saarland-informatics-campus.de

Leveraging LLVM’s ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau

Jan Reineke

UNIVERSITÄT DES SAARLANDES

SIC Saarland Informatics Campus

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

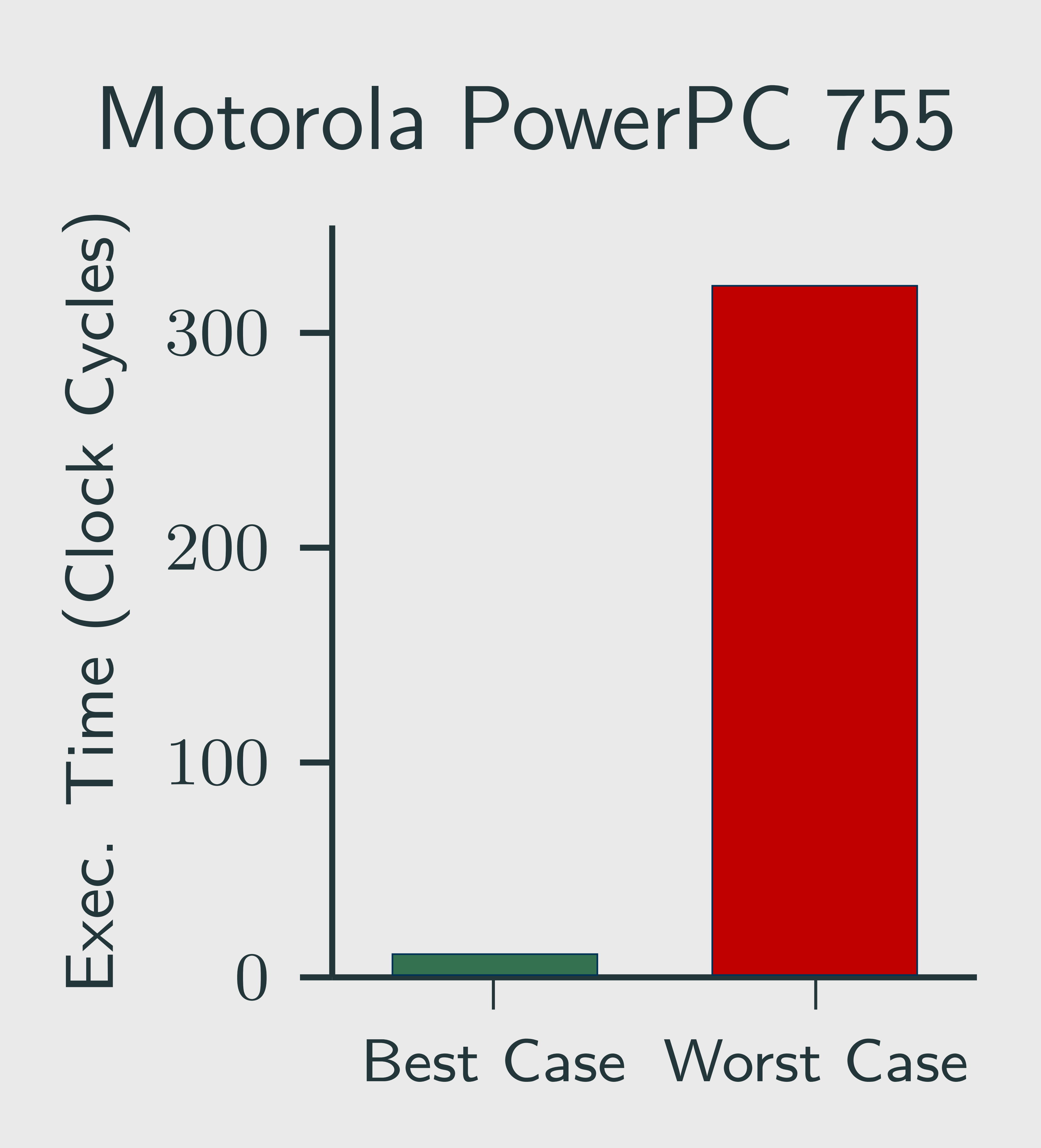

```
x = a + b
```

```
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
```



Instruction Cache Analysis

Source program

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

Instruction Cache Analysis

Source program

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

LLVM

Binary program


```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Instruction Cache Analysis

SIC Saarland Informatics Campus

Source program

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

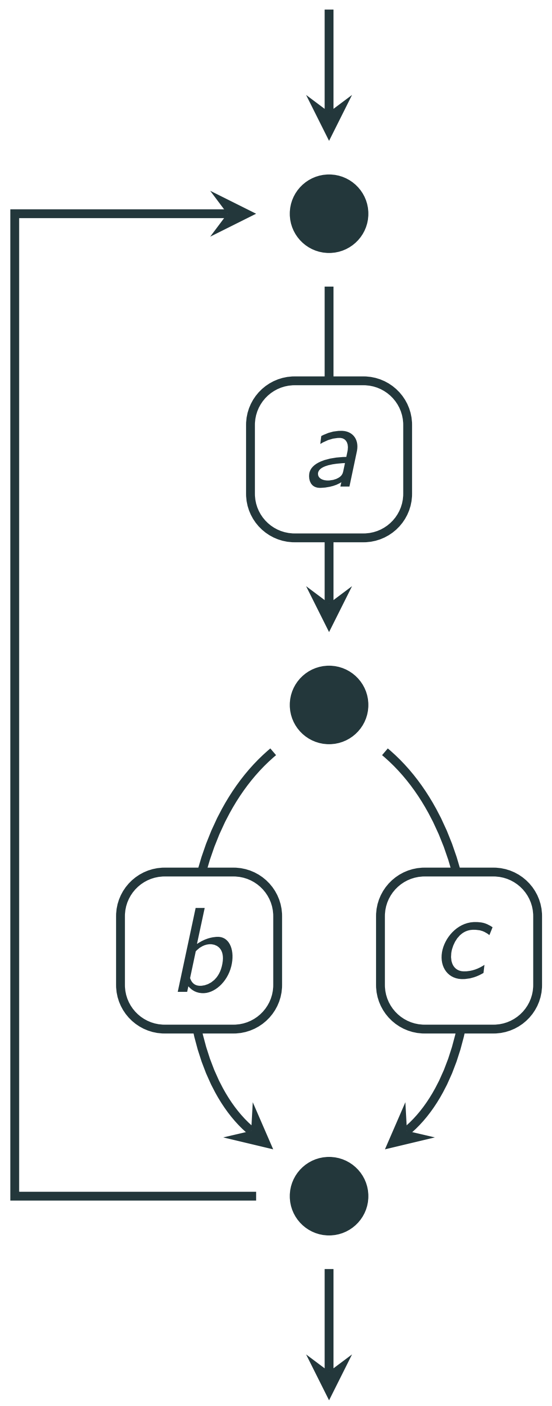
LLVM

Binary program


```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```


Abstraction

Control-flow graph



Instruction Cache Analysis

Source program

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

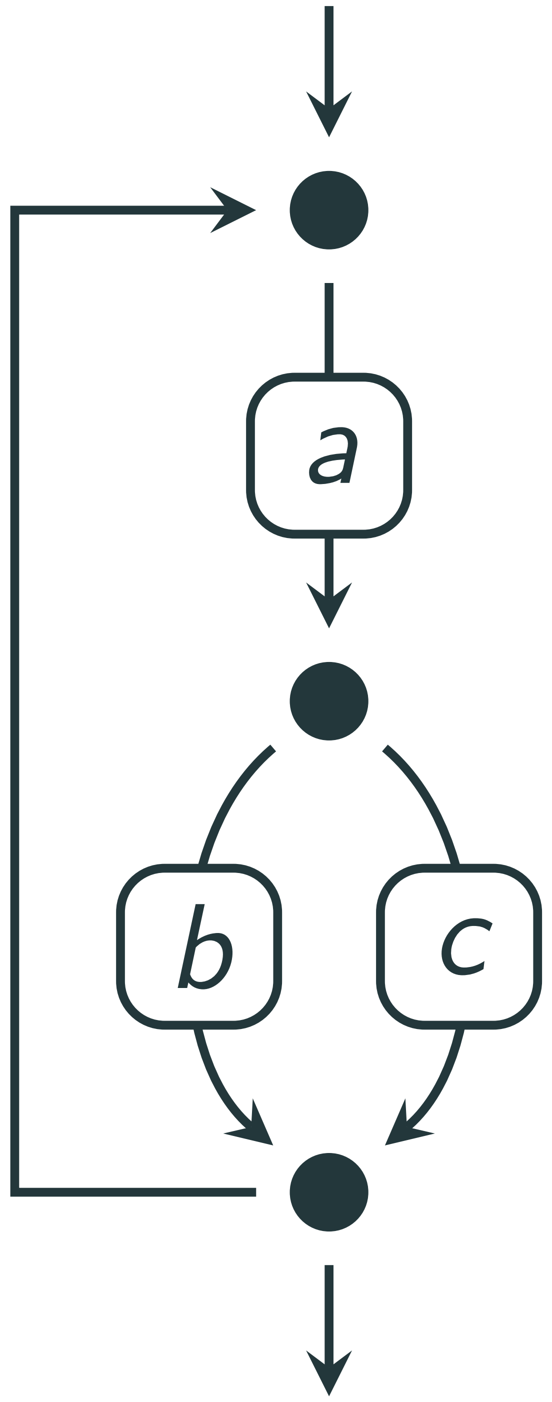
LLVM

Binary program


```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```


Abstraction

Control-flow graph



Instruction Cache Analysis

Classification:
- “always hit”
- “always miss”
- “unknown”

Challenges in Data Cache Analysis I

Source program

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

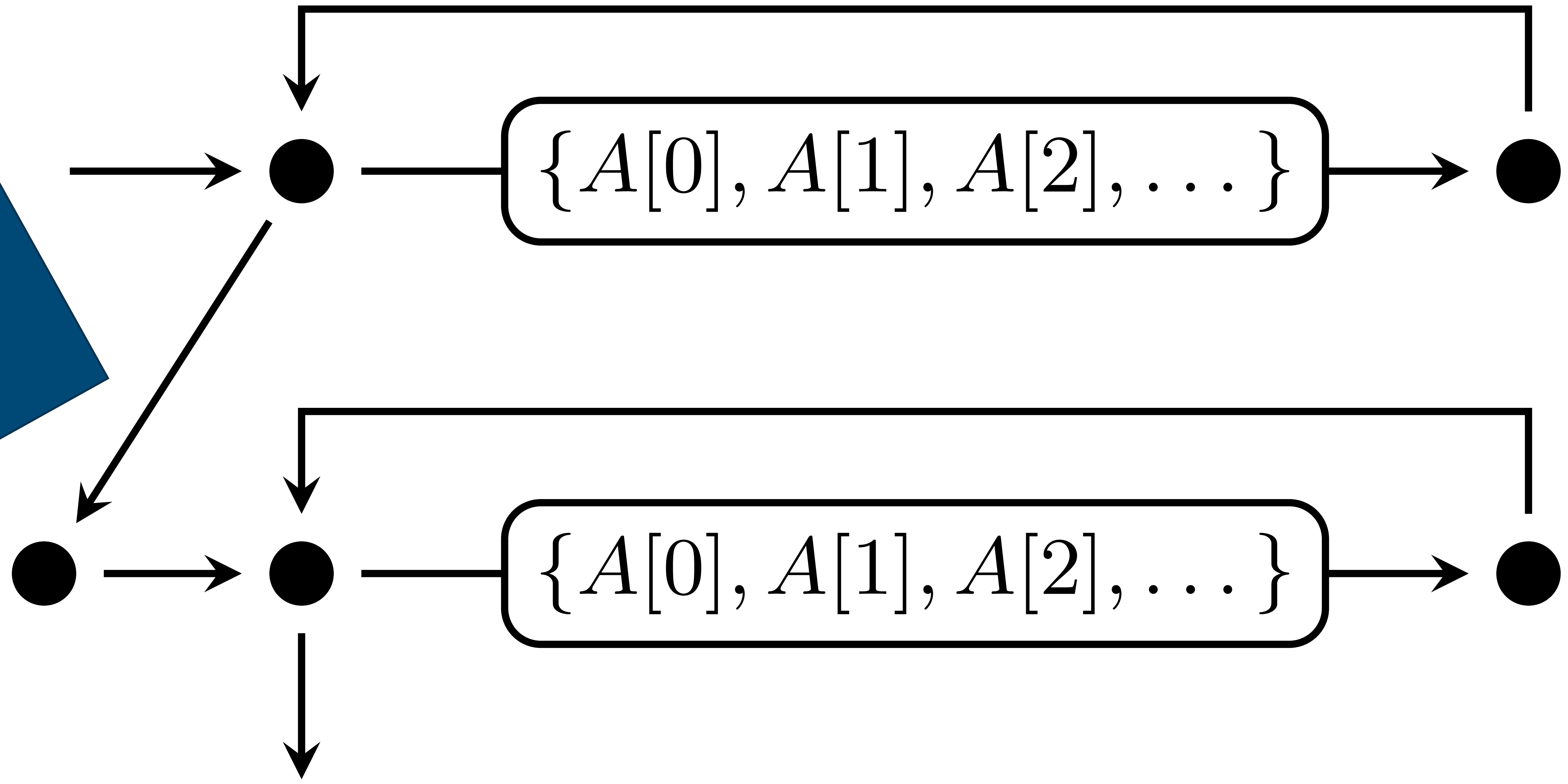
Challenges in Data Cache Analysis I

Source program

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

LLVM
+ Abstraction

Control-flow graph



Challenges in Data Cache Analysis I

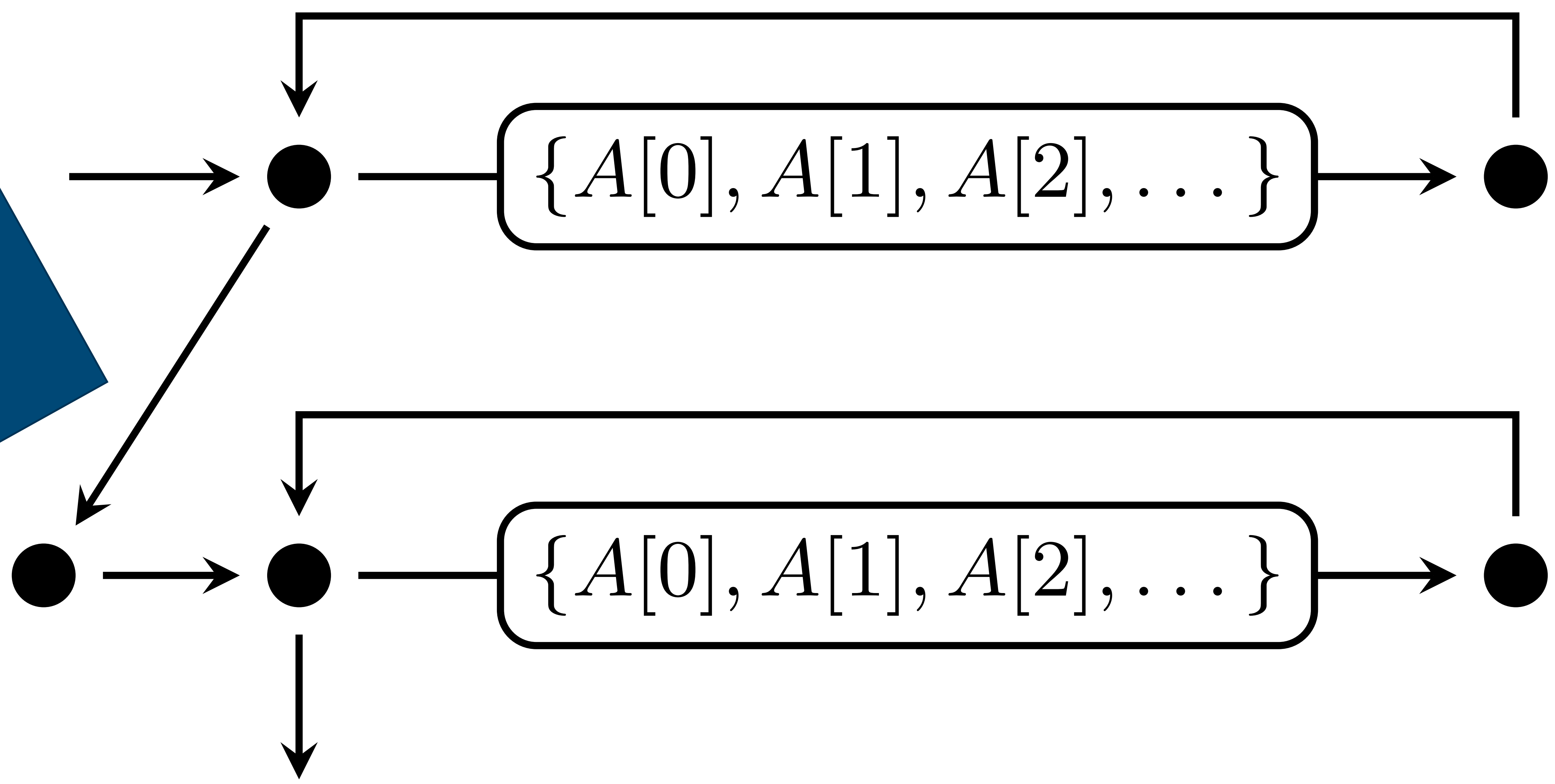
Source program

```
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

LLVM + Abstraction

Control-flow graph



Challenges in Data Cache Analysis I

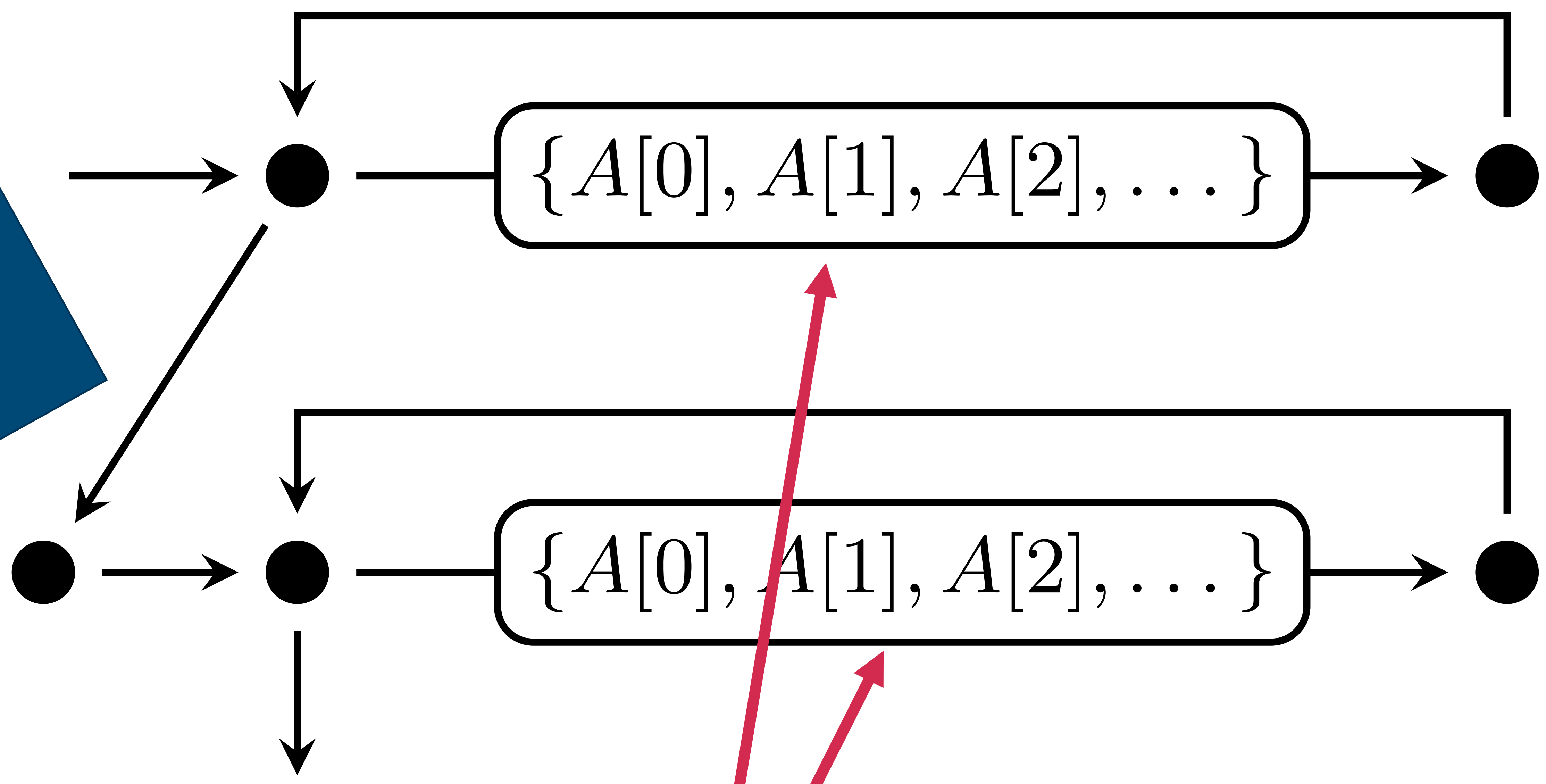
Source program

```
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

LLVM + Abstraction

Control-flow graph



Cannot express dependence of addresses on iteration!

First Contribution: Symbolic Control-Flow Graphs

Source program

```
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

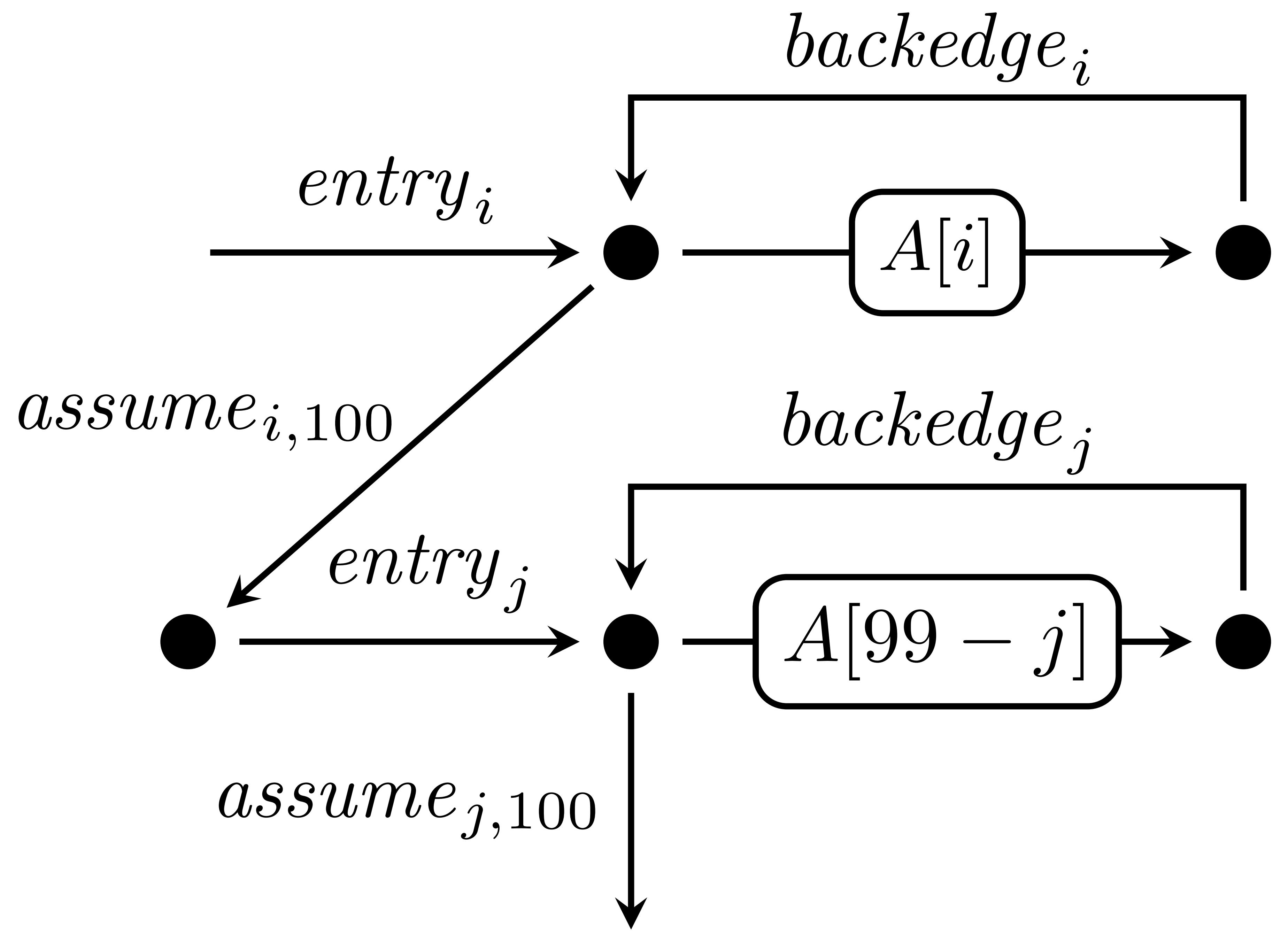
Source program

```
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

LLVM + Abstraction

Symbolic CFG



First Contribution: Symbolic Control-Flow Graphs

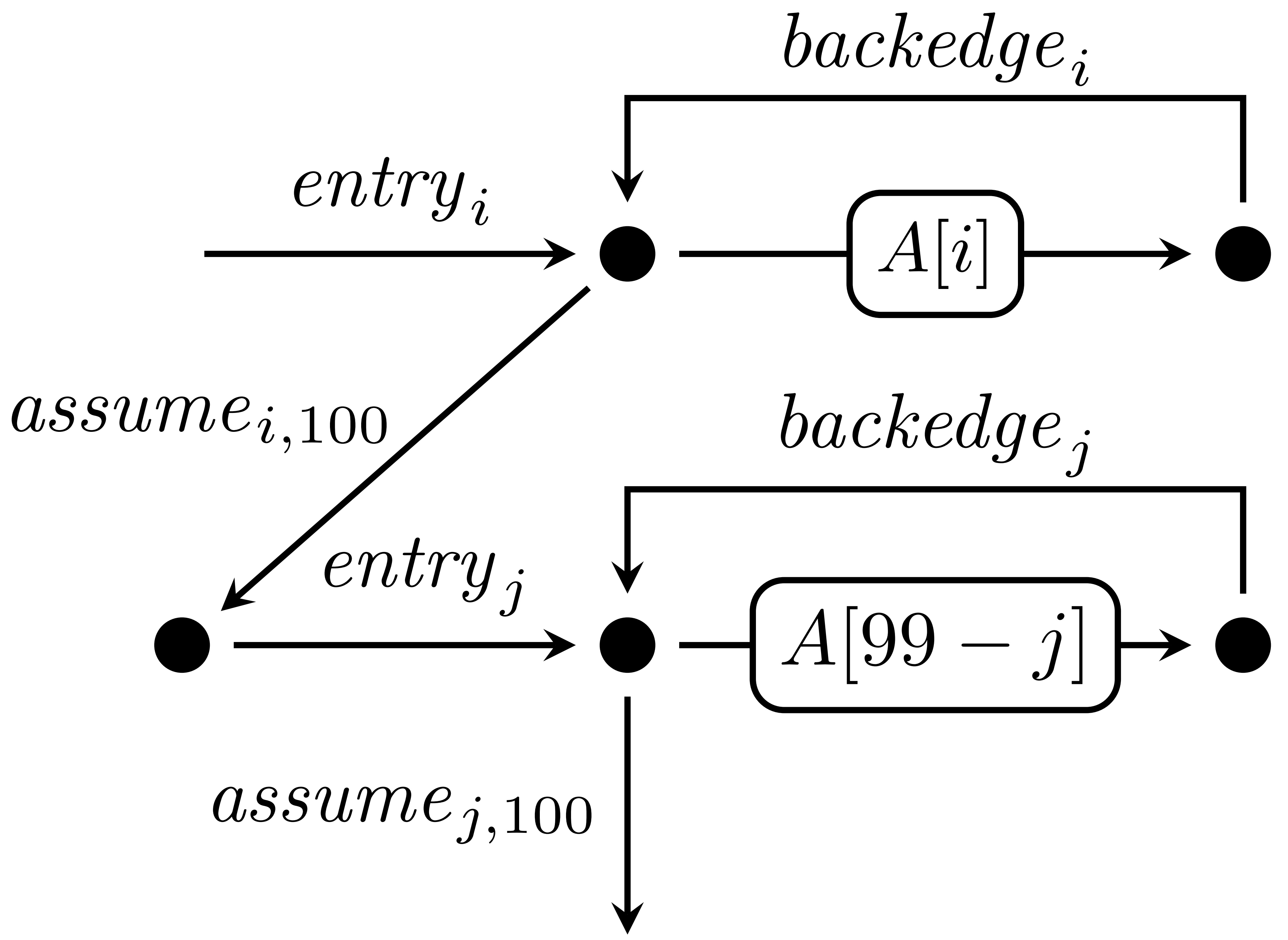
Source program

```
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

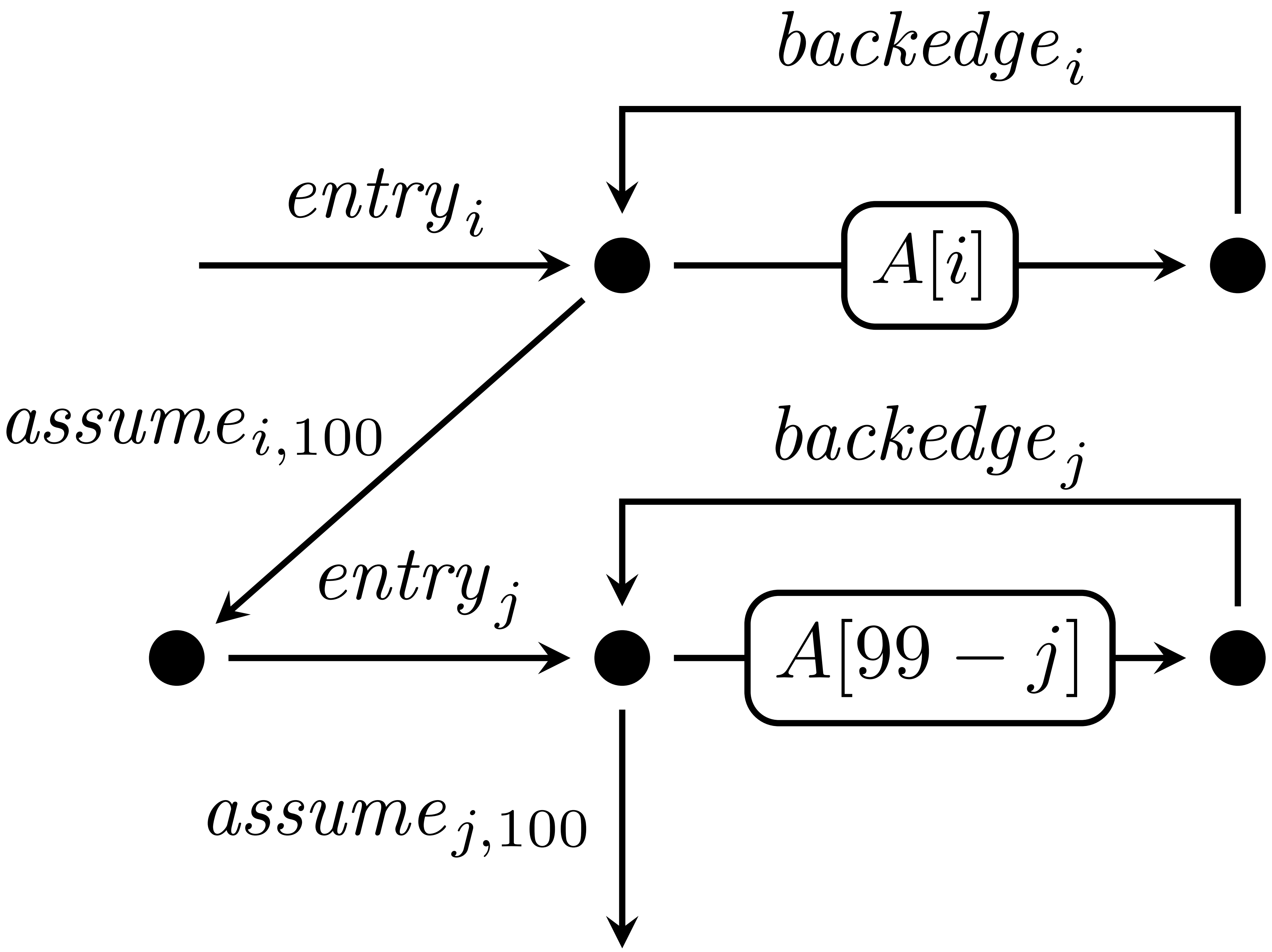
LLVM + Abstraction

Symbolic CFG



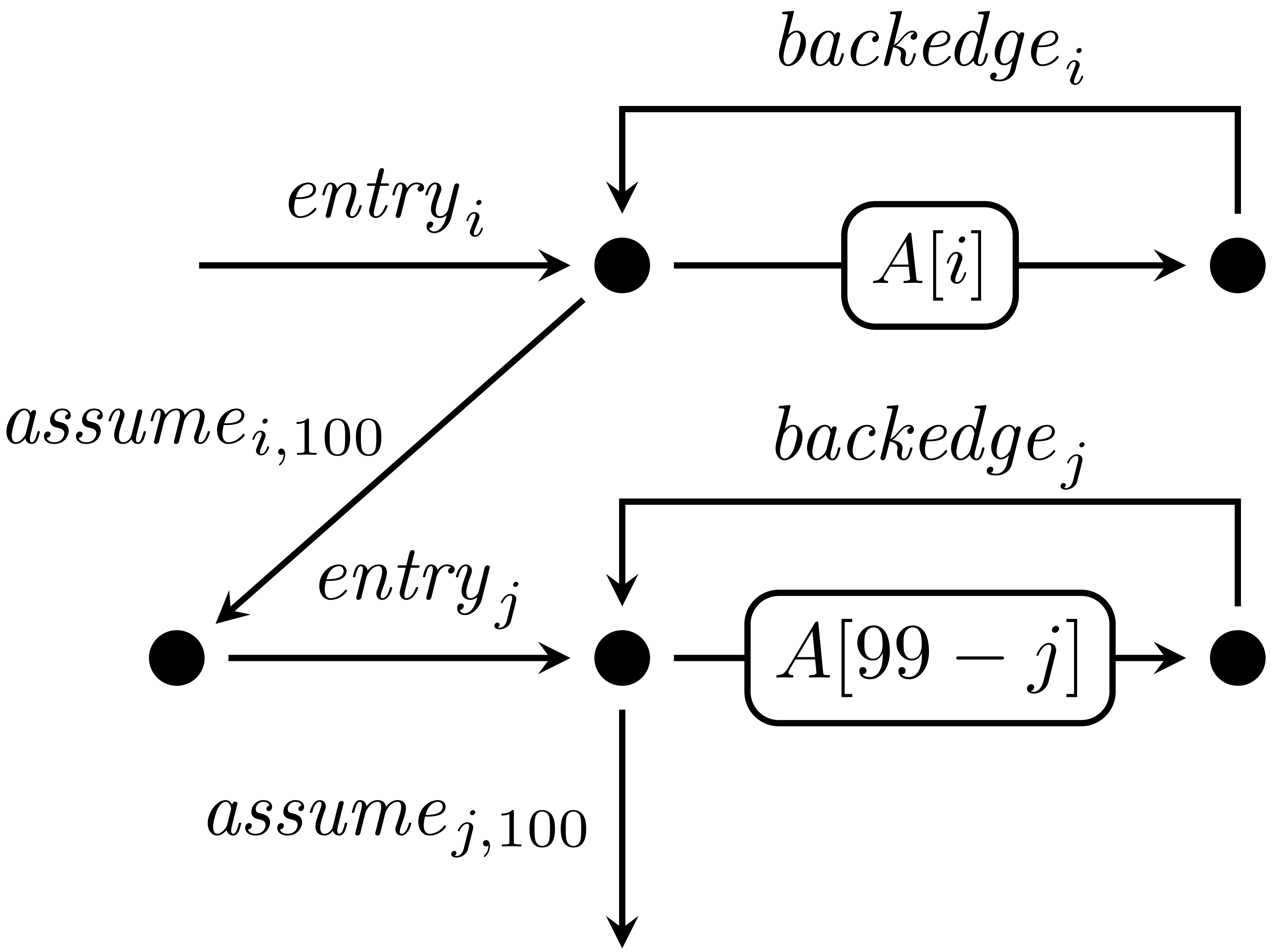
Captures dependence of addresses on loop iteration!

Symbolic CFG



Semantics

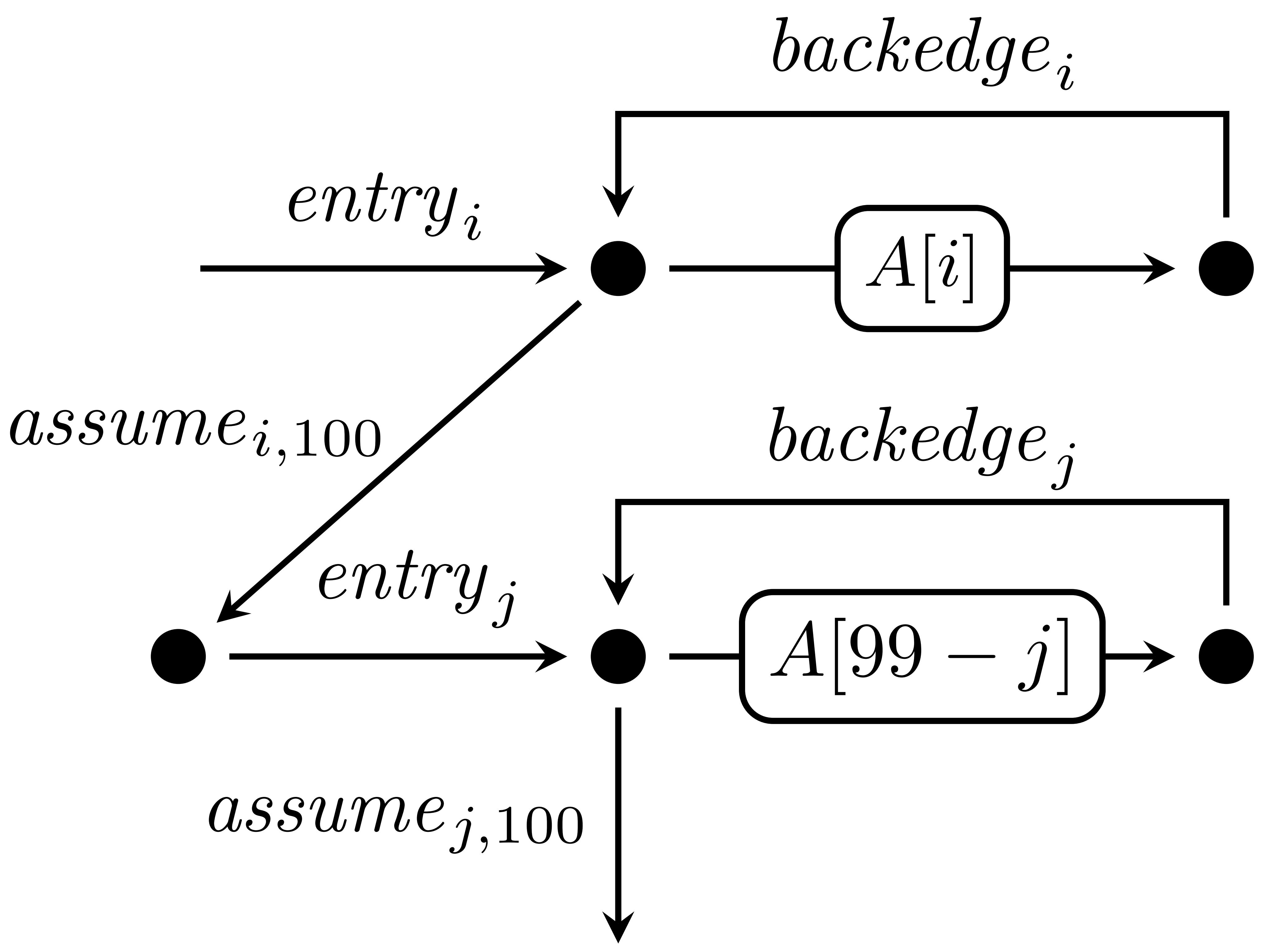
Symbolic CFG



Semantics

Loop variables capture iteration counts, here i and j.

Symbolic CFG

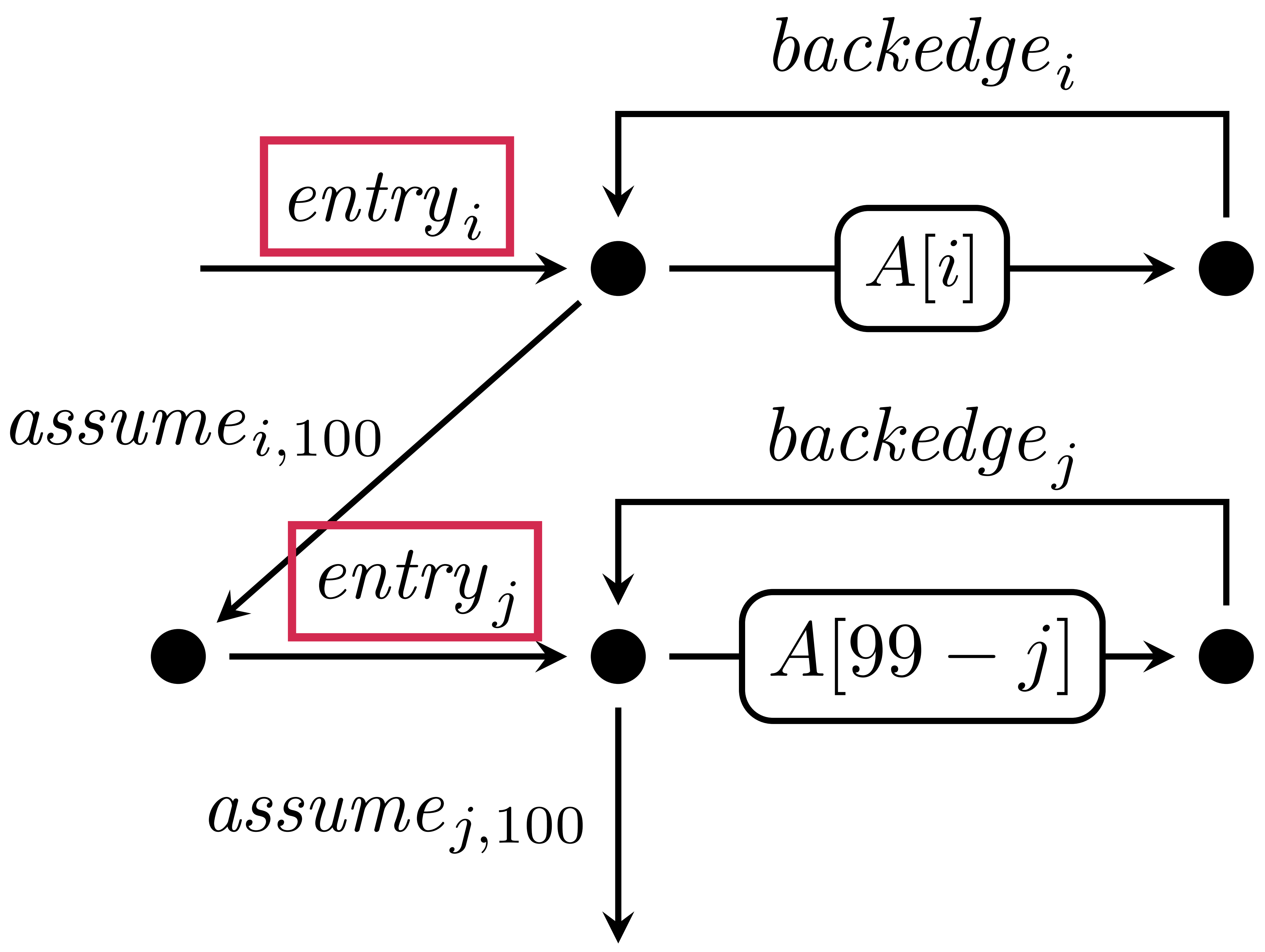


Semantics

Loop variables capture iteration counts, here i and j.

Three ways to manipulate variables:

Symbolic CFG



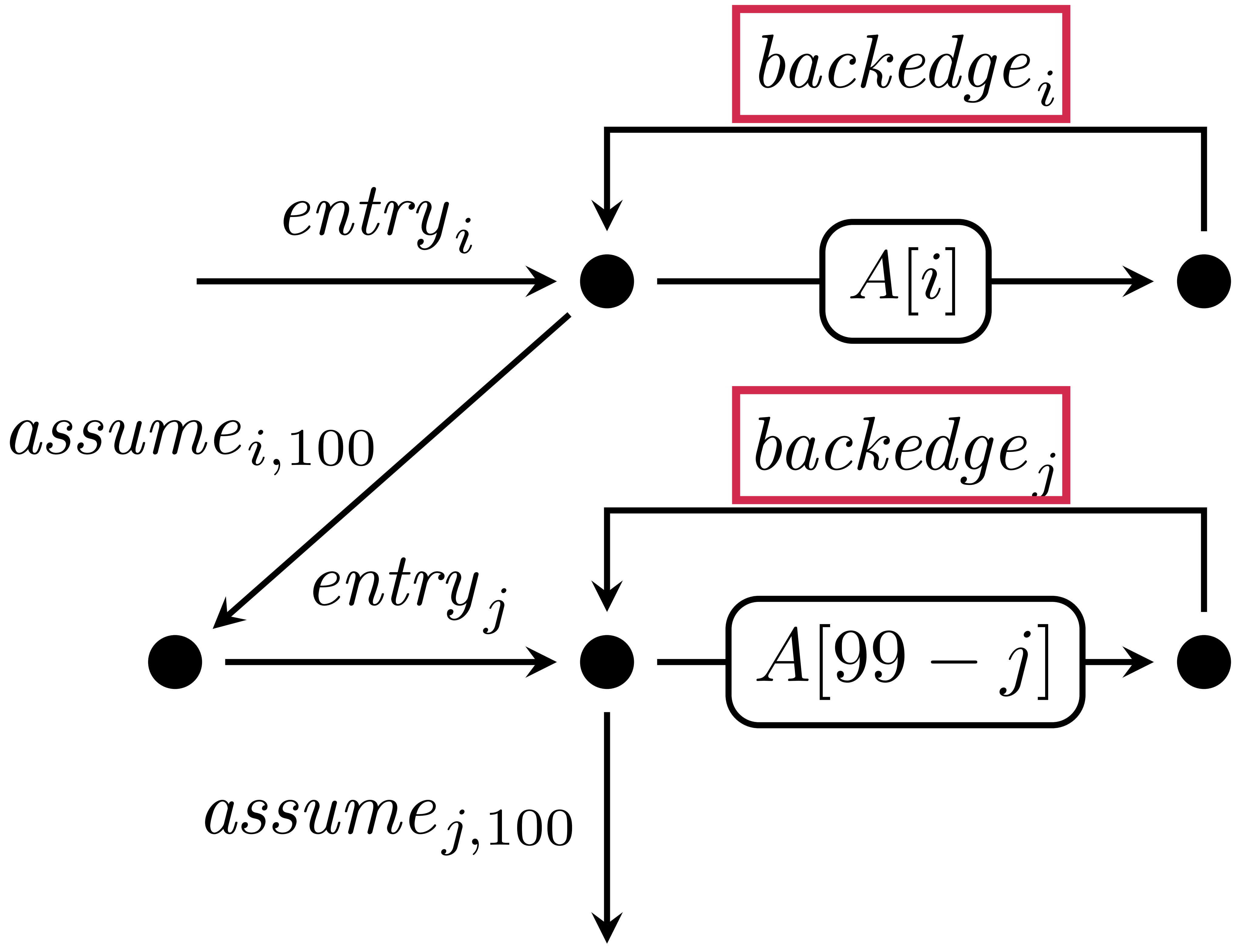
Semantics

Loop variables capture iteration counts, here i and j.

Three ways to manipulate variables:

$entry_i$ reset variable i to 0

Symbolic CFG



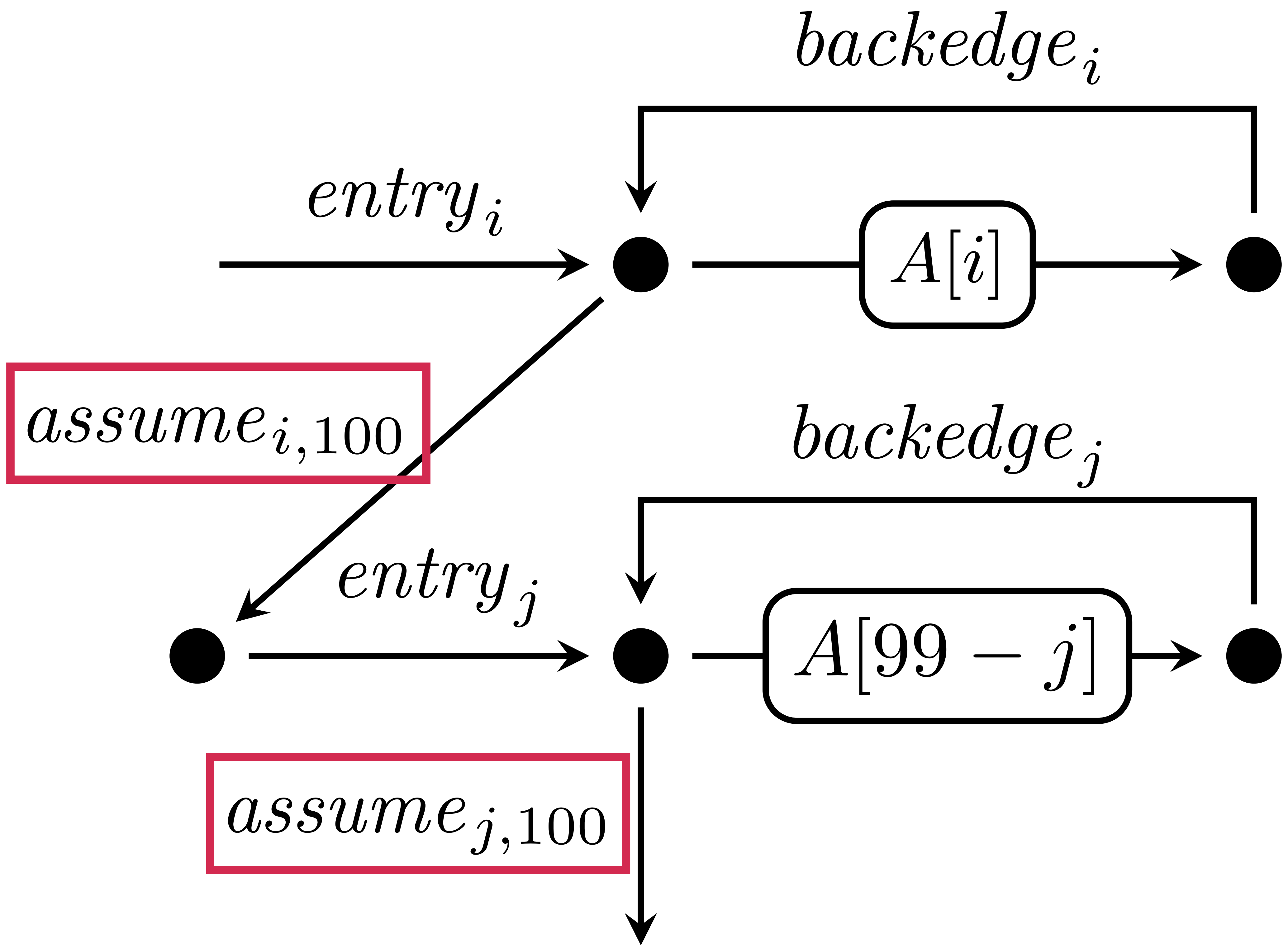
Semantics

Loop variables capture iteration counts, here i and j.

Three ways to manipulate variables:

$entry_i$	reset variable i to 0
$backedge_i$	increment variable i

Symbolic CFG



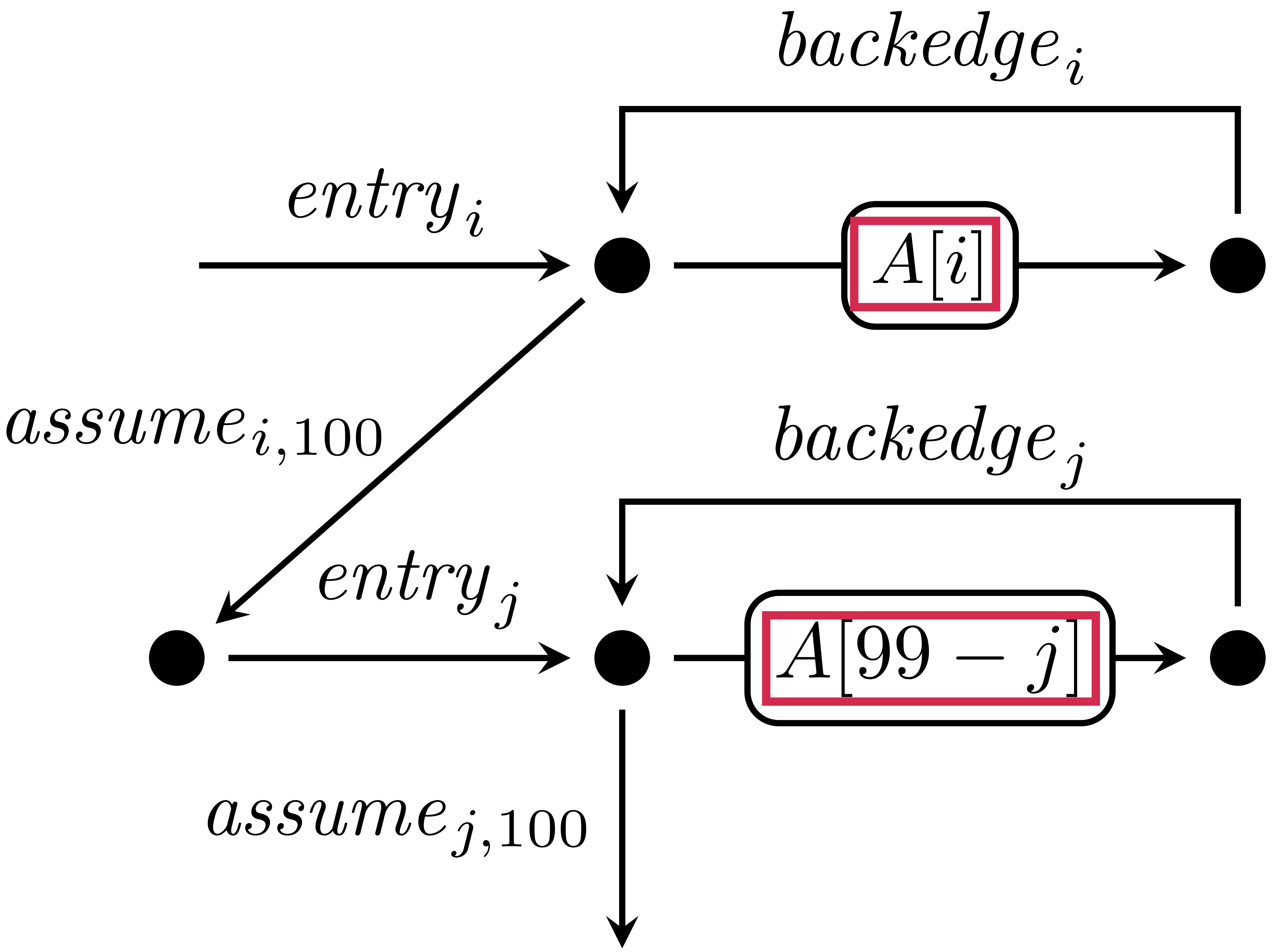
Semantics

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

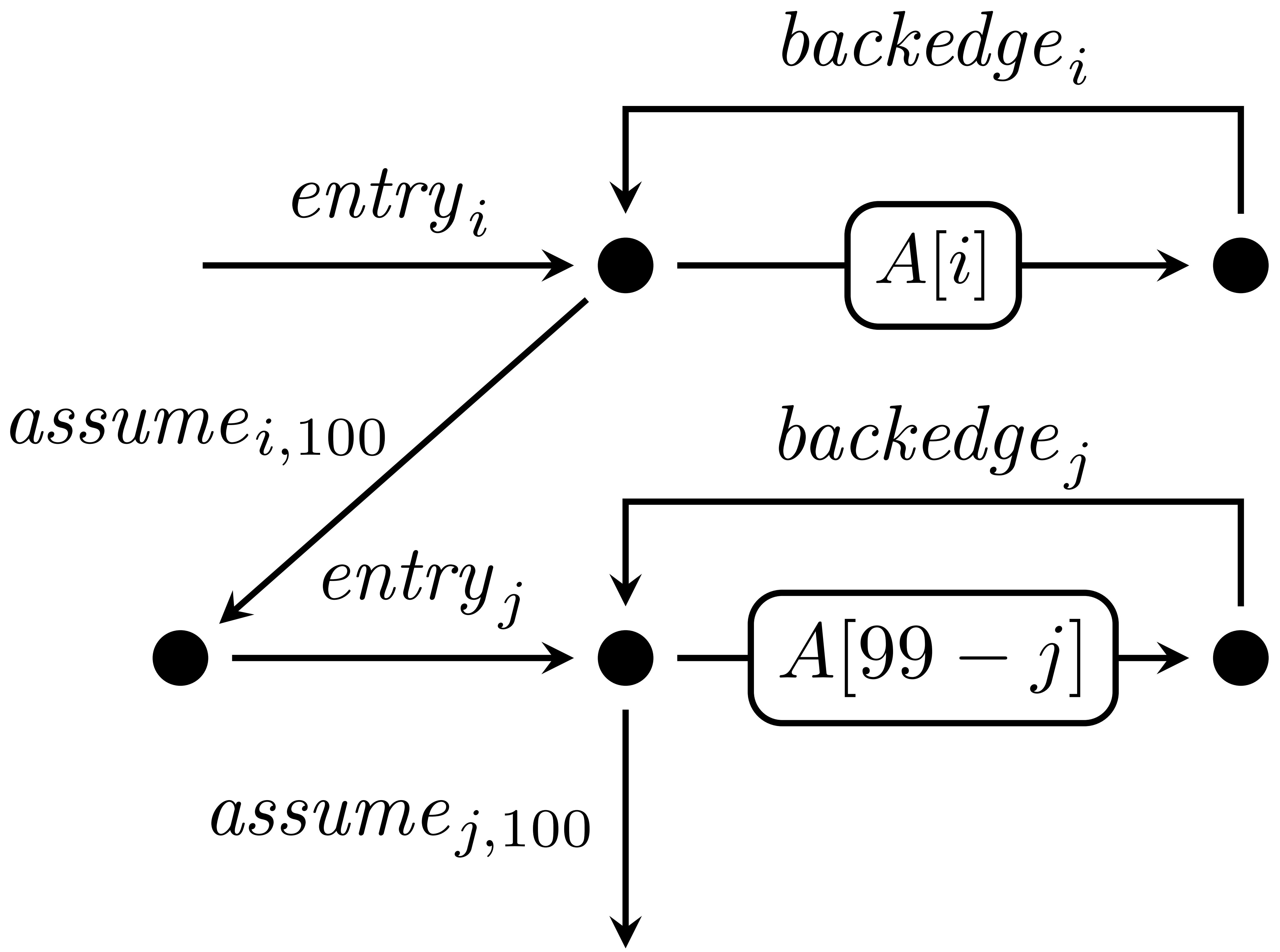
Loop variables capture iteration counts, here i and j.

Three ways to manipulate variables:

$entry_i$	reset variable i to 0
$backedge_i$	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.

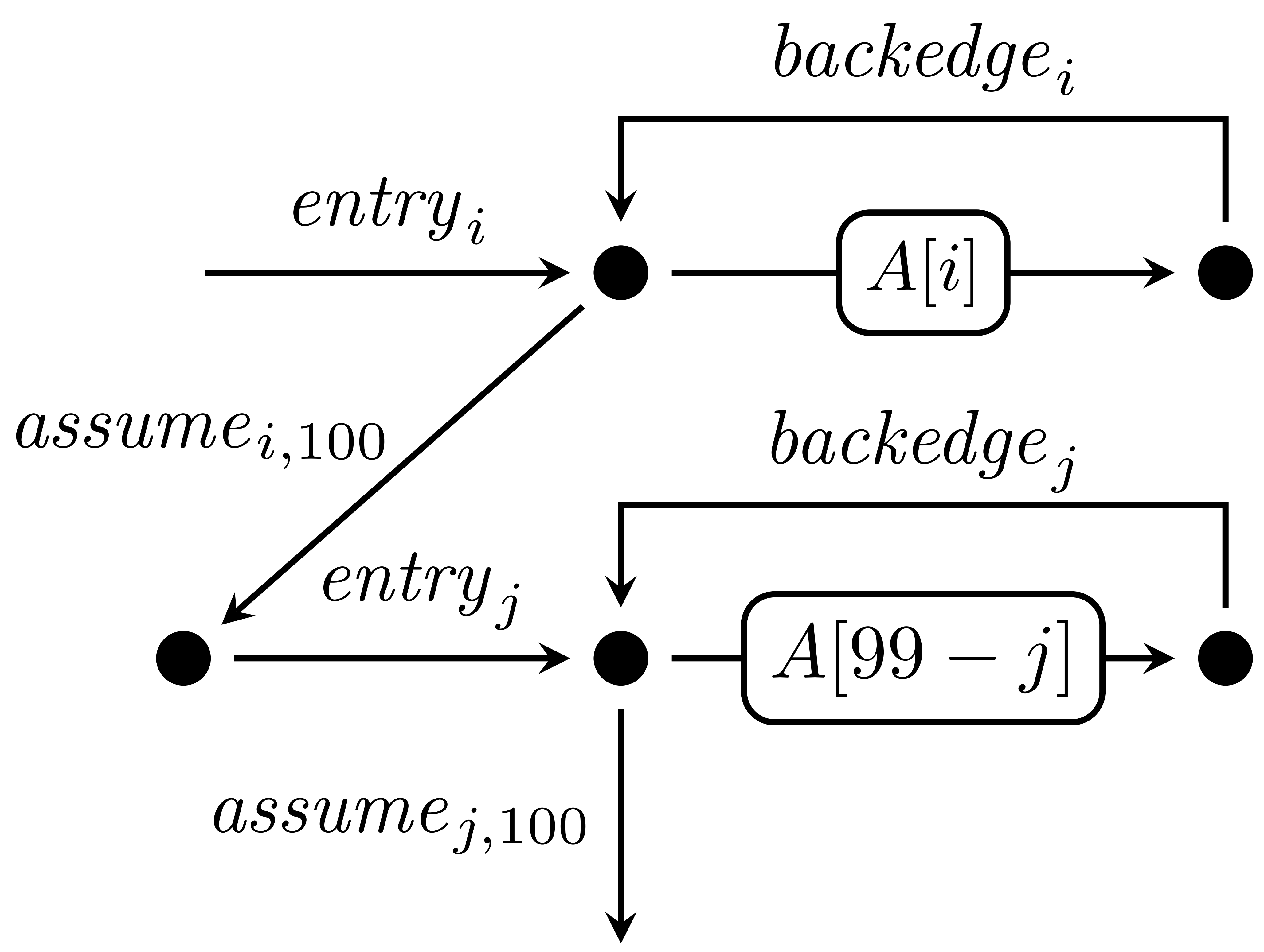
Three ways to manipulate variables:

$entry_i$	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

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

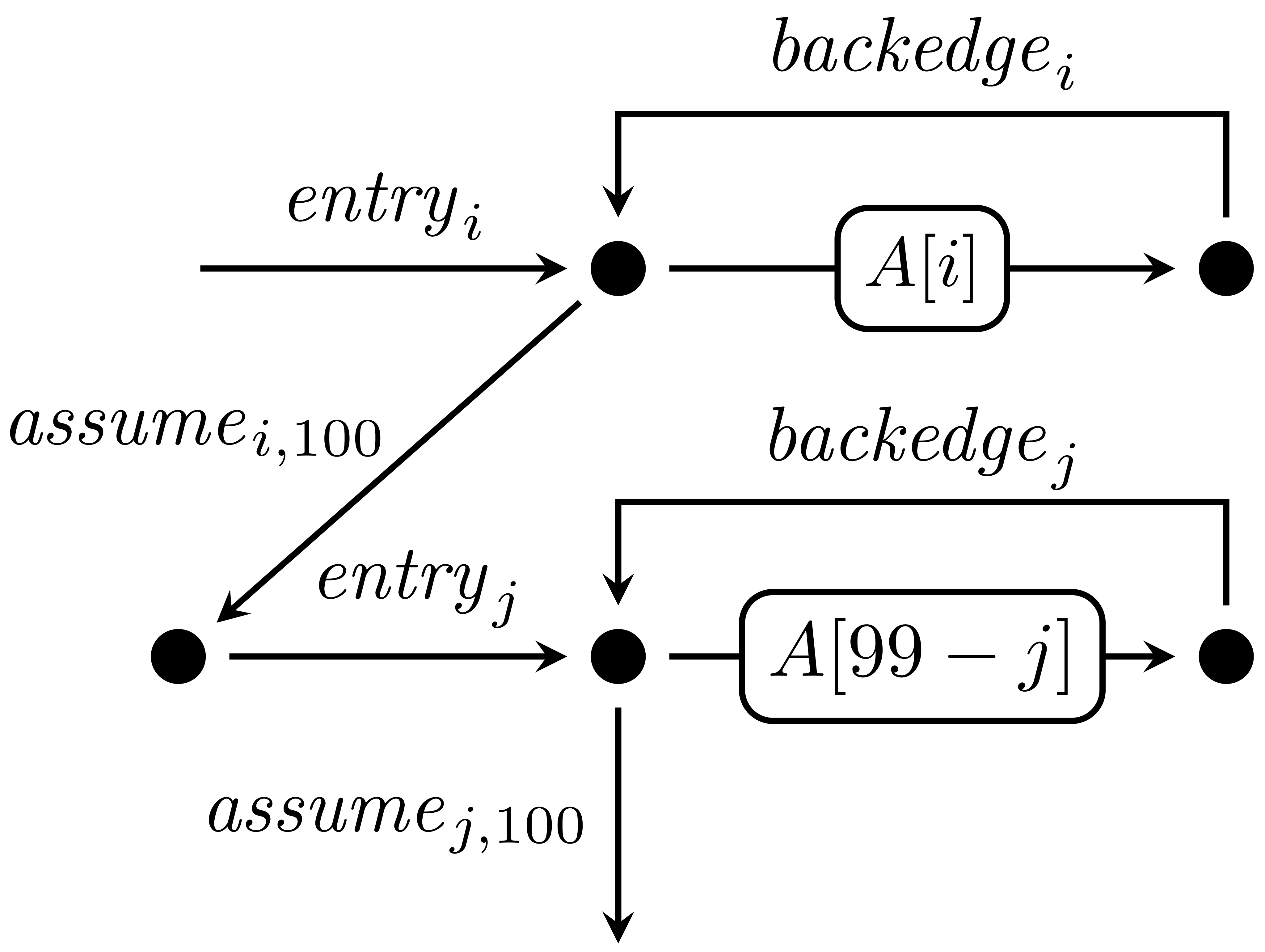
Obtained from LLVM's ScalarEvolution Analysis Pass

Symbolic CFG



Example

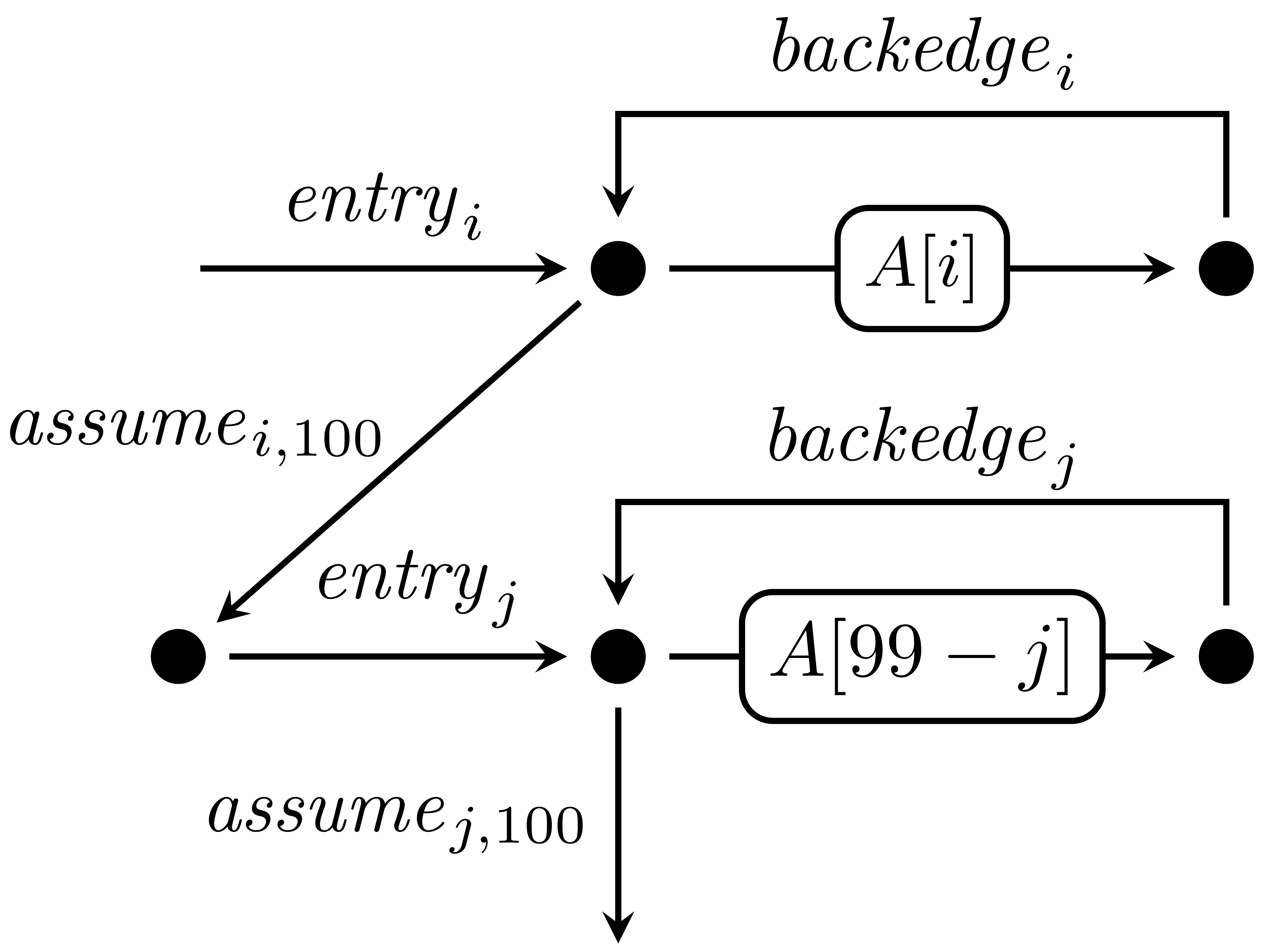
Symbolic CFG



Example

First loop:

Symbolic CFG

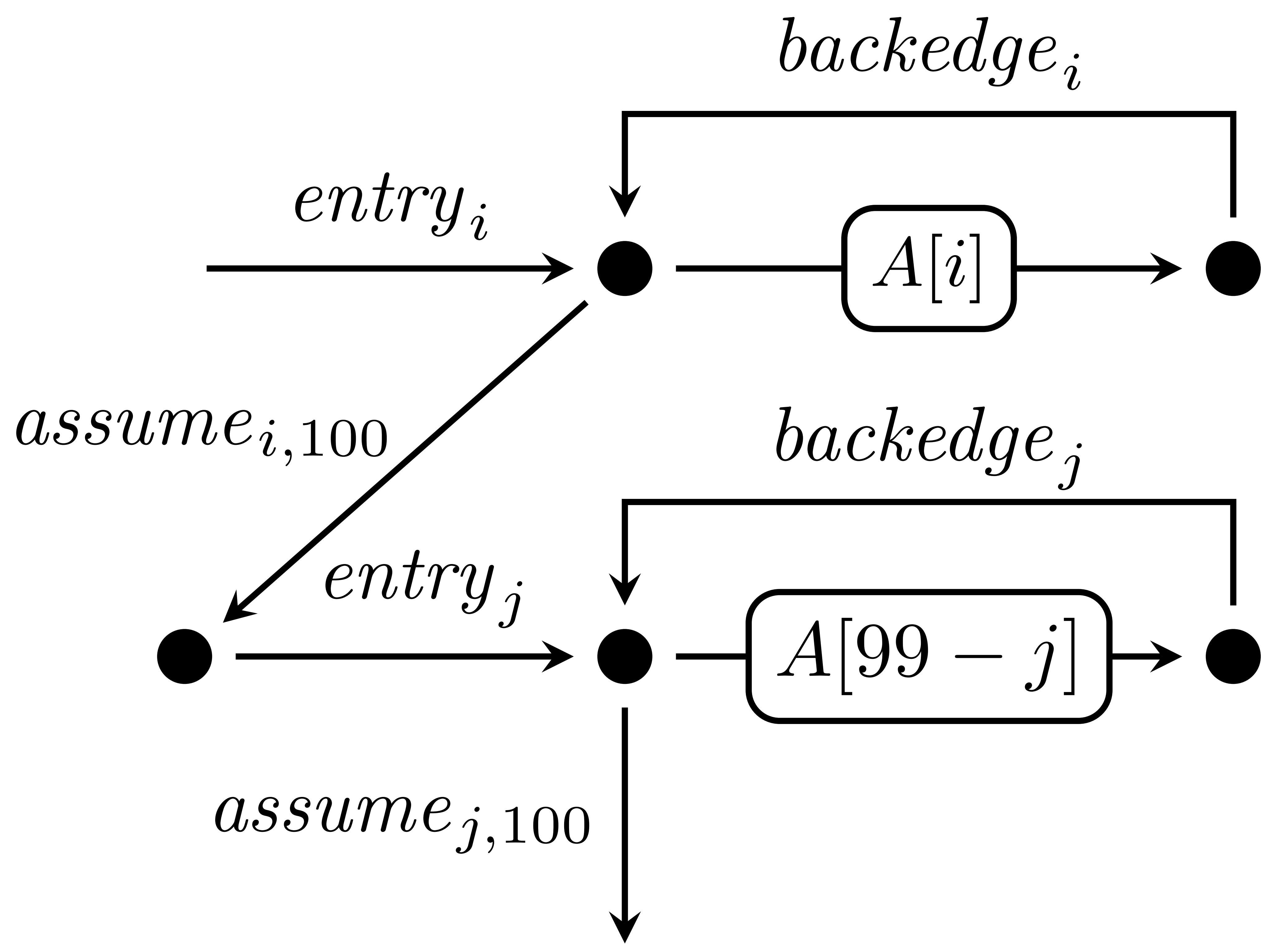


Example

First loop:

i = 0, A[0]

Symbolic CFG

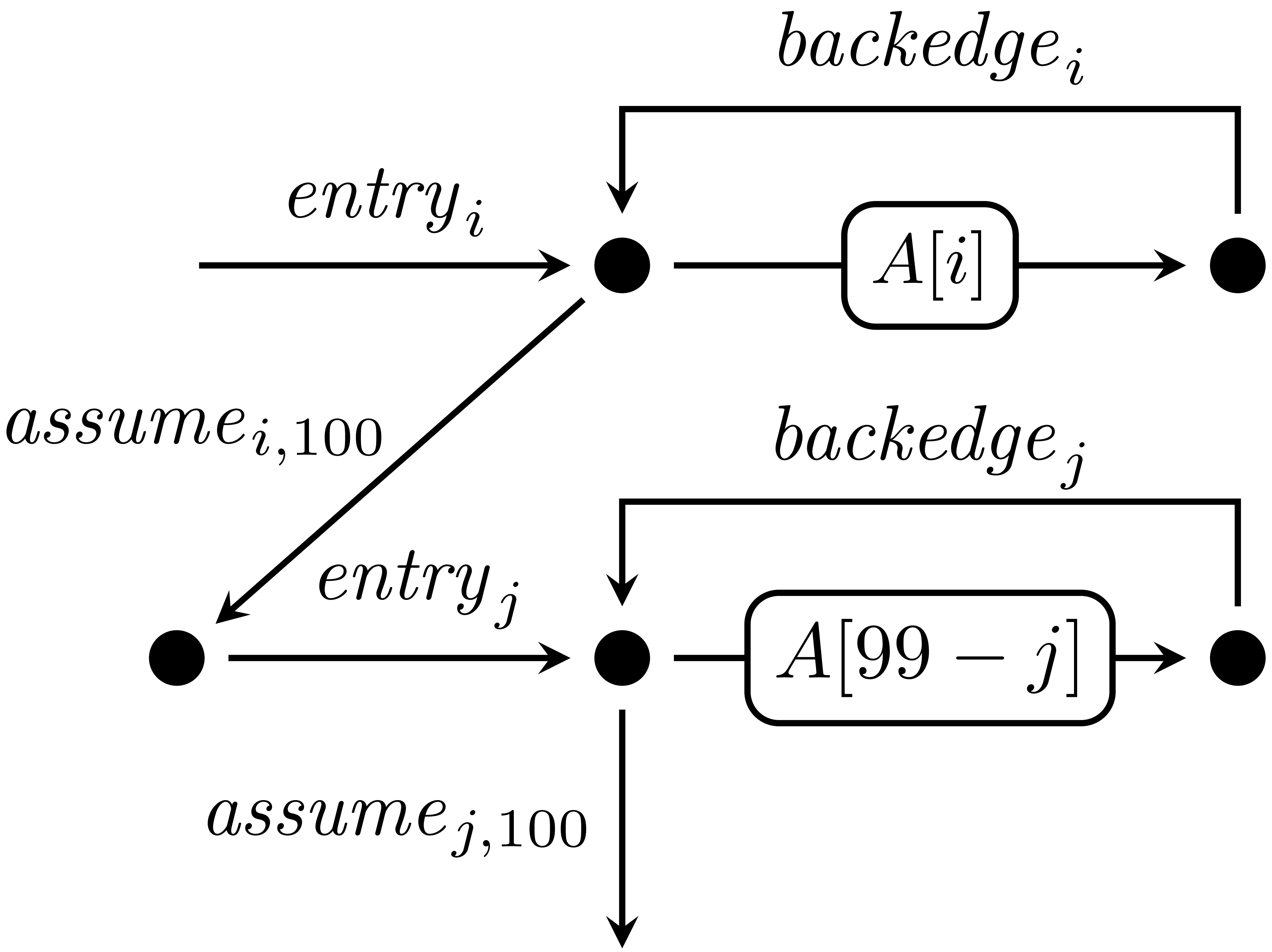


Example

First loop:

i = 0, A[0]   i = 1, A[1]

Symbolic CFG

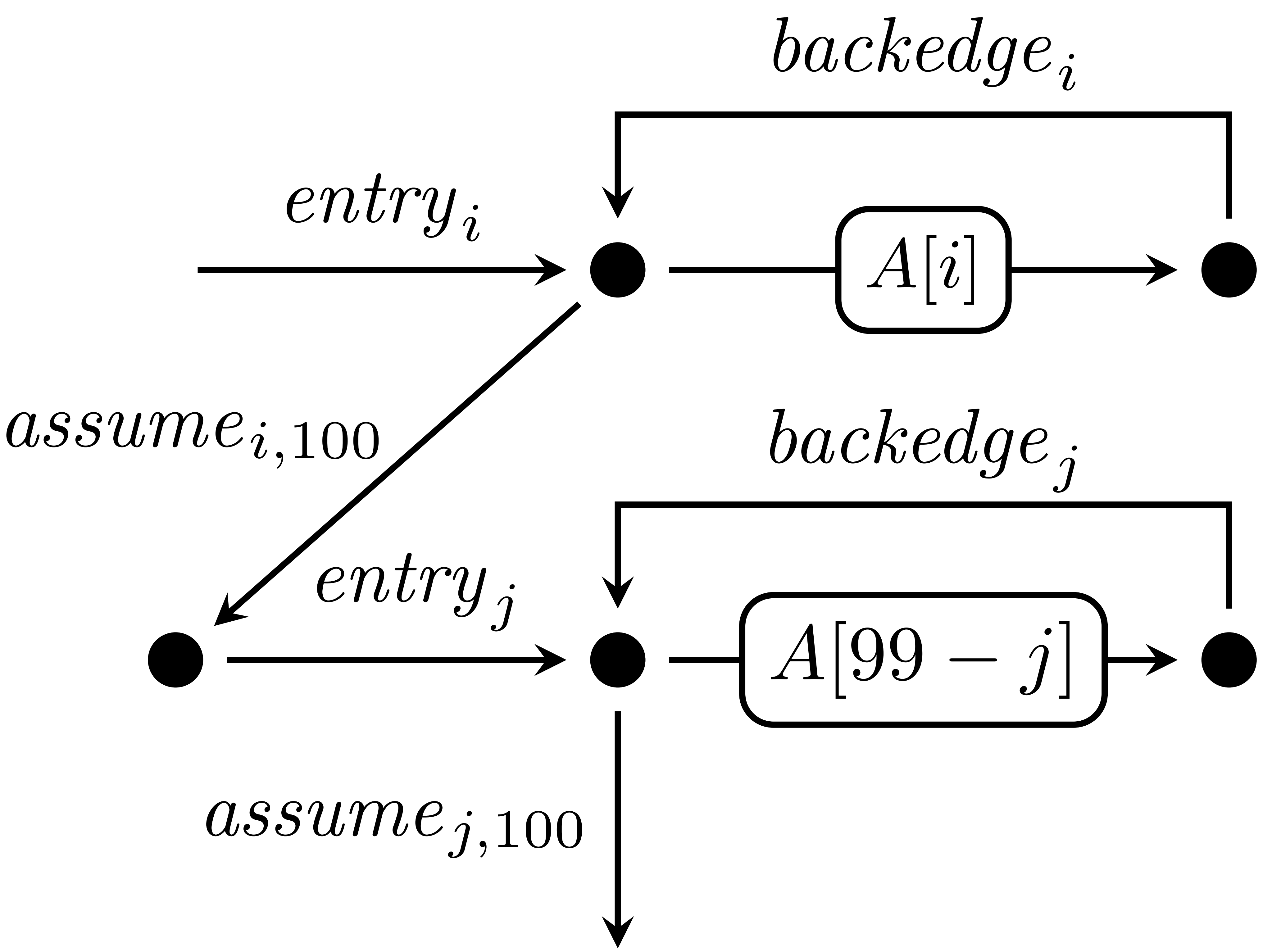


Example

First loop:

i = 0, A[0] i = 1, A[1] … i = 99, A[99]

Symbolic CFG



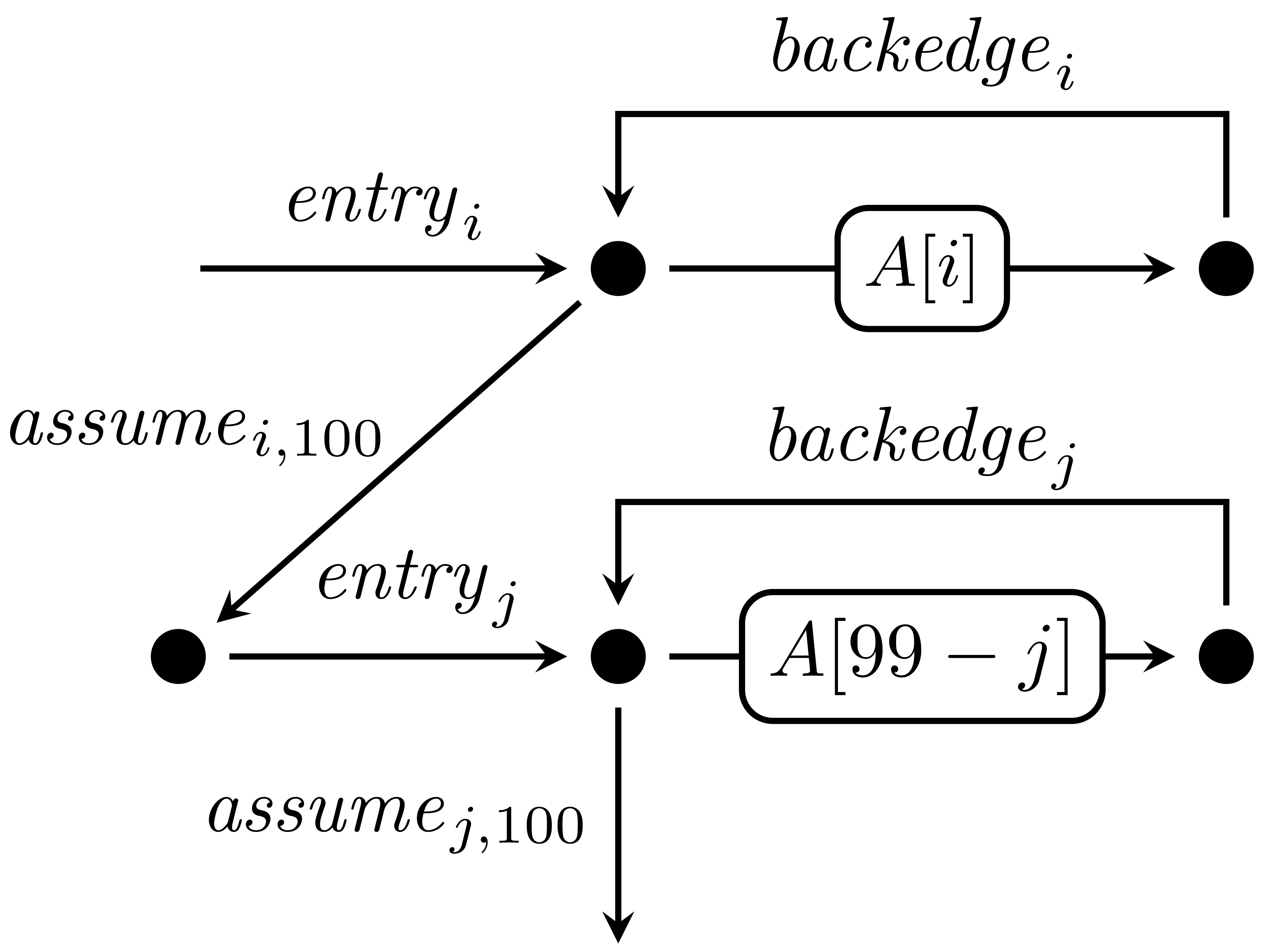
Example

First loop:

i = 0, A[0]   i = 1, A[1]   … i = 99, A[99]

Second loop:

Symbolic CFG



Example

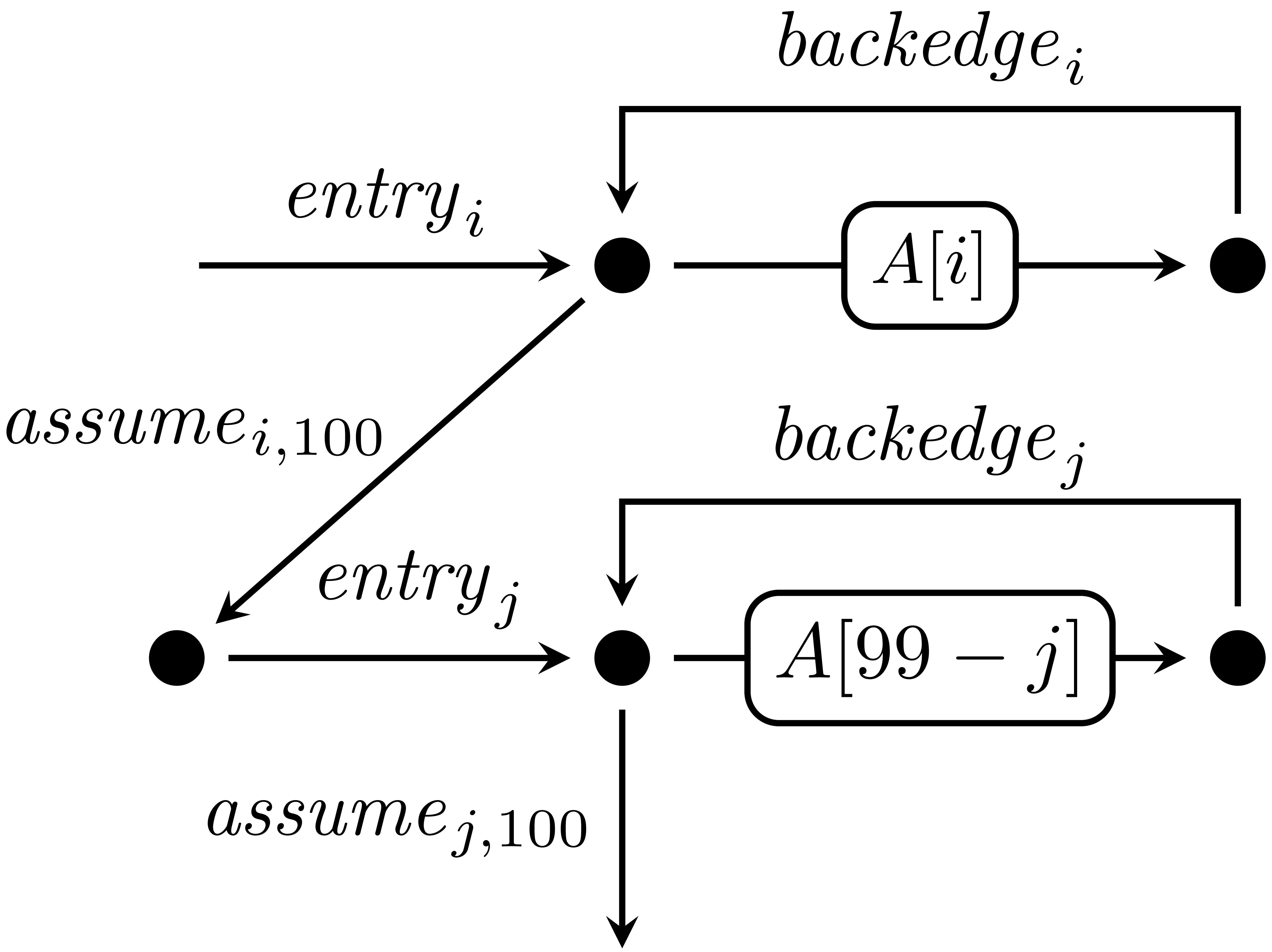
First loop:

i = 0, A[0]   i = 1, A[1]   … i = 99, A[99]

Second loop:

j = 0, A[99]

Symbolic CFG



Example

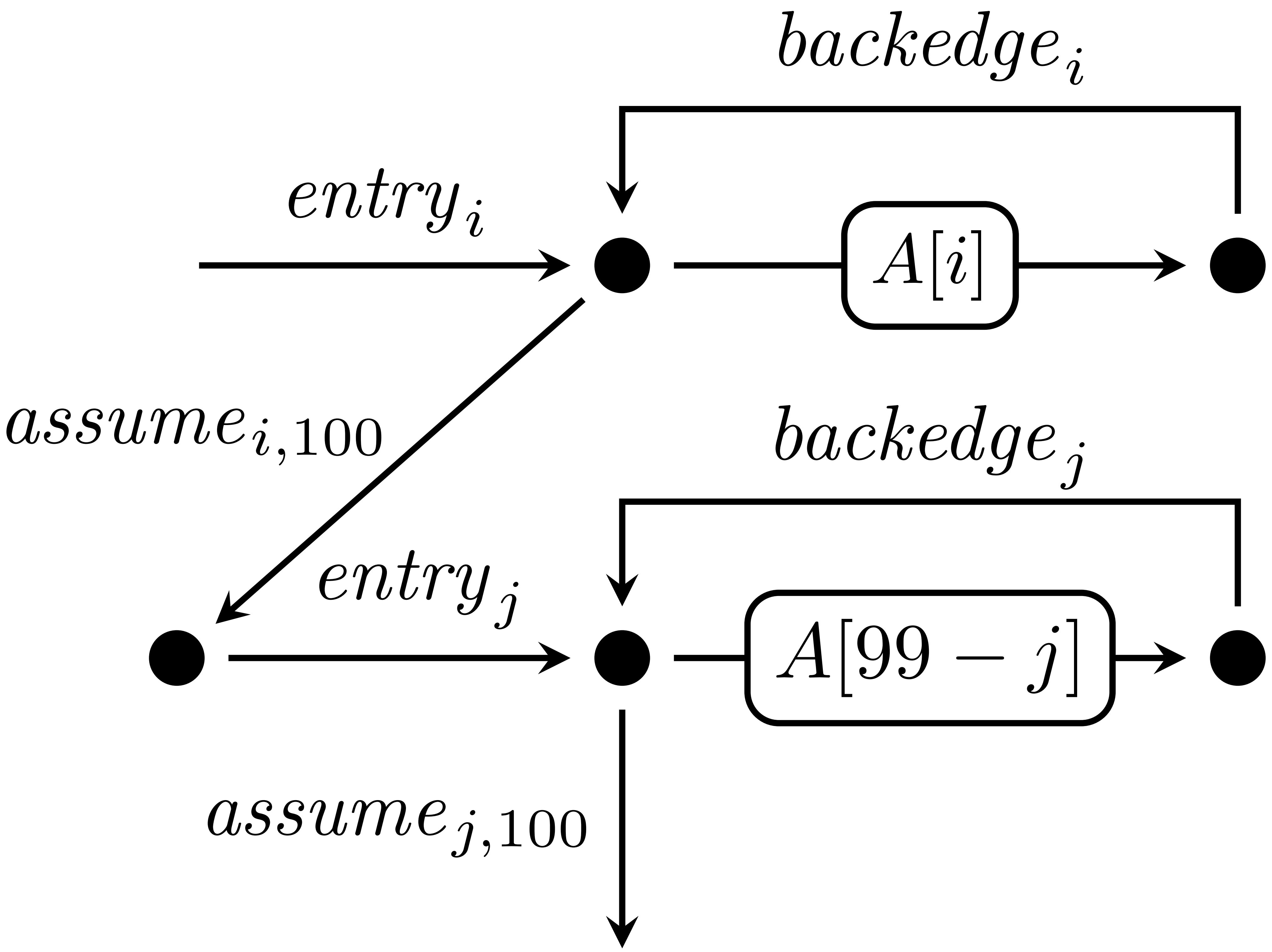
First loop:

i = 0, A[0]   i = 1, A[1]   … i = 99, A[99]

Second loop:

j = 0, A[99]   j = 1, A[98]

Symbolic CFG



Example

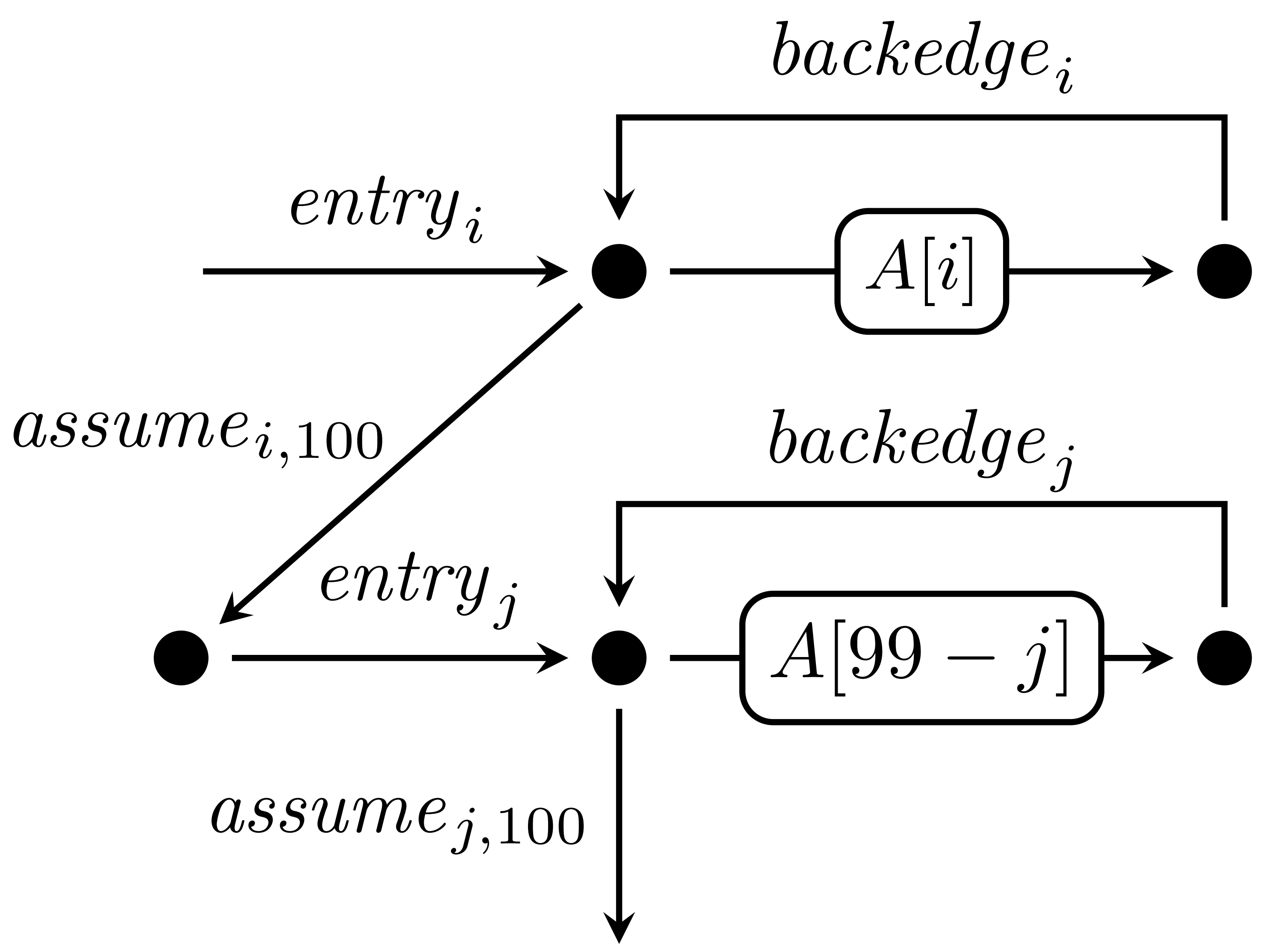
First loop:

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]

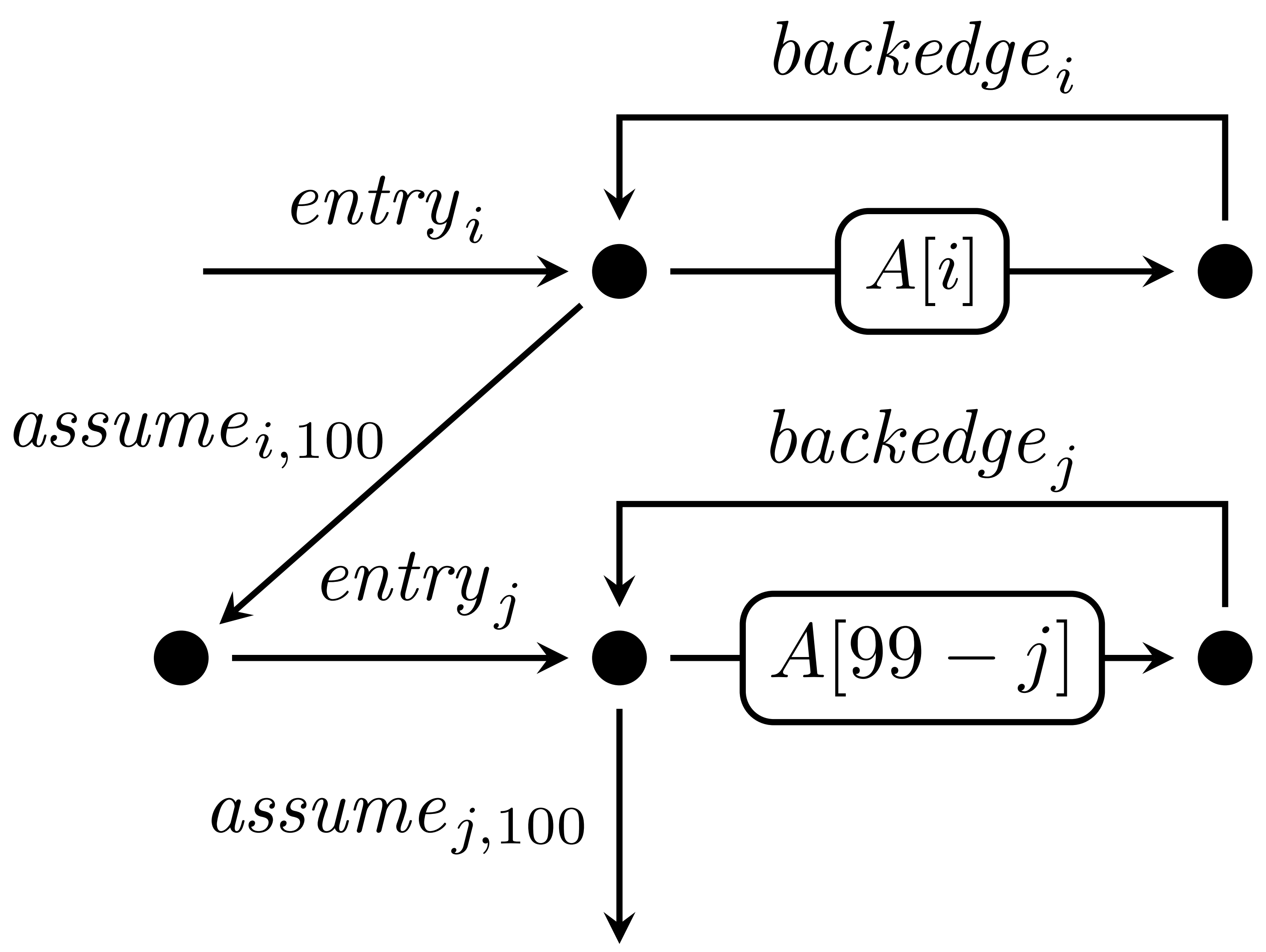
Symbolic CFG



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

Cache Analysis: Intuitively

Symbolic CFG

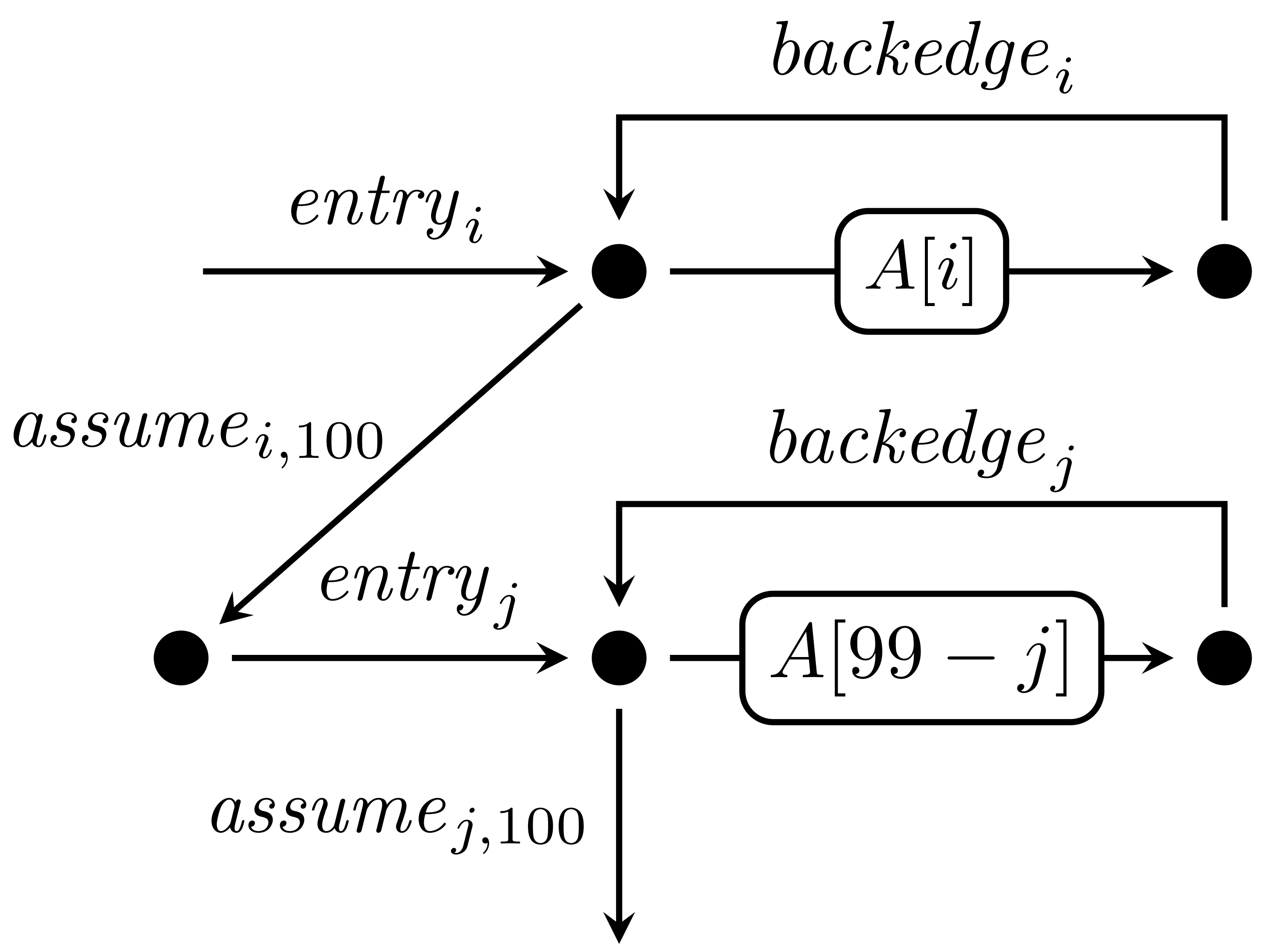


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

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

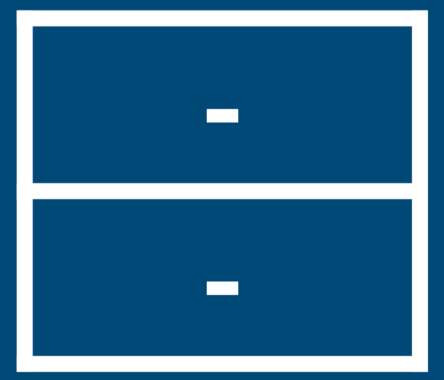
Symbolic CFG



Assumptions:

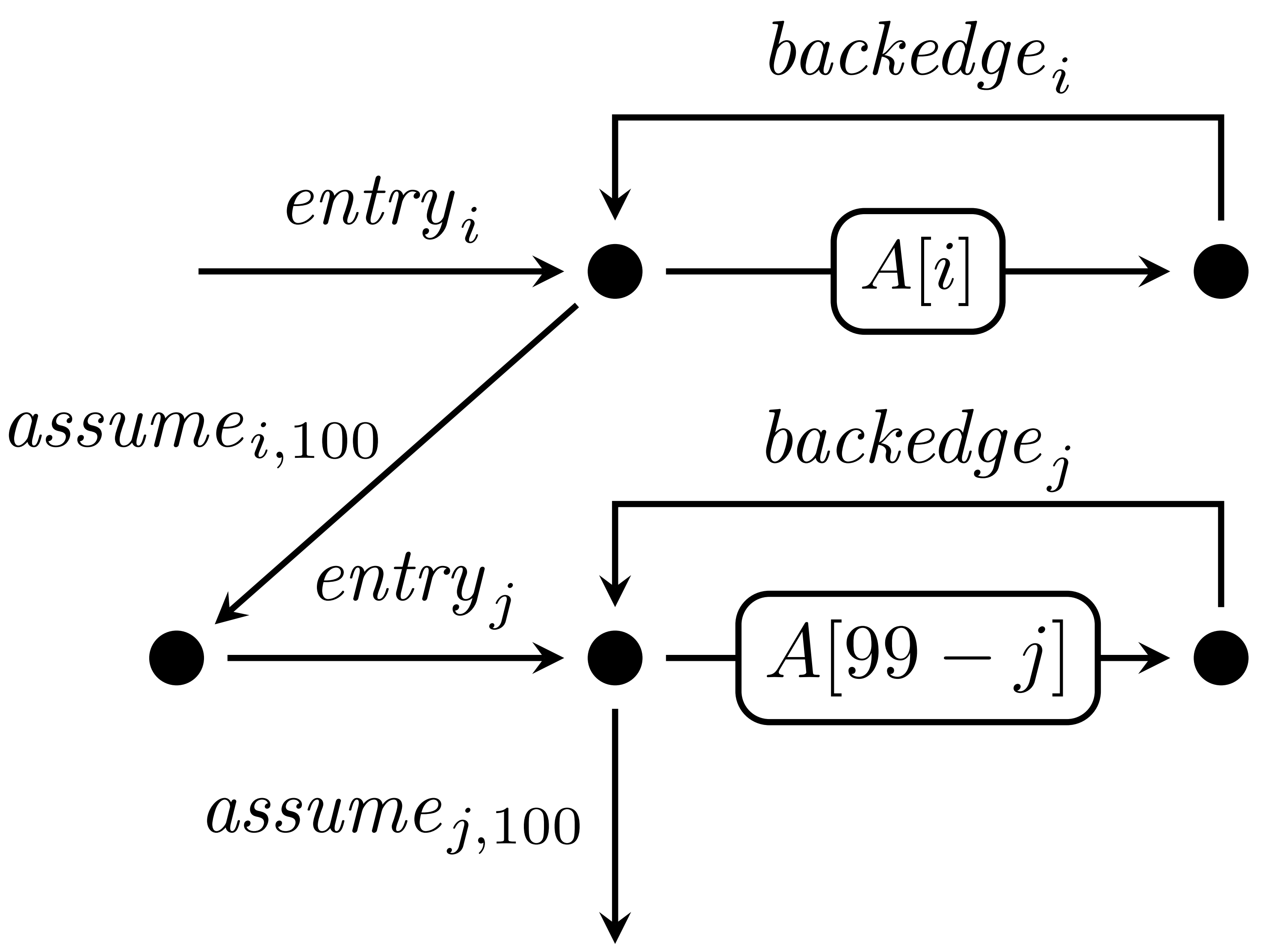
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively



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

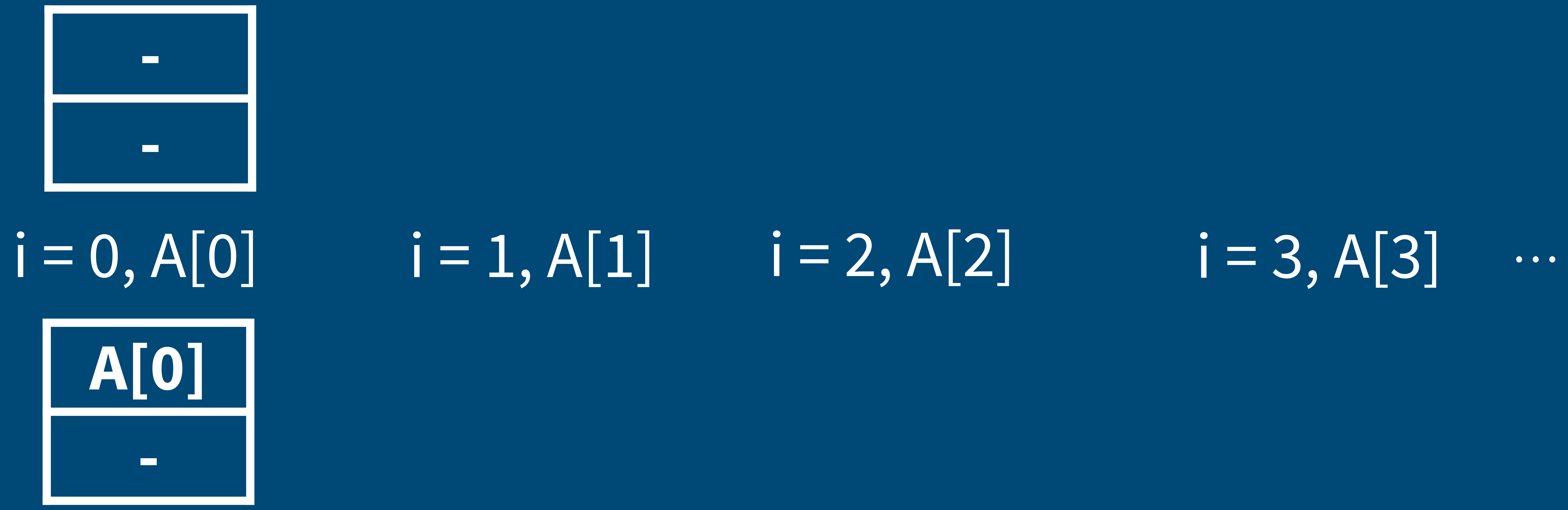
Symbolic CFG



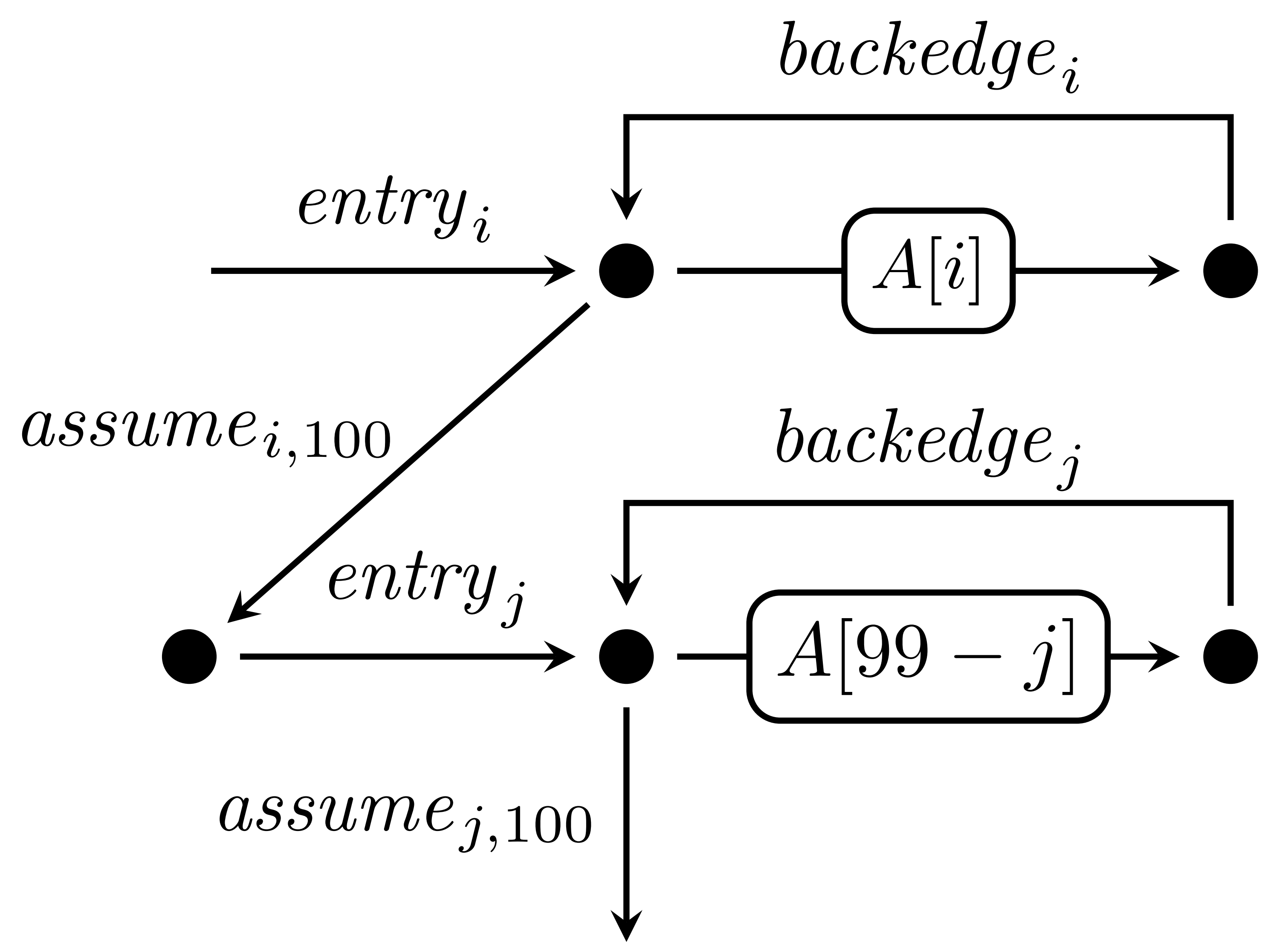
Assumptions:

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

Cache Analysis: Intuitively

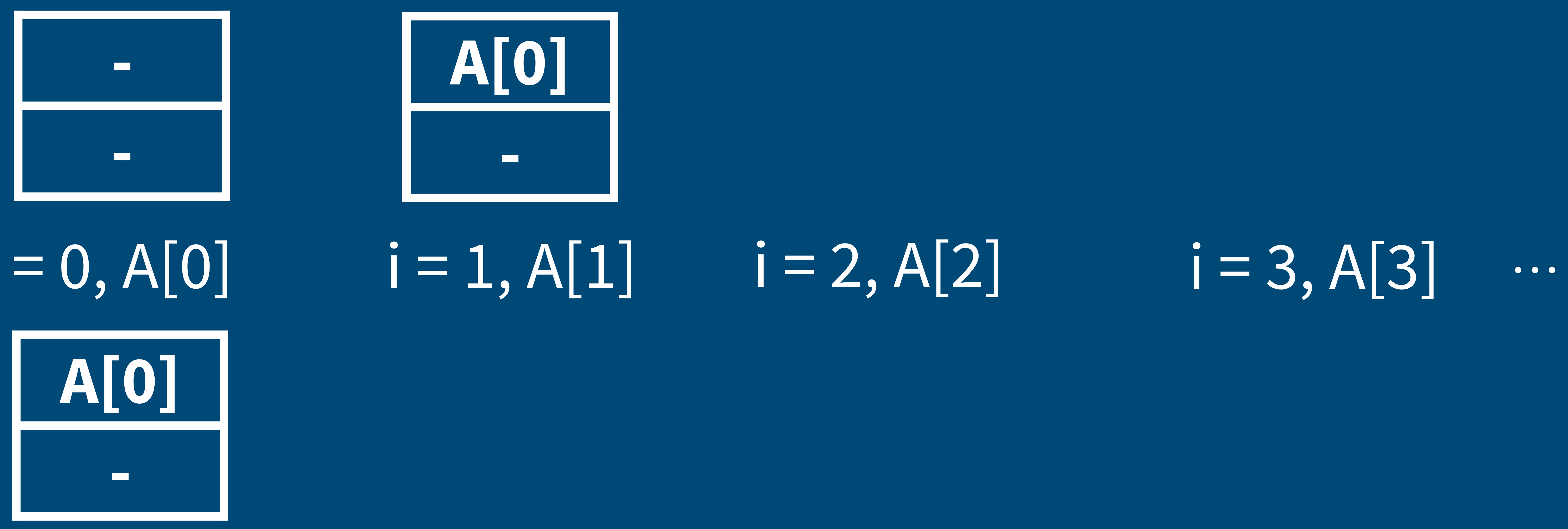


Symbolic CFG

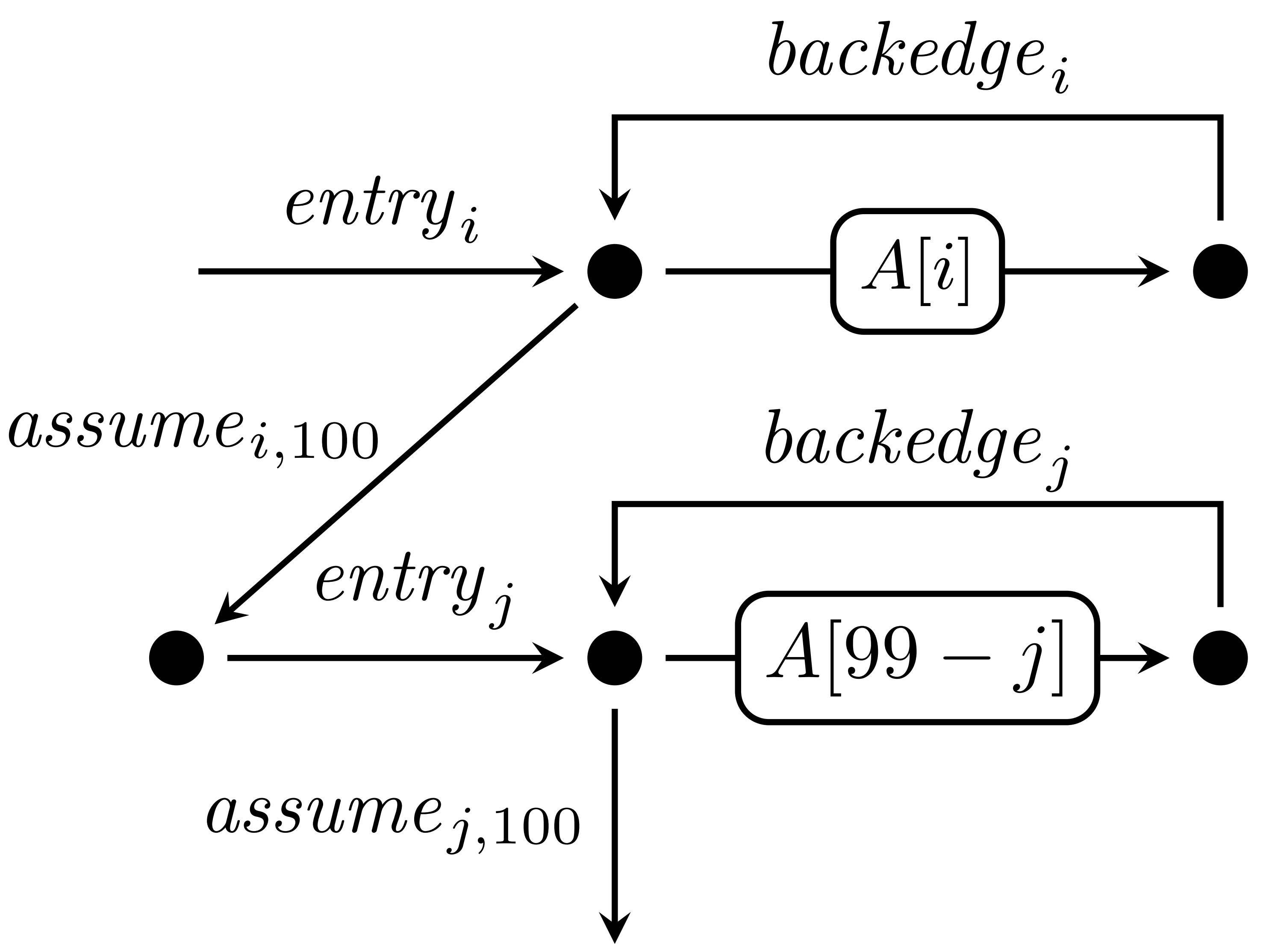


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

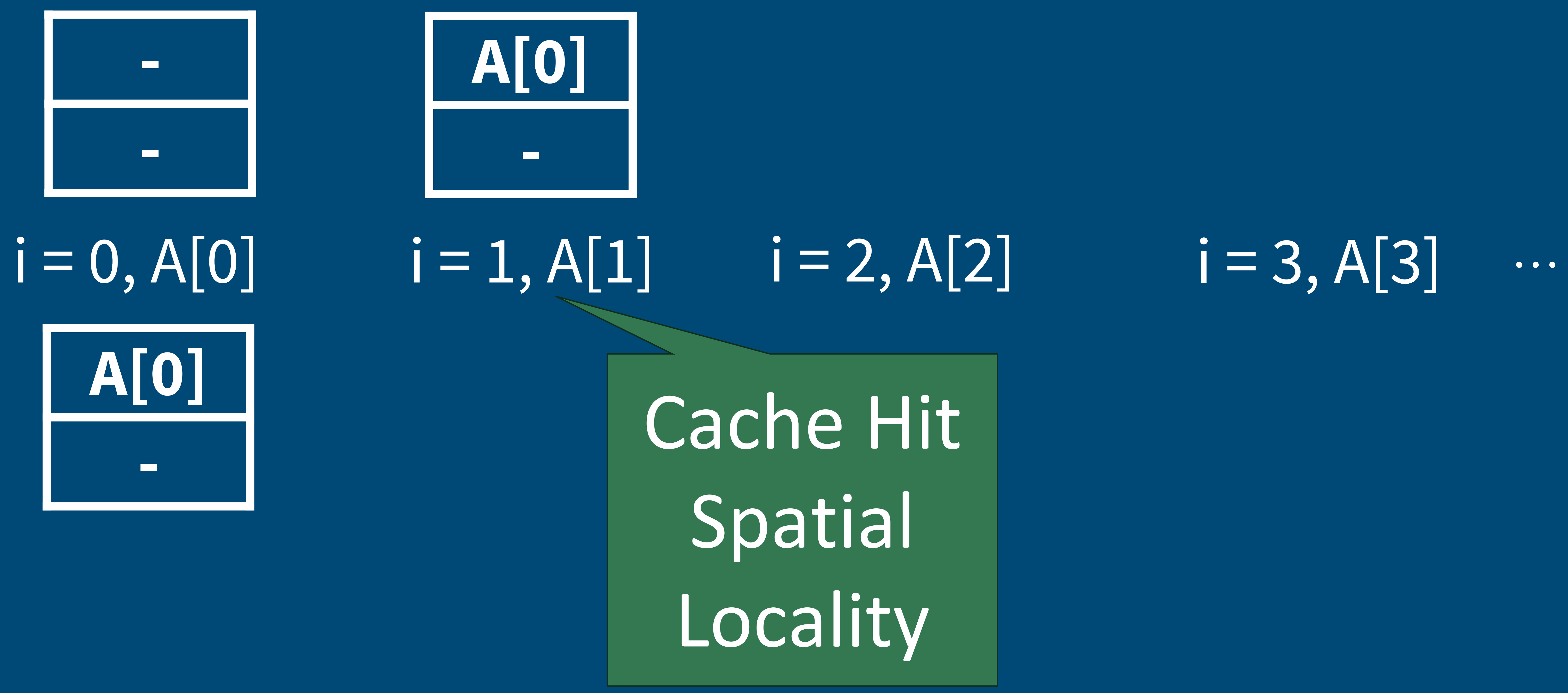


Symbolic CFG

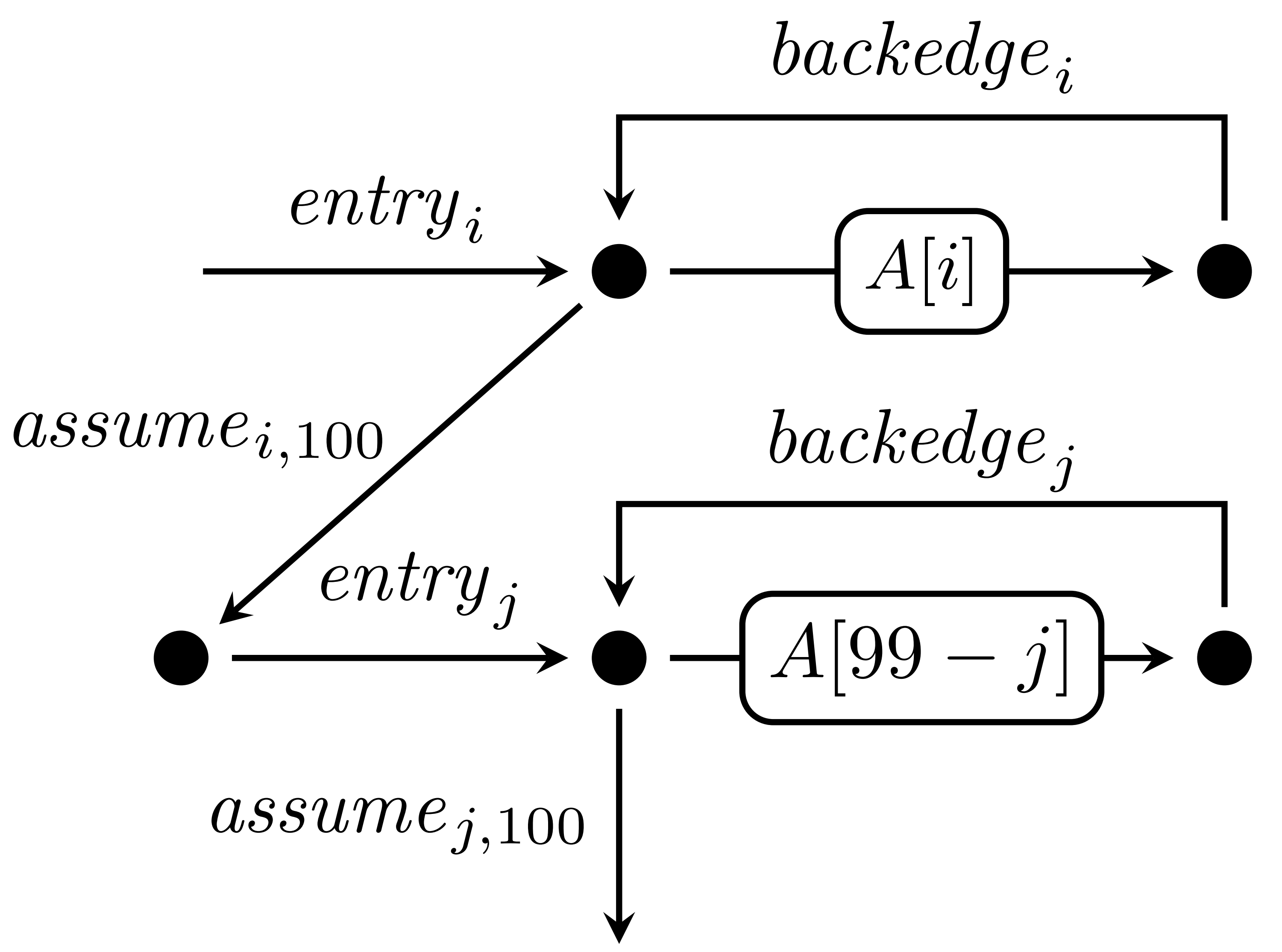


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

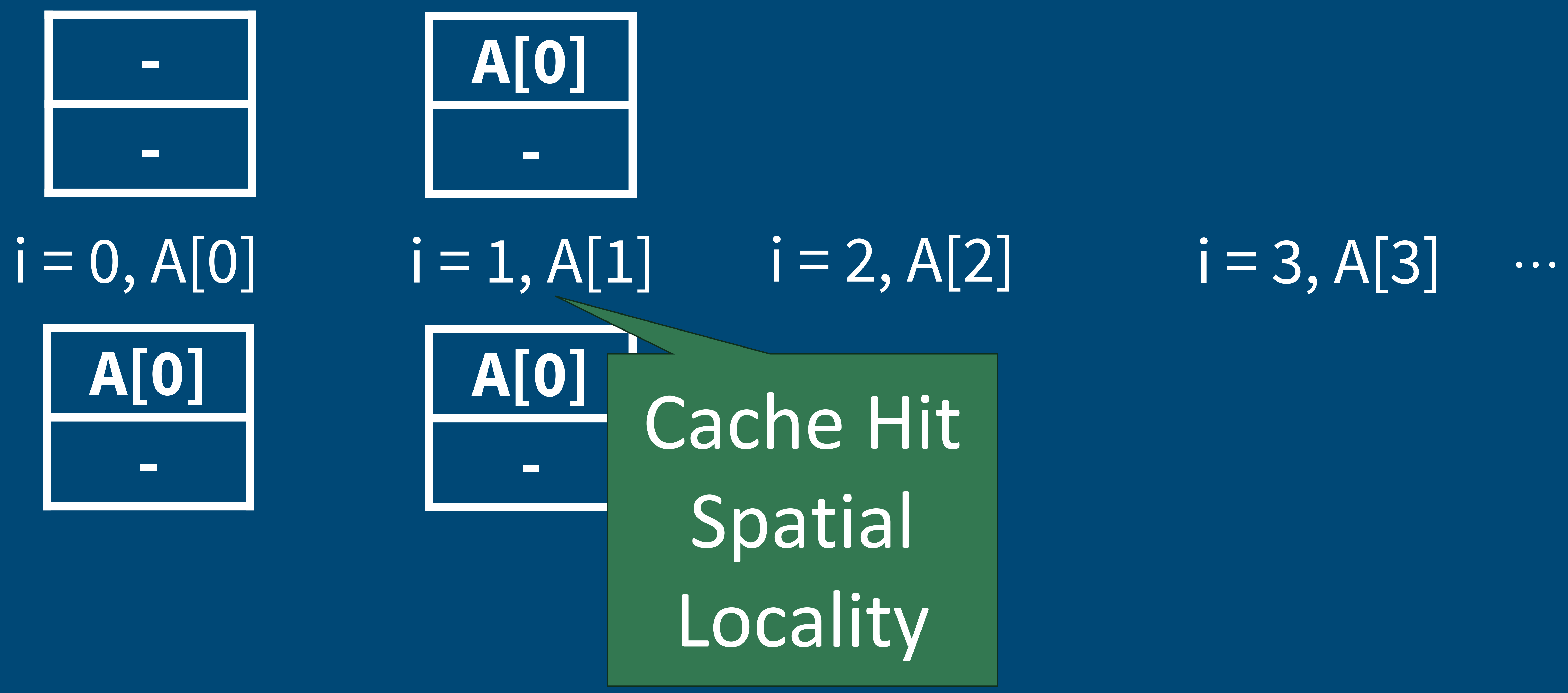


Symbolic CFG

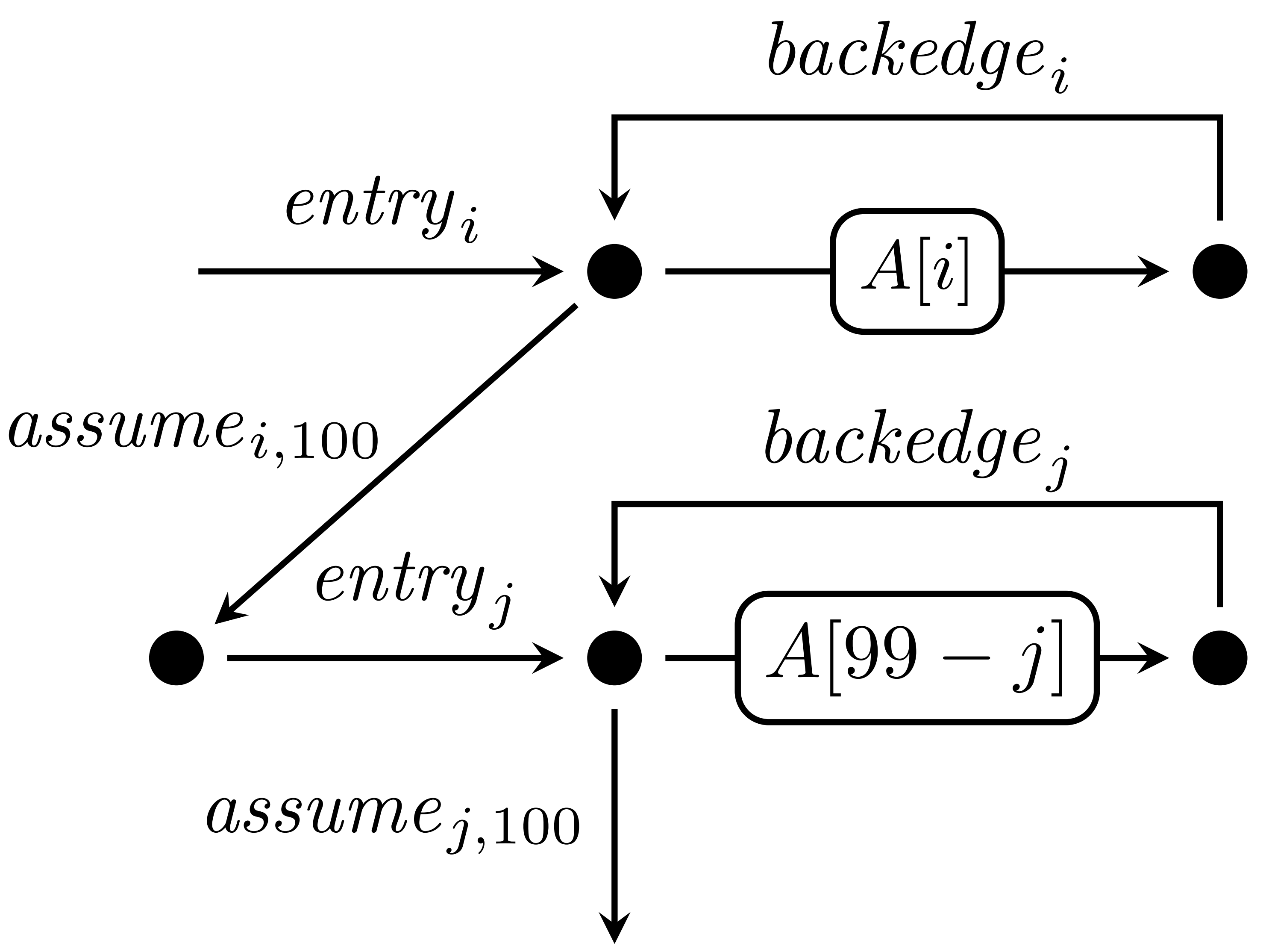


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

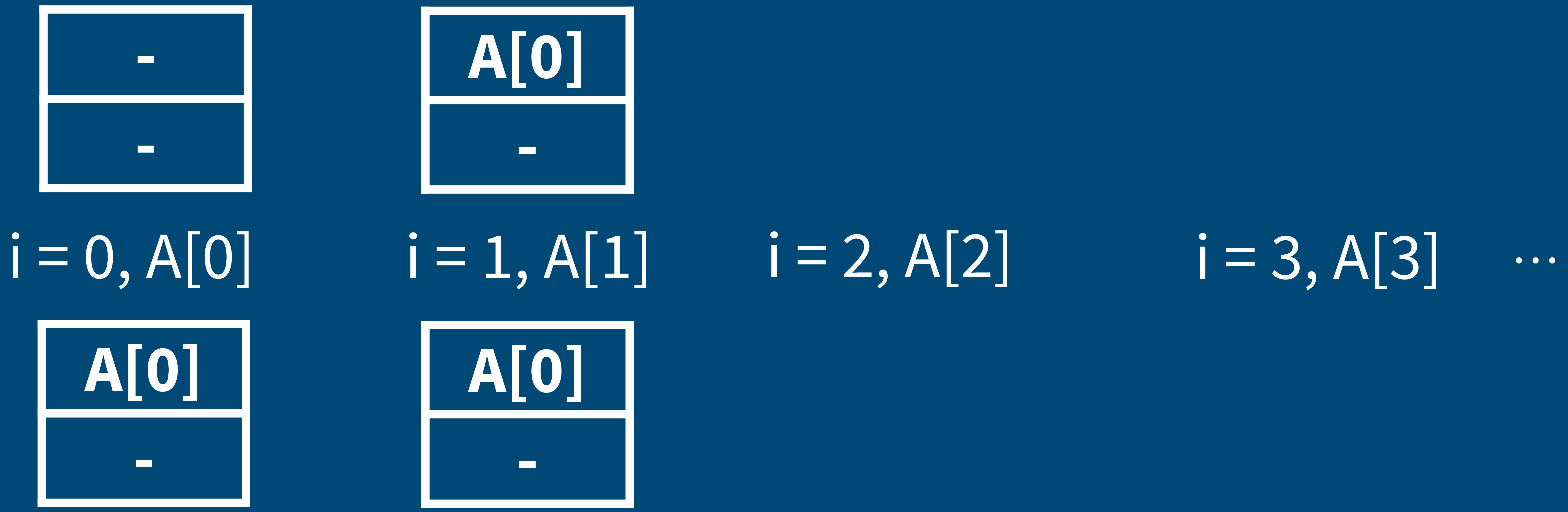


Symbolic CFG

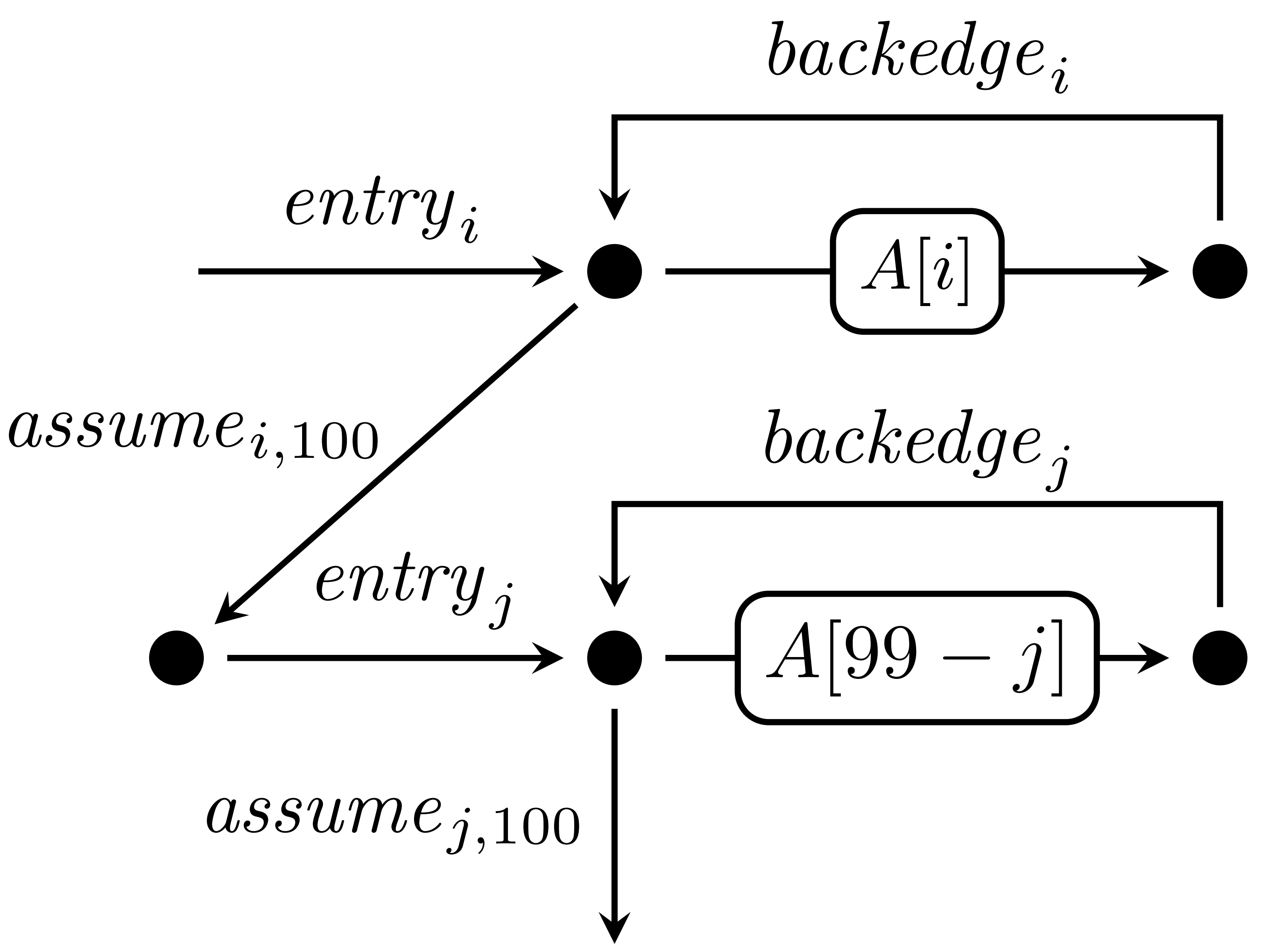


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

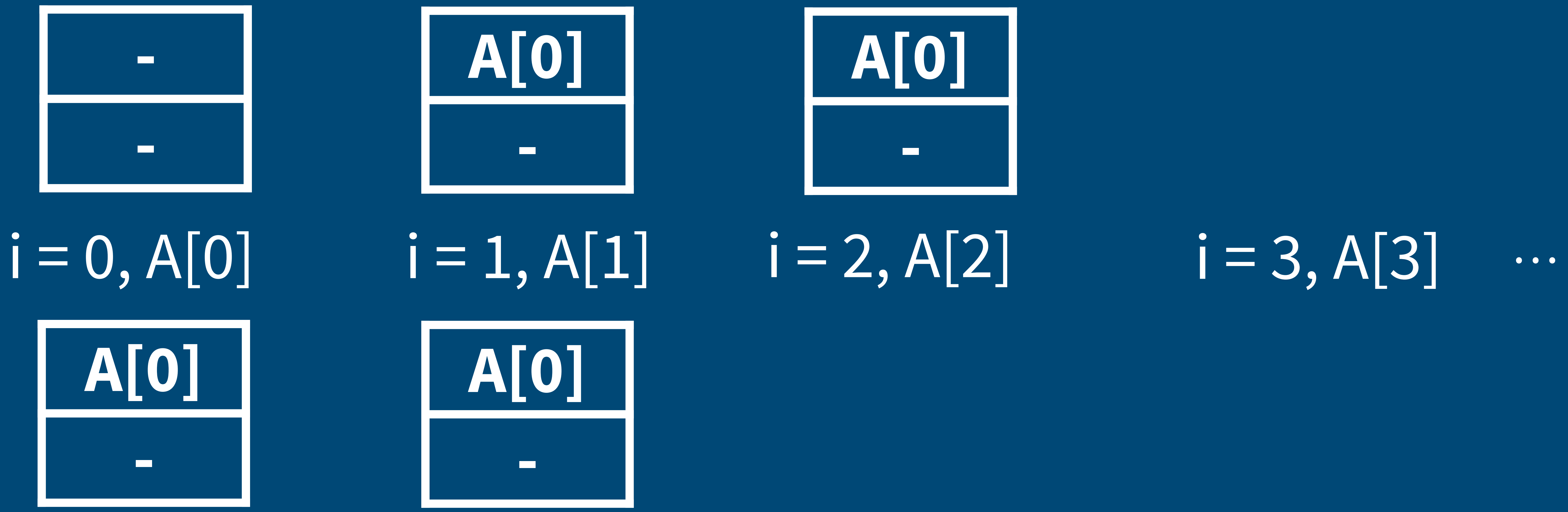


Symbolic CFG

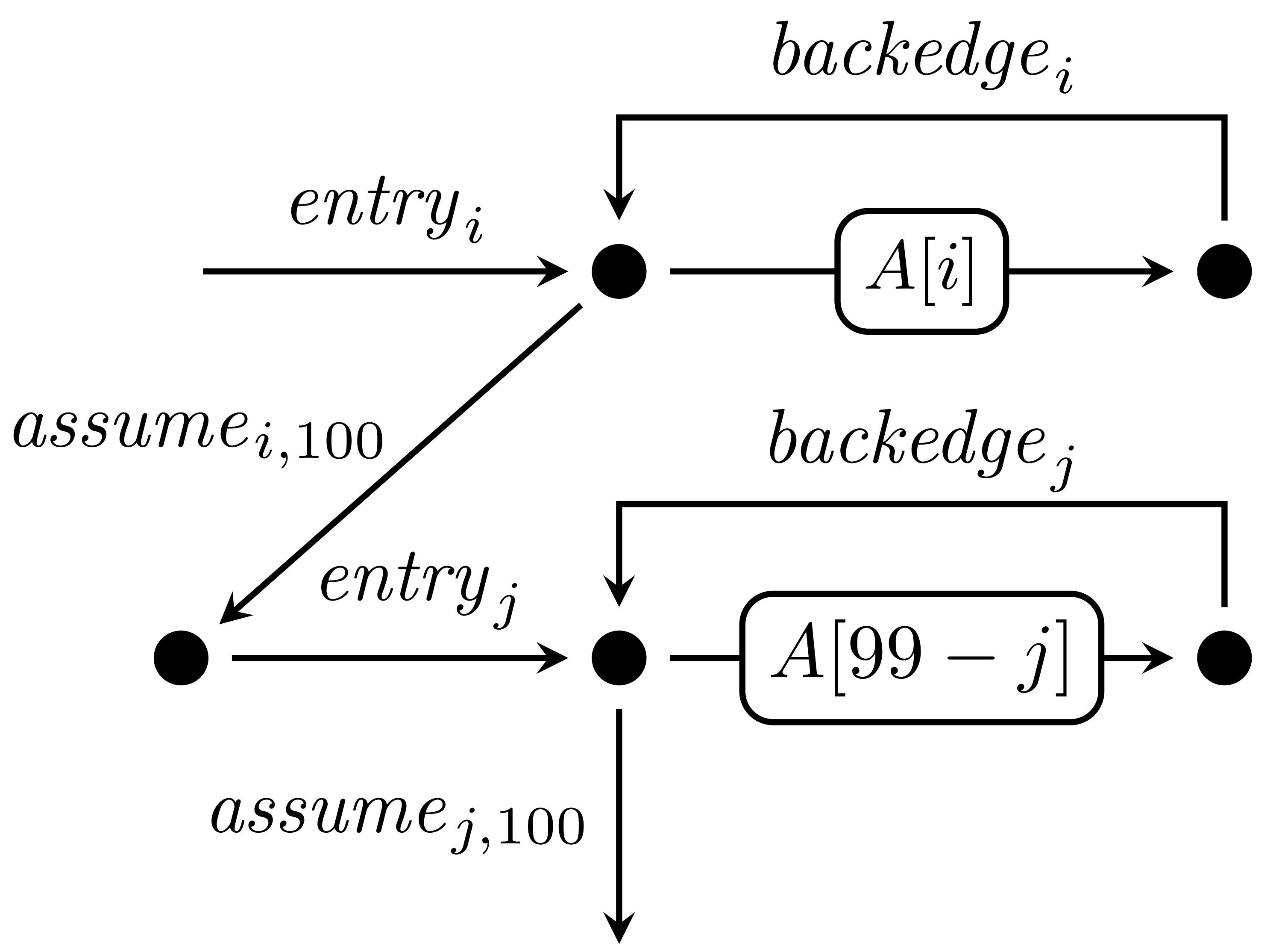


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively



Symbolic CFG



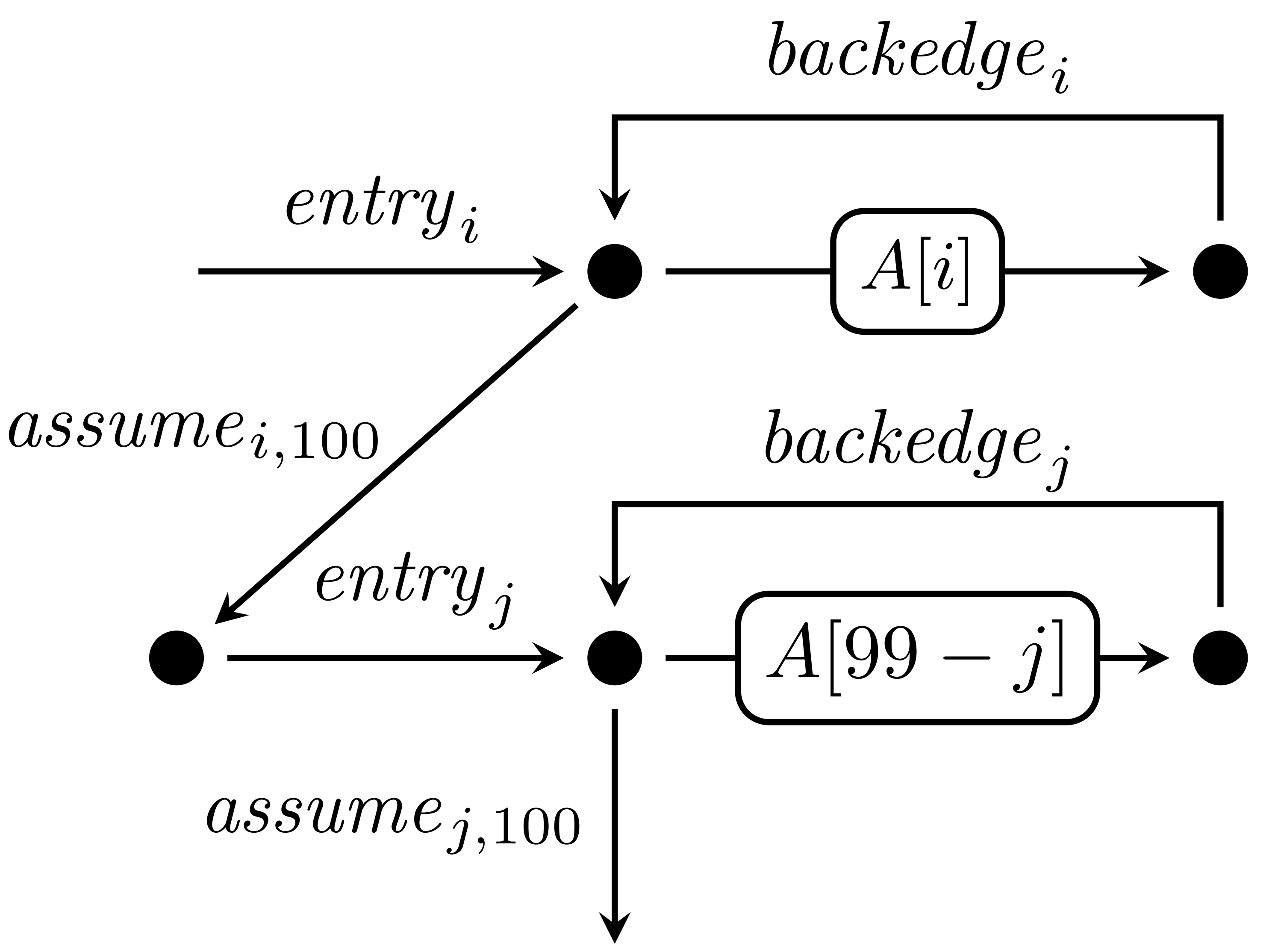
Assumptions:

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

Cache Analysis: Intuitively

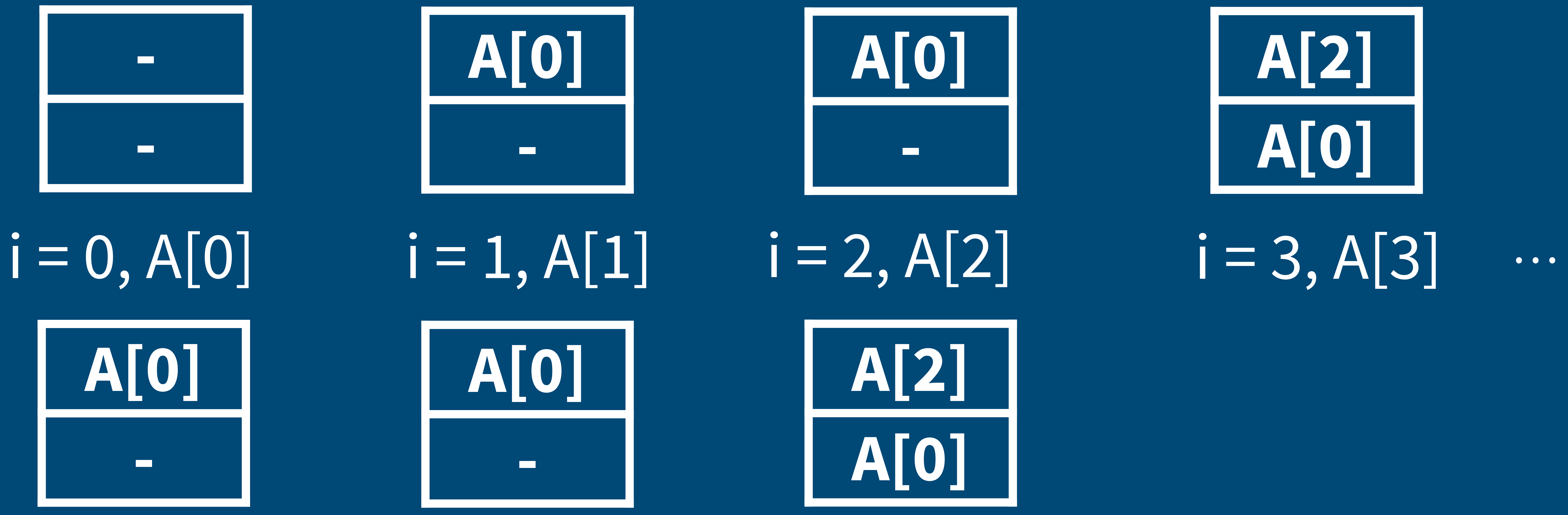
-	A[0]	A[0]		
-	-	-		
i = 0, A[0]	i = 1, A[1]	i = 2, A[2]	i = 3, A[3]	…
A[0]	A[0]	A[2]		
-	-	A[0]		

Symbolic CFG

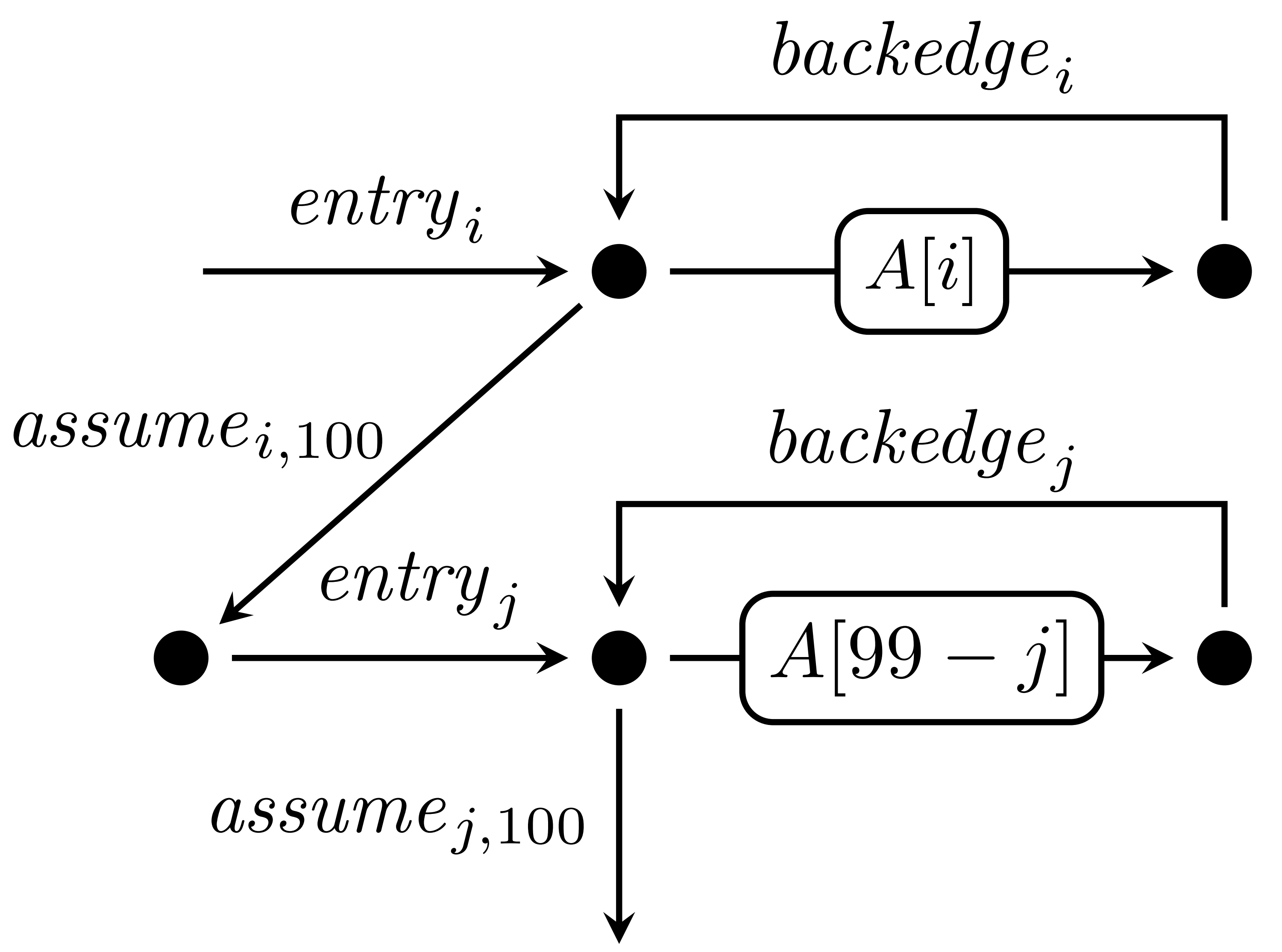


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

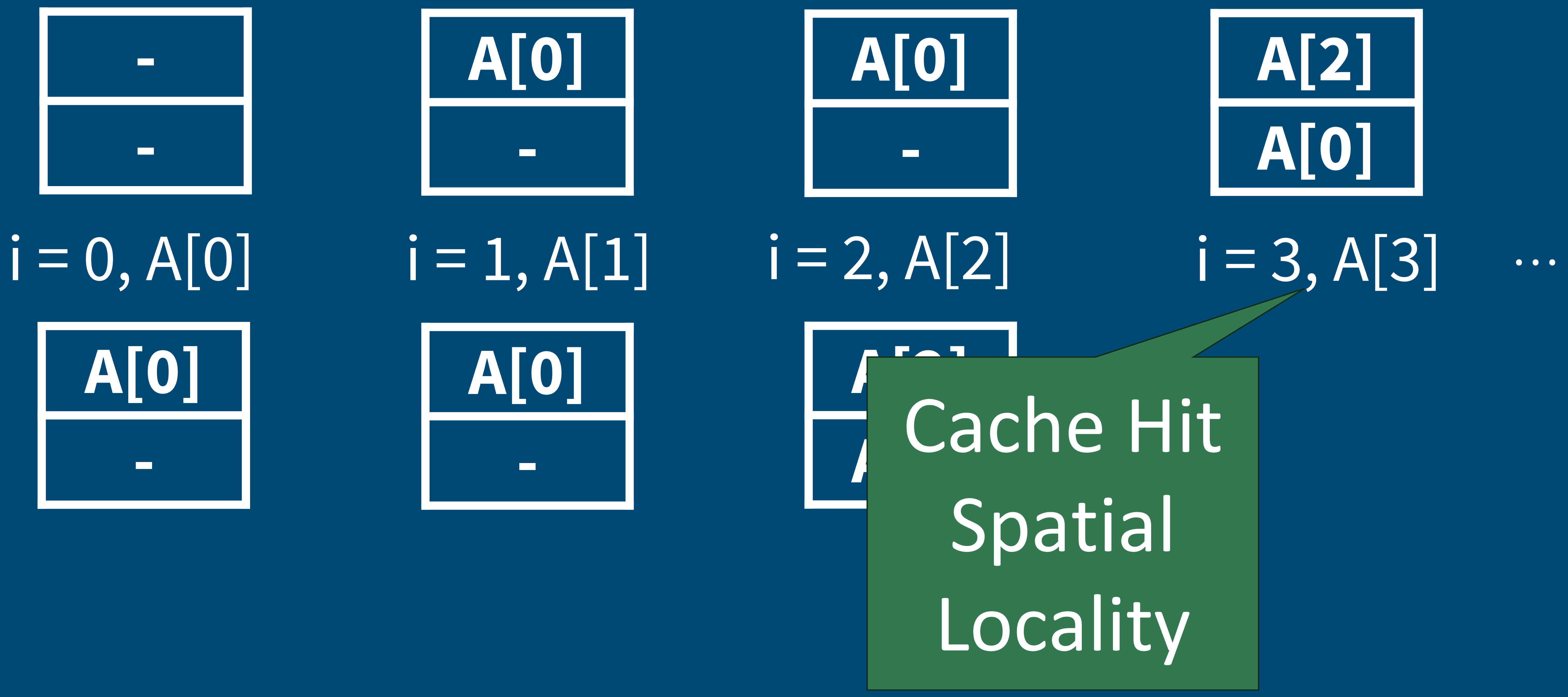


Symbolic CFG

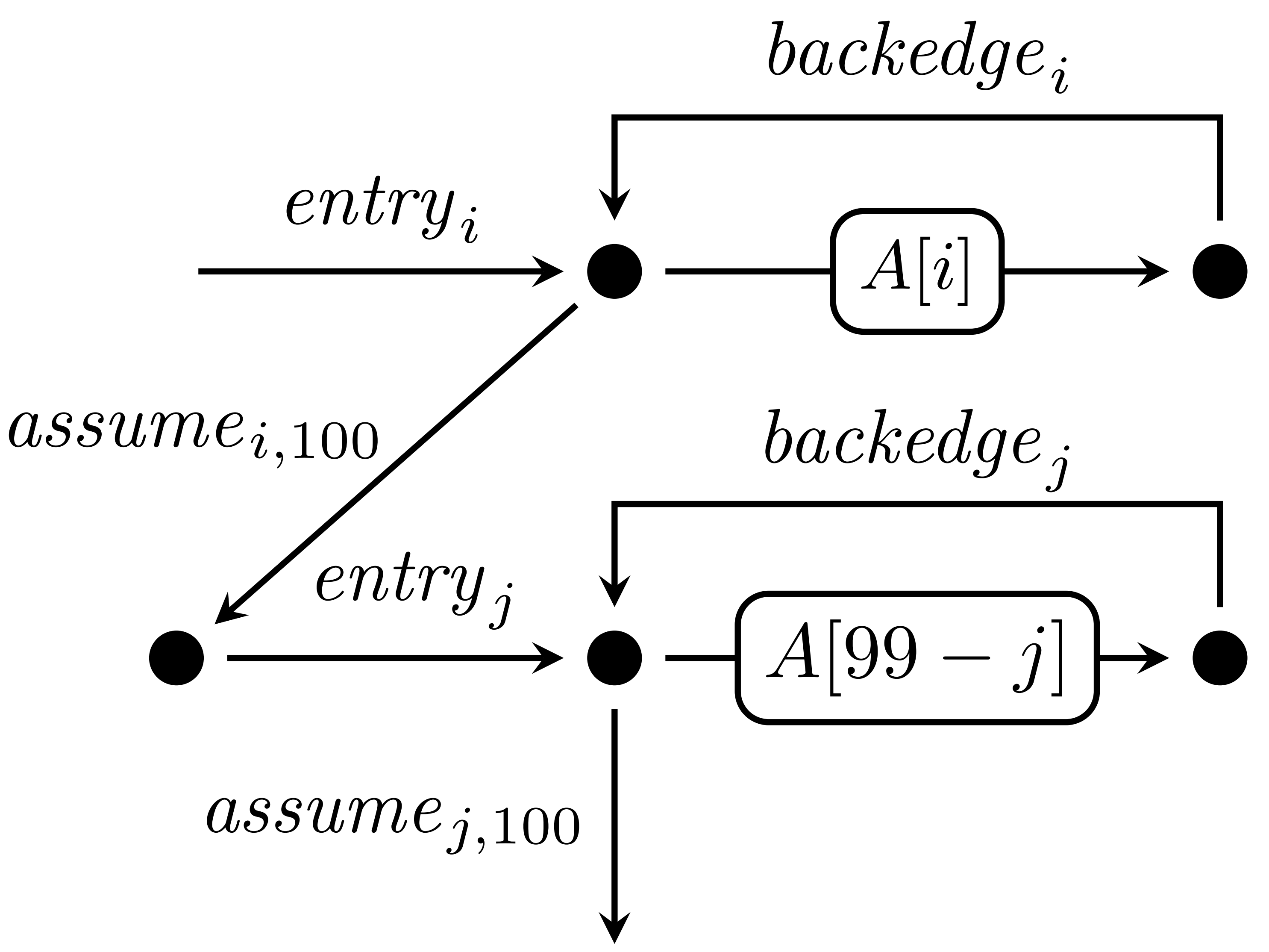


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

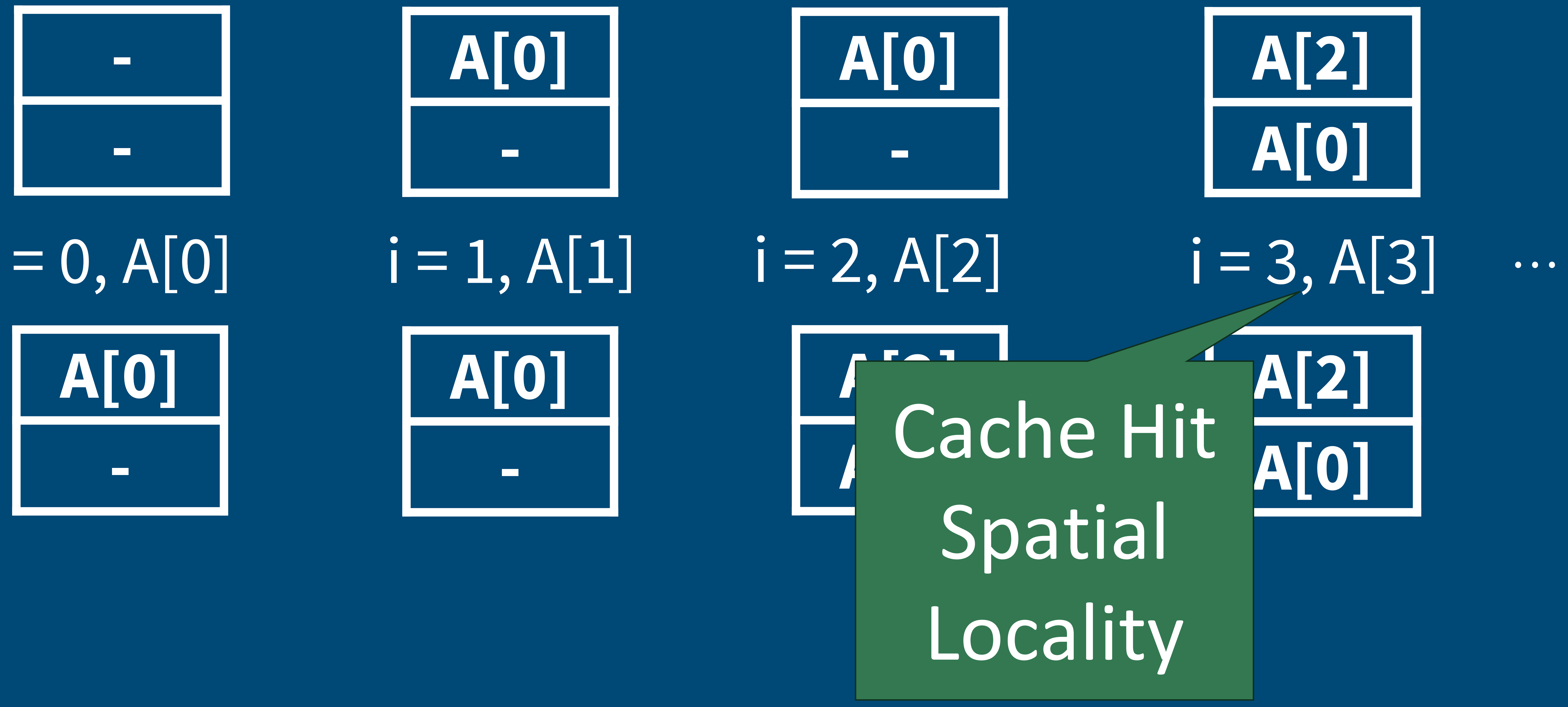


Symbolic CFG

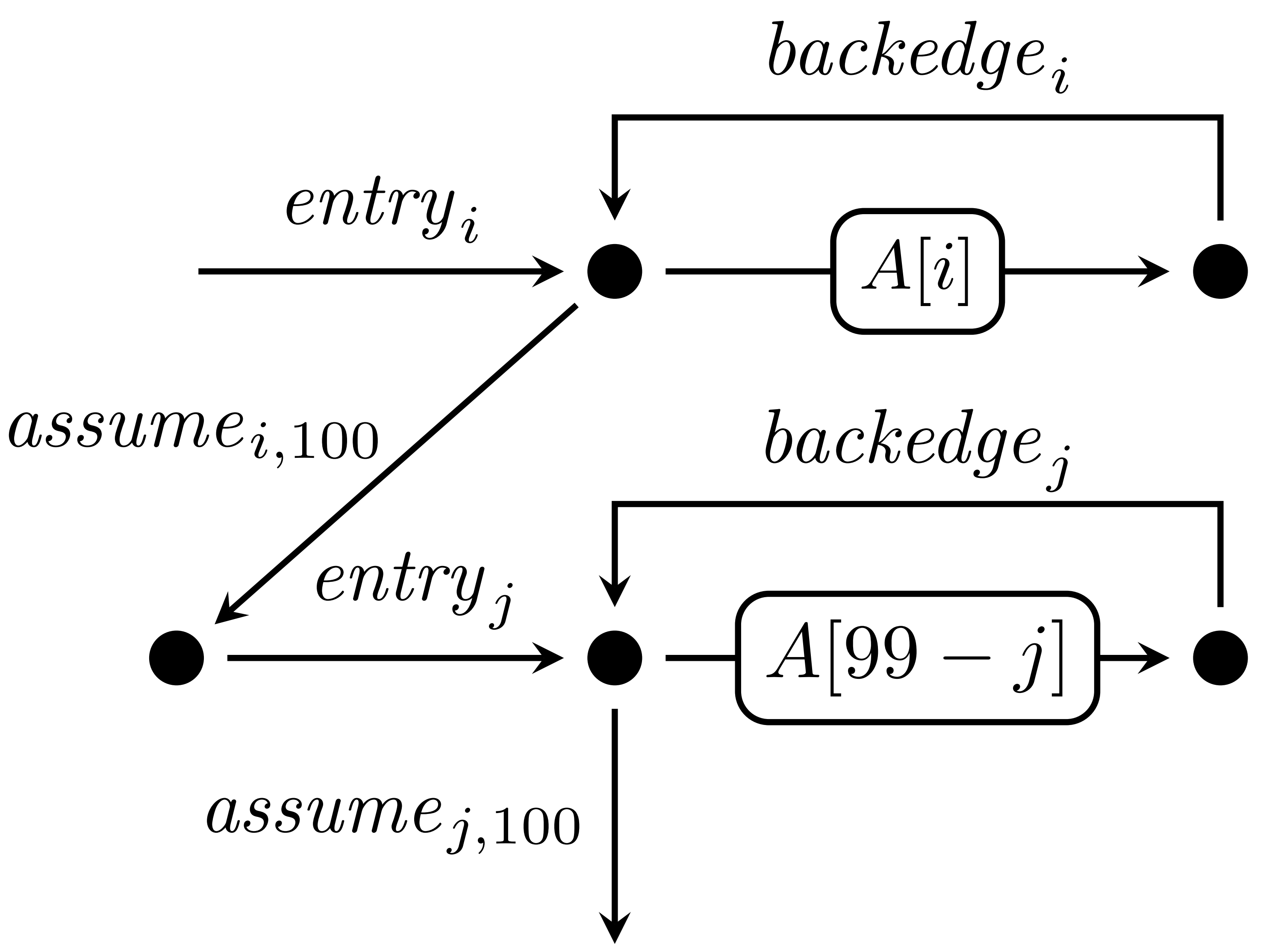


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

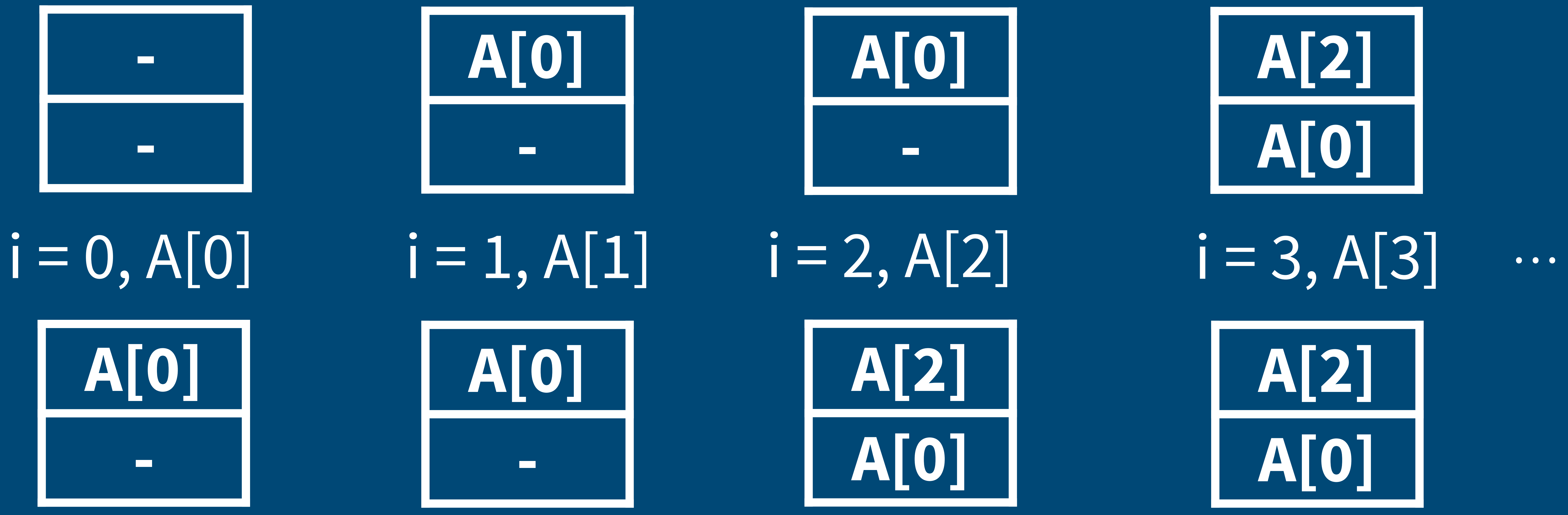


Symbolic CFG

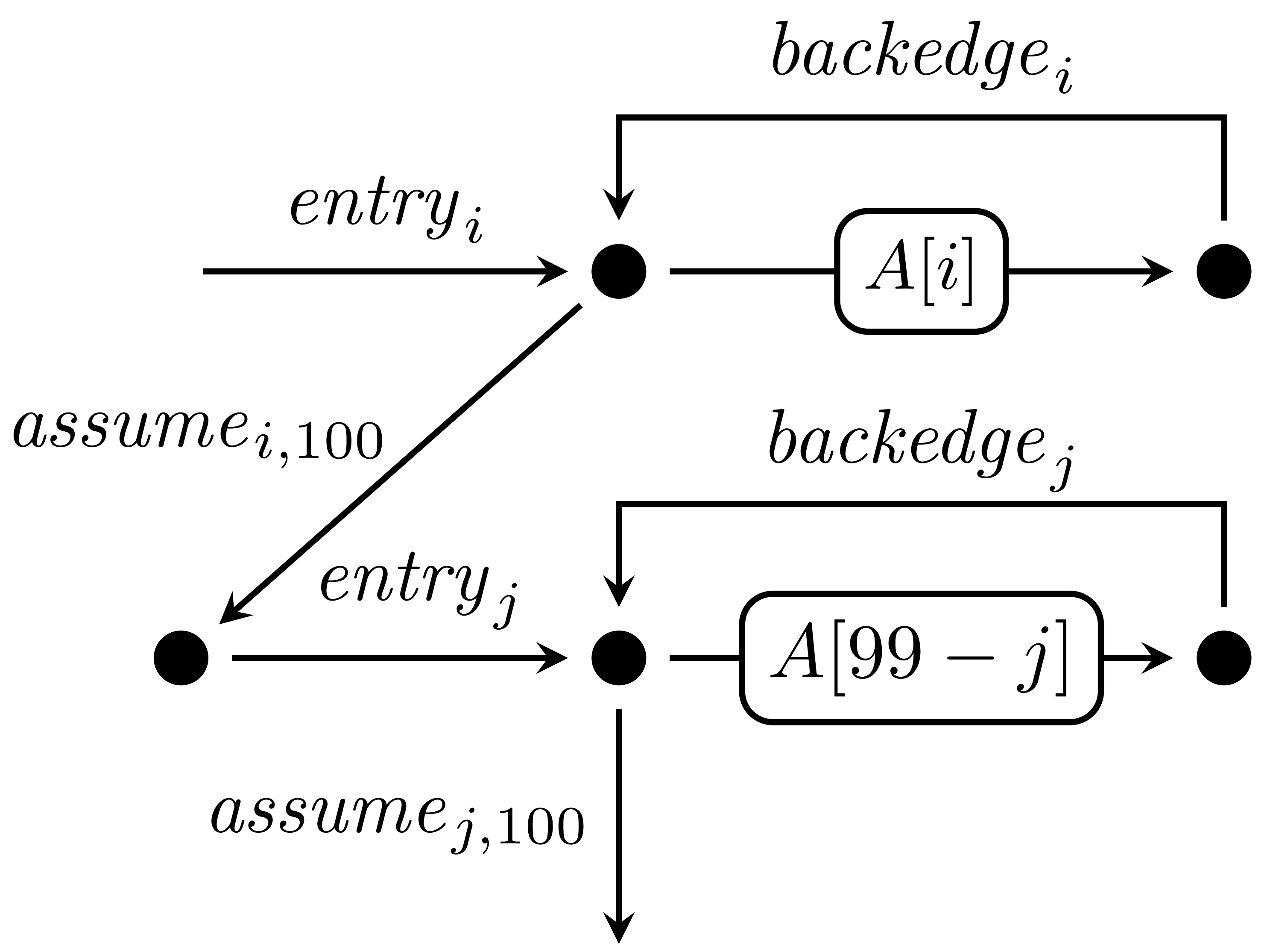


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

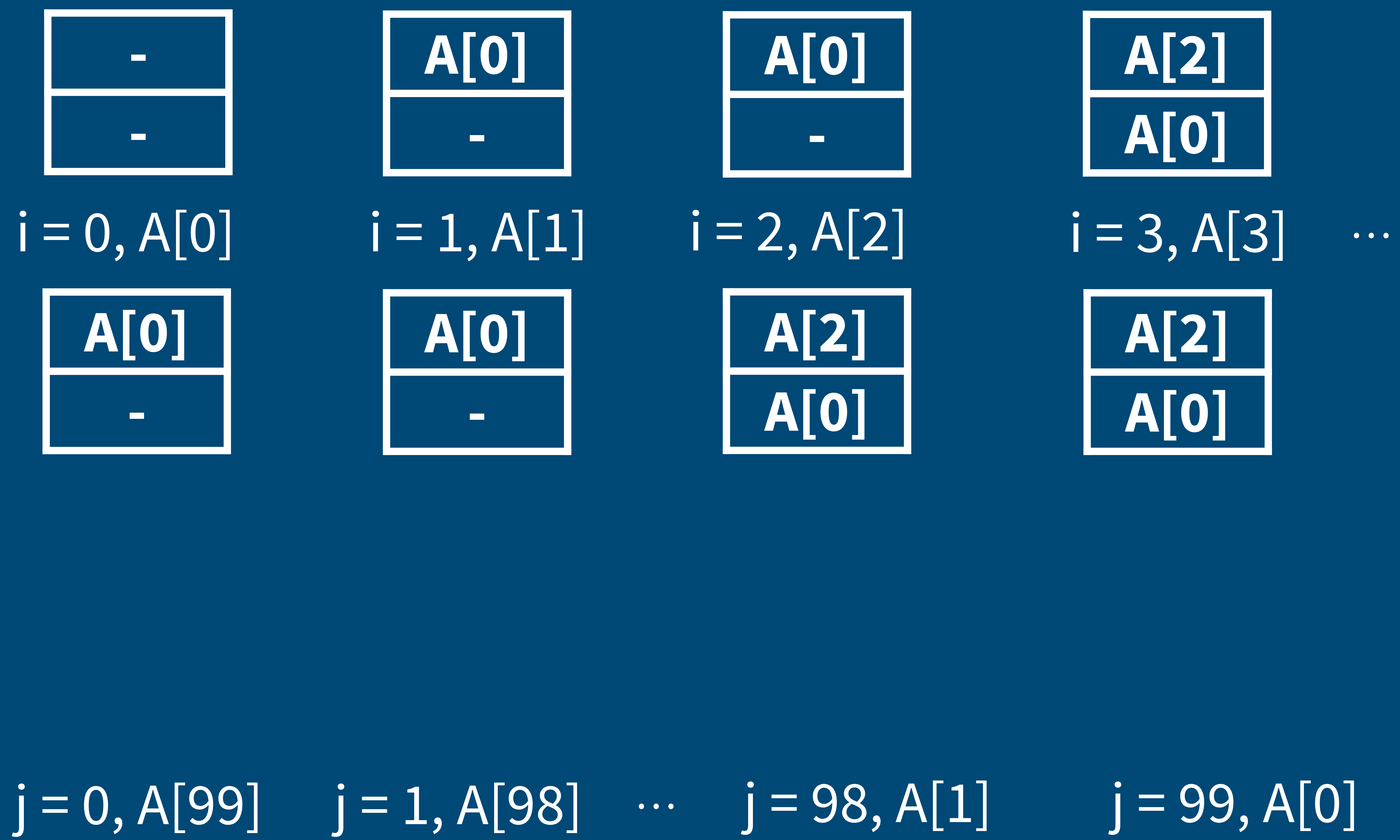


Symbolic CFG

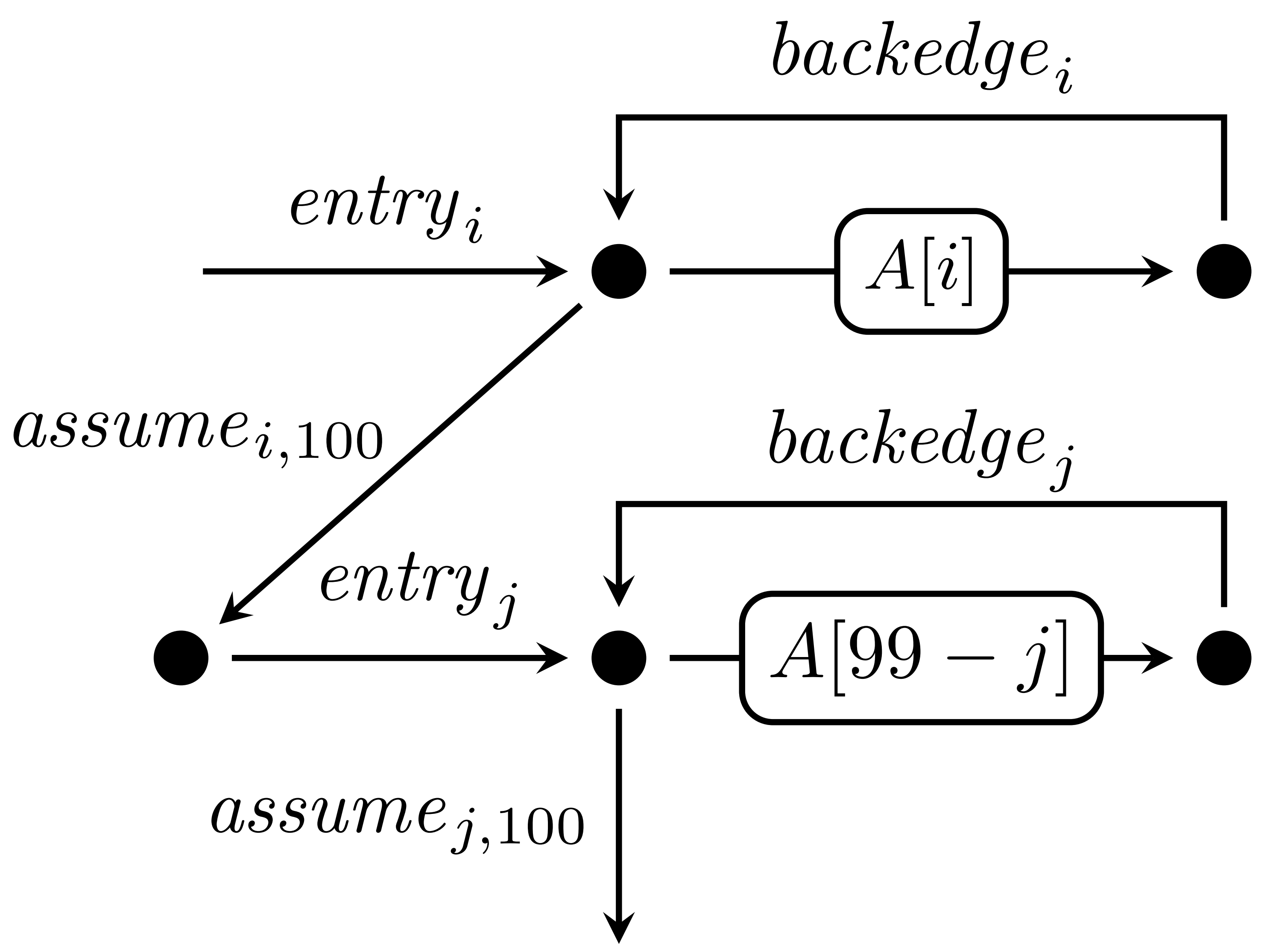


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

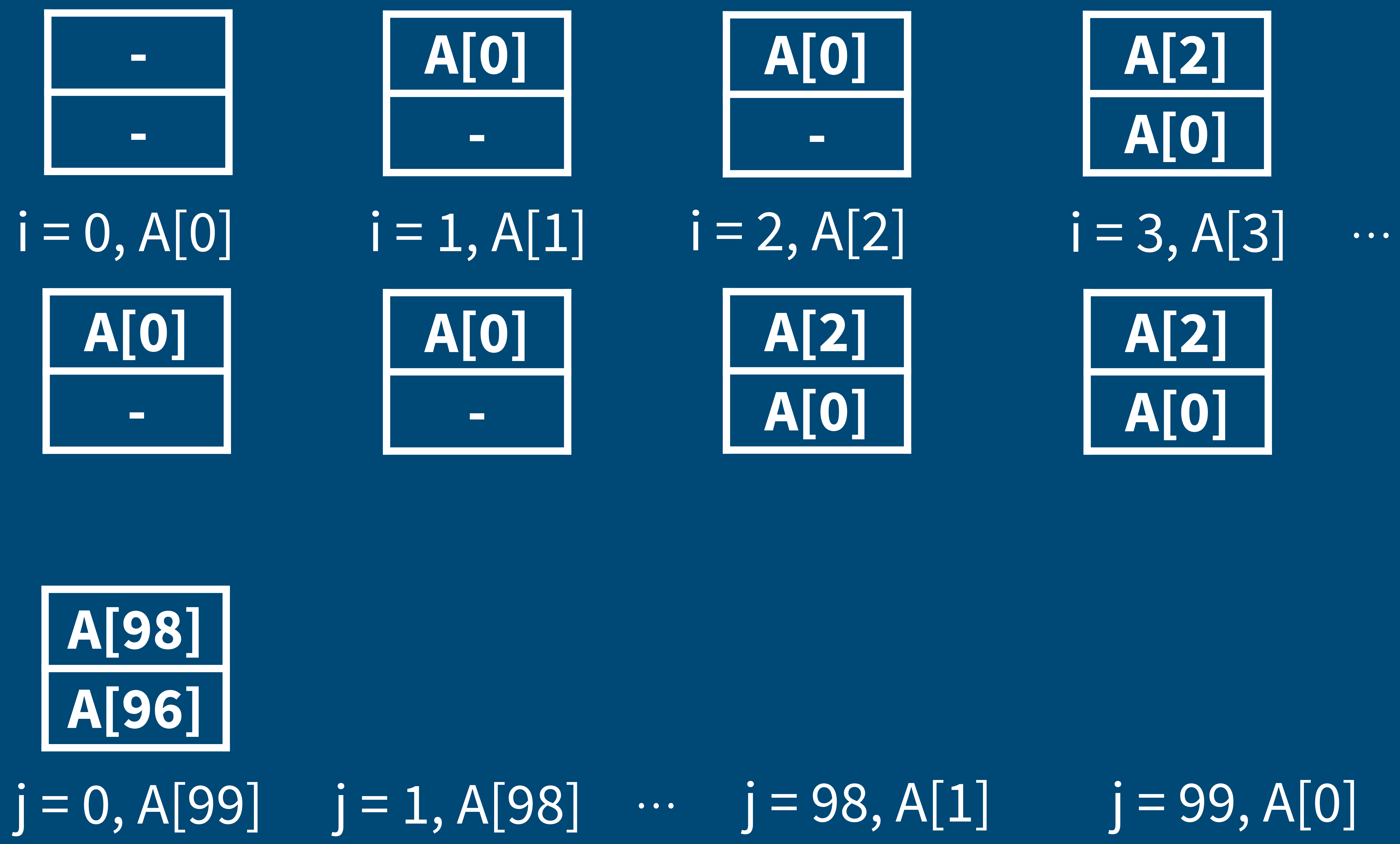


Symbolic CFG

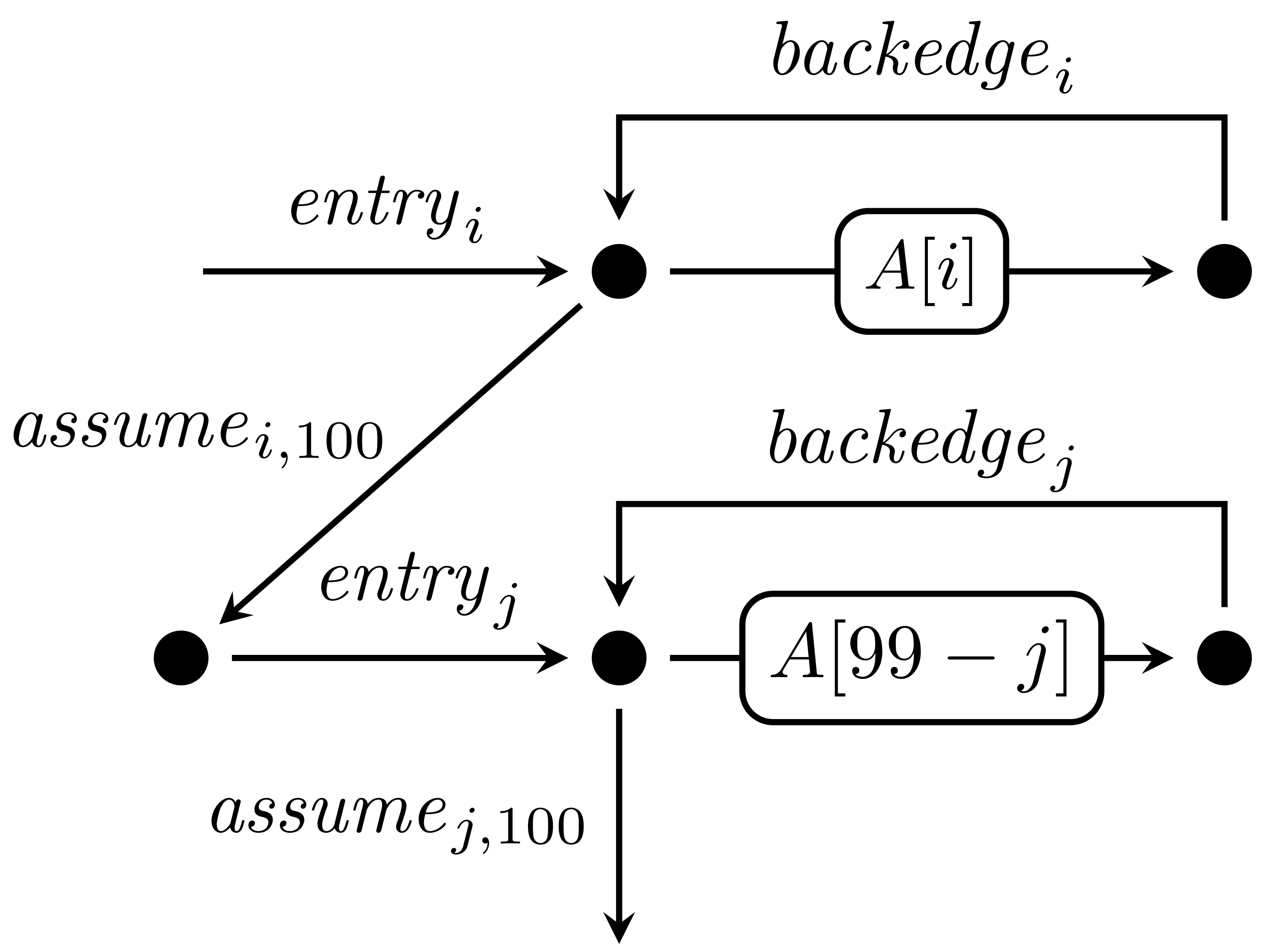


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

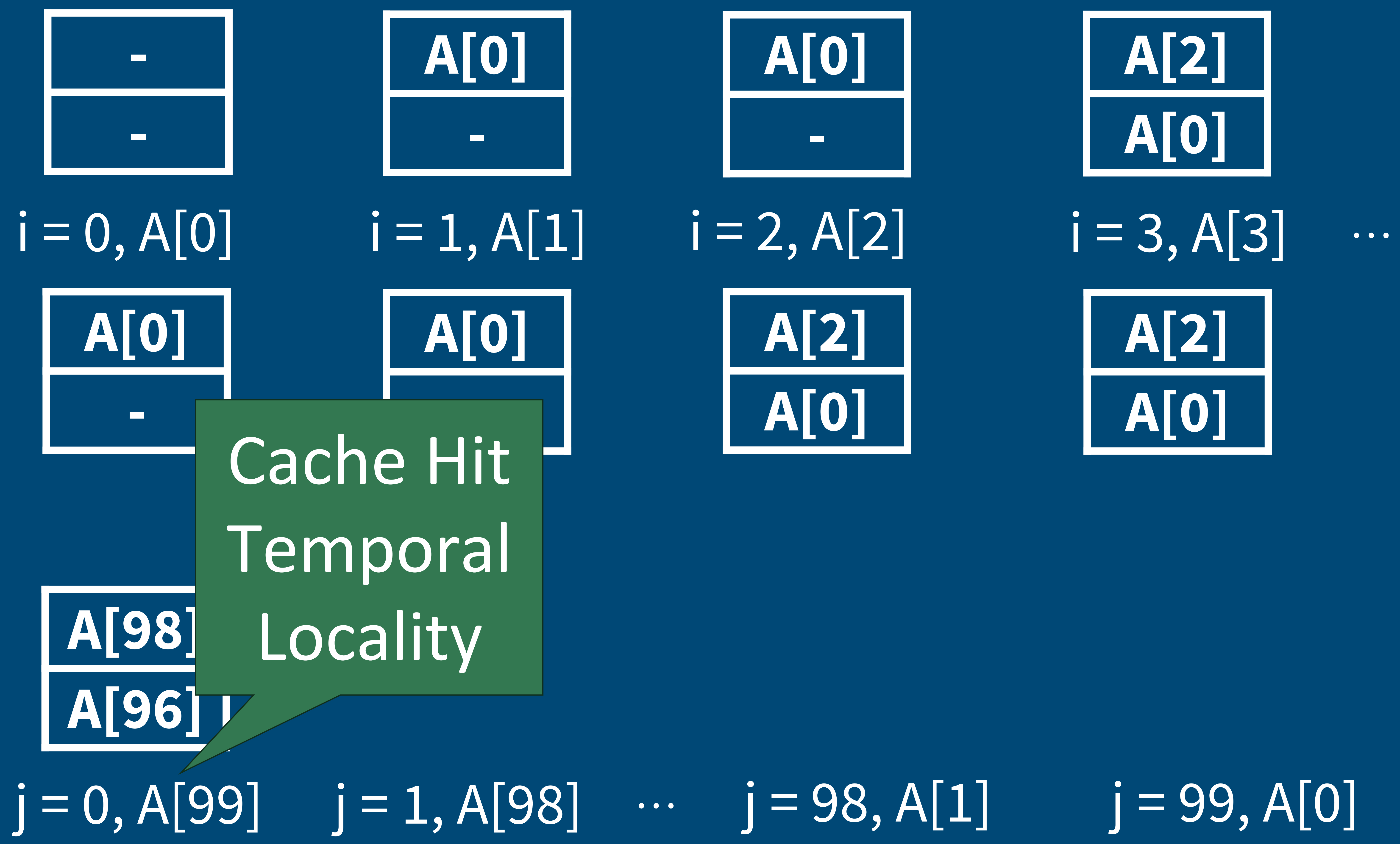


Symbolic CFG

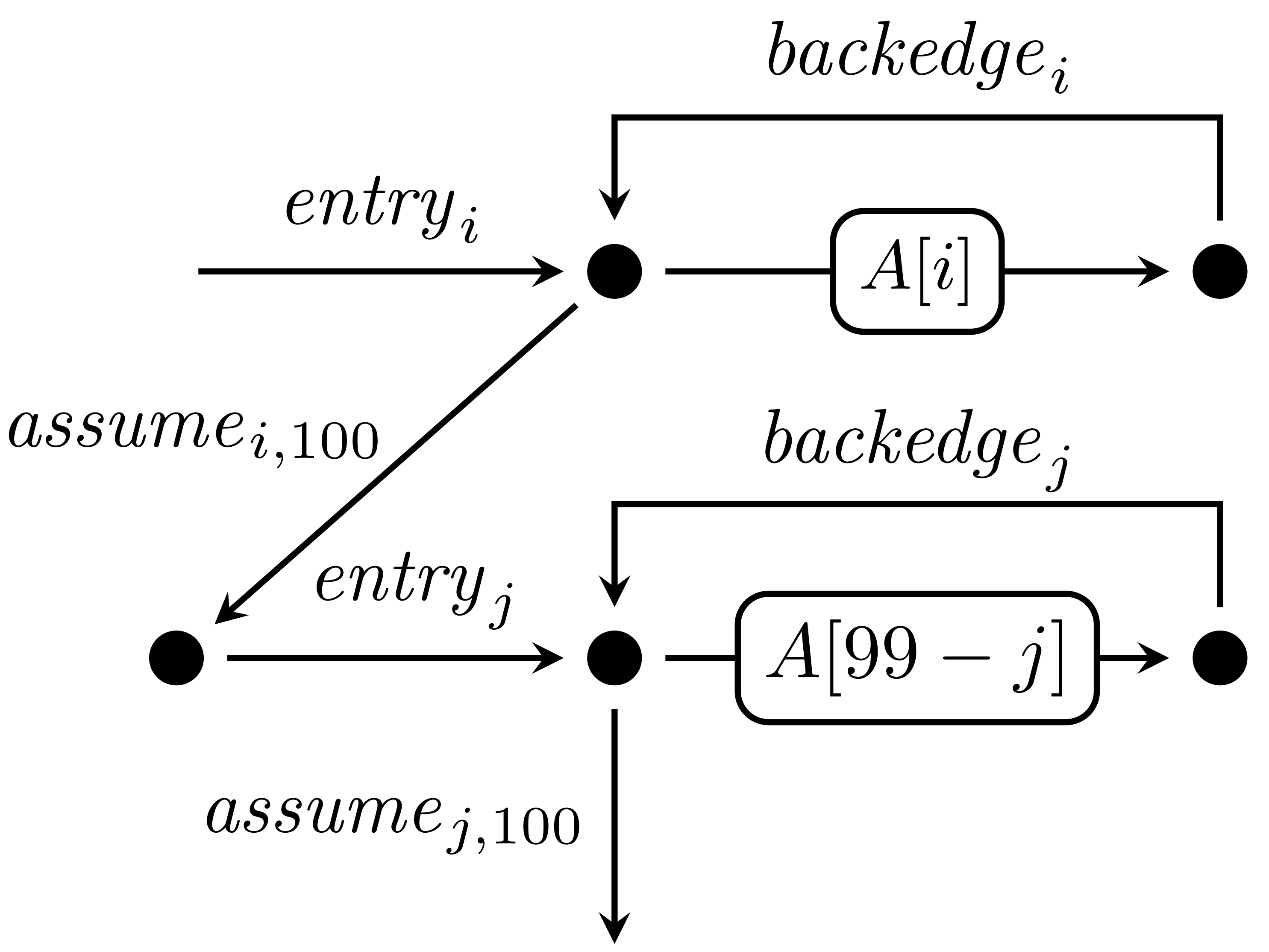


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively



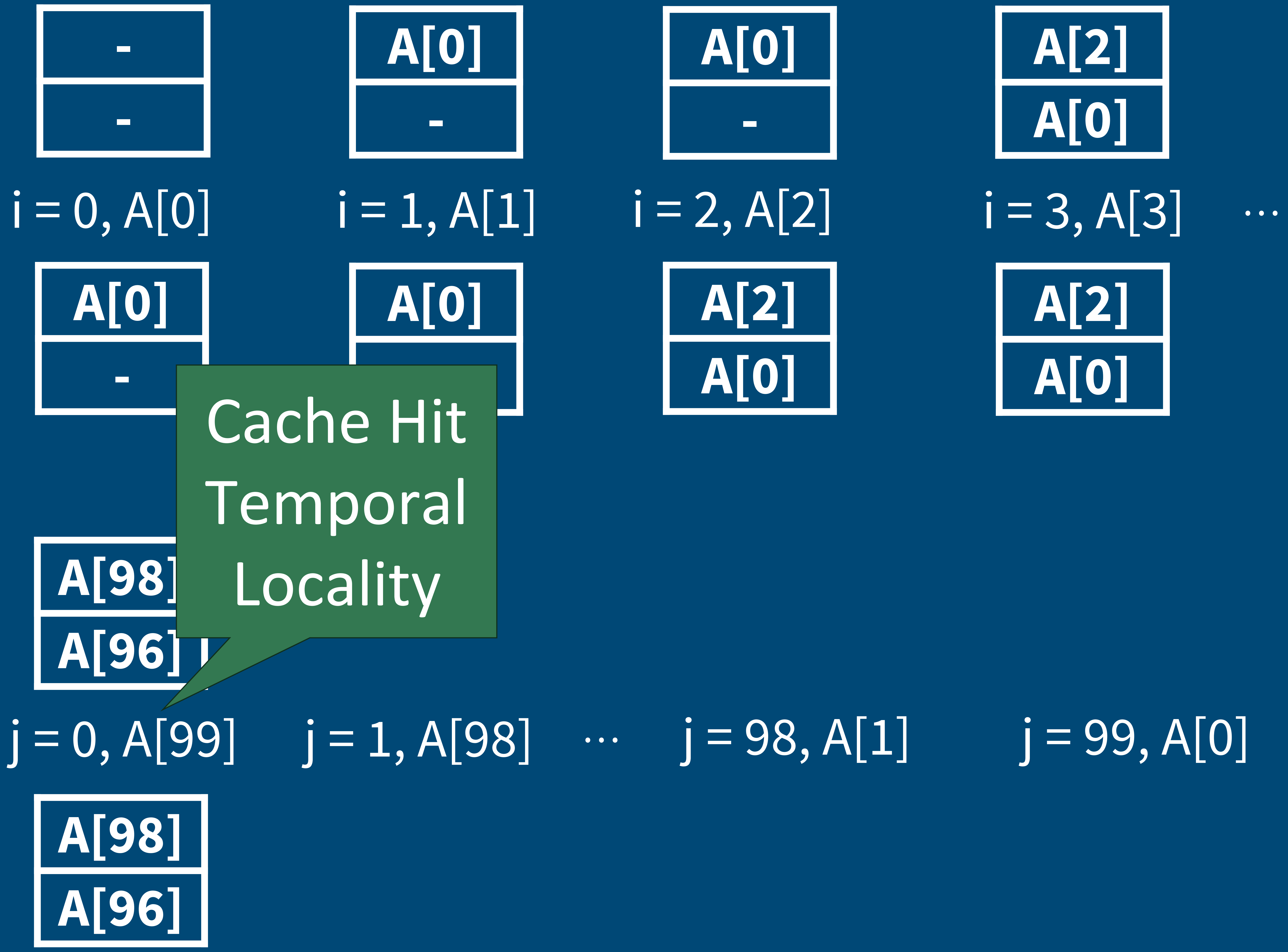
Symbolic CFG



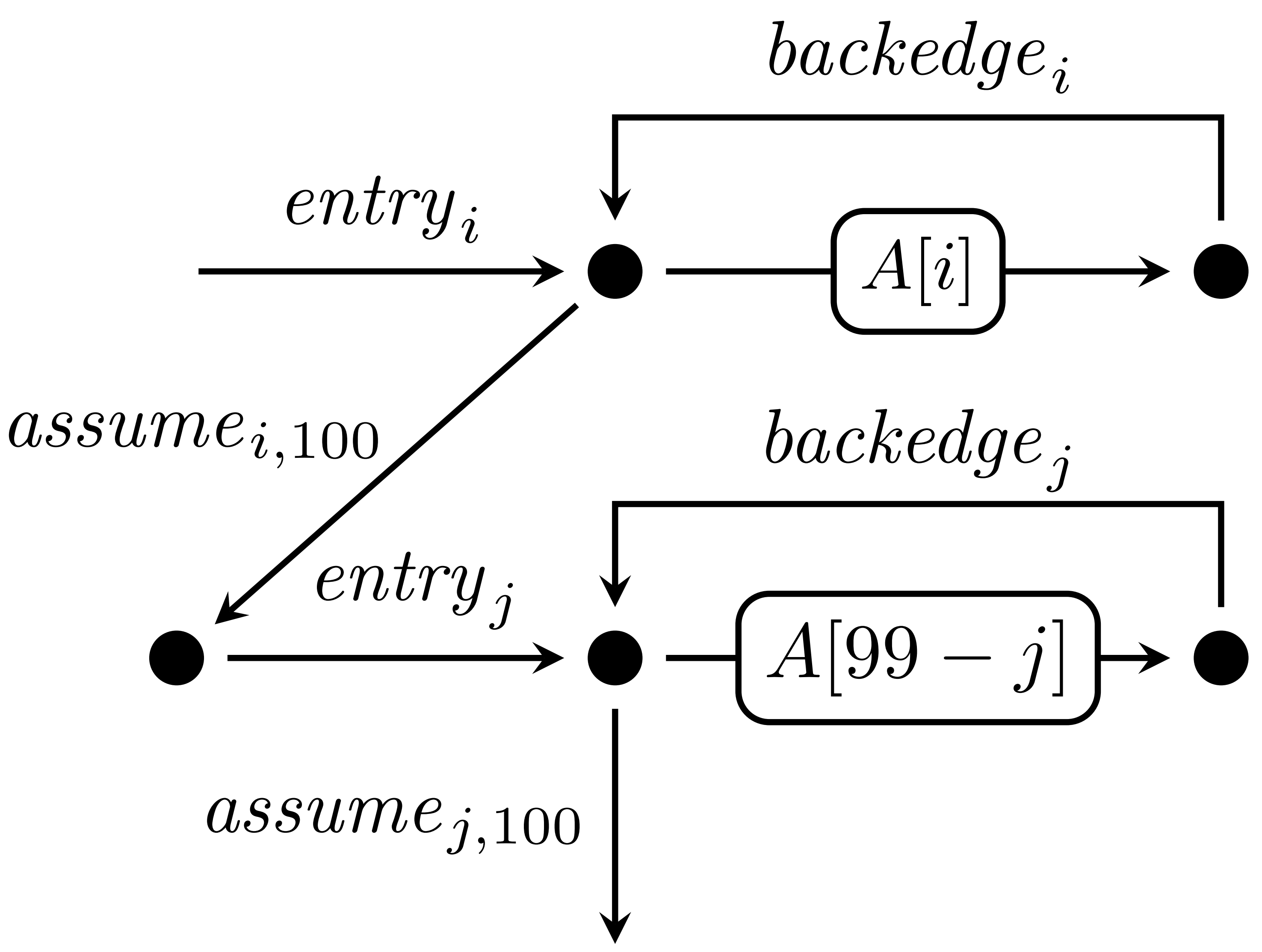
Assumptions:

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

Cache Analysis: Intuitively



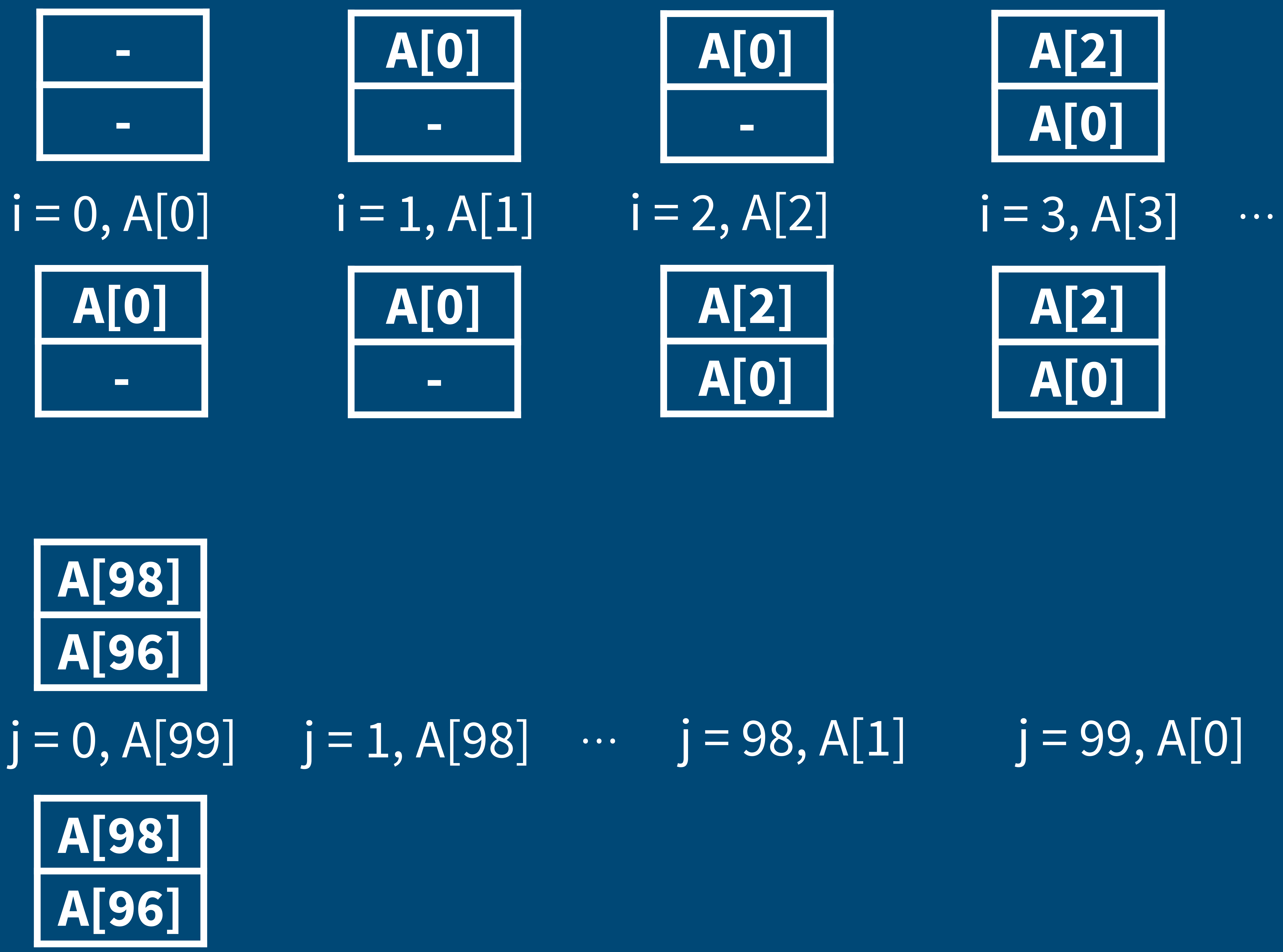
Symbolic CFG



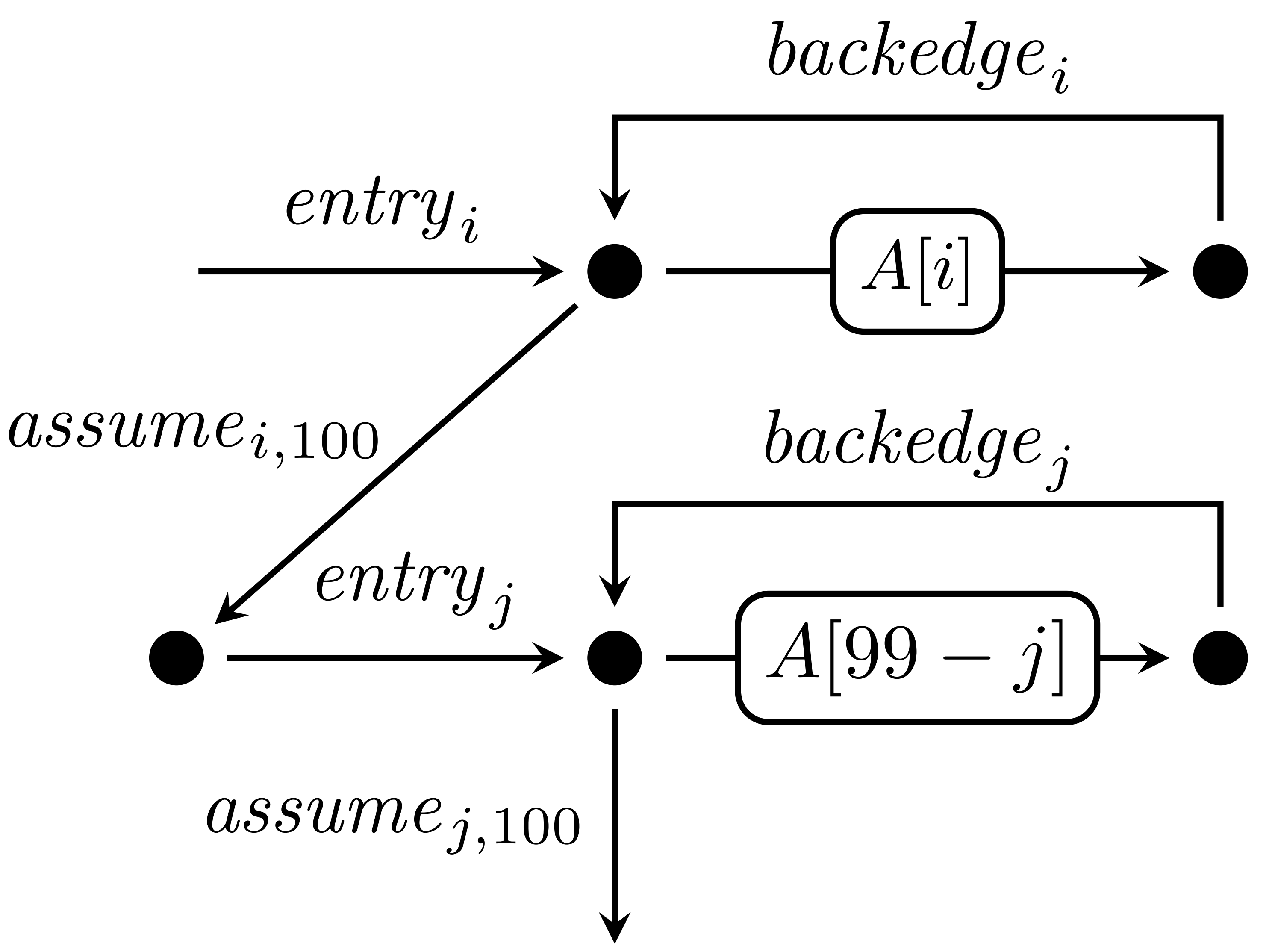
Assumptions:

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

Cache Analysis: Intuitively

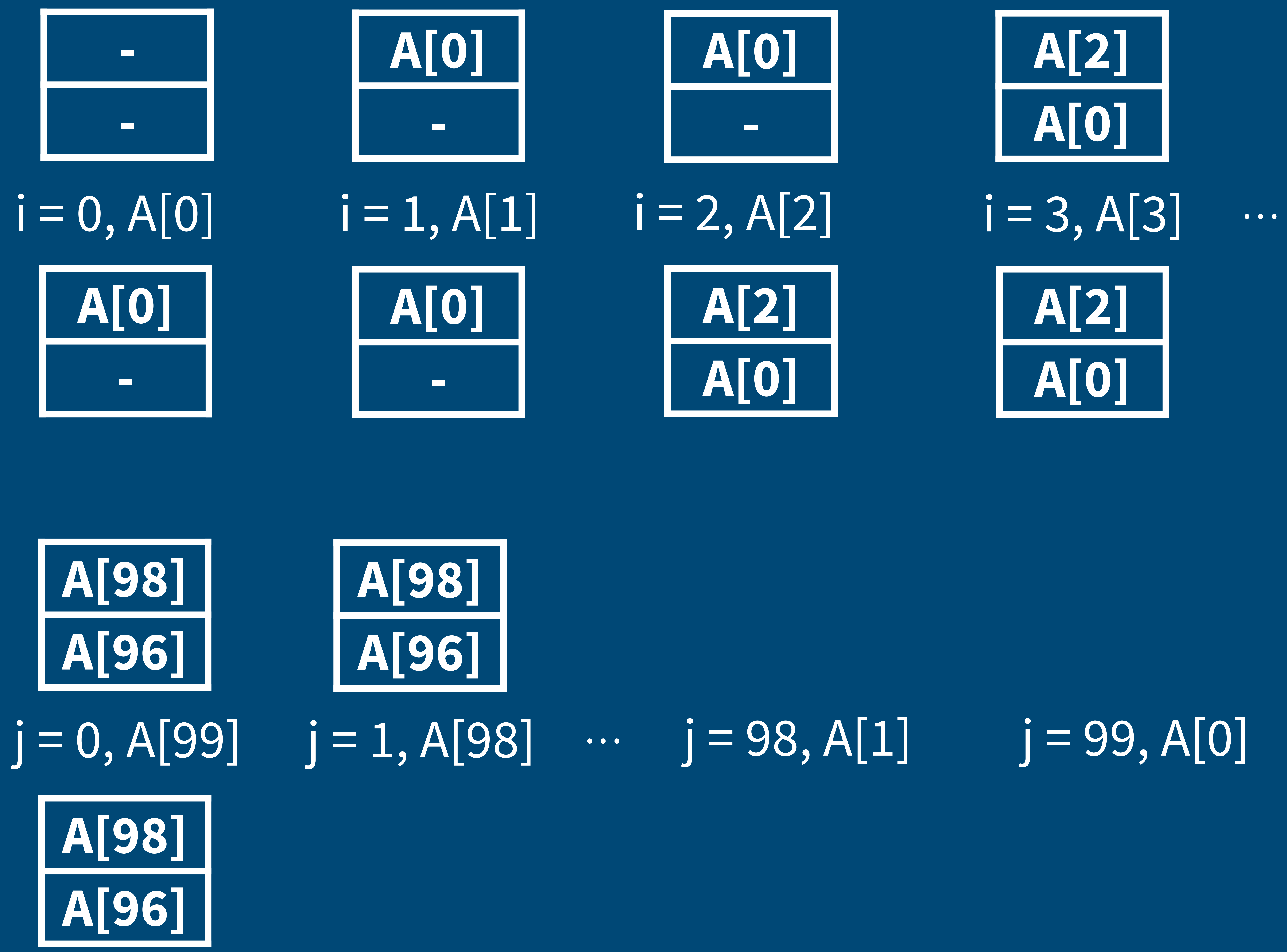


Symbolic CFG

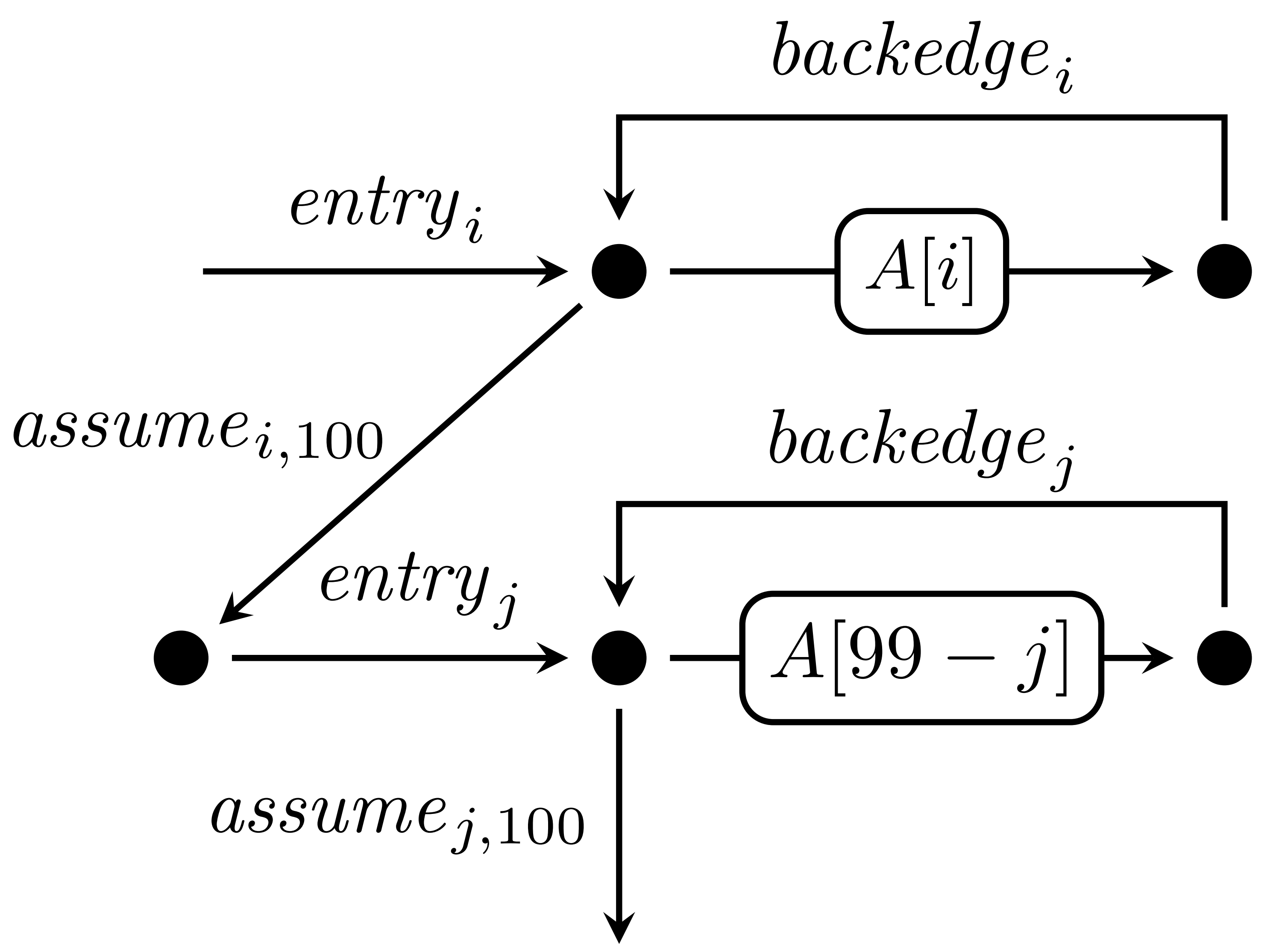


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively



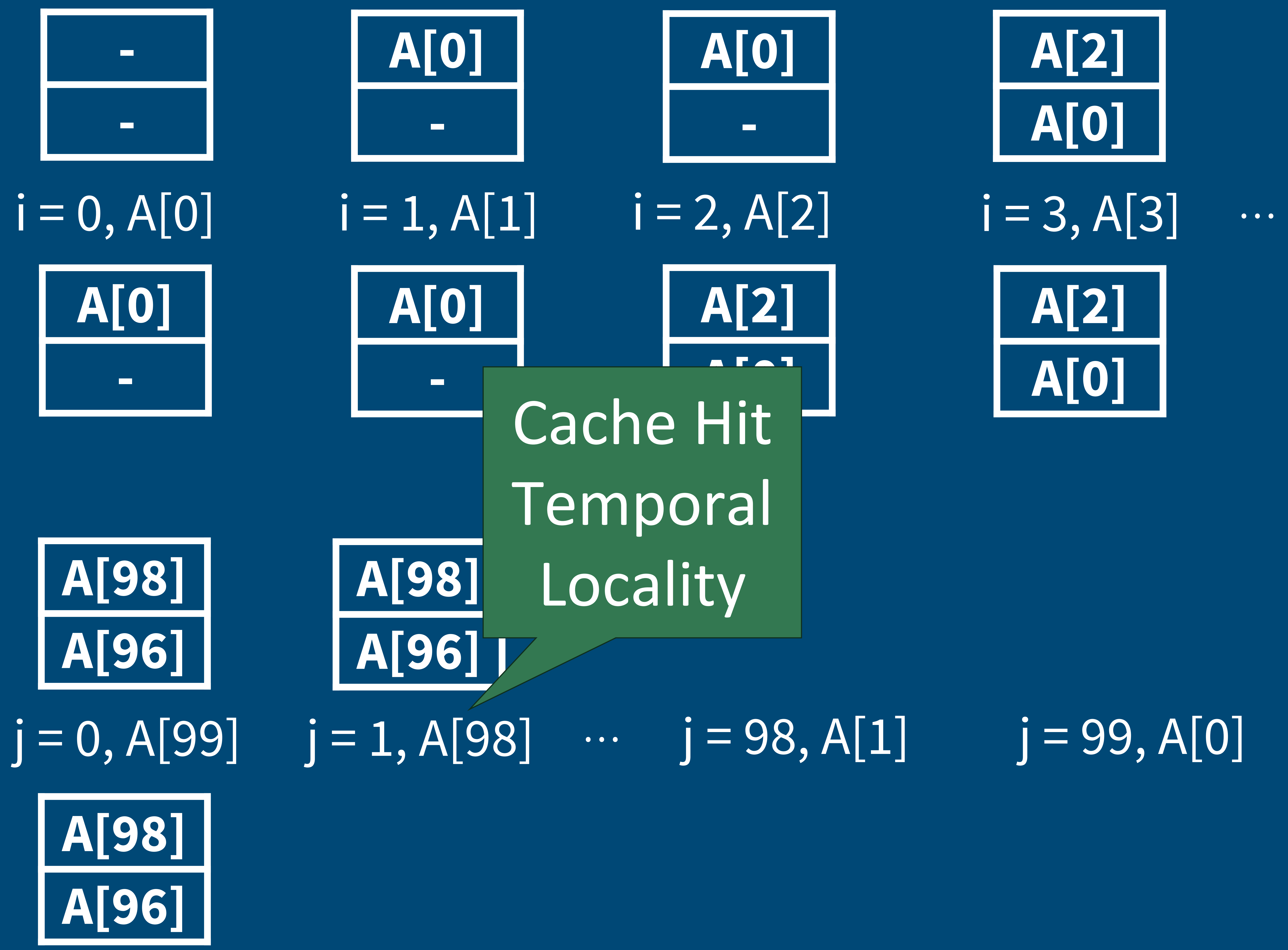
Symbolic CFG



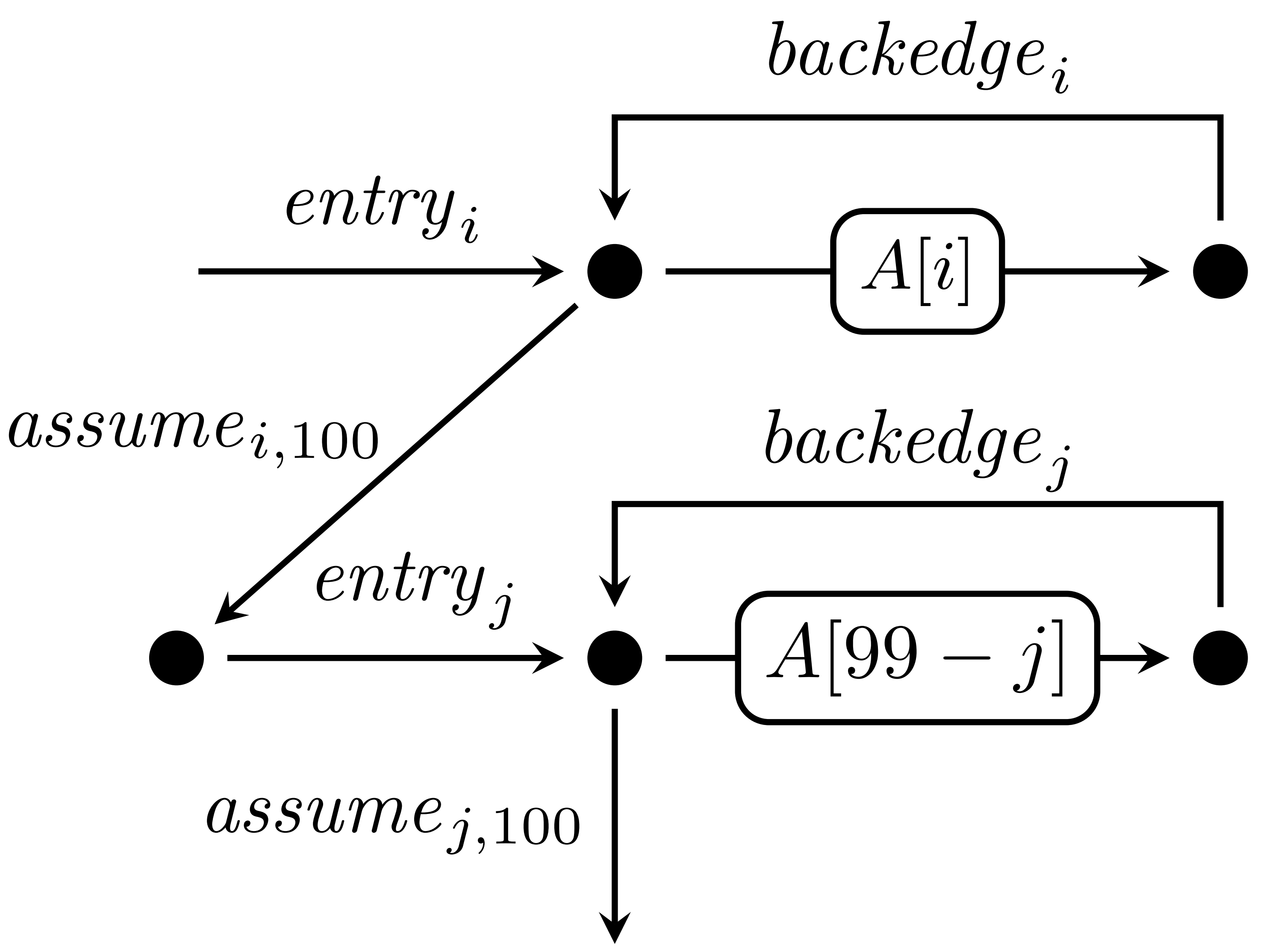
Assumptions:

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

Cache Analysis: Intuitively

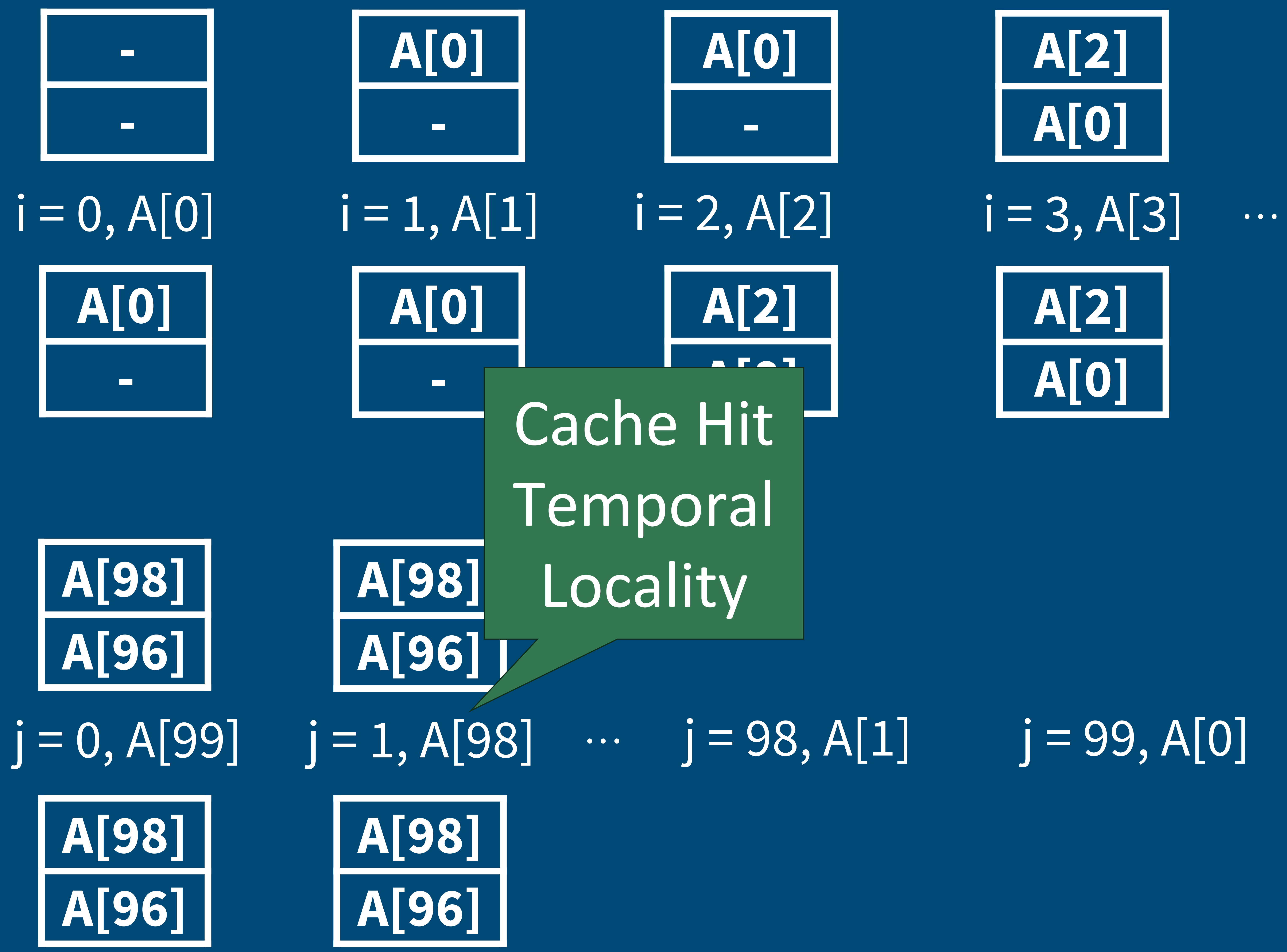


Symbolic CFG

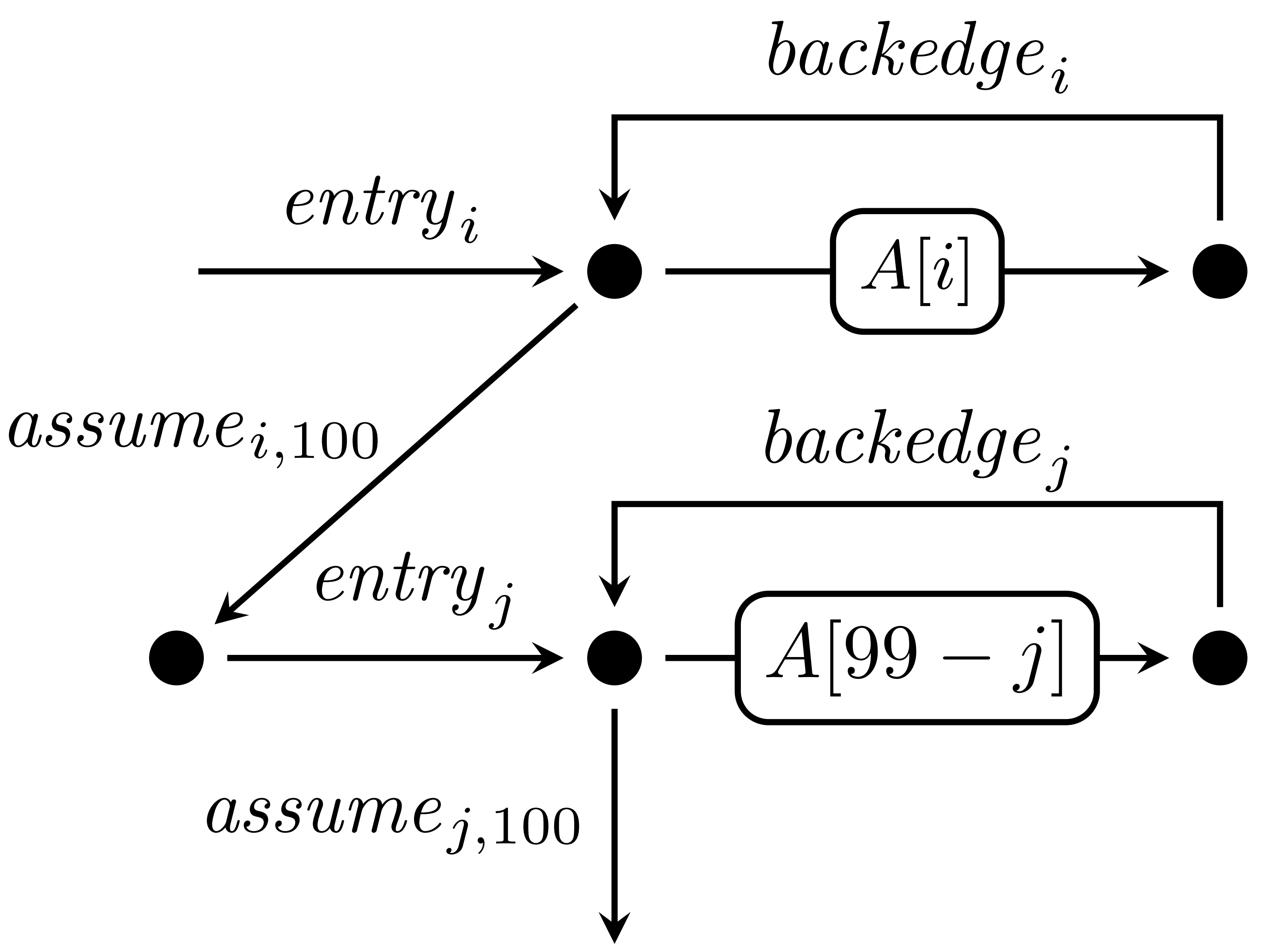


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively



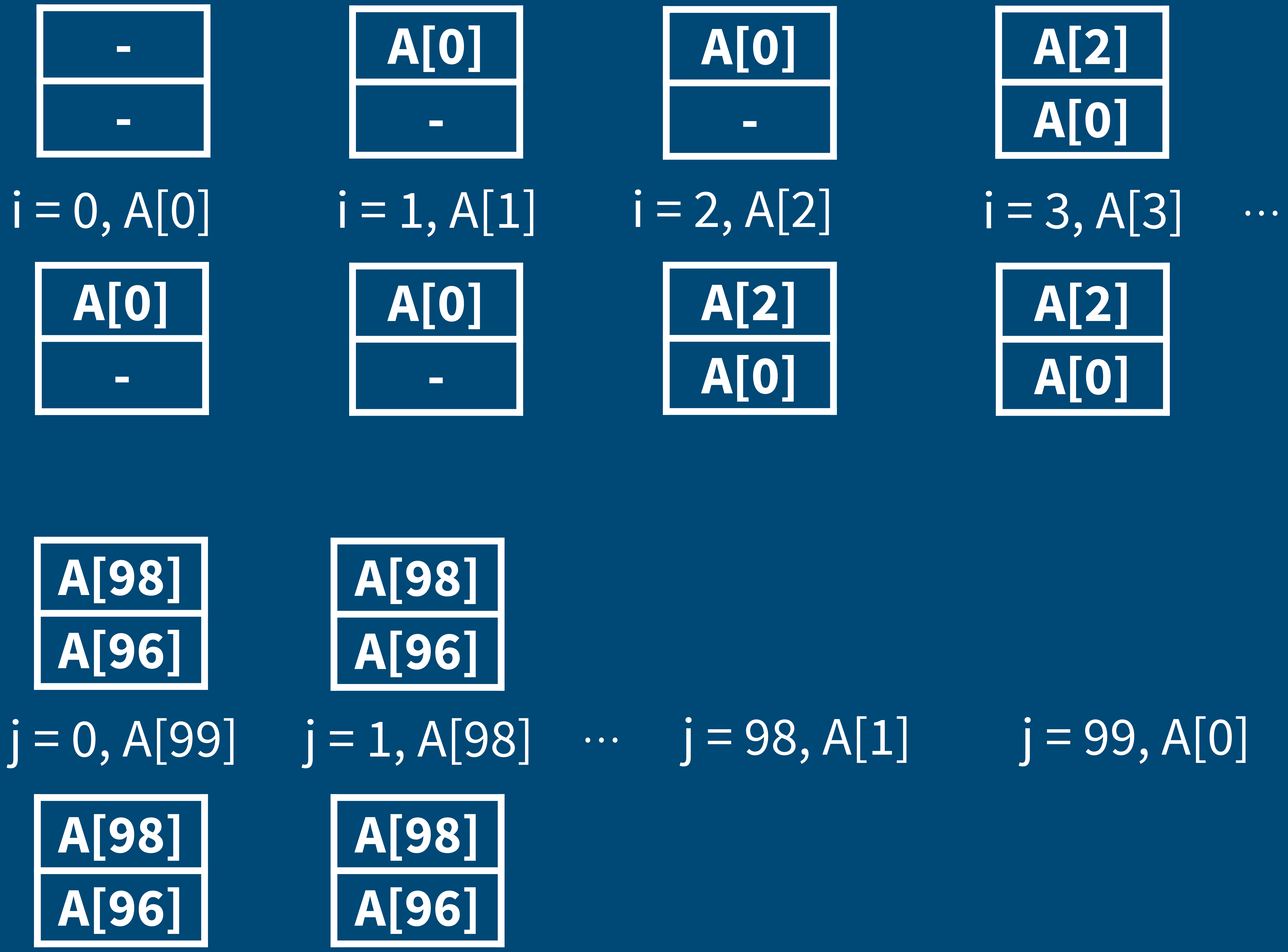
Symbolic CFG



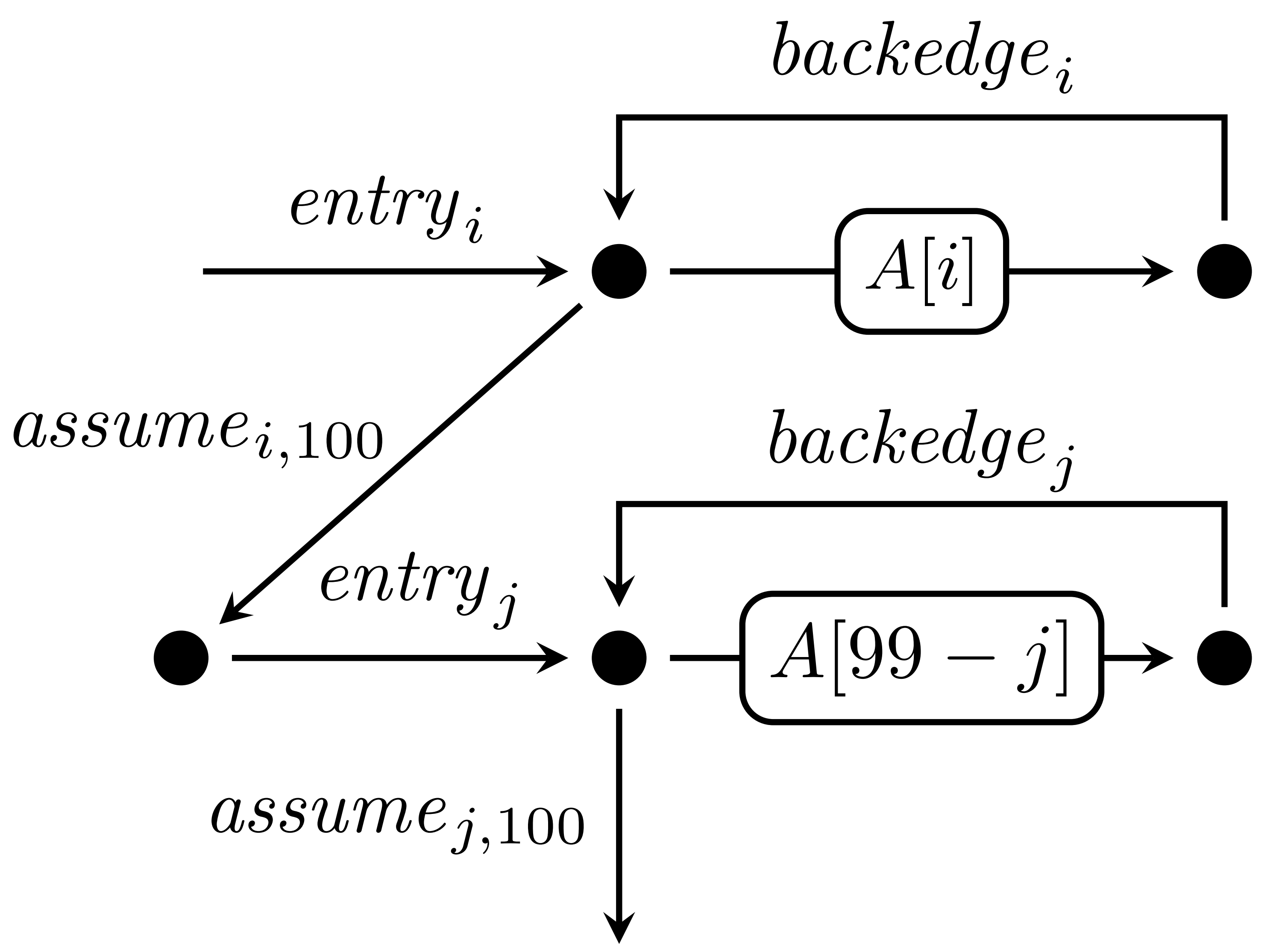
Assumptions:

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

Cache Analysis: Intuitively

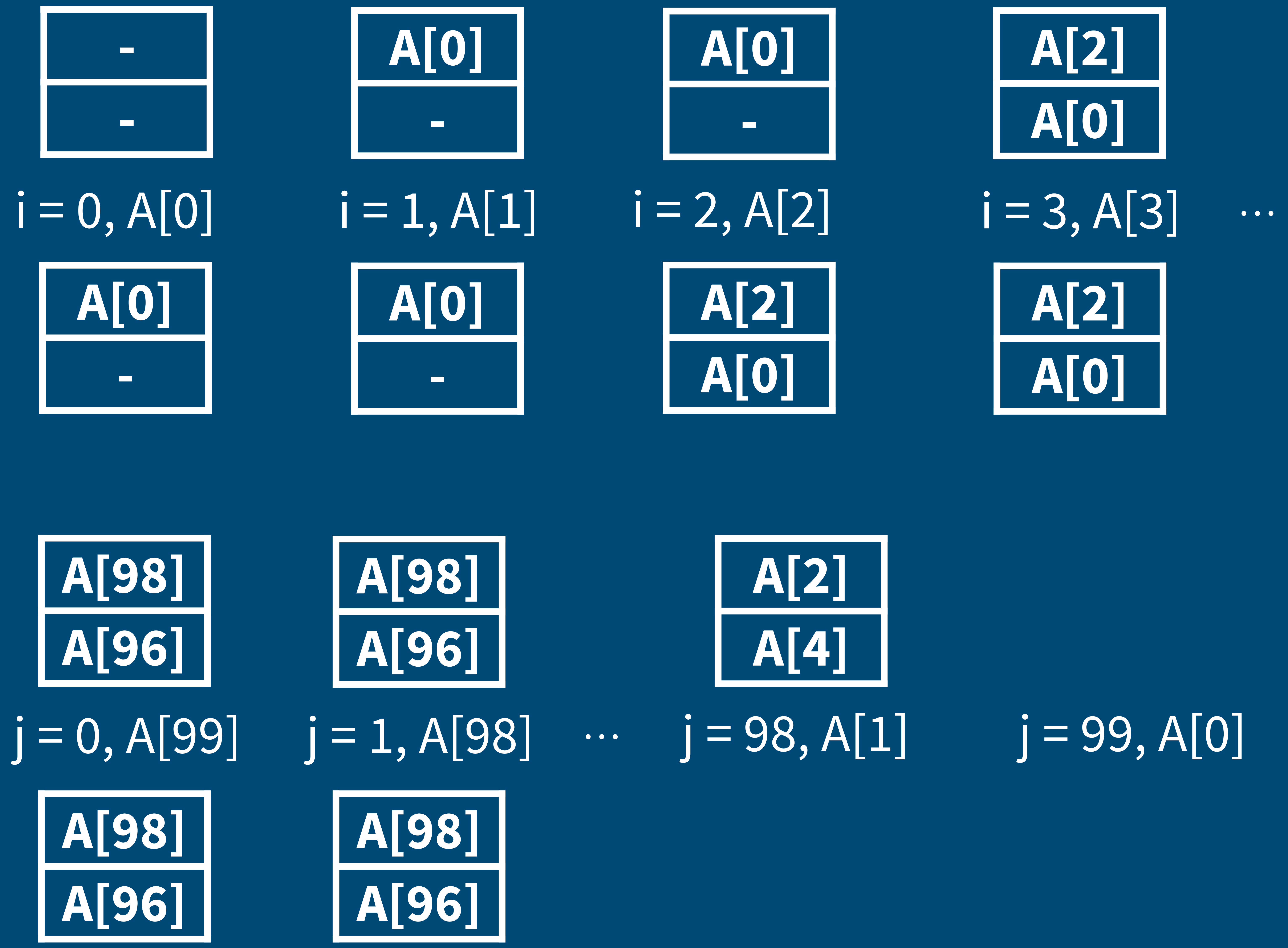


Symbolic CFG

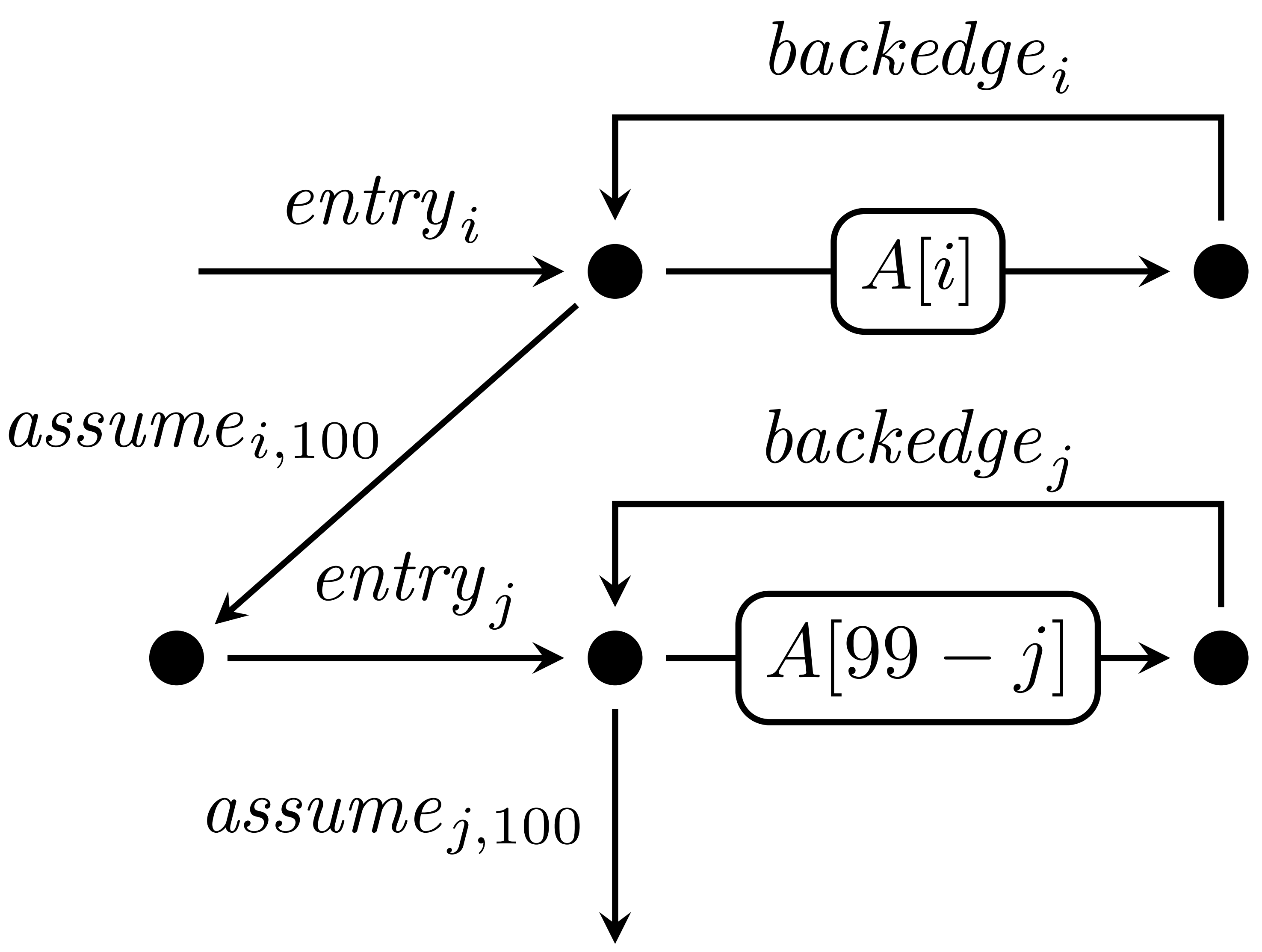


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

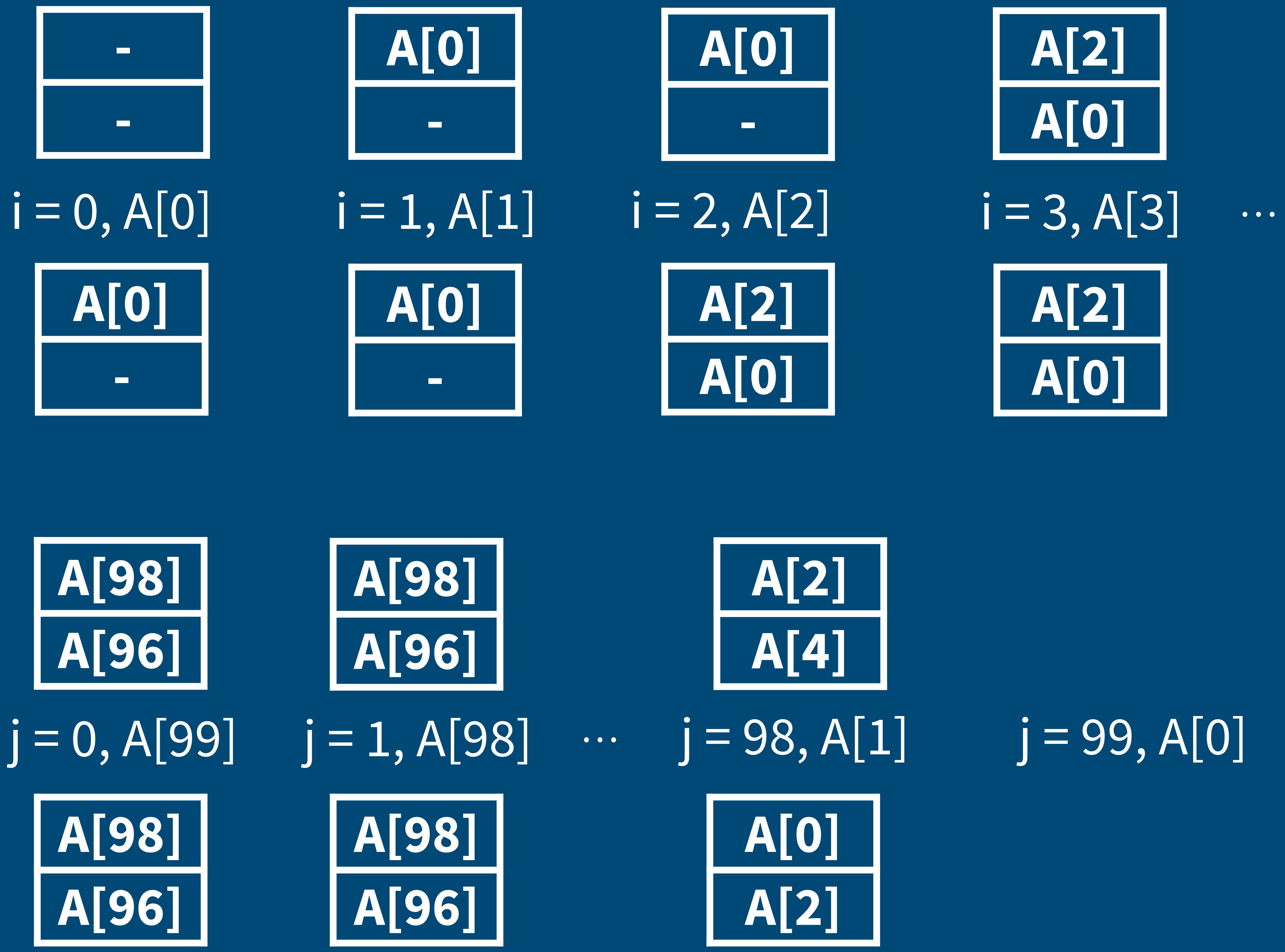


Symbolic CFG

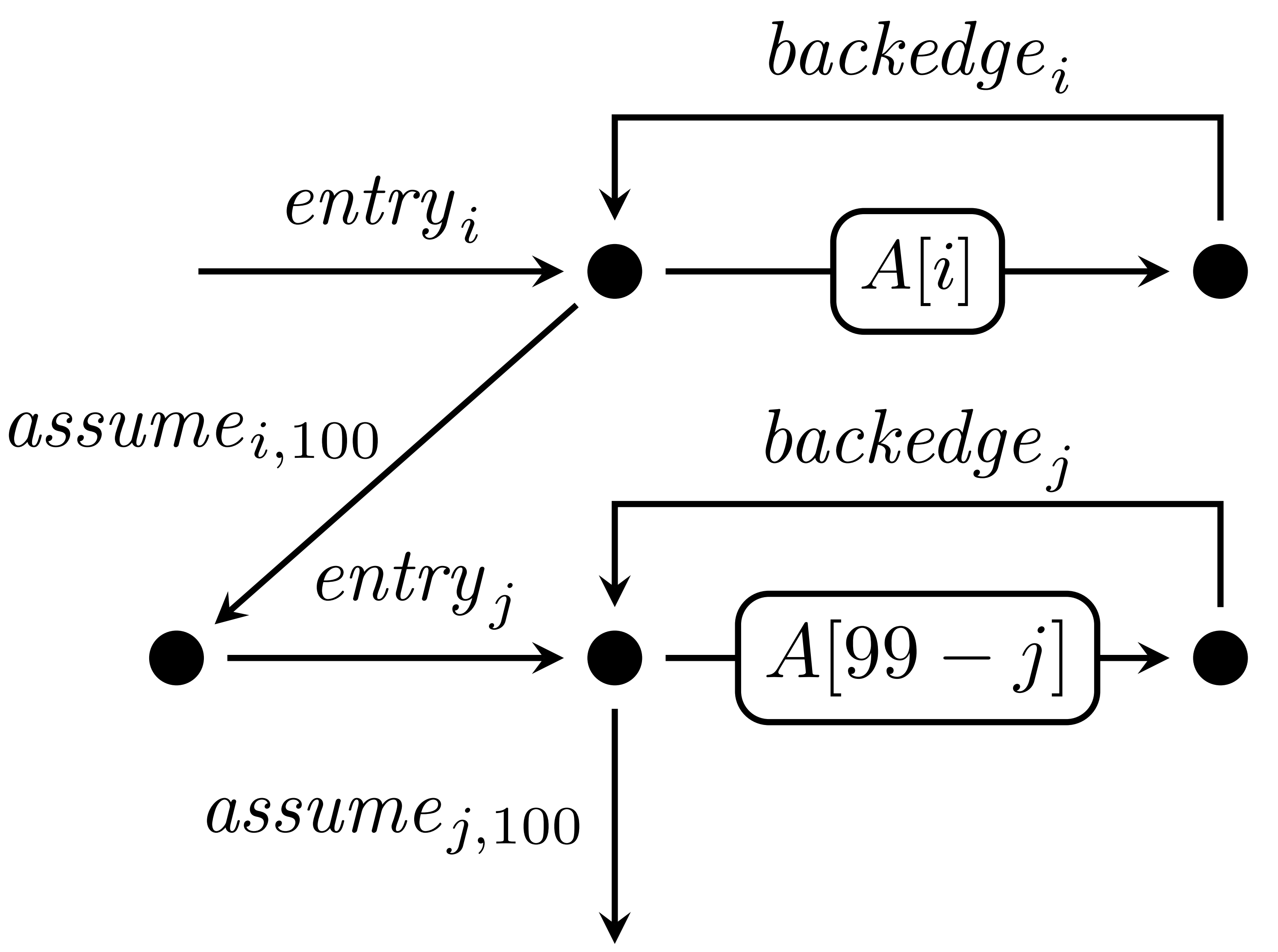


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

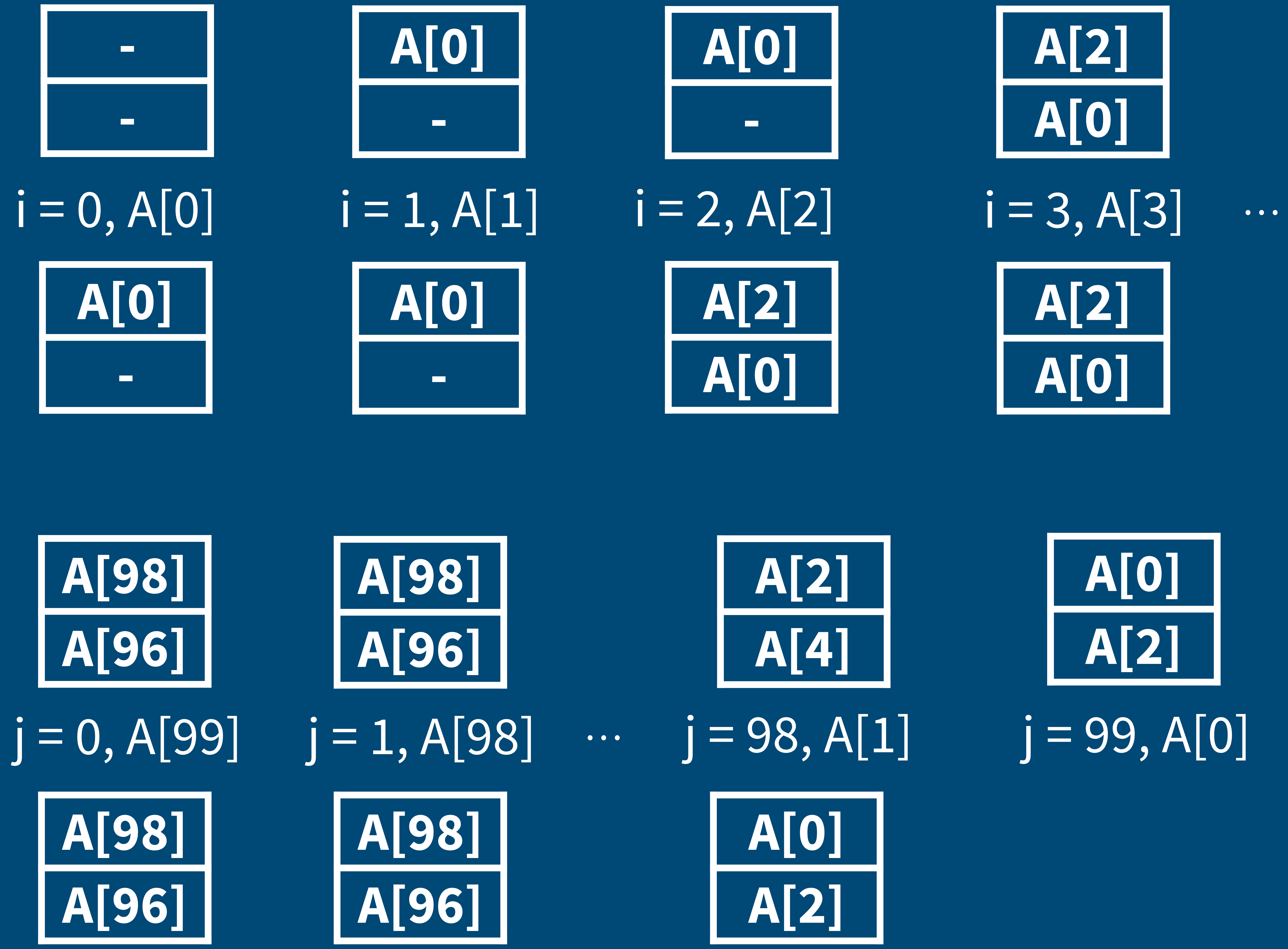


Symbolic CFG

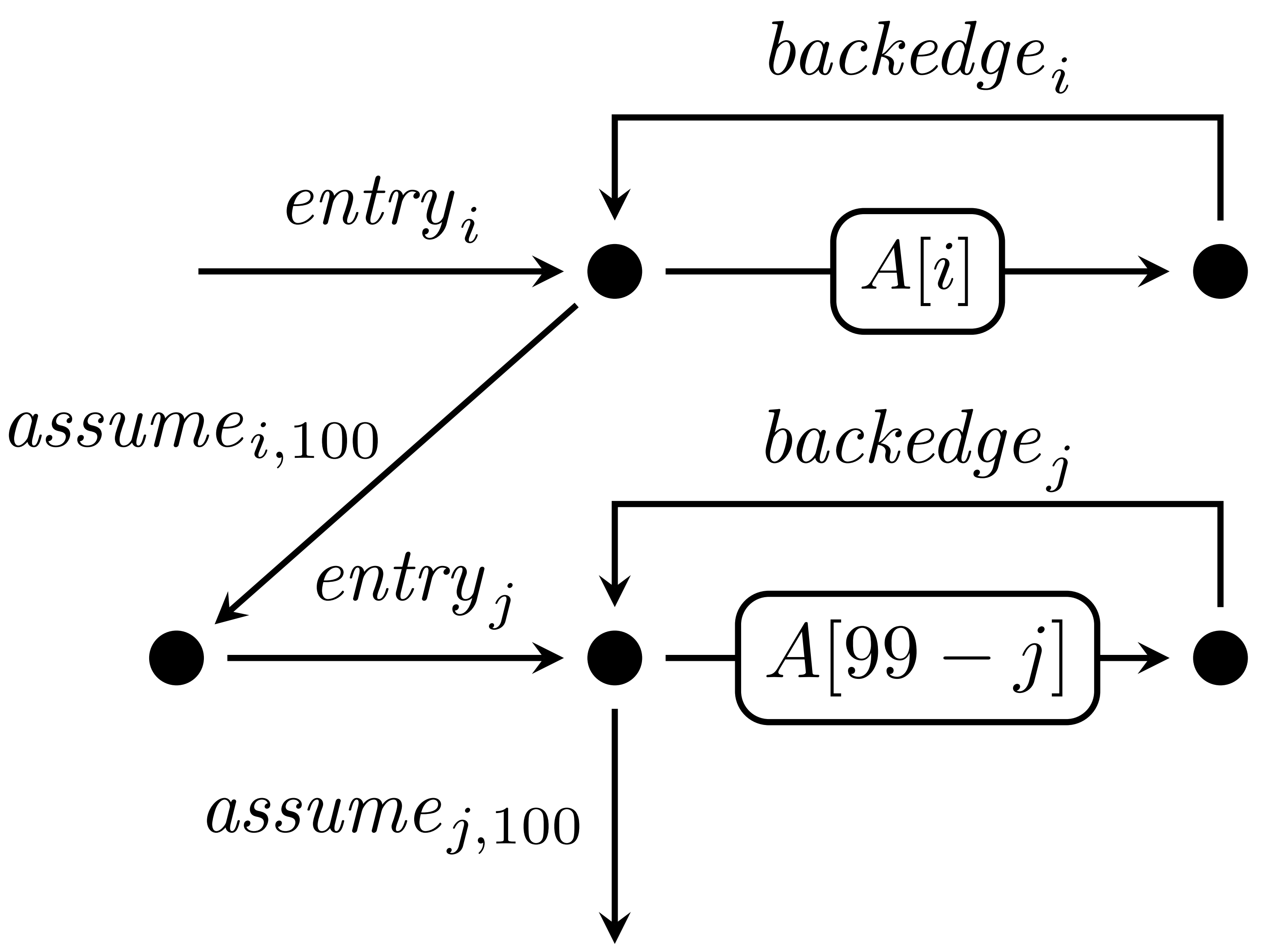


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

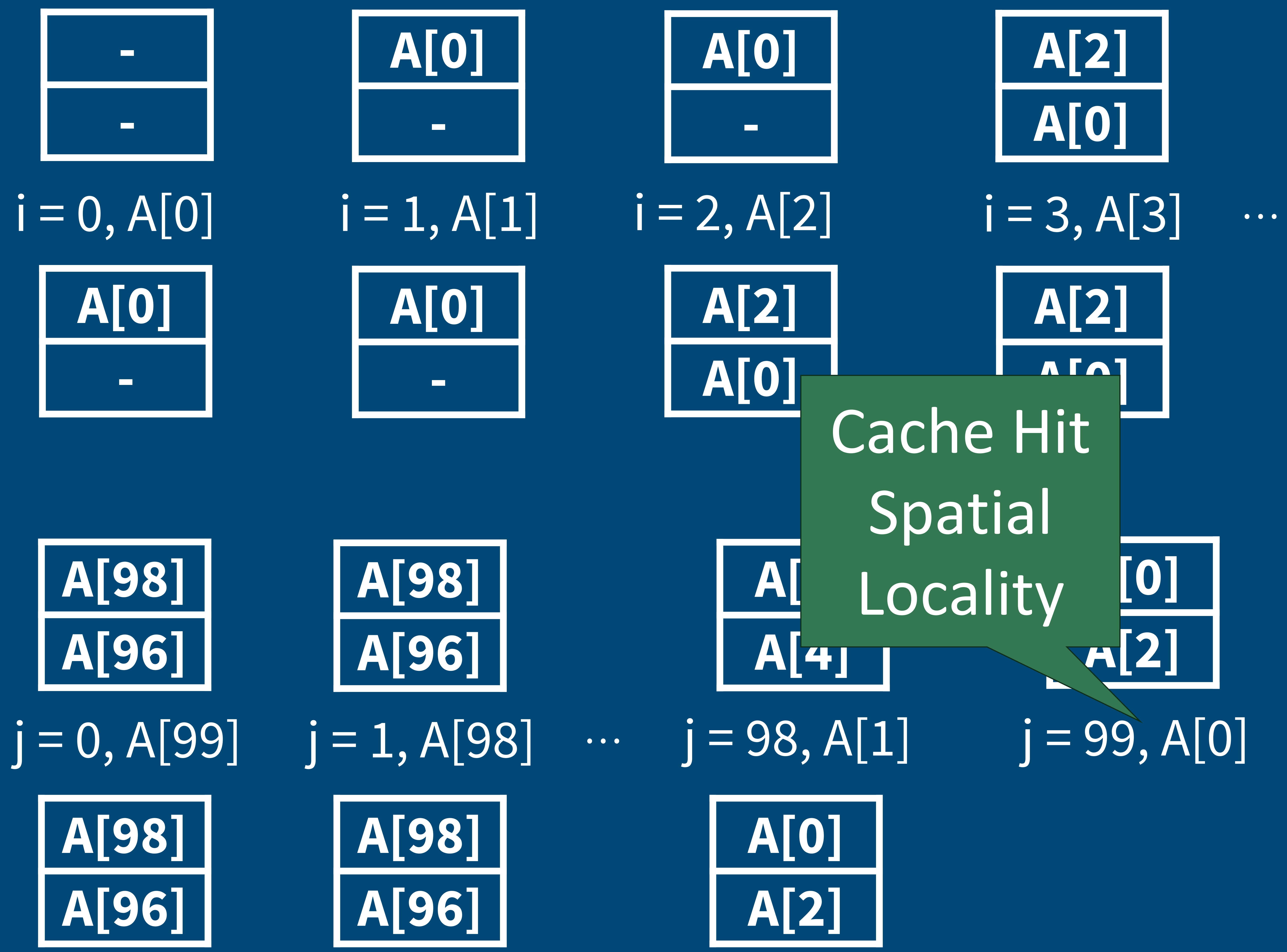


Symbolic CFG

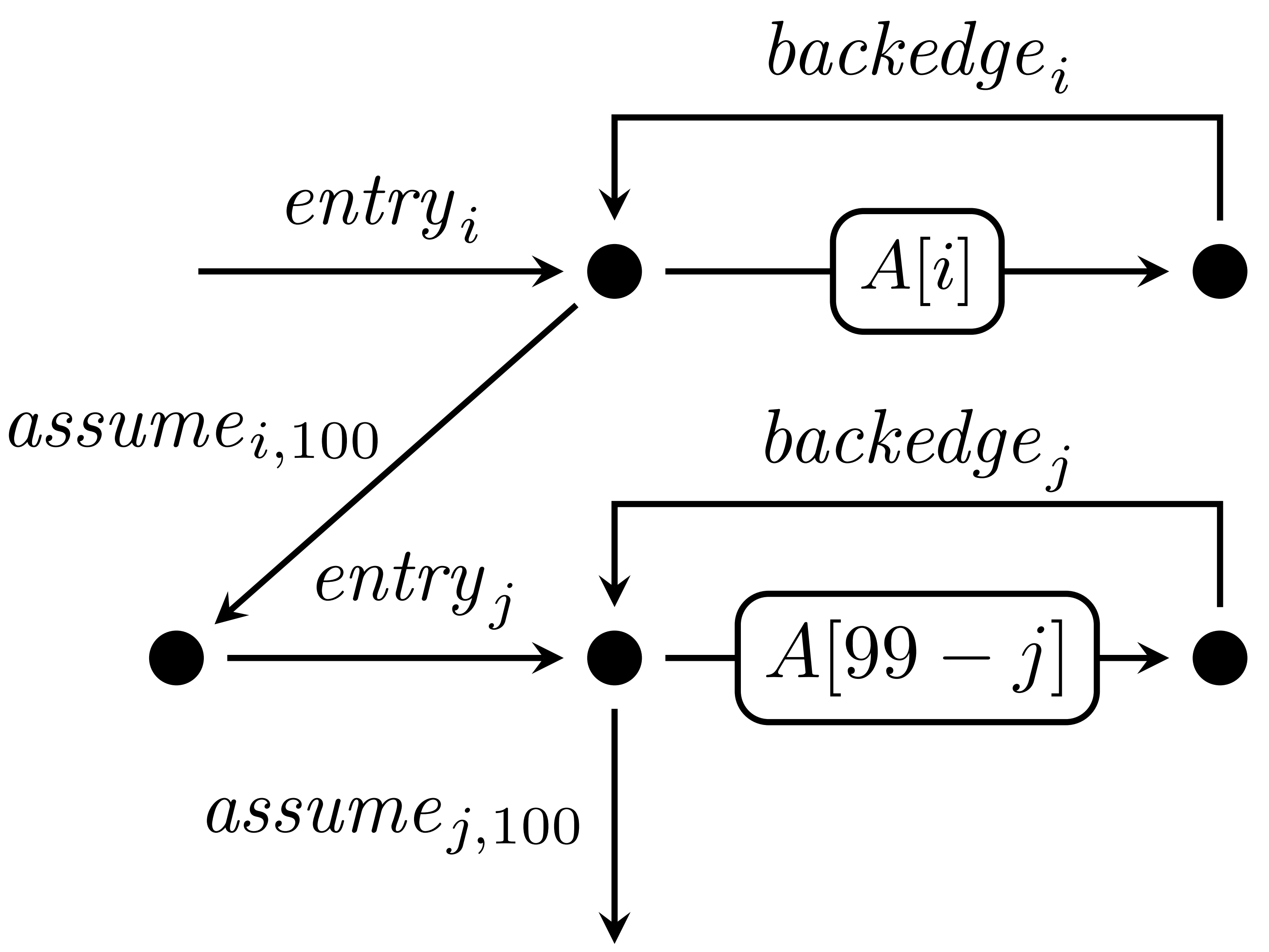


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

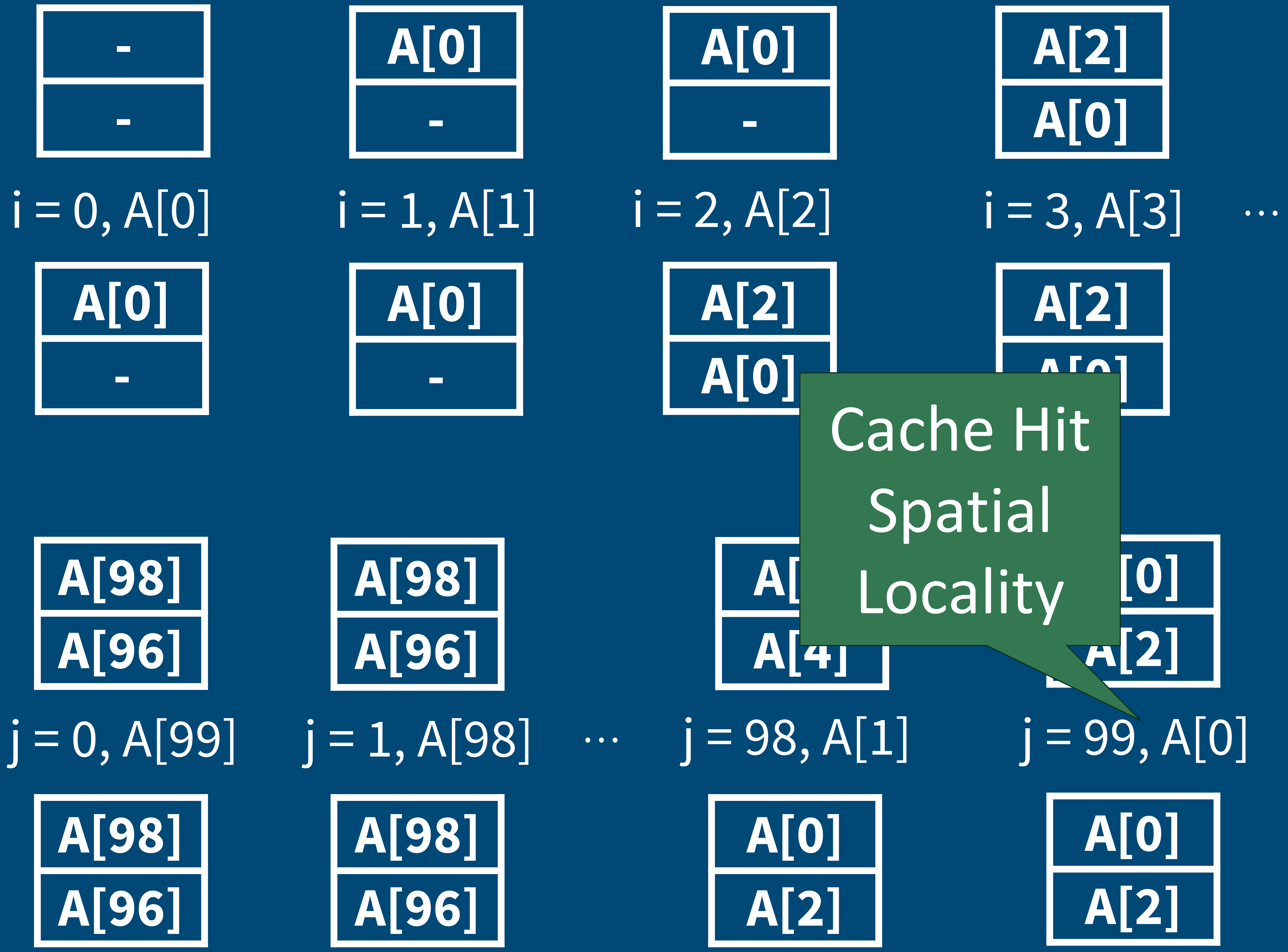


Symbolic CFG

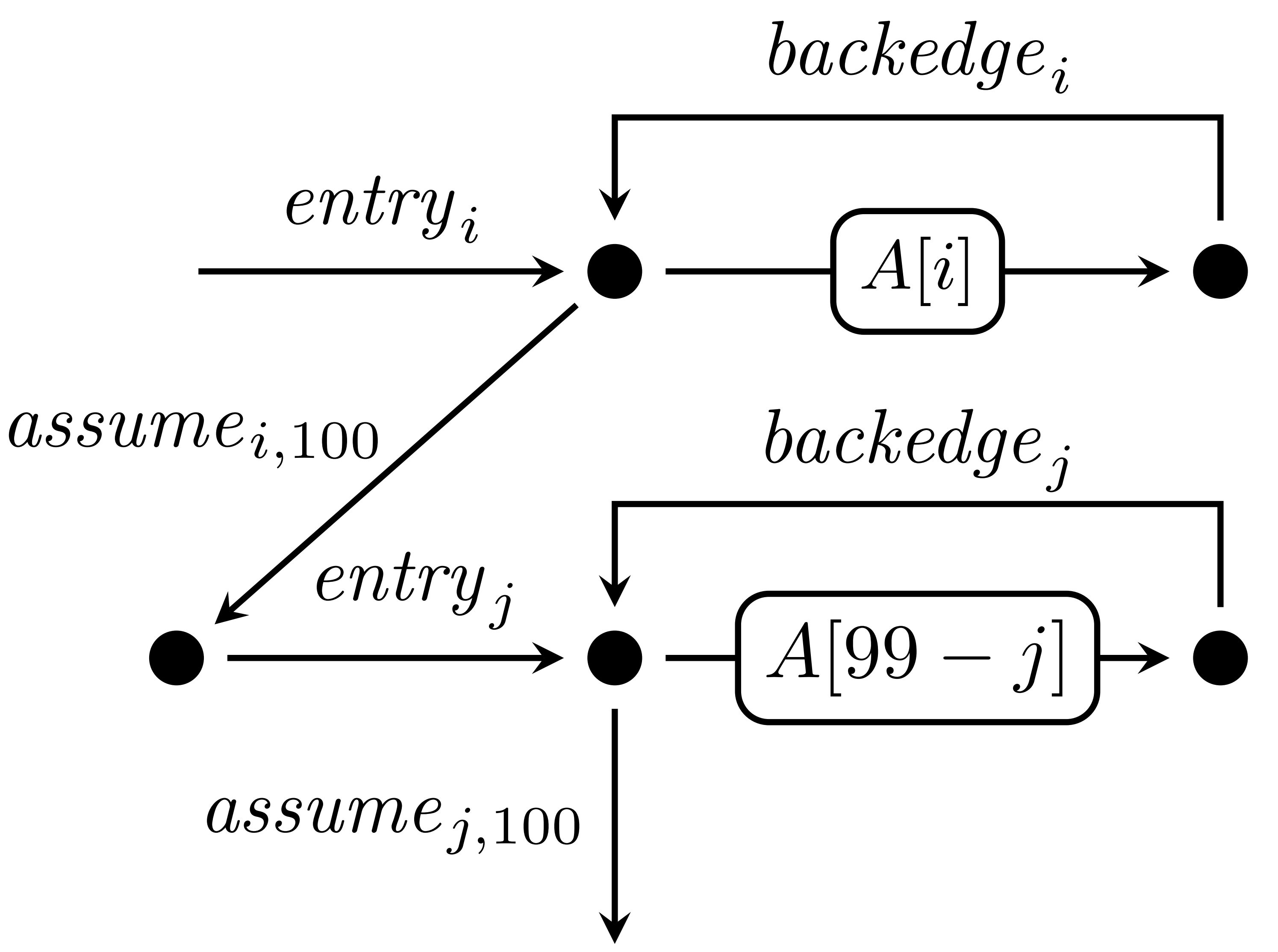


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

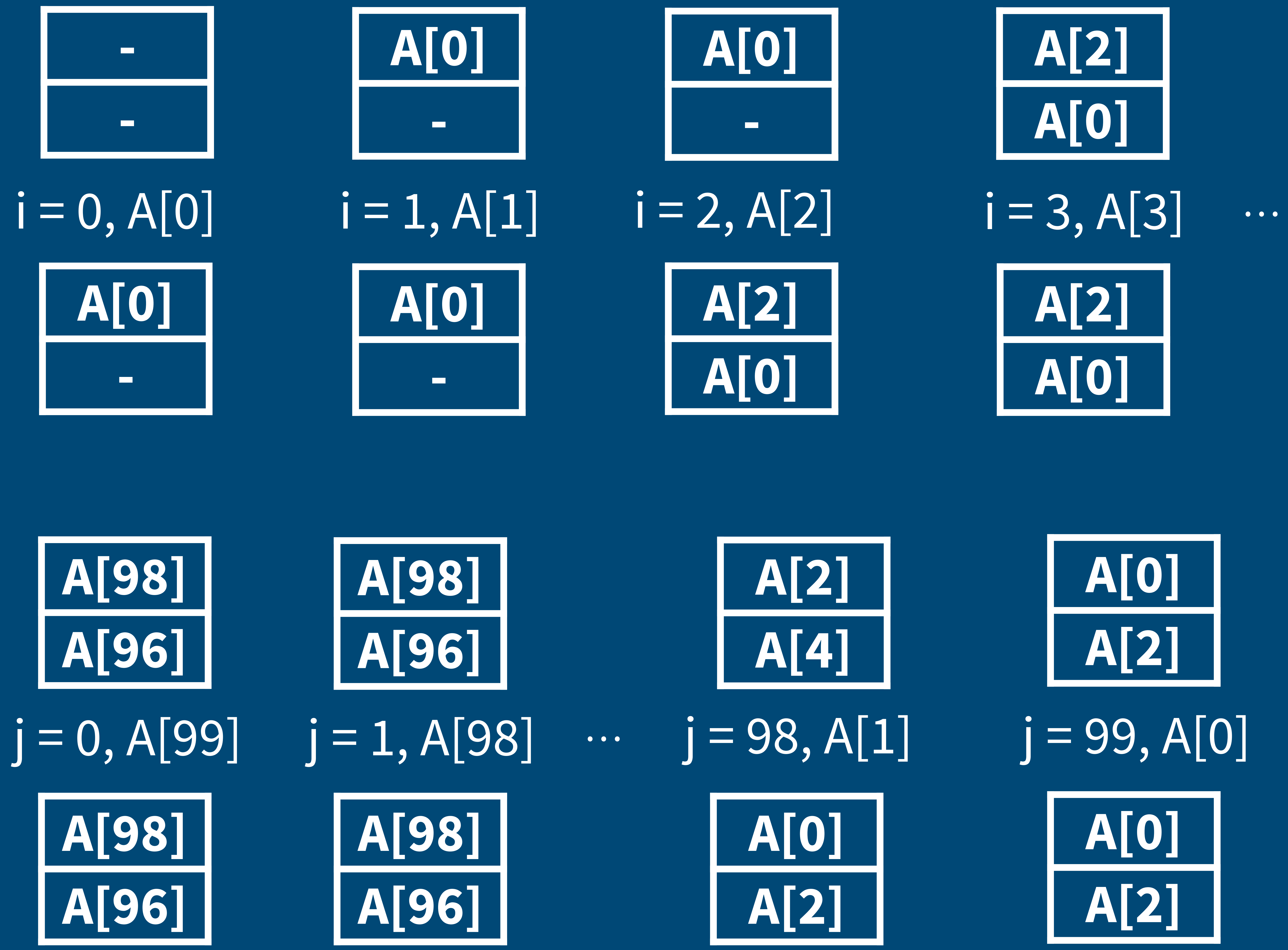


Symbolic CFG

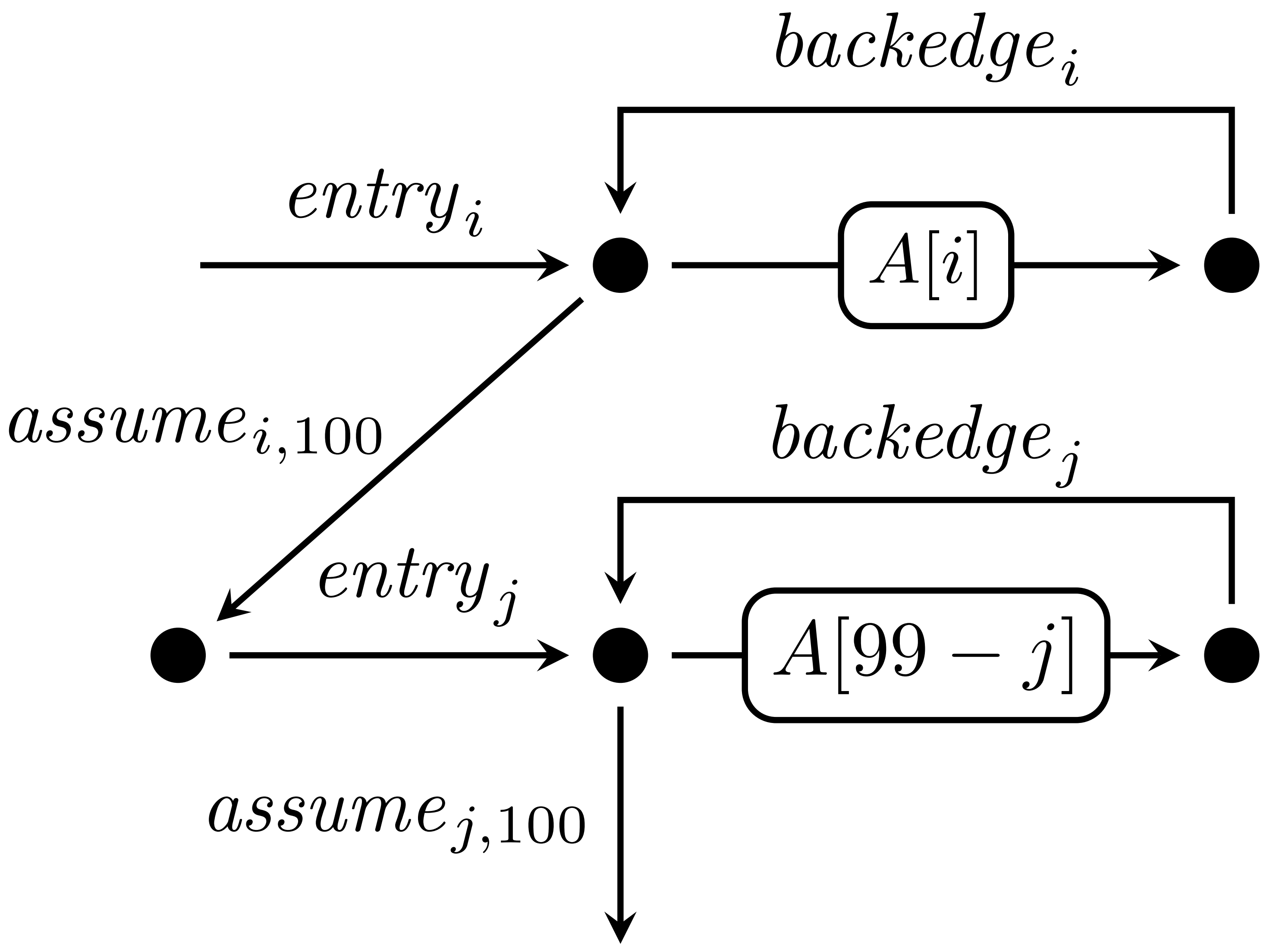


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively



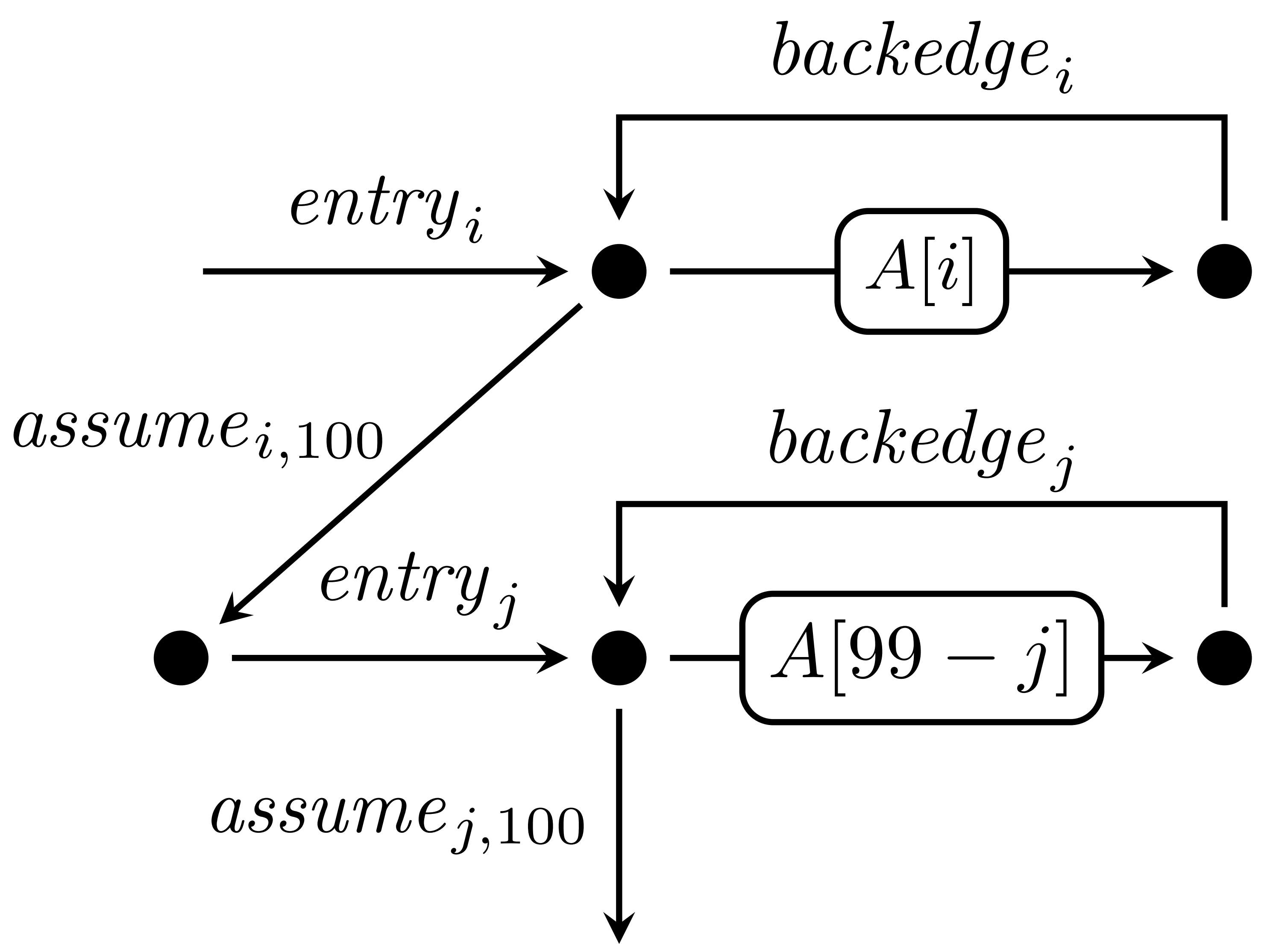
Symbolic CFG



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

Challenges in Data Cache Analysis II

Symbolic CFG

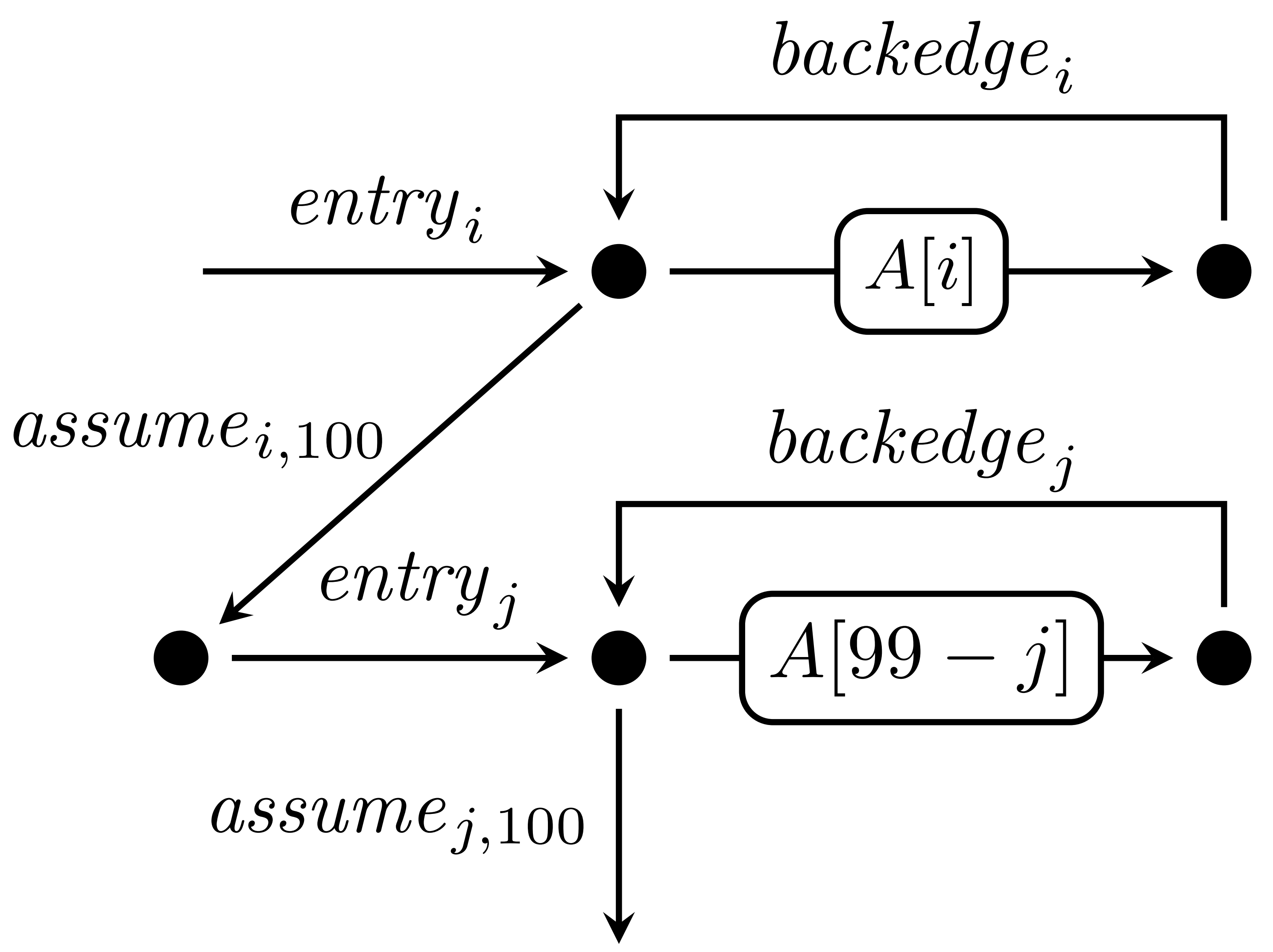


Assumptions:
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Challenges in Data Cache Analysis II

1. Cache states depend on loop iteration

Symbolic CFG



Assumptions:

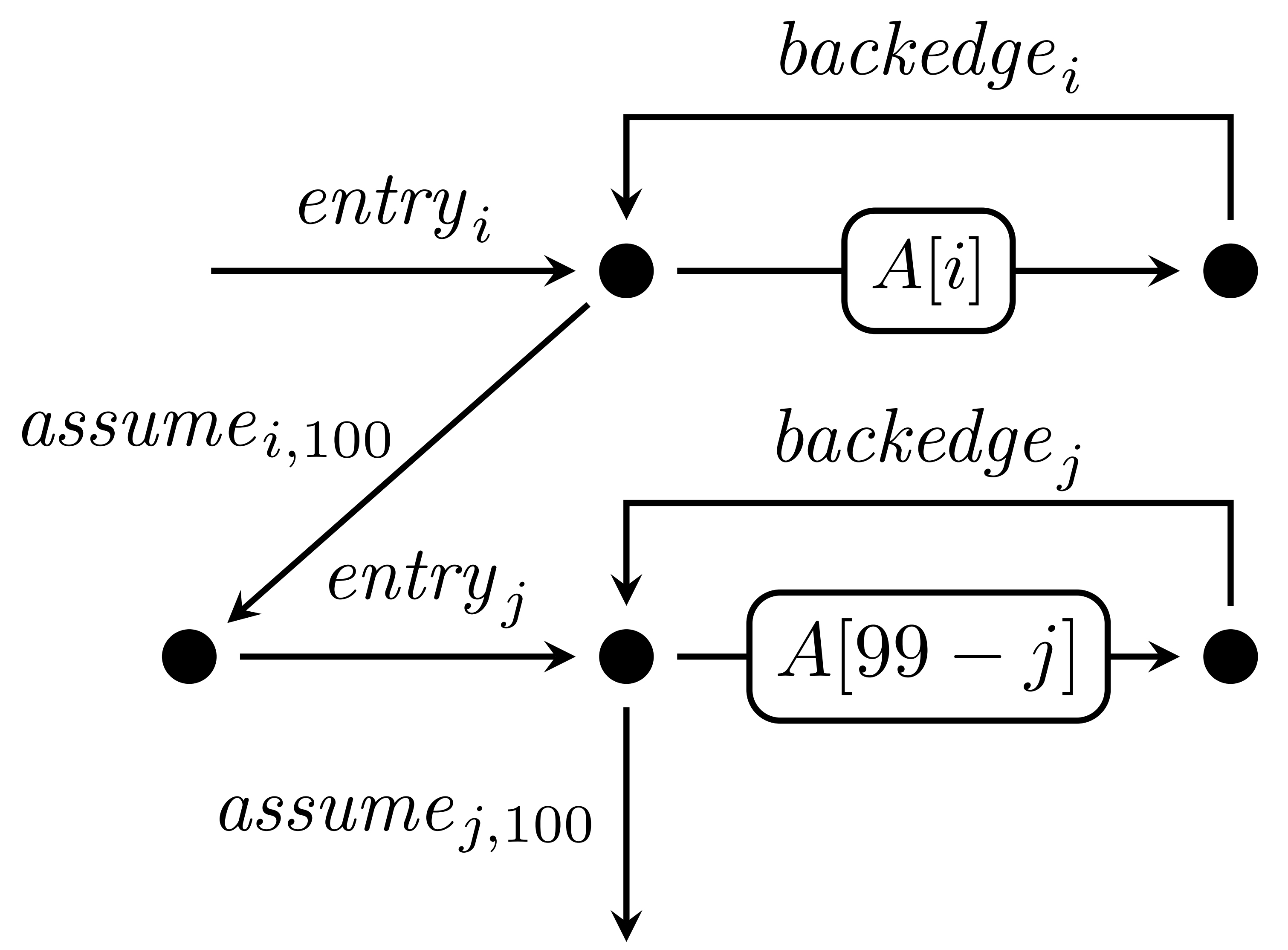
- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Challenges in Data Cache Analysis II

1. Cache states depend on loop iteration

Contribution: **Symbolic Cache States**

Symbolic CFG



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

Challenges in Data Cache Analysis II

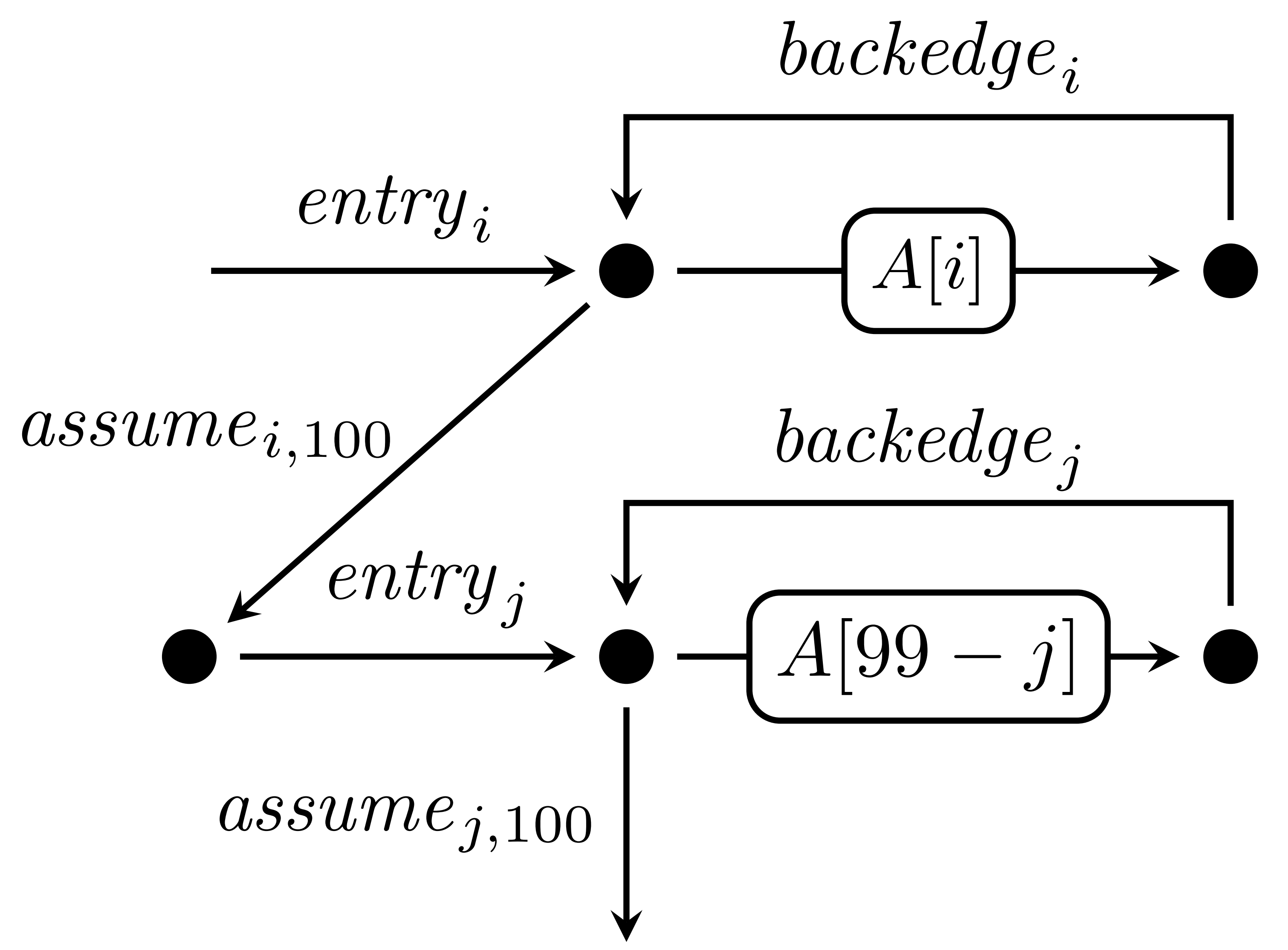
1. Cache states depend on loop iteration

Contribution: **Symbolic Cache States**

2. Behavior is phase dependent:

- Warm-up phase:
  hits/misses depending on initial state
- Steady-state phase:
  repetitive patterns

Symbolic CFG



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

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**

Symbolic Cache States

Saarland Informatics Campus

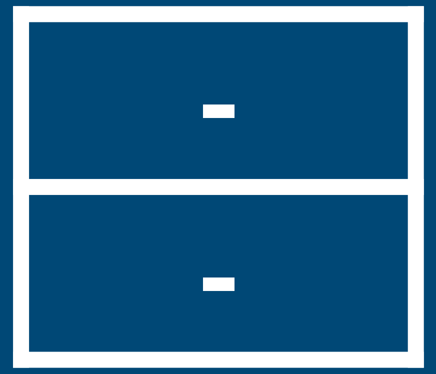
Symbolic Cache States

First loop:

Symbolic Cache States

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

Symbolic Cache States

-
-

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

A[i]
-

Symbolic Cache States

-	A[i-1]
-	-

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

A[i]
-

Symbolic Cache States

-
-

A[i-1]
-

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

A[i]
-

Cache Hit
Spatial
Locality

Symbolic Cache States

-	A[i-1]
-	-

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

A[i]	A[i-1]
-	-

Cache Hit
Spatial
Locality

Symbolic Cache States

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

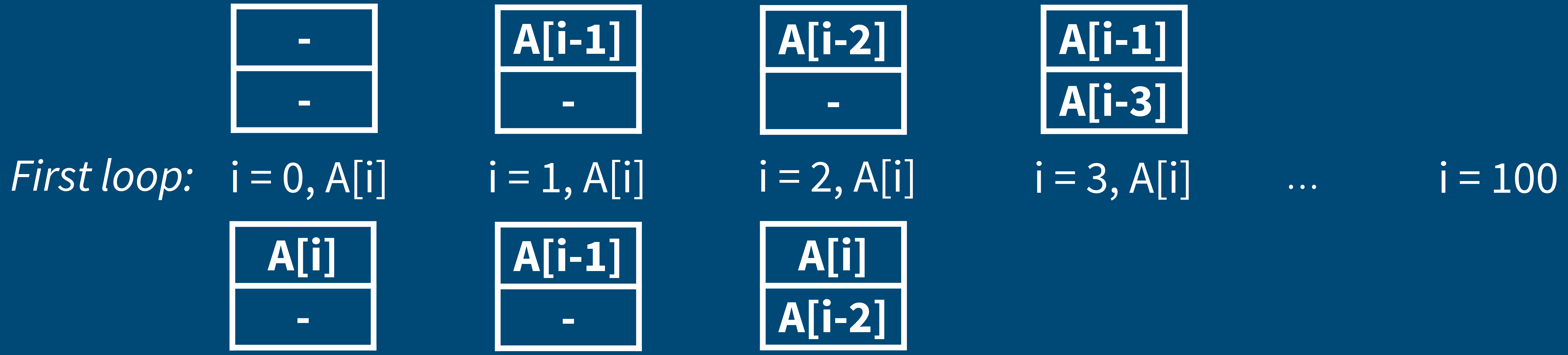
Symbolic Cache States

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

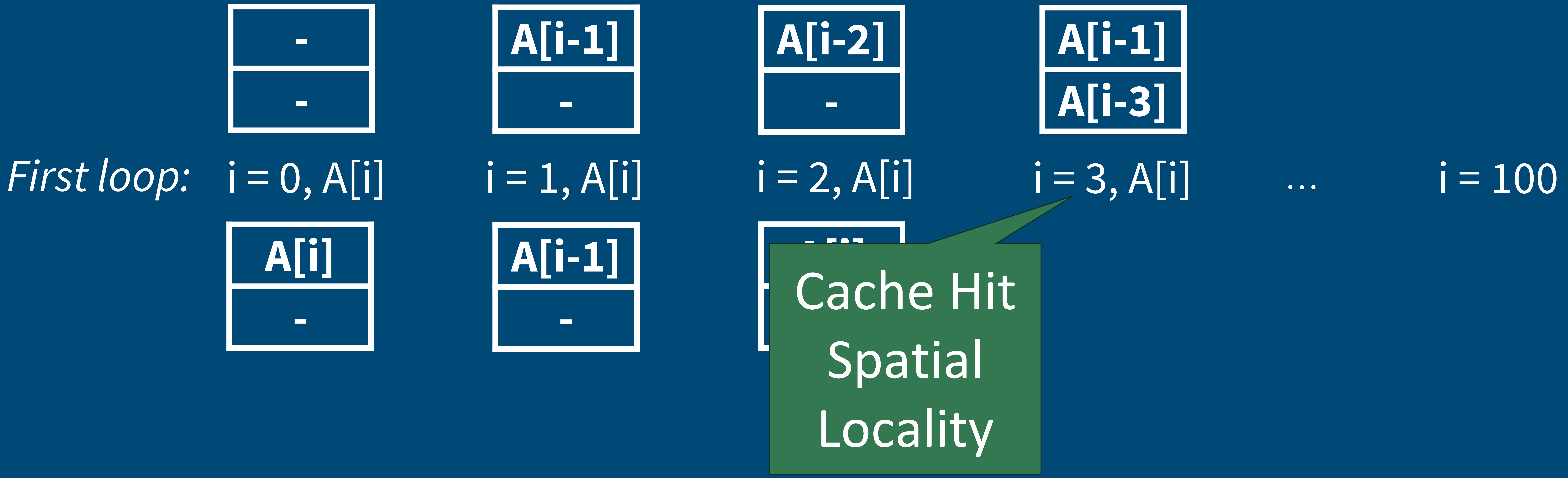
Symbolic Cache States

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

Symbolic Cache States



Symbolic Cache States



Symbolic Cache States

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

Cache Hit
Spatial
Locality

Symbolic Cache States

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

Symbolic Cache States

	-	A[i-1]	A[i-2]	A[i-1]		A[i-2]
	-	-	-	A[i-3]		A[i-4]
First loop:	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	A[i]	A[i-1]	A[i]	A[i-1]		
	-	-	A[i-2]	A[i-3]		

Symbolic Cache States

	- / -	A[i-1] / -	A[i-2] / -	A[i-1] / A[i-3]		A[i-2] / A[i-4]
First loop:	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	A[i] / -	A[i-1] / -	A[i] / A[i-2]	A[i-1] / A[i-3]		

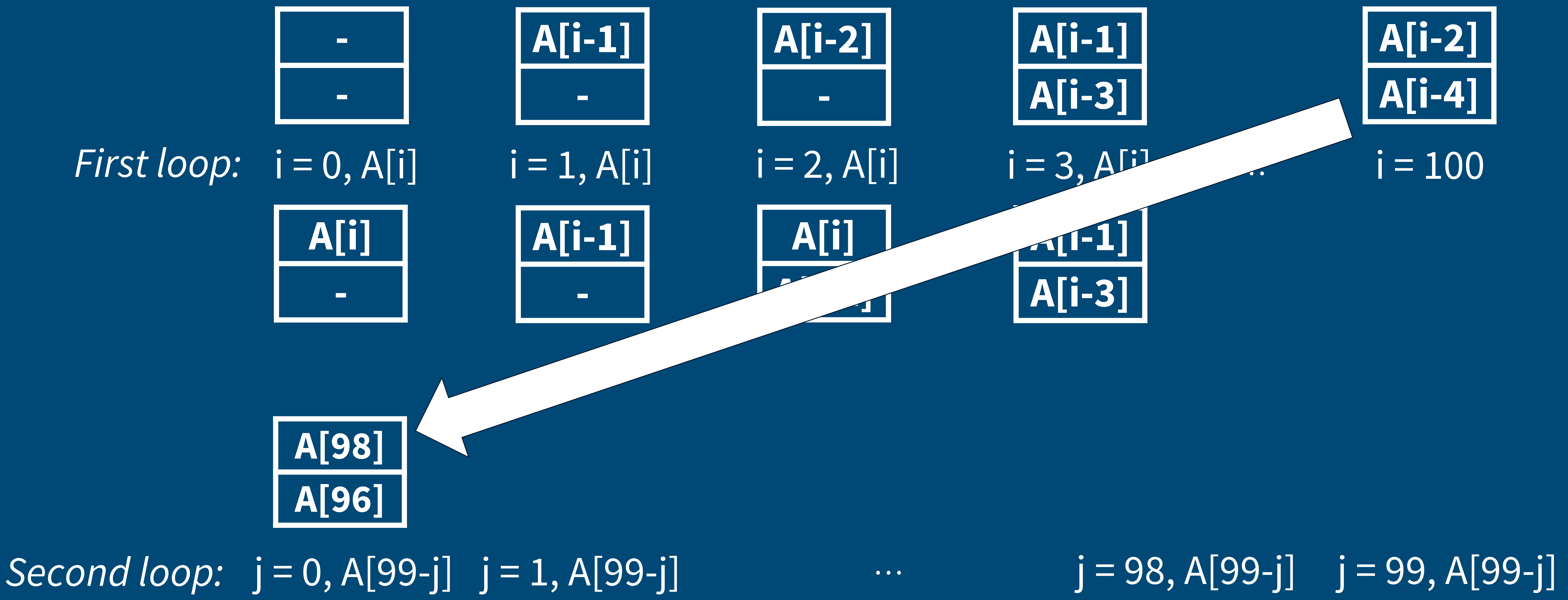
Second loop:

Symbolic Cache States

	-	A[i-1]	A[i-2]	A[i-1]		A[i-2]
	-	-	-	A[i-3]		A[i-4]
First loop:	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	A[i]	A[i-1]	A[i]	A[i-1]		
	-	-	A[i-2]	A[i-3]		

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

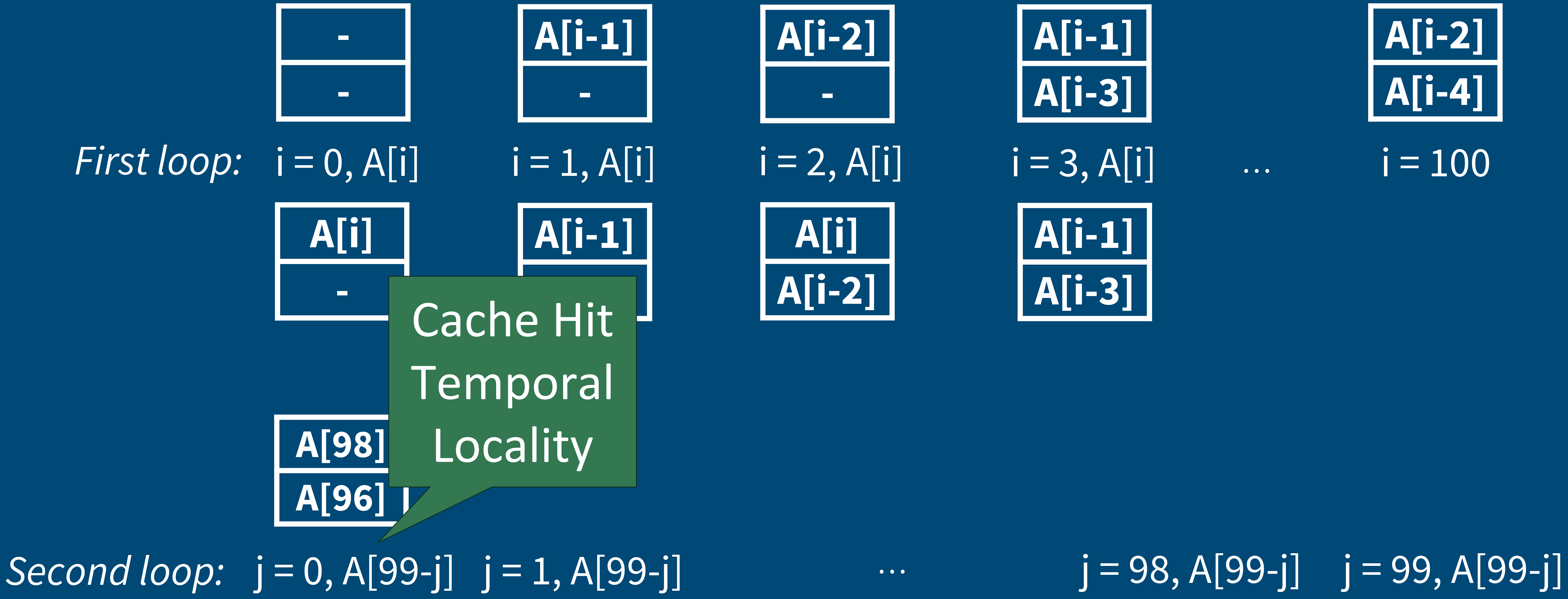


Symbolic Cache States

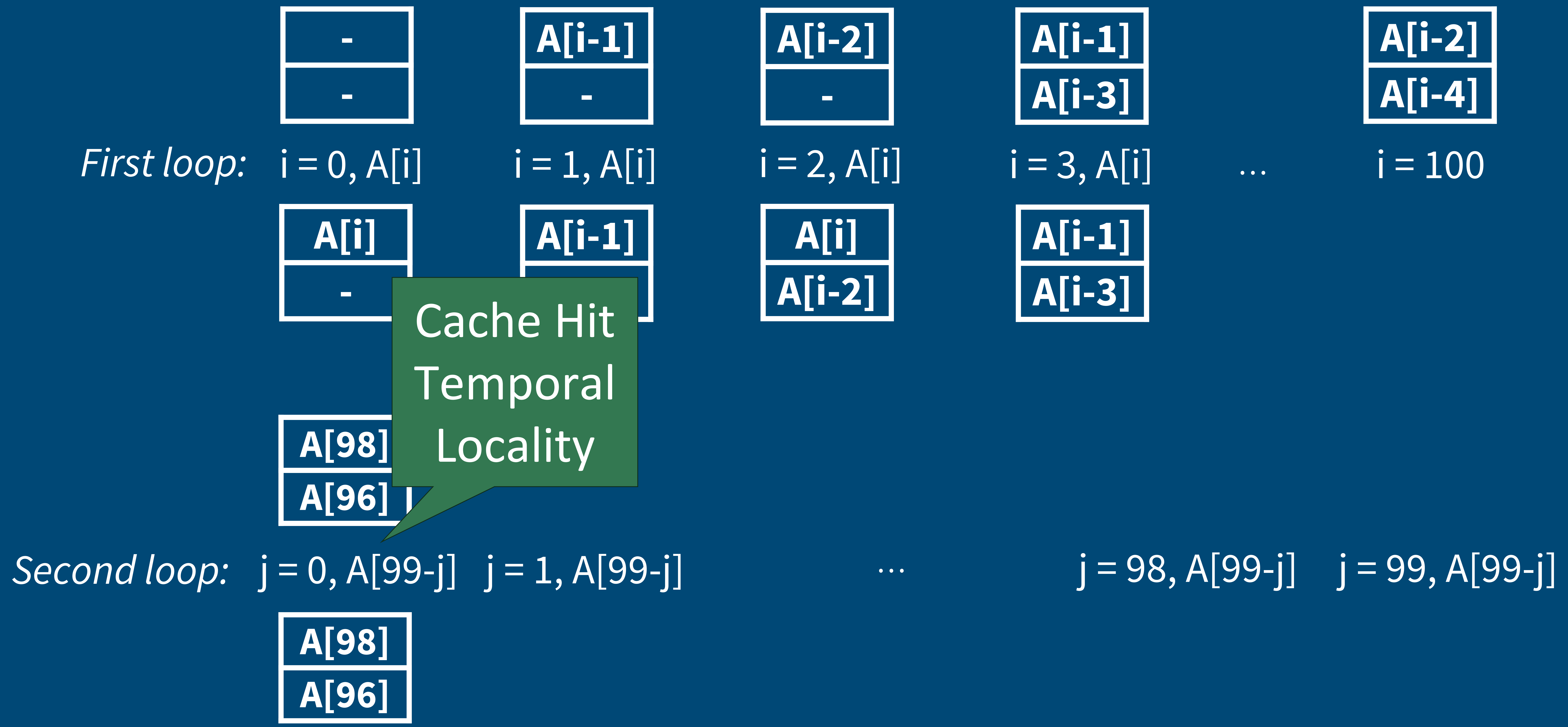
	- / -	A[i-1] / -	A[i-2] / -	A[i-1] / A[i-3]		A[i-2] / A[i-4]
First loop:	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	A[i] / -	A[i-1] / -	A[i] / A[i-2]	A[i-1] / A[i-3]		

	A[98] / A[96]				
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



Symbolic Cache States



Symbolic Cache States

First loop:

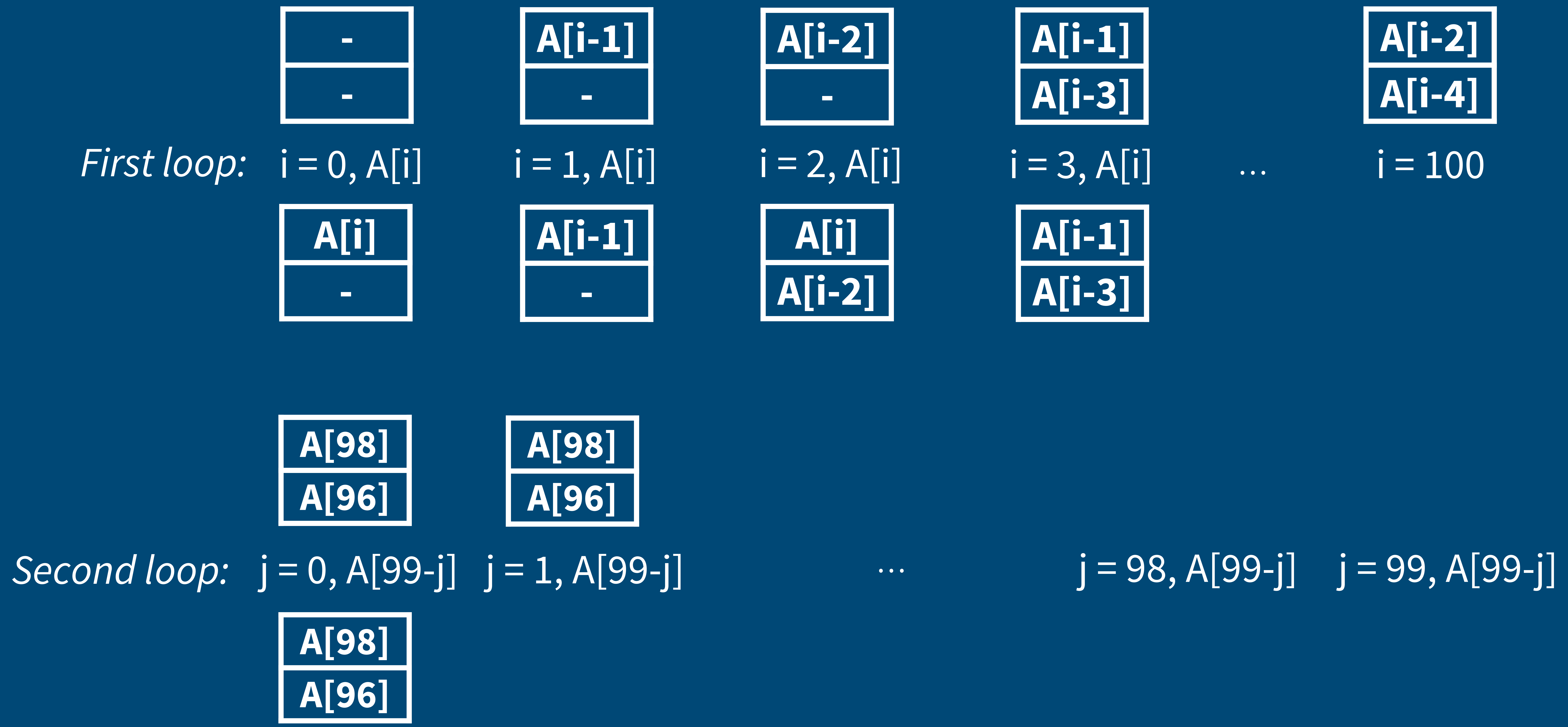
	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	- -	A[i-1] -	A[i-2] -	A[i-1] A[i-3]		A[i-2] A[i-4]
	A[i] -	A[i-1] -	A[i] A[i-2]	A[i-1] A[i-3]		

Second loop:

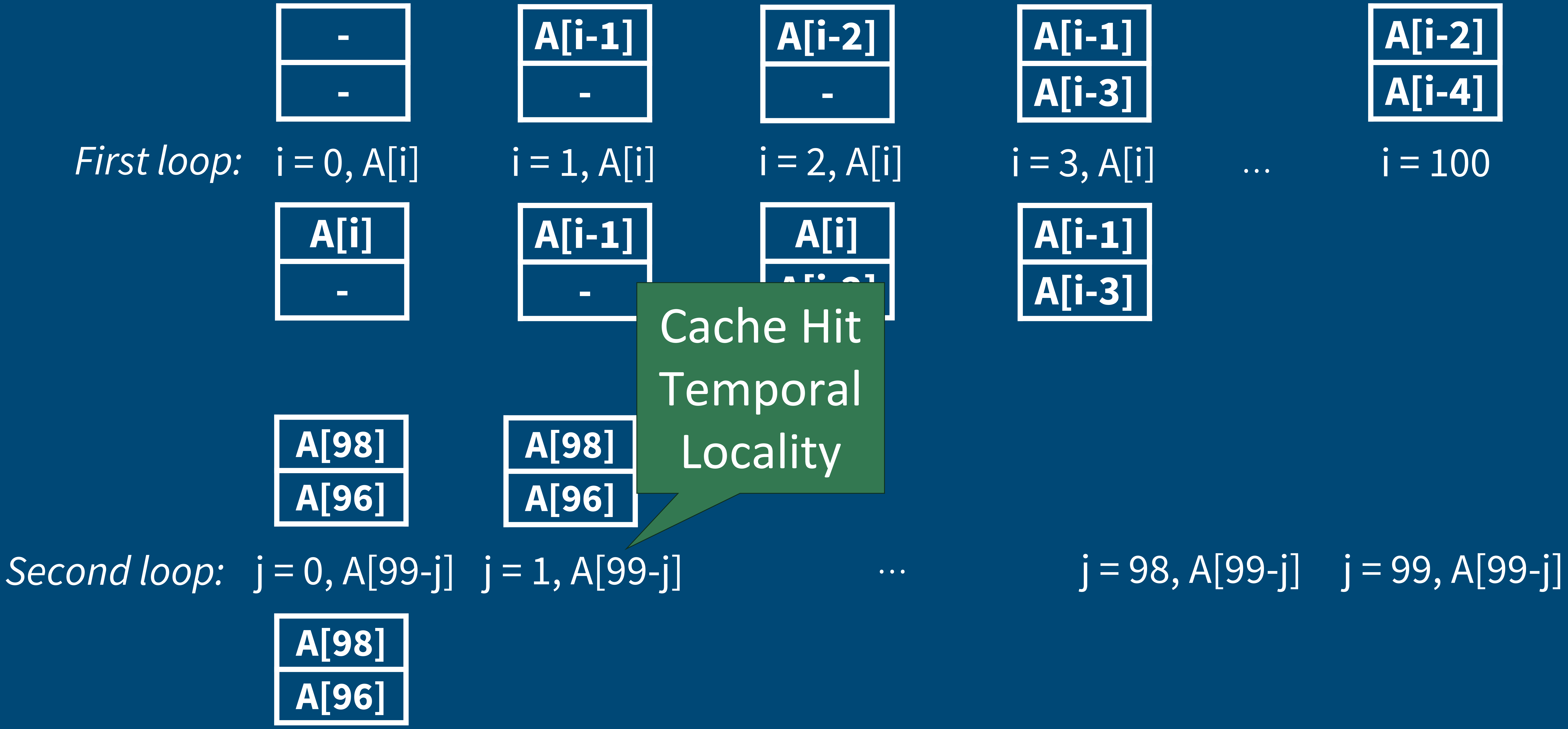
	j = 0, A[99-j]	j = 1, A[99-j]	…	j = 98, A[99-j]	j = 99, A[99-j]
	A[98] A[96]				
	A[98] A[96]				

Symbolic Cache States

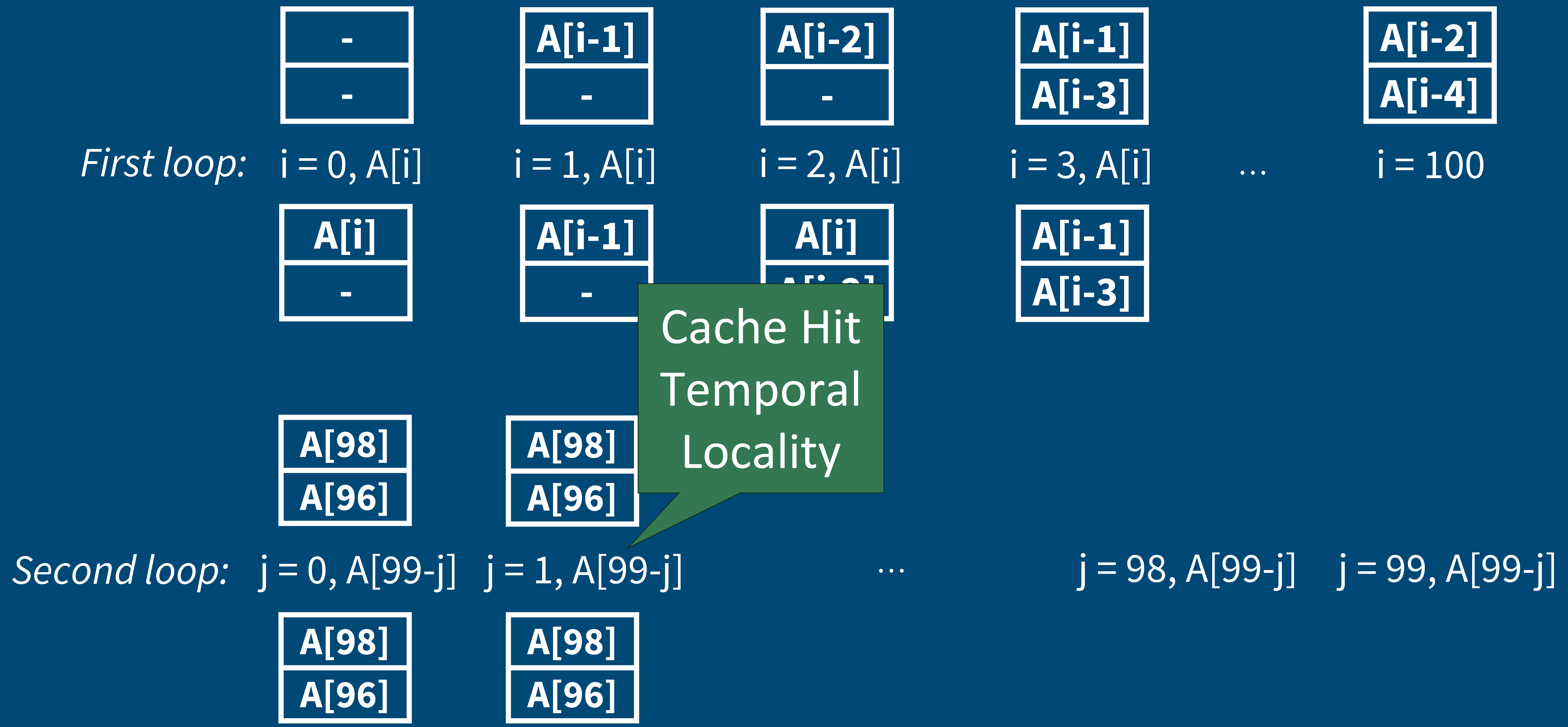
SIC Saarland Informatics Campus



Symbolic Cache States



Symbolic Cache States



Symbolic Cache States

	- -	A[i-1] -	A[i-2] -	A[i-1] A[i-3]		A[i-2] A[i-4]
First loop:	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	A[i] -	A[i-1] -	A[i] A[i-2]	A[i-1] A[i-3]		

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

Symbolic Cache States

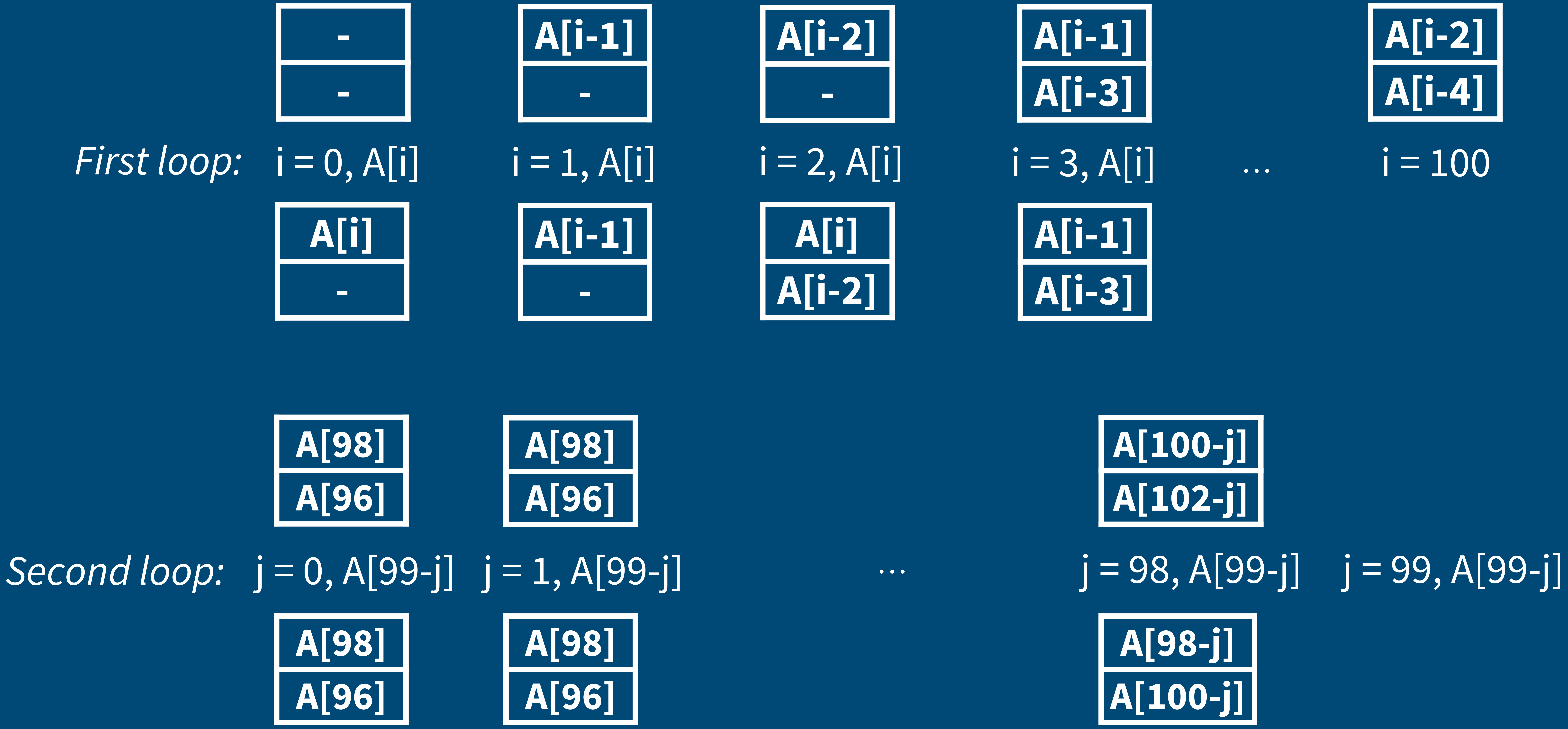
First loop:

	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
Before	- / -	A[i-1] / -	A[i-2] / -	A[i-1] / A[i-3]		A[i-2] / A[i-4]
After	A[i] / -	A[i-1] / -	A[i] / A[i-2]	A[i-1] / A[i-3]		

Second loop:

	j = 0, A[99-j]	j = 1, A[99-j]	…	j = 98, A[99-j]	j = 99, A[99-j]
Before	A[98] / A[96]	A[98] / A[96]		A[100-j] / A[102-j]	
After	A[98] / A[96]	A[98] / A[96]			

Symbolic Cache States

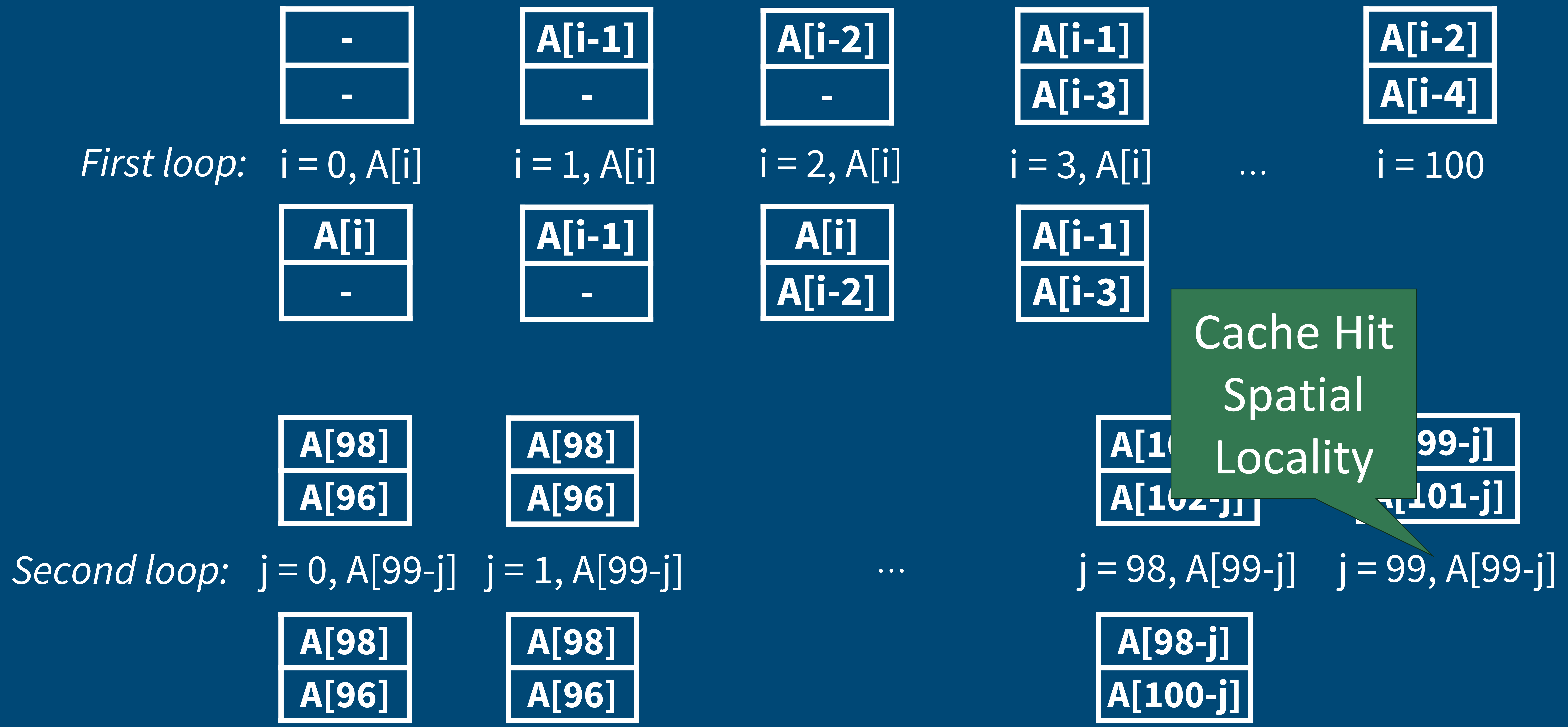


Symbolic Cache States

	- -	A[i-1] -	A[i-2] -	A[i-1] A[i-3]		A[i-2] A[i-4]
First loop:	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	A[i] -	A[i-1] -	A[i] A[i-2]	A[i-1] A[i-3]		

	A[98] A[96]	A[98] A[96]		A[100-j] A[102-j]	A[99-j] A[101-j]
Second loop:	j = 0, A[99-j]	j = 1, A[99-j]	…	j = 98, A[99-j]	j = 99, A[99-j]
	A[98] A[96]	A[98] A[96]		A[98-j] A[100-j]	

Symbolic Cache States



Symbolic Cache States

	- / -	A[i-1] / -	A[i-2] / -	A[i-1] / A[i-3]		A[i-2] / A[i-4]
First loop:	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	A[i] / -	A[i-1] / -	A[i] / A[i-2]	A[i-1] / A[i-3]		

	A[98] / A[96]	A[98] / A[96]		A[1… / A[1…-j]	…99-j] / …[101-j]
Second loop:	j = 0, A[99-j]	j = 1, A[99-j]	…	j = 98, A[99-j]	j = 99, A[99-j]
	A[98] / A[96]	A[98] / A[96]		A[98-j] / A[100-j]	A[99-j] / A[101-j]

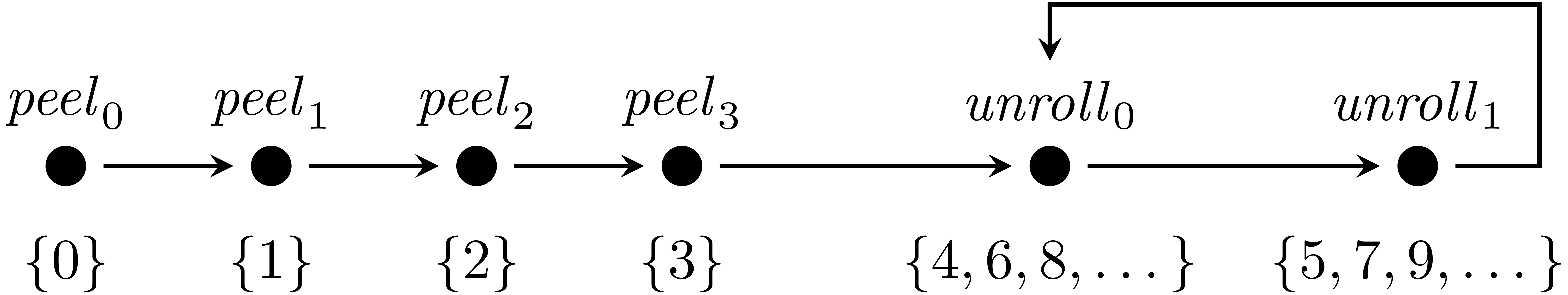
Cache Hit
Spatial
Locality

Symbolic Cache States

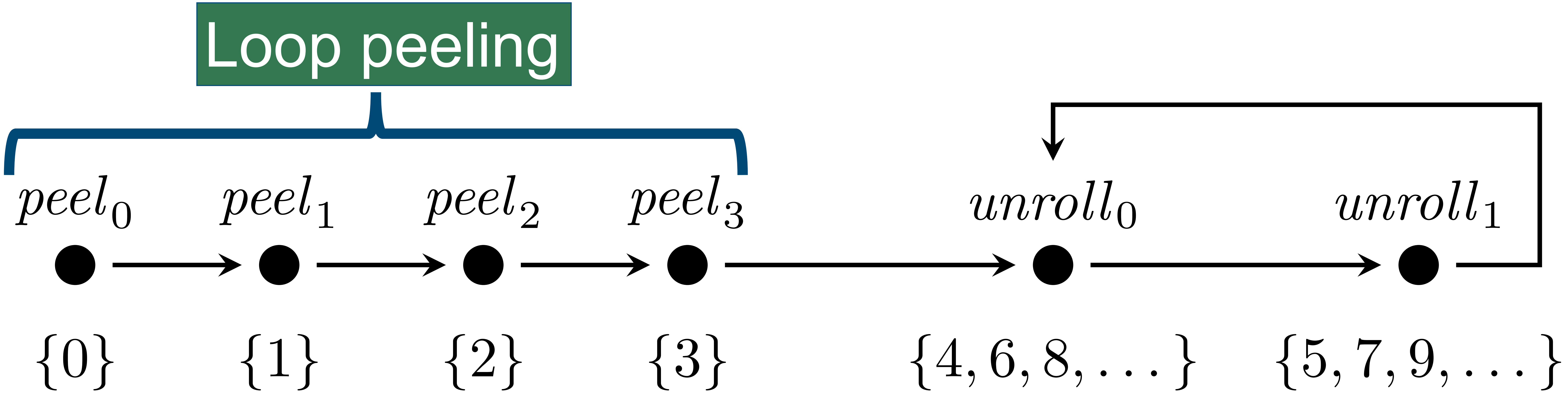
	-	A[i-1]	A[i-2]	A[i-1]		A[i-2]
	-	-	-	A[i-3]		A[i-4]
First loop:	i = 0, A[i]	i = 1, A[i]	i = 2, A[i]	i = 3, A[i]	…	i = 100
	A[i]	A[i-1]	A[i]	A[i-1]		
	-	-	A[i-2]	A[i-3]		

	A[98]	A[98]		A[100-j]	A[99-j]
	A[96]	A[96]		A[102-j]	A[101-j]
Second loop:	j = 0, A[99-j]	j = 1, A[99-j]	…	j = 98, A[99-j]	j = 99, A[99-j]
	A[98]	A[98]		A[98-j]	A[99-j]
	A[96]	A[96]		A[100-j]	A[101-j]

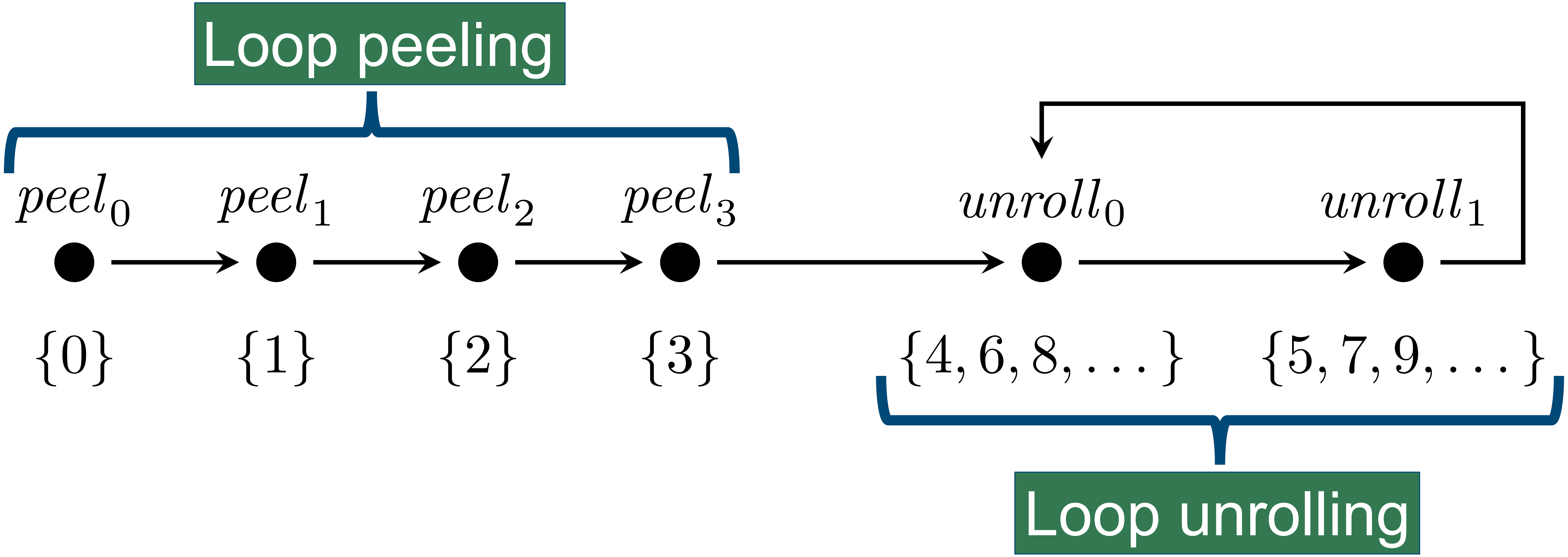
Context-sensitive Analysis



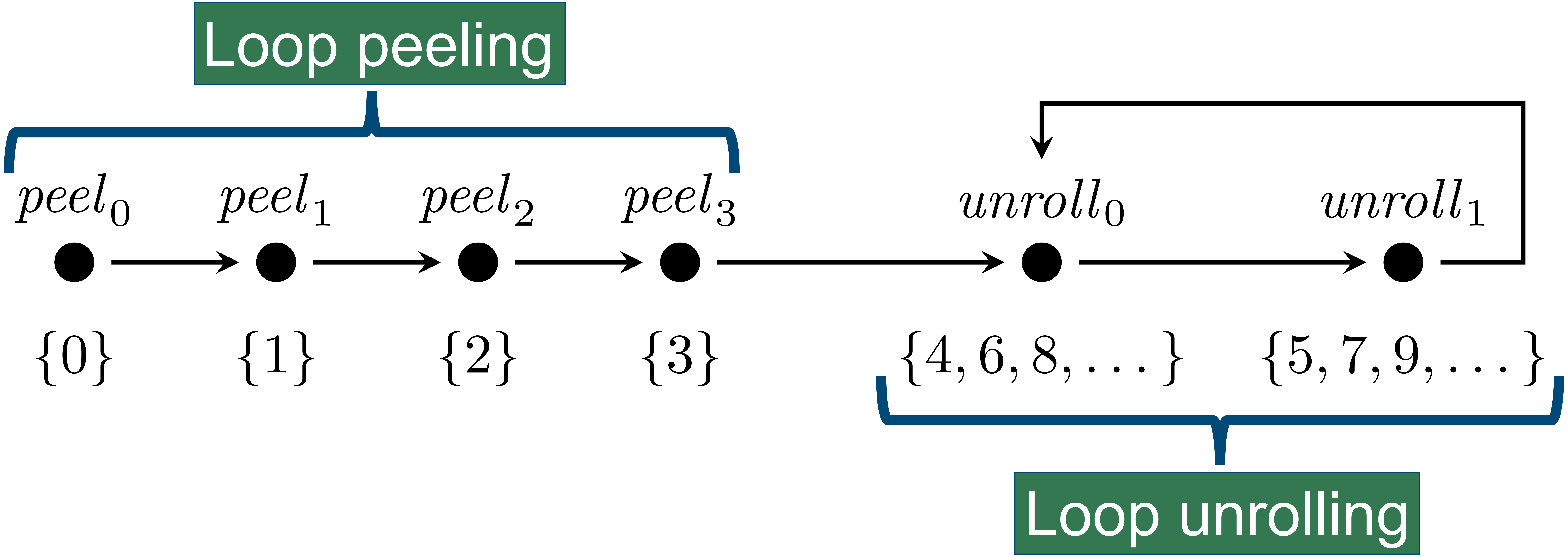
Context-sensitive Analysis



Context-sensitive Analysis



Context-sensitive Analysis



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

$$\frac{\#\text{old misses}}{\#\text{new misses}}$$

Accuracy

$$\frac{\#\text{old misses}}{\#\text{new misses}}$$

time limit
= 1 hour

Accuracy

$$\frac{\#\text{old misses}}{\#\text{new misses}}$$

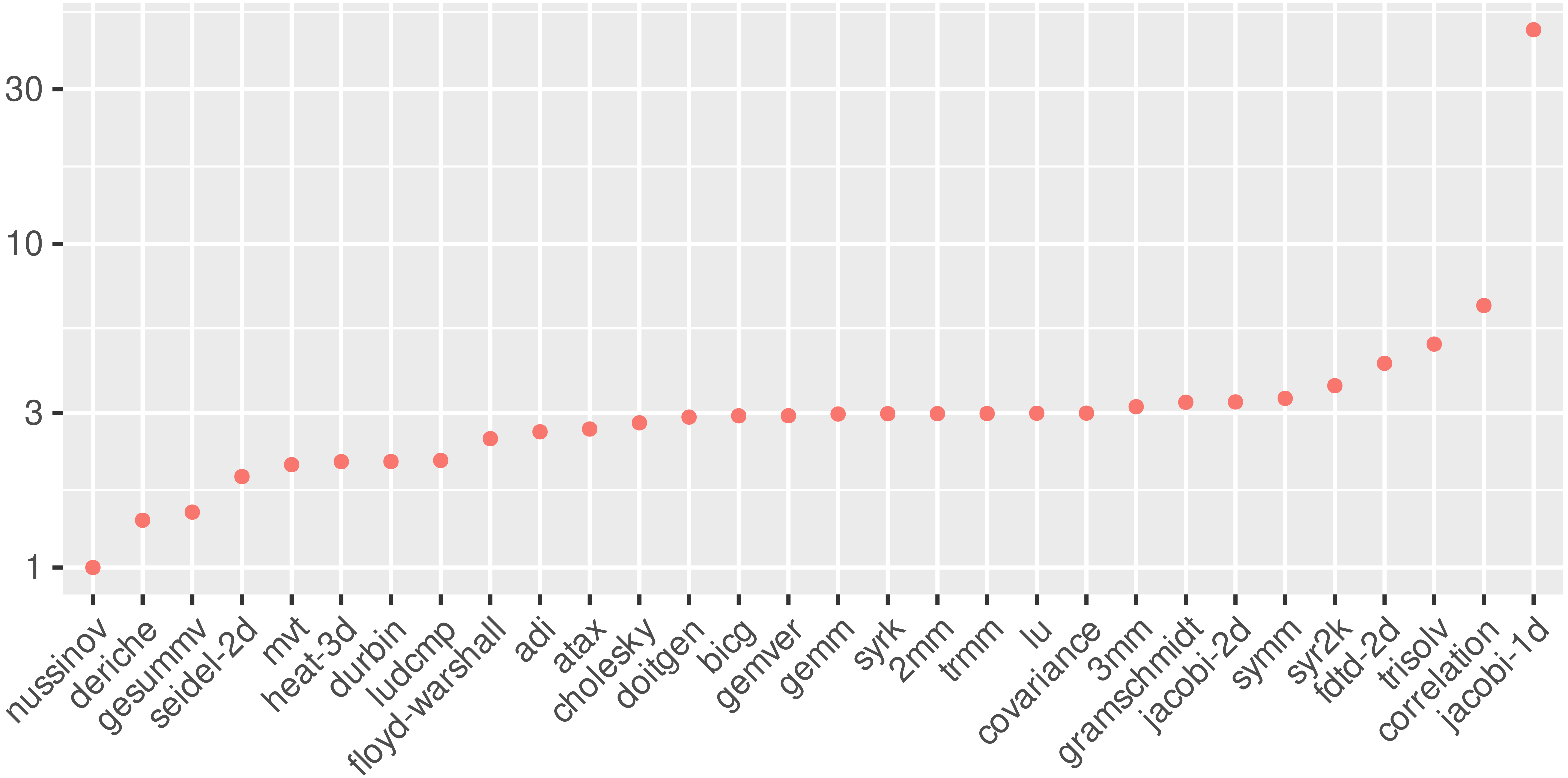
time limit
= 1 hour

PolyBench = Polyhedral Benchmark Suite

Accuracy

$$\frac{\#\text{old misses}}{\#\text{new misses}}$$

time limit
= 1 hour



PolyBench = Polyhedral Benchmark Suite

Scalability

Data size
XS to XL

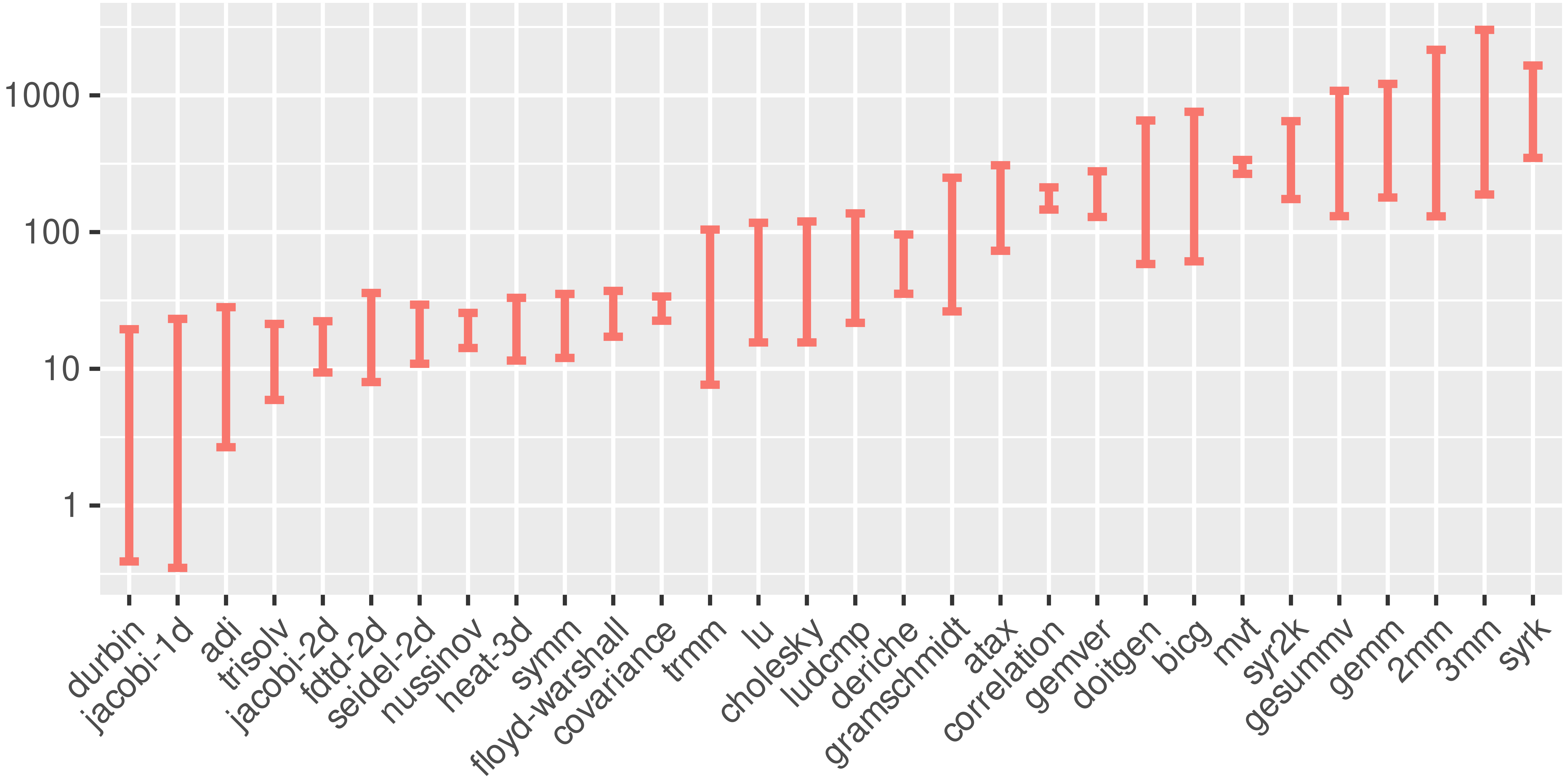
Analysis time
(seconds)

PolyBench = Polyhedral Benchmark Suite

Scalability

Data size
XS to XL

Analysis time
(seconds)



PolyBench = Polyhedral Benchmark Suite

In our paper

SIC Saarland Informatics Campus

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
valentin.touzeau@cs.uni-saarland.de

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-saarland.de

***Abstract*—While instruction cache analysis is essentially a solved problem, data cache analysis is more challenging. In contrast to instruction fetches, the data accesses generated by a memory instruction may vary with the program's inputs and across dynamic occurrences of the same instruction in loops.**

We observe that the plain control-flow graph (CFG) abstraction employed in classical cache analyses is inadequate to capture the dynamic behavior of memory instructions. On top of plain CFGs, accurate analysis of the underlying program's cache behavior is impossible.

Thus, our first contribution is the definition of a more expressive program abstraction coined symbolic control-flow graphs, which can be obtained from LLVM's ScalarEvolution analysis. To exploit this richer abstraction, our main contribution is the development of *symbolic data cache analysis*, a smooth generalization of classical LRU must analysis from plain to symbolic control-flow graphs.

The experimental evaluation demonstrates that symbolic data cache analysis consistently outperforms classical LRU must analysis both in terms of accuracy and analysis runtime.

***Index Terms*—cache analysis, chains of recurrences, data caches, symbolic analysis**

I. INTRODUCTION

Due to technological developments, the latency of accesses to DRAM-based main memory is much higher than the latency of arithmetic and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches between the processor cores and main memory.

In the presence of caches, the latency of a memory access may vary widely depending on the level of the memory hierarchy that is able to serve the access. Hits to the first-level cache take just a few processor cycles, while accesses that miss in all cache levels and thus need to be served by main memory can take hundreds of cycles.

This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The timing variability induced by caches also introduces security challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Cache analysis aims to statically characterize a program's cache behavior by classifying memory accesses in the program as guaranteed cache hits or misses. One perspective on cache analysis is that it is the composition of two phases:

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses.
2) An analysis of this decorated CFG that classifies accesses as "always hit", "always miss", or "unknown".

For instruction cache analysis this two-phase approach works well, as CFGs accurately captures most programs' instruction fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the following simple loop:

```
for (int x = 0; x < 100; x++)
  sum += A[x]
```

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes impossible to make accurate predictions about the program's cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed.

Our first contribution is the definition of symbolic control-flow graphs in Section IV, which is our formalization of the output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic CFGs accurately capture the link between loop iterations and accessed memory blocks via chains of recurrences [8], [9] in a manner that is amenable to static analysis.

To exploit this more expressive program abstraction our main contribution is the development of *symbolic data cache analysis* in Section V, a smooth generalization of Ferdinand's classical LRU must analysis [10], [11] from plain to symbolic control-flow graphs. To fully realize the potential of symbolic data cache analysis we further introduce a context-sensitive analysis combining loop peeling and unrolling in Section VI and various implementation tricks in Section VII.

The experimental evaluation on the PolyBench benchmark suite in Section VIII demonstrates that symbolic cache analysis compares favorably to classical LRU must analysis both in terms of accuracy and analysis runtime.

In our paper

SIC Saarland Informatics Campus

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
valentin.touzeau@cs.uni-saarland.de

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-saarland.de

***Abstract*—While instruction cache analysis is essentially a solved problem, data cache analysis is more challenging. In contrast to instruction fetches, the data accesses generated by a memory instruction may vary with the program's inputs and across dynamic occurrences of the same instruction in loops.**

We observe that the plain control-flow graph (CFG) abstraction employed in classical cache analyses is inadequate to capture the dynamic behavior of memory instructions. On top of plain CFGs, accurate analysis of the underlying program's cache behavior is impossible.

Thus, our first contribution is the definition of a more expressive program abstraction coined symbolic control-flow graphs, which can be obtained from LLVM's ScalarEvolution analysis. To exploit this richer abstraction, our main contribution is the development of *symbolic data cache analysis*, a smooth generalization of classical LRU must analysis from plain to symbolic control-flow graphs.

The experimental evaluation demonstrates that symbolic data cache analysis consistently outperforms classical LRU must analysis both in terms of accuracy and analysis runtime.

***Index Terms*—cache analysis, chains of recurrences, data caches, symbolic analysis**

I. Introduction

Due to technological developments, the latency of accesses to DRAM-based main memory is much higher than the latency of arithmetic and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches between the processor cores and main memory.

In the presence of caches, the latency of a memory access may vary widely depending on the level of the memory hierarchy that is able to serve the access. Hits to the first-level cache take just a few processor cycles, while accesses that miss in all cache levels and thus need to be served by main memory can take hundreds of cycles.

This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The timing variability induced by caches also introduces security challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Cache analysis aims to statically characterize a program's cache behavior by classifying memory accesses in the program as guaranteed cache hits or misses. One perspective on cache analysis is that it is the composition of two phases:

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses.
2) An analysis of this decorated CFG that classifies accesses as "always hit", "always miss", or "unknown".

For instruction cache analysis this two-phase approach works well, as CFGs accurately captures most programs' instruction fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the following simple loop:

```
for (int x = 0; x < 100; x++)
  sum += A[x]
```

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes impossible to make accurate predictions about the program's cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed.

Our first contribution is the definition of symbolic control-flow graphs in Section IV, which is our formalization of the output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic CFGs accurately capture the link between loop iterations and accessed memory blocks via chains of recurrences [8], [9] in a manner that is amenable to static analysis.

To exploit this more expressive program abstraction our main contribution is the development of *symbolic data cache analysis* in Section V, a smooth generalization of Ferdinand's classical LRU must analysis [10], [11] from plain to symbolic control-flow graphs. To fully realize the potential of symbolic data cache analysis we further introduce a context-sensitive analysis combining loop peeling and unrolling in Section VI, and various implementation tricks in Section VII.

The experimental evaluation in Section VIII demonstrates that symbolic cache analysis compares favorably to classical LRU must analysis both in terms of accuracy and analysis runtime.

- Formalization as Abstract Interpretation

In our paper

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
valentin.touzeau@cs.uni-saarland.de

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-saarland.de

***Abstract*—While instruction cache analysis is essentially a solved problem, data cache analysis is more challenging. In contrast to instruction fetches, the data accesses generated by a memory instruction may vary with the program's inputs and across dynamic occurrences of the same instruction in loops.**

We observe that the plain control-flow graph (CFG) abstraction employed in classical cache analyses is inadequate to capture the dynamic behavior of memory instructions. On top of plain CFGs, accurate analysis of the underlying program's cache behavior is impossible.

Thus, our first contribution is the definition of a more expressive program abstraction coined symbolic control-flow graphs, which can be obtained from LLVM's ScalarEvolution analysis. To exploit this richer abstraction, our main contribution is the development of *symbolic data cache analysis*, a smooth generalization of classical LRU must analysis from plain to symbolic control-flow graphs.

The experimental evaluation demonstrates that symbolic data cache analysis consistently outperforms classical LRU must analysis both in terms of accuracy and analysis runtime.

***Index Terms*—cache analysis, chains of recurrences, data caches, symbolic analysis**

I. Introduction

Due to technological developments, the latency of accesses to DRAM-based main memory is much higher than the latency of arithmetic and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches between the processor cores and main memory.

In the presence of caches, the latency of a memory access may vary widely depending on the level of the memory hierarchy that is able to serve the access. Hits to the first-level cache take just a few processor cycles, while accesses that miss in all cache levels and thus need to be served by main memory can take hundreds of cycles.

This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The timing variability induced by caches also introduces security challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Cache analysis aims to statically characterize a program's cache behavior by classifying memory accesses in the program as guaranteed cache hits or misses. One perspective on cache analysis is that it is the composition of two phases:

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses.
2) An analysis of this decorated CFG that classifies accesses as "always hit", "always miss", or "unknown".

For instruction cache analysis this two-phase approach works well, as CFGs accurately captures most programs' instruction fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the following simple loop:

```
for (int x = 0; x < 100; x++)
    sum += A[x]
```

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes impossible to make accurate predictions about the program's cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed.

Our first contribution is the definition of symbolic control-flow graphs in Section IV, which is our formalization of the output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic CFGs accurately capture the link between loop iterations and accessed memory blocks via chains of recurrences [8], [9] in a manner that is amenable to static analysis.

To exploit this more expressive program abstraction our main contribution is the development of *symbolic data cache analysis* in Section V, a smooth generalization of Ferdinand's classical LRU must analysis [10], [11] from plain to symbolic control-flow graphs. To fully realize the potential of symbolic data cache analysis we further introduce a context-sensitive analysis combining loop peeling and unrolling in Section VI and various implementation tricks in Section VII.

The experimental evaluation on the PolyBench benchmark suite in Section VIII demonstrates that symbolic cache analysis compares favorably to classical LRU must analysis both in terms of accuracy and analysis runtime.

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

In our paper

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
valentin.touzeau@cs.uni-saarland.de

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-saarland.de

***Abstract*—While instruction cache analysis is essentially a solved problem, data cache analysis is more challenging. In contrast to instruction fetches, the data accesses generated by a memory instruction may vary with the program's inputs and across dynamic occurrences of the same instruction in loops.**

We observe that the plain control-flow graph (CFG) abstraction employed in classical cache analyses is inadequate to capture the dynamic behavior of memory instructions. On top of plain CFGs, accurate analysis of the underlying program's cache behavior is impossible.

Thus, our first contribution is the definition of a more expressive program abstraction coined symbolic control-flow graphs, which can be obtained from LLVM's ScalarEvolution analysis. To exploit this richer abstraction, our main contribution is the development of *symbolic data cache analysis*, a smooth generalization of classical LRU must analysis from plain to symbolic control-flow graphs.

The experimental evaluation demonstrates that symbolic data cache analysis consistently outperforms classical LRU must analysis both in terms of accuracy and analysis runtime.

***Index Terms*—cache analysis, chains of recurrences, data caches, symbolic analysis**

I. Introduction

Due to technological developments, the latency of accesses to DRAM-based main memory is much higher than the latency of arithmetic and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches between the processor cores and main memory.

In the presence of caches, the latency of a memory access may vary widely depending on the level of the memory hierarchy that is able to serve the access. Hits to the first-level cache take just a few processor cycles, while accesses that miss in all cache levels and thus need to be served by main memory can take hundreds of cycles.

This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The timing variability induced by caches also introduces security challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Cache analysis aims to statically characterize a program's cache behavior by classifying memory accesses in the program as guaranteed cache hits or misses. One perspective on cache analysis is that it is the composition of two phases:

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses.
2) An analysis of this decorated CFG that classifies accesses as "always hit", "always miss", or "unknown".

For instruction cache analysis this two-phase approach works well, as CFGs accurately captures most programs' instruction fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the following simple loop:

```
for (int x = 0; x < 100; x++)
  sum += A[x]
```

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes impossible to make accurate predictions about the program's cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed.

Our first contribution is the definition of symbolic control-flow graphs in Section IV, which is our formalization of the output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic CFGs accurately capture the link between loop iterations and accessed memory blocks via chains of recurrences [8], [9] in a manner that is amenable to static analysis.

To exploit this more expressive program abstraction our main contribution is the development of *symbolic data cache analysis* in Section V, a smooth generalization of Ferdinand's classical LRU must analysis [10], [11] from plain to symbolic control-flow graphs. To fully realize the potential of symbolic data cache analysis we further introduce a context-sensitive analysis combining loop peeling and unrolling in Section VI and various implementation tricks in Section VII.

The experimental evaluation in Section VIII demonstrates that symbolic cache analysis compares favorably to classical LRU must analysis both in terms of accuracy and analysis runtime.

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

In our paper

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
valentin.touzeau@cs.uni-saarland.de

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-saarland.de

Abstract—While instruction cache analysis is essentially a solved problem, data cache analysis is more challenging. In contrast to instruction fetches, the data accesses generated by a memory instruction may vary with the program's inputs and across dynamic occurrences of the same instruction in loops.

We observe that the plain control-flow graph (CFG) abstraction employed in classical cache analyses is inadequate to capture the dynamic behavior of memory instructions. On top of plain CFGs, accurate analysis of the underlying program's cache behavior is impossible.

Thus, our first contribution is the definition of a more expressive program abstraction coined symbolic control-flow graphs, which can be obtained from LLVM's ScalarEvolution analysis. To exploit this richer abstraction, our main contribution is the development of *symbolic data cache analysis*, a smooth generalization of classical LRU must analysis from plain to symbolic control-flow graphs.

The experimental evaluation demonstrates that symbolic data cache analysis consistently outperforms classical LRU must analysis both in terms of accuracy and analysis runtime.

***Index Terms*—cache analysis, chains of recurrences, data caches, symbolic analysis**

I. INTRODUCTION

Due to technological developments, the latency of accesses to DRAM-based main memory is much higher than the latency of arithmetic and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches between the processor cores and main memory.

In the presence of caches, the latency of a memory access may vary widely depending on the level of the memory hierarchy that is able to serve the access. Hits to the first-level cache take just a few processor cycles, while accesses that miss in all cache levels and thus need to be served by main memory can take hundreds of cycles.

This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The timing variability induced by caches also introduces security challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Cache analysis aims to statically characterize a program's cache behavior by classifying memory accesses in the program as guaranteed cache hits or misses. One perspective on cache analysis is that it is the composition of two phases:

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses.
2) An analysis of this decorated CFG that classifies accesses as "always hit", "always miss", or "unknown".

For instruction cache analysis this two-phase approach works well, as CFGs accurately captures most programs' instruction fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the following simple loop:

```
for (int x = 0; x < 100; x++)
    sum += A[x]
```

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes impossible to make accurate predictions about the program's cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed.

Our first contribution is the definition of symbolic control-flow graphs in Section IV, which is our formalization of the output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic CFGs accurately capture the link between loop iterations and accessed memory blocks via chains of recurrences [8], [9] in a manner that is amenable to static analysis.

To exploit this more expressive program abstraction our main contribution is the development of *symbolic data cache analysis* in Section V, a smooth generalization of Ferdinand's classical LRU must analysis [10], [11] from plain to symbolic control-flow graphs. To fully realize the potential of symbolic data cache analysis we further introduce a context-sensitive analysis combining loop peeling and unrolling in Section VI and various implementation tricks in Section VII.

The experimental evaluation on the PolyBench benchmark suite in Section VIII demonstrates that symbolic cache analysis compares favorably to classical LRU must analysis both in terms of accuracy and analysis runtime.

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

In our paper

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
valentin.touzeau@cs.uni-saarland.de

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-saarland.de

***Abstract*—While instruction cache analysis is essentially a solved problem, data cache analysis is more challenging. In contrast to instruction fetches, the data accesses generated by a memory instruction may vary with the program's inputs and across dynamic occurrences of the same instruction in loops.**

We observe that the plain control-flow graph (CFG) abstraction employed in classical cache analyses is inadequate to capture the dynamic behavior of memory instructions. On top of plain CFGs, accurate analysis of the underlying program's cache behavior is impossible.

Thus, our first contribution is the definition of a more expressive program abstraction coined symbolic control-flow graphs, which can be obtained from LLVM's ScalarEvolution analysis. To exploit this richer abstraction, our main contribution is the development of *symbolic data cache analysis*, a smooth generalization of classical LRU must analysis from plain to symbolic control-flow graphs.

The experimental evaluation demonstrates that symbolic data cache analysis consistently outperforms classical LRU must analysis both in terms of accuracy and analysis runtime.

***Index Terms*—cache analysis, chains of recurrences, data caches, symbolic analysis**

I. Introduction

Due to technological developments, the latency of accesses to DRAM-based main memory is much higher than the latency of arithmetic and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches between the processor cores and main memory.

In the presence of caches, the latency of a memory access may vary widely depending on the level of the memory hierarchy that is able to serve the access. Hits to the first-level cache take just a few processor cycles, while accesses that miss in all cache levels and thus need to be served by main memory can take hundreds of cycles.

This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The timing variability induced by caches also introduces security challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Cache analysis aims to statically characterize a program's cache behavior by classifying memory accesses in the program as guaranteed cache hits or misses. One perspective on cache analysis is that it is the composition of two phases:

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses.
2) An analysis of this decorated CFG that classifies accesses as "always hit", "always miss", or "unknown".

For instruction cache analysis this two-phase approach works well, as CFGs accurately captures most programs' instruction fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the following simple loop:

```
for (int x = 0; x < 100; x++)
    sum += A[x]
```

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes impossible to make accurate predictions about the program's cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed.

Our first contribution is the definition of symbolic control-flow graphs in Section IV, which is our formalization of the output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic CFGs accurately capture the link between loop iterations and accessed memory blocks via chains of recurrences [8], [9] in a manner that is amenable to static analysis.

To exploit this more expressive program abstraction our main contribution is the development of *symbolic data cache analysis* in Section V, a smooth generalization of Ferdinand's classical LRU must analysis [10], [11] from plain to symbolic control-flow graphs. To fully realize the potential of symbolic data cache analysis we further introduce a context-sensitive analysis combining loop peeling and unrolling in Section VI and various implementation tricks in Section VII.

The experimental evaluation on the PolyBench benchmark suite in Section VIII demonstrates that symbolic cache analysis compares favorably to classical LRU must analysis both in terms of accuracy and analysis runtime.

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

Questions?