addresses of memory accesses captured as polynomial expressions of loop variables.
Symbolic CFG

Semantics
Loop variables capture iteration counts, here \( i \) and \( j \).

Three ways to manipulate variables:

- \( \text{entry}_i \) reset variable \( i \) to 0
- \( \text{backedge}_i \) increment variable \( i \)
- \( \text{assume}_{i,e} \) can only take edge if variable \( i \) is equal to expression \( e \)

Addresses of memory accesses captured as polynomial expressions of loop variables.

Obtained from LLVM’s ScalarEvolution Analysis Pass
Symbolic Control-Flow Graphs and Cache Analysis

Symbolic CFG

Example
Symbolic CFG

Example

First loop:
Symbolic CFG

Example

First loop:
i = 0, A[0]
Symbolic CFG

Example

First loop:
i = 0, A[0]  
i = 1, A[1]

\[\text{Symbolic Control-Flow Graphs and Cache Analysis}\]

ScalarEvolution Analysis [6], [7].

Symbolic CFGs are our formalization of the output of LLVM's analysis, because the link between loop iterations and accessed data is of no use here.

Memory blocks is lost. Thus, our first step towards accurate analysis could then be employed successfully on this exploded representation.

While this abstraction is adequate for instruction execution of the program, let us arrive at the symbolic cache state depicted in Figure 2c.

Assume that loop variables other than themselves. An edge annotated with a concrete memory access. Ferdinand's must exploit this information. Simply applying Ferdinand's must cache states. A relatively straightforward approach would be to virtually unroll the loops for the sake of the analysis, resulting in at most one cache miss. However, in our example, the symbolic CFG perfectly captures the sequence of memory accesses generated by the program.

In fact, in our example, the symbolic CFG perfectly captures the sequence of memory accesses more precisely than plain CFGs. In fact, in our example, the symbolic CFG perfectly captures the sequence of memory accesses more precisely than plain CFGs.
Symbolic Control-Flow Graphs and Cache Analysis

A Symbolic Control-Flow Graph (CFG) is a formalization of the output of LLVM's ScalarEvolution Analysis \[6\], \[7\].

Symbolic CFGs capture the link between loop iterations and accessed data. To express such information, symbolic CFGs may annotate edges with expressions that are equal to the values of loop variables. For example, if we have two loops, the symbolic state will be reached at the beginning of the first loop, where each loop is annotated accordingly with an `assume` statement. The symbolic state will then be reached at the entry to the second loop, where each loop is annotated with a `backedge`.

First loop:
- `i = 0, A[0]`
- `i = 1, A[1]`
- ... `i = 99, A[99]`

In our example, the symbolic CFG perfectly captures the sequence of memory accesses in terms of the loop variables. For instance, if the entire set of memory blocks that can potentially be accessed does not fully fit into the cache, and so persistence analysis \[16\], \[17\], \[18\], \[19\], \[15\] may deduce that each of these blocks is lost. Thus, our first step towards accurate data cache analysis is to employ what we coin symbolic data cache analysis.
Symbolic CFG

Example

First loop:
i = 0, A[0]  i = 1, A[1]  …  i = 99, A[99]

Second loop:
Symbolic CFG

Example

First loop:
i = 0, A[0]   i = 1, A[1]   …   i = 99, A[99]

Second loop:
j = 0, A[99]
Symbolic Control-Flow Graphs and Cache Analysis

We define symbolic CFGs in Section IV. There we also discuss multivariate chains of recurrences [8], [9], [20], which are used to represent access expressions and loop bounds.

To this end, our first basic idea are symbolic cache states that capture how cache states depend on the loop iteration. To motivate this approach, consider Figures 2a and 2b, which represent the states from Figures 1a and 1b in an exploded plain CFG in which each edge could once more be annotated with a concrete memory access. Ferdinand’s must exploit this information. Simply applying Ferdinand’s must analysis determines upper bounds on the ages of memory blocks. However, instead of associating bounds with concrete memory accesses, our approach associates upper bounds with symbolic memory accesses.

Symbolic CFGs are useful for data cache analysis as the same instructions are accessed in each loop iteration, it is inadequate for data cache analysis, as the link between loop iterations and accessed memory addresses is lost. Thus, our first step towards accurate data cache analysis is of no use here.

Symbolic CFGs are our formalization of the output of LLVM’s ScalarEvolution Analysis [6], [7].

First loop:

\[
i = 0, A[0] \quad i = 1, A[1] \quad \ldots \quad i = 99, A[99]
\]

Second loop:

\[
j = 0, A[99] \quad j = 1, A[98]
\]
Symbolic CFG

Example

First loop:
- i = 0, A[0]
- i = 1, A[1]
- ... i = 99, A[99]

Second loop:
- j = 0, A[99]
- j = 1, A[98]
- ... j = 99, A[0]
Symbolic Control-Flow Graphs and Cache Analysis

Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively
Symbolic Control-Flow Graphs and Cache Analysis

A. Symbolic Control-Flow Graphs

B. Symbolic Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

\[ i = 0, A[0] \quad i = 1, A[1] \quad i = 2, A[2] \quad i = 3, A[3] \quad \ldots \]
may refer to loop variables other than contain entry to exit. To express such information, symbolic CFGs may have been taken. Then, the accessed address is that with is the identifier of a loop. Consider the edge annotated by annotating edges with new loop iteration begins. These transitions are indicated CFGs make it explicit when a loop is entered and when a iterations of their enclosing loops. To this end, symbolic of memory accesses are expressed in terms of the loop program. In a symbolic CFG—where possible—the addresses captures the link between loop iterations and accessed data. a simple yet powerful program representation that concisely alysis, because the link between loop iterations and accessed cache states. A relatively straightforward approach would be to exploit this information. Simply applying Ferdinand’s must analysis determines upper bounds on the ages of memory seek a precise analysis whose runtime is independent of particular for programs with large loop bounds. We are thus cache states. A relatively straightforward approach would be to assume that symbolic cache states, consider Figures 2a and 2b, which are used to represent access expressions and loop bounds. In our example, the back edges of both loops are taken exactly 100 times, and so the exit edges of the symbolic cache state depicted in Figure 2c. Furthermore, the mapping to that block. Our idea is to represent memory blocks array. We represent each memory block by the first array cell cache lines of size 8. This is to be interpreted as follows: In an Figure 1c shows a symbolic CFG for our example. If the entire set of memory blocks that can potentially be accessed fits into the cache, then persistence analysis may deduce that each of these blocks does not fully fit into the cache, and so persistence analysis could then be employed successfully on this exploded plain CFG. However, this approach would be very costly, in seeking a precise analysis whose runtime is independent of the plain CFG. However, this approach would be very costly, in virtually unroll the loops for the sake of the analysis, resulting assumptions.

**Assumptions:**
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

\[ \begin{array}{c}
\text{Assume } i = 0, A[0] \\
\text{Assume } j = 0, A[99 - j] \\
\text{Assume } i = 1, A[1] \\
\text{Assume } j = 1, A[99 - j] \\
\text{Assume } i = 2, A[2] \\
\text{Assume } j = 2, A[99 - j] \\
\text{Assume } i = 3, A[3] \\
\text{...}
\end{array} \]
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>A[0]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A[0]</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

entry_i → A[i] → backedge_i → entry_i

entry_j → A[99 – j] → backedge_j → entry_j

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Figure 1: Simple program and its plain and symbolic control-flow-graph abstractions.
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

Symbols:
- entry
- backedge
- assume
- A

Assumptions:
- i
- j
- 100
- 99
- 0
- 1
- 2
- 3
- Cache Hit
- Spatial Locality
Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Cache Analysis: Intuitively


A[0]  A[0]

Cache Hit Spatial Locality
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively
Symbolic Control-Flow Graphs are our formalization of the output of LLVM's scalar evolution analysis [6], [7]. Symbolic CFGs make it explicit when a loop is entered and when a new loop iteration begins. These transitions are indicated by annotating edges with assume statements, where

\begin{align*}
\text{assume}_{i,100} & \quad \text{entry}_i \\
\text{assume}_{j,100} & \quad \text{entry}_j
\end{align*}

Furthermore, the symbolic state will be reached at the end of each odd loop iteration, starting from iteration 1, i.e.,

\begin{align*}
A[0] & \quad \text{if } i = 0, A[0] \\
A[0] & \quad \text{if } i = 1, A[1] \\
A[0] & \quad \text{if } i = 2, A[2] \\
\vdots & \quad \text{if } i = 3, A[3]
\end{align*}

### Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A[0]</td>
<td>A[0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A[0]</td>
</tr>
</tbody>
</table>

Figure 1c shows a symbolic CFG for our example.
Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

- Symbolic CFG
- Cache Analysis:
  - Intuitively
  - Symbolic Cache Analysis:
    - Intuitively
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

- 
- 

i = 0, A[0]

- 
A[0]

i = 1, A[1]

- 
A[0]

i = 2, A[2]

A[0]

A[0]

i = 3, A[3] ...

A[2]

A[2]

A[2]

A[0]

A[0]

A[0]

A[0]
Symbolic CFG

Cache Analysis: Intuitively

**Assumptions:**
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Hit Temporal Locality

Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

Temporal Locality

Cache Hit

A[98]
A[96]

A[0]
A[0]
A[0]
A[0]
A[2]
A[2]
A[2]
A[2]

8
ScalarEvolution Analysis \[6\], \[7\].

C. Symbolic Control-Flow Graphs and Cache Analysis

Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

|-------|----------|-------------|-------------|-------------|-------------|

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

ScalarEvolution Analysis [6], [7].

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively
Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

Symbolic Control-Flow Graphs and Cache Analysis

A simple yet powerful program representation that concisely captures the link between loop iterations and accessed data.

Data cache analysis is of no use here. Because the link between loop iterations and accessed memory accesses in the program to be cache hits or misses.

We have seen that symbolic CFGs are useful for data cache analysis as they capture a program’s memory access behavior more precisely than plain CFGs. In fact, in our analysis, we employ what we coin symbolic data cache states. To motivate symbolic cache states, consider Figures 2a and 2b, which show the concrete cache states at the ends of iterations of memory accesses generated by the program.

It remains to define a static analysis that can efficiently exploit this information. Simply applying Ferdinand’s must analysis our symbolic data cache states, consider Figures 2a and 2b, which represent the states from Figures 2a and 2b in this way.

To this end, our first basic idea are symbolic cache states that symbolically contain the number of times that a loop’s back edges are taken from entry to exit. To express such information, symbolic CFGs may refer to loop variables other than a simple yet powerful program representation that concisely captures the link between loop iterations and accessed data.

ScalarEvolution Analysis [6], [7].

Symbolic CFGs are our formalization of the output of LLVM’s C. Symbolic Control-Flow Graphs and Cache Analysis

A relatively straightforward approach would be to virtually unroll the loops for the sake of the analysis, resulting in an exploded plain CFG in which each edge could once more be annotated with a concrete memory access. Ferdinand’s must analysis, however, this approach would be very costly, in particular for programs with large loop bounds. We are thus seeking a precise analysis whose runtime is independent of the loop bounds of the program.
ScalarEvolution Analysis [6], [7].

Symbolic CFGs are our formalization of the output of LLVM’s analysis, because the link between loop iterations and accessed data is captured by symbolic CFGs. When analyzing cache states, symbolic CFGs make it explicit when a loop is entered and when a new loop iteration begins. These transitions are indicated by annotating edges with assume statements, where the identifier of a loop is the symbol for the edge.

ScalarEvolution Analysis [6], [7]. The symbolic cache state is the abstraction that suits the link between the loop iteration and the accessed address.

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

- A[0] -
A[0] - A[0]


Fig. 1: Simple program and its plain and symbolic control-flow-graph abstractions.

ScalarEvolution Analysis [6], [7].

Symbolic CFGs are our formalization of the output of LLVM's data cache analysis to employ what we coin the plain CFG abstraction is inadequate for data cache analysis, as the link between loop iterations and accessed addresses is lost. As a consequence, it is impossible to predict any of the memory accesses generated by the program to be cache hits or misses. When this abstraction is inadequate for data cache analysis, we define symbolic CFGs in Section IV. There we also discuss multivariate chains of recurrences [8], [9], [20], which may deduce that each of these blocks of memory accesses are expressed in terms of the loop variables. For some loops, ScalarEvolution is also able to derive the exact number of times that a loop's back edges are taken from execution of the program, let \( i \) be the identifier of a loop. Consider the edge annotated by annotating edges with new loop iteration begins. These transitions are indicated in a symbolic CFG—where possible—the addresses of memory accesses are expressed in terms of the loop iteration, it is inadequate for data cache analysis, as the link between the loop iteration and the accessed address is lost.
Symbolic CFG

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Hit
Spatial Locality
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Cache Analysis: Intuitively
C. Symbolic Control-Flow Graphs and Cache Analysis

Symbolic CFGs are useful for data cache analysis as they capture a program's memory access behavior more precisely than plain CFGs. In fact, in our example, the symbolic CFG perfectly captures the sequence of memory accesses generated by the program. While this abstraction is adequate for instruction analysis, it is inadequate for data cache analysis, as the same instructions are accessed in each loop iteration. To this end, our first basic idea are symbolic cache states that symbolically capture symbolic state will be reached at the end of each odd loop iteration, starting from iteration 1.

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

- for data cache analysis as they capture a program's memory access behavior more precisely than plain CFGs. In fact, in our example, the symbolic CFG perfectly captures the sequence of memory accesses generated by the program. While this abstraction is adequate for instruction analysis, it is inadequate for data cache analysis, as the same instructions are accessed in each loop iteration. To this end, our first basic idea are symbolic cache states that symbolically capture symbolic state will be reached at the end of each odd loop iteration, starting from iteration 1.

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Challenges in Data Cache Analysis II
Assumptions:
- fully-associative cache
- least-recently-used
- 2 array cells per cache line

1. Cache states depend on loop iteration

Data Cache Analysis II

Challenges in

Symbolic CFG
Symbolic CFG

Challenges in Data Cache Analysis II

1. Cache states depend on loop iteration

Contribution: Symbolic Cache States

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic CFG

Challenges in Data Cache Analysis II

1. Cache states depend on loop iteration

Contribution: **Symbolic Cache States**

2. Behavior is phase dependent:
   - Warm-up phase:
     hits/misses depending on initial state
   - Steady-state phase:
     repetitive patterns

**Assumptions:**
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Symbolic CFG

Fig. 1: Simple program and its plain and symbolic control-flow-graph abstractions.
Symbolic CFG

Challenges in Data Cache Analysis II

1. Cache states depend on loop iteration
   Contribution: Symbolic Cache States

2. Behavior is phase dependent:
   - Warm-up phase:
     hits/misses depending on initial state
   - Steady-state phase:
     repetitive patterns

Contribution: Context-sensitive Analysis

Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line
Symbolic Cache States
Symbolic Cache States

First loop:
Symbolic Cache States

First loop: \( i = 0, A[i] \) \( i = 1, A[i] \) \( i = 2, A[i] \) \( i = 3, A[i] \) \( \ldots \) \( i = 100 \)
## Symbolic Cache States

**First loop:**

<table>
<thead>
<tr>
<th>i</th>
<th>A[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

*Note: The values A[i] are placeholders for the actual cache state values.*
Symbolic Cache States

First loop:

\[ \begin{array}{ccccccc}
\text{ } & i = 0, A[i] & i = 1, A[i] & i = 2, A[i] & i = 3, A[i] & \ldots & i = 100 \\
\end{array} \]
Symbolic Cache States

First loop:

- $i = 0, A[i]$
- $i = 1, A[i]$
- $i = 2, A[i]$
- $i = 3, A[i]$
- ... $i = 100$
Symbolic Cache States

First loop: $i = 0, A[i]$  $i = 1, A[i]$  $i = 2, A[i]$  $i = 3, A[i]$  ...  $i = 100$

Cache Hit Spatial Locality
### Symbolic Cache States

**First loop:**

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

- 
  - 
  -  
  -  
  -  
  -  

Cache Hit
Spatial Locality
Symbolic Cache States

First loop:

\[
\begin{align*}
\text{i = 0, } & A[i] \\
\text{i = 1, } & A[i] \\
\text{i = 2, } & A[i] \\
\text{i = 3, } & A[i] \\
\text{...} & \\
\text{i = 100} & \\
\end{align*}
\]
Symbolic Cache States

First loop: $i = 0, A[i]$

$A[i-1]$

$A[i]$

$A[i-2]$

$i = 1, A[i]$

$A[i]$

$A[i-1]$

$i = 2, A[i]$

$A[i]$

$A[i-2]$

$i = 3, A[i]$

...$

$i = 100$
Symbolic Cache States

First loop:

\[
\begin{align*}
&i = 0, A[i] \\
&i = 1, A[i] \\
&i = 2, A[i] \\
&i = 3, A[i] \\
&\ldots \\
&i = 100
\end{align*}
\]
Symbolic Cache States

First loop: i = 0, A[i]  
A[i] - 

i = 1, A[i]  
A[i-1] - 

i = 2, A[i]  
A[i-2] - 

i = 3, A[i]  
A[i-1] 

...  

i = 100  
A[i-2]
Symbolic Cache States

First loop:

- i = 0, A[i]
- i = 1, A[i]
A[i] - i = 2, A[i]
A[i-1] - i = 3, A[i]
A[i-1] - ... i = 100

Cache Hit
Spatial Locality
### Symbolic Cache States

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>A[i-1]</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A[i-2]</td>
<td>-</td>
<td>A[i-1]</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>A[i-3]</td>
<td>A[i-1]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**First loop:**

- i = 0, A[i]
- i = 1, A[i]
- i = 2, A[i]
- i = 3, A[i]
- ... i = 100

---

**Cache Hit Spatial Locality**

---

80
Symbolic Cache States

First loop:
- \(i = 0, A[i]\)
- \(i = 1, A[i]\)
- \(i = 2, A[i]\)
- \(i = 3, A[i]\)
- \(i = 100\)
Symbolic Cache States

First loop:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>A[i]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>A[i-1]</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>A[i-2]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>A[i-1]</td>
<td>A[i-3]</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A[i-2]</td>
</tr>
</tbody>
</table>

Symbolic Cache States

First loop:

- i = 0, A[i]
- i = 1, A[i]
- i = 2, A[i]
- i = 3, A[i]

Second loop:

- i = 100

A[i]
A[i-1]
A[i-2]
A[i-3]
A[i-2]
A[i-3]
A[i-4]
Symbolic Cache States

First loop:
- i = 0, A[i]
- A[i-1]
- A[i-2]
- A[i-3]
- A[i-4]

Second loop:
- j = 0, A[99-j]
- j = 1, A[99-j]
- ...
- j = 98, A[99-j]
- j = 99, A[99-j]
Symbolic Cache States

First loop:

\[
\begin{align*}
&i = 0, A[i] \\
&i = 1, A[i] \\
&A[i-1] \\
&A[i-2] \\
&A[i-3] \\
&A[i-4] \\
&i = 100
\end{align*}
\]

Second loop:

\[
\begin{align*}
&j = 0, A[99-j] \\
&j = 1, A[99-j] \\
&\ldots \\
&j = 98, A[99-j] \\
&j = 99, A[99-j]
\end{align*}
\]
Symbolic Cache States

First loop:  
\[ i = 0, A[i] \]
\[ i = 1, A[i] \]
\[ i = 2, A[i] \]
\[ i = 3, A[i] \]
\[ \ldots \]
\[ i = 100 \]

Second loop:  
\[ j = 0, A[99-j] \]
\[ j = 1, A[99-j] \]
\[ \ldots \]
\[ j = 98, A[99-j] \]
\[ j = 99, A[99-j] \]
Symbolic Cache States

First loop:
- i = 0, A[i]
- i = 1, A[i]
A[i]
A[i-1]
A[i-2]
A[i-3]
A[i-4]

Second loop:
- j = 0, A[99-j]
- j = 1, A[99-j]
A[98]
A[97]
A[96]
A[99-j]
A[99-j]

Cache Hit Temporal Locality

i = 2, A[i]
i = 3, A[i]
... i = 100
Symbolic Cache States

First loop: i = 0, A[i]  i = 1, A[i]  i = 2, A[i]  i = 3, A[i]  ...  i = 100


Cache Hit Temporal Locality
Symbolic Cache States

First loop:

\[
\begin{align*}
&i = 0, A[i] \\
&i = 1, A[i] \\
&i = 2, A[i] \\
&i = 3, A[i] \\
&\vdots \\
&i = 100
\end{align*}
\]

Second loop:

\[
\begin{align*}
&j = 0, A[99-j] \\
&j = 1, A[99-j] \\
&\vdots \\
&j = 98, A[99-j] \\
&j = 99, A[99-j]
\end{align*}
\]
Symbolic Cache States

First loop:  

\[ i = 0, A[i] \]
\[ i = 1, A[i] \]
\[ i = 2, A[i] \]
\[ i = 3, A[i] \]
\[ \ldots \]
\[ i = 100 \]

Second loop:  

\[ j = 0, A[99-j] \]
\[ j = 1, A[99-j] \]
\[ \ldots \]
\[ j = 98, A[99-j] \]
\[ j = 99, A[99-j] \]
Symbolic Cache States

First loop:

\[
\begin{align*}
&i = 0, A[i] \\
&A[i] \\
\end{align*}
\]

\[
\begin{align*}
&i = 1, A[i] \\
&A[i-1] \\
\end{align*}
\]

\[
\begin{align*}
&i = 2, A[i] \\
&A[i-2] \\
\end{align*}
\]

\[
\begin{align*}
&i = 3, A[i] \\
&A[i-1] \\
&A[i-3] \\
\end{align*}
\]

\[
\begin{align*}
&\ldots \\
&A[i-2] \\
&A[i-4] \\
\end{align*}
\]

Second loop:

\[
\begin{align*}
&j = 0, A[99-j] \\
&A[98] \\
&A[96] \\
\end{align*}
\]

\[
\begin{align*}
&j = 1, A[99-j] \\
&A[98] \\
&A[96] \\
\end{align*}
\]

\[
\begin{align*}
&\ldots \\
&A[98] \\
&A[96] \\
\end{align*}
\]

\[
\begin{align*}
&j = 98, A[99-j] \\
&A[98] \\
&A[96] \\
\end{align*}
\]

\[
\begin{align*}
&A[98] \\
&A[96] \\
\end{align*}
\]

Cache Hit
Temporal Locality
Symbolic Cache States

First loop:


  i = 0, A[i]  i = 1, A[i]  i = 2, A[i]  i = 3, A[i]  ...  i = 100


Cache Hit
Temporal Locality
# Symbolic Cache States

**First loop:**
- $i = 0, A[i]$
- $i = 1, A[i]$
- $i = 2, A[i]$
- $i = 3, A[i]$
- $i = 100$

**Second loop:**
- $j = 0, A[99-j]$
- $j = 1, A[99-j]$
- $j = 98, A[99-j]$
- $j = 99, A[99-j]$
Symbolic Cache States

First loop:  
i = 0, A[i]  i = 1, A[i]  i = 2, A[i]  i = 3, A[i]  ...  i = 100

Second loop:  
## Symbolic Cache States

1. **First loop:** 
   - For $i = 0$, $A[i]$ 
   - For $i = 1$, $A[i]$ 
   - For $i = 2$, $A[i]$ 
   - For $i = 3$, $A[i]$ 
   - ... 
   - For $i = 100$, $A[i]$ 

2. **Second loop:** 
   - For $j = 0$, $A[99-j]$ 
   - For $j = 1$, $A[99-j]$ 
   - ... 
   - For $j = 98$, $A[99-j]$ 
   - For $j = 99$, $A[99-j]$ 

Symbolic Cache States

First loop:

\[
\begin{align*}
&i = 0, A[i] \\
&i = 1, A[i] \\
&i = 2, A[i] \\
&i = 3, A[i] \\
&\vdots \\
&i = 100
\end{align*}
\]

Second loop:

\[
\begin{align*}
&j = 0, A[99-j] \\
&j = 1, A[99-j] \\
&\vdots \\
&j = 98, A[99-j] \\
&j = 99, A[99-j]
\end{align*}
\]
### Symbolic Cache States

#### First loop:
- \( i = 0, A[i] \)
- \( i = 1, A[i] \)
- \( i = 2, A[i] \)
- \( i = 3, A[i] \)
- \( i = 100 \)

#### Second loop:
- \( j = 0, A[99-j] \)
- \( j = 1, A[99-j] \)
- \( j = 98, A[99-j] \)
- \( j = 99, A[99-j] \)

---

**Cache Hit Spatial Locality**

- \( A[i-2] \)
- \( A[i-3] \)
- \( A[100-j] \)
- \( A[101-j] \)
Symbolic Cache States

First loop:
- i = 0, A[i]
- i = 1, A[i]
- i = 2, A[i]
- i = 3, A[i]
... i = 100

Second loop:
- j = 0, A[99-j]
- j = 1, A[99-j]
... j = 98, A[99-j]
- j = 99, A[99-j]

Cache Hit Spatial Locality
Symbolic Cache States

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First loop: i = 0, A[i]
Second loop: j = 0, A[99-j]

j = 0, A[99-j]

j = 98, A[99-j]

j = 99, A[99-j]
Context-sensitive Analysis

$$peel_0 \quad peel_1 \quad peel_2 \quad peel_3 \quad unroll_0 \quad unroll_1$$

$$\{0\} \quad \{1\} \quad \{2\} \quad \{3\} \quad \{4, 6, 8, \ldots \} \quad \{5, 7, 9, \ldots \}$$
Context-sensitive Analysis

Loop peeling

\[ \text{peel}_0 \rightarrow \text{peel}_1 \rightarrow \text{peel}_2 \rightarrow \text{peel}_3 \]

\{0\} \rightarrow \{1\} \rightarrow \{2\} \rightarrow \{3\} \rightarrow \{4, 6, 8, \ldots\} \rightarrow \{5, 7, 9, \ldots\}
Context-sensitive Analysis

Loop peeling

\(peel_0\)  \(peel_1\)  \(peel_2\)  \(peel_3\)

\{0\}  \{1\}  \{2\}  \{3\}

unrolling loops

\(unroll_0\)  \(unroll_1\)

\{4, 6, 8, \ldots\}  \{5, 7, 9, \ldots\}

Loop unrolling
Peeling and unrolling parameters

- influence analysis accuracy + cost
- chosen heuristically based on cache geometry + loop structure
Does it work?
Does it work?

Accuracy: Does symbolic analysis improve bounds on cache misses?
Does it work?

**Accuracy:** Does symbolic analysis improve bounds on cache misses?

**Scalability:** How does symbolic analysis runtime scale with program loop bounds?
Accuracy
Accuracy

#old misses
#new misses
Accuracy

\[
\frac{\text{#old misses}}{\text{#new misses}}
\]

time limit = 1 hour
Accuracy

#old misses
#new misses

time limit = 1 hour

PolyBench = Polyhedral Benchmark Suite
Accuracy

#old misses
#new misses

time limit = 1 hour

PolyBench = Polyhedral Benchmark Suite
Scalability

Data size
XS to XL

Analysis time
(seconds)

PolyBench = Polyhedral Benchmark Suite
Scalability

Data size
XS to XL

Analysis time (seconds)

PolyBench = Polyhedral Benchmark Suite
In our paper, we focus on leveraging LLVM’s ScalarEvolution for symbolic data cache analysis. Our approach involves formalizing the analysis as abstract interpretation, using multivariate chains of recurrences to represent symbolic expressions. The implementation is carried out on top of LLVM. We discuss related work in the context of our approach.
In our paper

Leveraging LLVM’s ScalarEvolution for Symbolic Data Cache Analysis

Wouter Smeets
Eindhoven University of Technology
wouter.smeets@tue.nl

ABSTRACT

This paper presents a method to automatically compute symbolic expressions of data cache behavior for programs written in the C programming language. The method is based on symbolic analysis of the LLVM intermediate representation, which is a data flow analysis technique. The symbolic expressions are then used to derive constraints on the data cache behavior, allowing for efficient cache optimization.

1. INTRODUCTION

Data cache behavior is an important factor in determining the performance of programs. The behavior of a program can be modeled using symbolic expressions, which can then be used to derive constraints on the data cache behavior. This paper presents a method to automatically compute symbolic expressions of data cache behavior for programs written in the C programming language.

The method is based on symbolic analysis of the LLVM intermediate representation, which is a data flow analysis technique. The symbolic expressions are then used to derive constraints on the data cache behavior, allowing for efficient cache optimization.

2. IMPLEMENTATION

The method was implemented on top of the LLVM compiler infrastructure. The symbolic expressions were computed using the symbolic execution engine of LLVM, which is a data flow analysis technique. The constraints on the data cache behavior were derived using a constraint solver, which is a system for solving symbolic expressions.

3. DISCUSSION OF RELATED WORK

Related work on symbolic analysis of the LLVM intermediate representation includes [1, 2]. These papers present methods for computing symbolic expressions of program behavior, but they do not focus on the specific application of cache behavior.

4. CONCLUSION

This paper presents a method to automatically compute symbolic expressions of data cache behavior for programs written in the C programming language. The method is based on symbolic analysis of the LLVM intermediate representation, which is a data flow analysis technique. The symbolic expressions are then used to derive constraints on the data cache behavior, allowing for efficient cache optimization.

REFERENCES


Copyright © 2023. All rights reserved. This paper is subject to the copyright laws of the United States and other countries. No part of this paper may be reproduced or transmitted in any form or by any means without written permission from the publisher.

Formalization as Abstract Interpretation

In our paper
In our paper

- Formalization as Abstract Interpretation
- Multivariate chains of recurrences to represent symbolic expressions
In our paper

- Formalization as Abstract Interpretation
- Multivariate chains of recurrences to represent symbolic expressions
- Implementation on top of LLVM
In our paper

- Formalization as Abstract Interpretation
- Multivariate chains of recurrences to represent symbolic expressions
- Implementation on top of LLVM
- Discussion of related work
In our paper:

- Formalization as Abstract Interpretation
- Multivariate chains of recurrences to represent symbolic expressions
- Implementation on top of LLVM
- Discussion of related work