FAT POINTERS

Arjun Menon

IIT Madras

What is a Fat Pointer?

- Typically metadata contains the “base” and “bounds” of the pointer which is essentially the valid accessible memory region by the pointer
- if((ADDRESS >= PTR.base) && (ADDRESS <= PTR.bound))
        perform load or store
  else
        jump to error handler

Recap of Memory-based attacks

- Spatial (Buffer overflow)
  - Stack overflow
  - Heap overflow
  - Format string attacks
- Temporal
  - Use-after-free
  - Double free

Object based

Key concept: Base and bounds associated per object

Advantage:

- Memory layout of objects is not changed
  - Improves source and binary compatibility

Disadvantage:

- Overflows can occur on a sub-object basis
- Performance bottleneck: Object lookup is a range lookup
  - Typically implemented using splay trees
- Out-of-bounds pointers need special care

Examples: [1], [2], [3]

```
struct {
   char id[8];
   int account_balance;
 }  bank_account;
char* ptr = &(bank_account.id);
strcpy(ptr, "overflow...");
```

Pointer based

Key concept: Base and bounds associated per pointer

Advantages:

- Can enforce complete spatial safety
- Out-of-bounds pointers are taken care implicitly

Disadvantage:

- Performance overhead: Propagation and checking of base and bounds
- Changes memory layout in a programmer visible way
- Do not handle arbitrary casts
- May be not support dynamic linking of libraries

Examples: [4], [5], [6], [7]

Agenda

1. **SoftBound [4]**
2. Low-fat Pointers [5]
3. WatchDog [6]
4. **Shakti-T [7]**

1.SoftBound (PLDI ‘09)

SoftBound

- Tries to combine advantages of both object and pointer based solutions
- Source code compatibility
  - Disjoint metadata: Avoids any programmer visible memory layout changes
  - Allows arbitrary casts
- Completeness
  - Guarantees spatial safety
  - Includes a formal proof
- Separate compilation
  - Allows library code to be recompiled with SoftBound and dynamically linked

Pointer dereference check

```
check (ptr, ptr_base, ptr_bound, sizeof(*ptr))
value= *ptr;
```

```
void check(ptr, base, bound, size) {
    if ((ptr < base) || (ptr+size > bound)) {
        abort();
    }
}
```

Creating pointers

1. Explicit memory allocation i.e. malloc()

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL)
   ptr_bound = NULL;
```

2. Taking the address of a global or a stack allocated variable using the “&” operator

```
int array[100];
ptr = &array;
ptr_base = &array[0];
ptr_bound = ptr_base+
                sizeof(array);
```

Pointer arithmetic and pointer assignment

- new_ptr= ptr + index
- No checks are required
  - Out-of-bounds value of newptr_bound is fine as long as “newptr” is not dereferenced

```
newptr = ptr + index;
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

Optional narrowings of pointer bounds

1. Creating a pointer to a field of a structure.

NARROWED

```
struct { ... int num; ... } *n;
... p = &(n->num);
p_base = max(&(n->num), n_base);
p_bound = min(p_base + sizeof(n->num), n_bound);
```

2. Creating a pointer to an element of an array.

NOT NARROWED

```
memset(&arr[4], 0, size);
p_base = arr_base;
p_bound = arr_bound;
```

In-Memory Pointer Metadata Encoding

1. Load

```
int** ptr;
int* new_ptr;
new_ptr= *ptr;
```

⇒

```
int** ptr;
int *new_ptr;
. . .
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
newptr = *ptr;
newptr_base = table_lookup(ptr)->base;
newptr_bound = table_lookup(ptr)->bound;
```

2. Store

```
int** ptr;
int* new_ptr;
(*ptr)= new_ptr;
```

⇒

```
int** ptr;
int *new_ptr;
. . .
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
(*ptr) = new_ptr;
table_lookup(ptr)->base = newptr_base;
table_lookup(ptr)->bound = newptr_bound;
```

Metadata Propagation with Function Calls

```
int func(char *s)
{ . . . }
int val = func(ptr);
```

→

```
int sb_func(char *s, void* s_base, void* s_bound)
{ . . . }
int val = sb_func(ptr, ptr_base, ptr_bound);
```

- Functions that return a pointer are changed to return a 3-element structure by value

Disadvantages

- Performance overhead of 67% on average
- Does not provide security against temporal attacks

2.Low-Fat Pointers (CCS ‘13)

Low-fat Pointers

- Use the upper unused bits of virtual address to store the base and bounds
- New, compact fat-pointer encoding and implementation (BIMA)
- Dedicated hardware checks in parallel if the Effective Address (EA) is within the valid base and bounds
  - Does not affect the processor clock speed
- Assumptions:
  - The memory is tagged
    - Every word has a type associated with it

Aligned Encoding

- Assumption
  - The pointer is aligned on a boundary that is a power of 2
  - The size of the segment the pointer is referencing is also a power of two (i.e. 2^B for some B)
- The base can be determined by replacing B bits in the LSB with 0's

  base= A - (A & ((1 << B) -1))
- The bound can be determined by replacing B bits in the LSB with 1's
- Therefore, only B bits are required to represent both the base and the bounds
- Disadvantage:
  - Very high memory fragmentation

BIMA encoding

63–58	57–52	51–46	45–0
B	I	M	A
6	6	6	46

- B: Block size exponent
- I: Minimum bound
- M: Maximum bound
- A: Address

The formula

$$\text{carry} = 1 \ll (B + |I|)$$

$$\text{Atop} = (A \;\&\; \overline{(\text{carry}-1)})$$

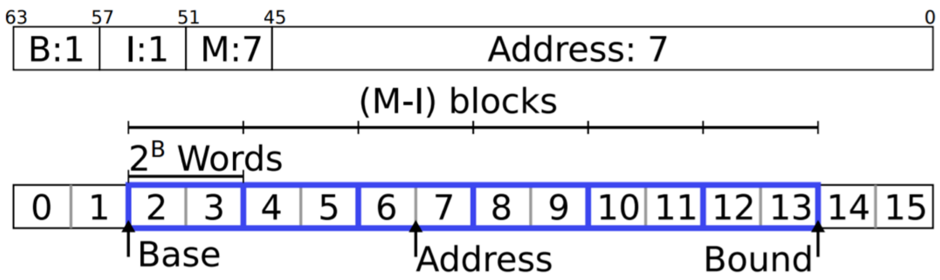
$$\text{Mshift} = M \ll B$$

$$\text{Ishift} = I \ll B$$

$$D_{under} = (A \gg B)[5{:}0] < I \;?\; (\text{carry} \mid \text{Atop}) - \text{Ishift} : \text{Atop} - \text{Ishift}$$

$$D_{over} = (A \gg B)[5{:}0] > M \;?\; (\text{carry} \mid \text{Mshift}) - \text{Atop} : \text{Mshift} - \text{Atop}$$

Example



Base = 2
Bound = 13
Address = 7

carry = 1 << (B + |I|)
Atop = (A & (carry-1))
Mshift = M << B
Ishift = I << B
D_{under} = (A >> B)[5:0] < I ?
(carry | Atop) - Ishift : Atop - Ishift
D_{over} = (A >> B)[5:0] > M?
(carry | Mshift) - Atop : Mshift - Atop

carry = 1 << (1+6) = ‘b1000_0000
Atop = ‘b111 & (‘b0111_111) = ‘b111 = 7
Mshift = 7 << 1 = 14
Ishift = 1 << 1 = 2
D_{under} = 3 < 1 ?
(carry | Atop) - Ishift : 7 - 2 = **5**
D_{over} = (A >> B)[5:0] > M?
(carry | Mshift) - Atop : 14 - 7 = **7**

Drawbacks

- Cannot express Out-of-Bounds pointer implicitly
- Memory fragmentation (~3%)
- Managing the base and bounds of stack allocated variables
- Prevents only spatial, and not temporal memory attacks

3.WatchDog (ISCA ‘12)

Key idea

- Associate a base, bound, lock and a key with every pointer
- Hardware is responsible for propagation and checking of metadata
- Software manages the values of these metadata
- To prevent temporal attacks, fetch the value at the lock address, and check if it matches the value of the key

Temporal protection (Conceptual)

- Assumptions:
  - Every register has a sidecar part which stores the metadata (id or lock)
  - Every memory address has a shadow region which stores the id of the pointer stored in that memory location

(a) Load

```
ld R1 <- memory[R2]
```

```
check R2.id
R1. id <- shadow[R2.val].id
R1.val <-memory[R2.val].val
```

(b) Store

```
st memory[R2]<- R1
```

```
check R2.id
shadow[R2.val].id <- R1.id
memory[R2.val].val <- R1.val
```

(c) Add immediate

```
add R1 <- R2,imm
```

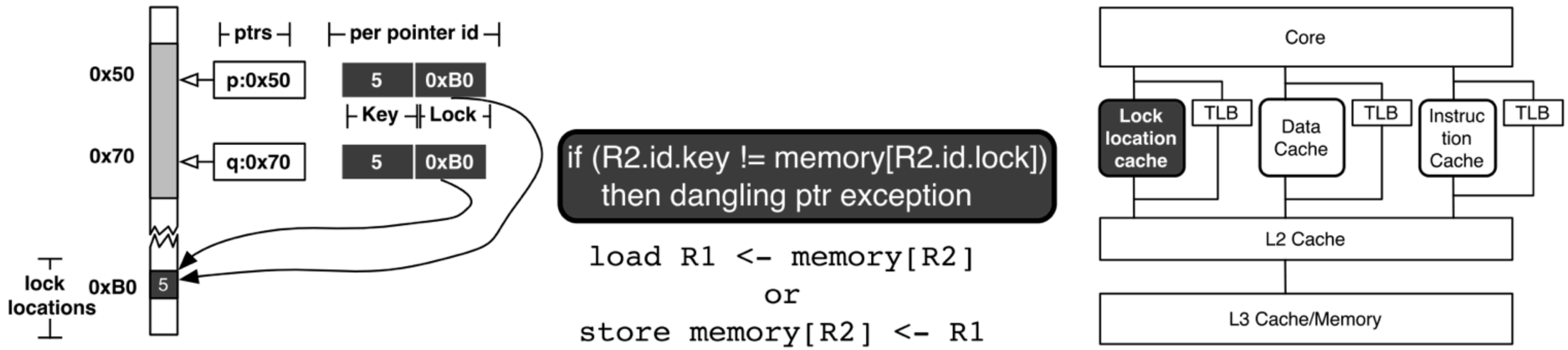
```
R1. id <-  R2.id
R1.val <- R2.val + imm
```

(d) Add

```
add R1 <- R2, R3
```

```
if (R2. id  != INVALID)
   R1.id <-  R2.id
else
    R1.id <- R3.id
R1.val <- R2.val + R3.val
```

Lock and Key Mechanism



a Lock and key identifier b Watchdog check c Lock location cache

Code instrumentation

(a) Heap allocation (runtime)

```
p = malloc(size)
```

```
key = unique_identifier++;
lock = allocate_new_lock();
*(lock) = key;
id = (key, lock);
q = setident(p, id);
```

(b) Heap deallocation (runtime)

```
id = getident(p);
*(id.lock) = INVALID;
add_to_free_list(id.lock)
```

```
free(p)
```

(c) Stack allocation (hardware)

```
call
```

```
stack_key = stack_key + 1
stack_lock = stack_lock + 8
memory[stack_lock] = stack_key
%rsp.id  = (stack_key, stack_lock)
```

(d) Stack deallocation (hardware)

```
return
```

```
memory[stack_lock] = INVALID
stack_lock = stack_lock - 8
current_key = memory[stack_lock]
%rsp.id = (current_key, stack_lock)
```

Drawbacks

- The metadata overhead per pointer is 256bits
- Separate lock location cache

Existing Hardware Solutions (Common design choice)

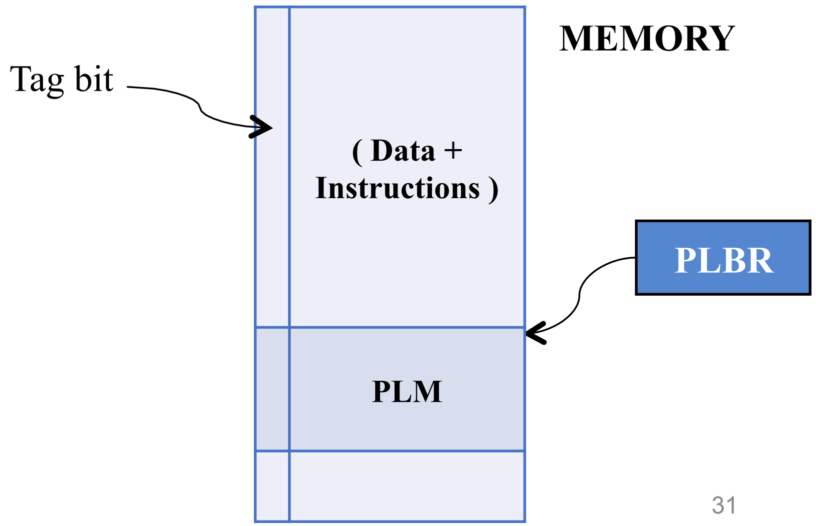
- Store the base and bound values (in shadow registers) in the register file alongside the value.

- It has the following implications:
  - Most of the base and bound shadow registers remain unused
  - When register spilling occurs, the base and bounds are also discarded
  - If aliased pointers exists in the registers, the base and bound values will have duplicate entries

4.Shakti-T (HASP ‘17)

Proposed solution

1. Have a common memory region called Pointer Limits Memory (PLM) to store the values of base and bounds
   - Declare a new register which points the base address of PLM
   - Base and bounds are associated with a pointer by the value of the offset (*pointer_id*)

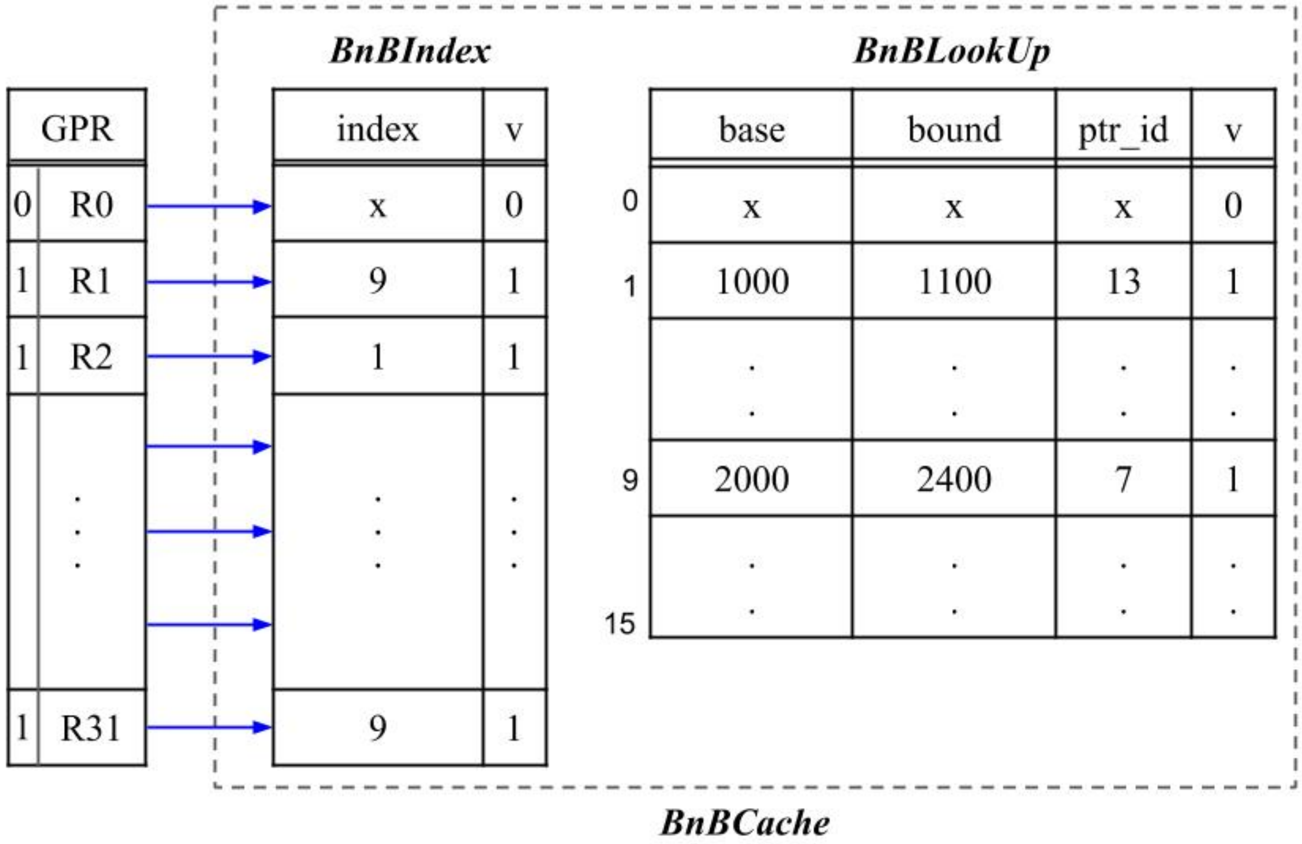


2. Add a 1-bit tag to every memory word
   - 0: Data/Instruction
   - 1: Pointer

Proposed solution

3. Maintain a separate table alongside the register file that stores the values of base and bounds (and the *pointer_id*)
    - One level indexing is used to associate a GPR holding a pointer with its corresponding values of base and bounds

Proposed solution



New Instructions

- Write tag [*wrtag* rd, imm]
- Write PLM [*wrplm* rs1, r2, rs3]
- Load base and bounds [*ldbnb* rd, rs1]
- Load pointer [*ldptr* rd, rs1, imm]
- Write special register [*wrspreg* rs1, imm]
- Read special register [*rdspreg* rd, imm]
- Function store [*fnst* rs1, imm(rs2)]
- Function load [*fnld* rd, imm(rs1)]

Example programs

- Dynamic memory allocation

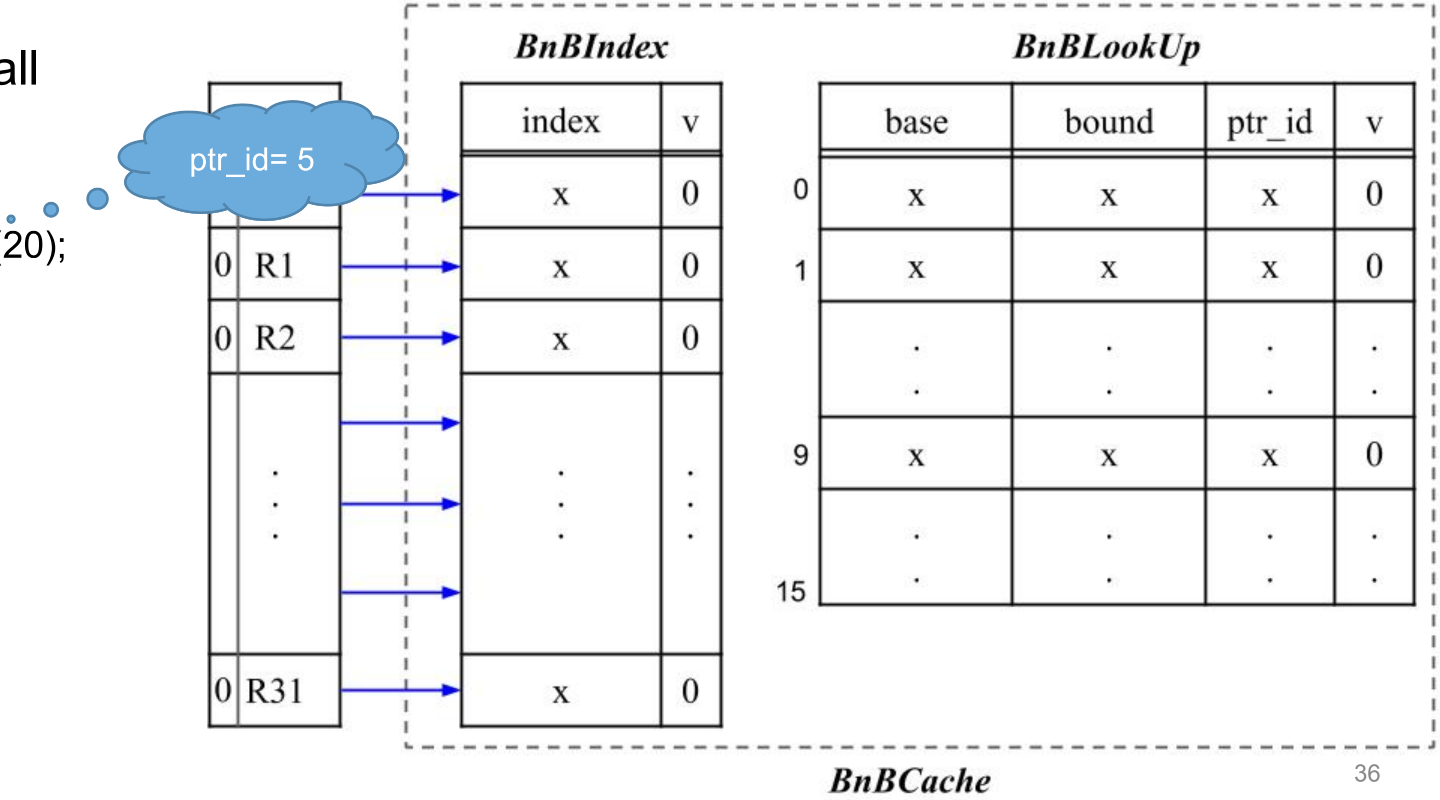
char **ptr* = malloc(n);

1. After malloc returns with the base address, the bounds is computed as
   bound = base + n
2. Store the value of base and bound in the PLM at the address *PLBR+ptr_id* using the *wrplm* instruction.
3. When storing the initialized value of *ptr* in the memory at an address *addr*, store the value of *ptr_id* at *addr*+8

Example programs

- A function call

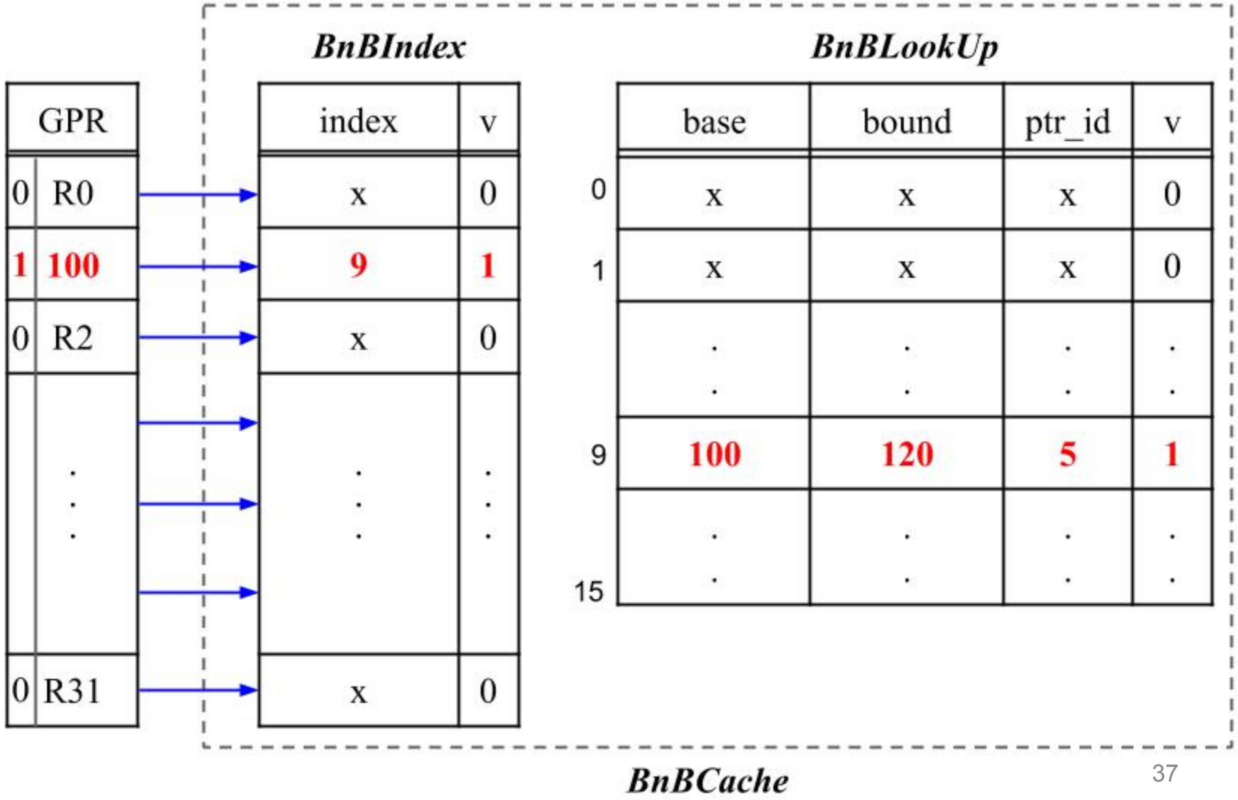
```
function foo( ) {
    char *ptr5;
    ptr5= malloc(20);
    …
    bar( );
    …
}
```



Example programs

- A function call

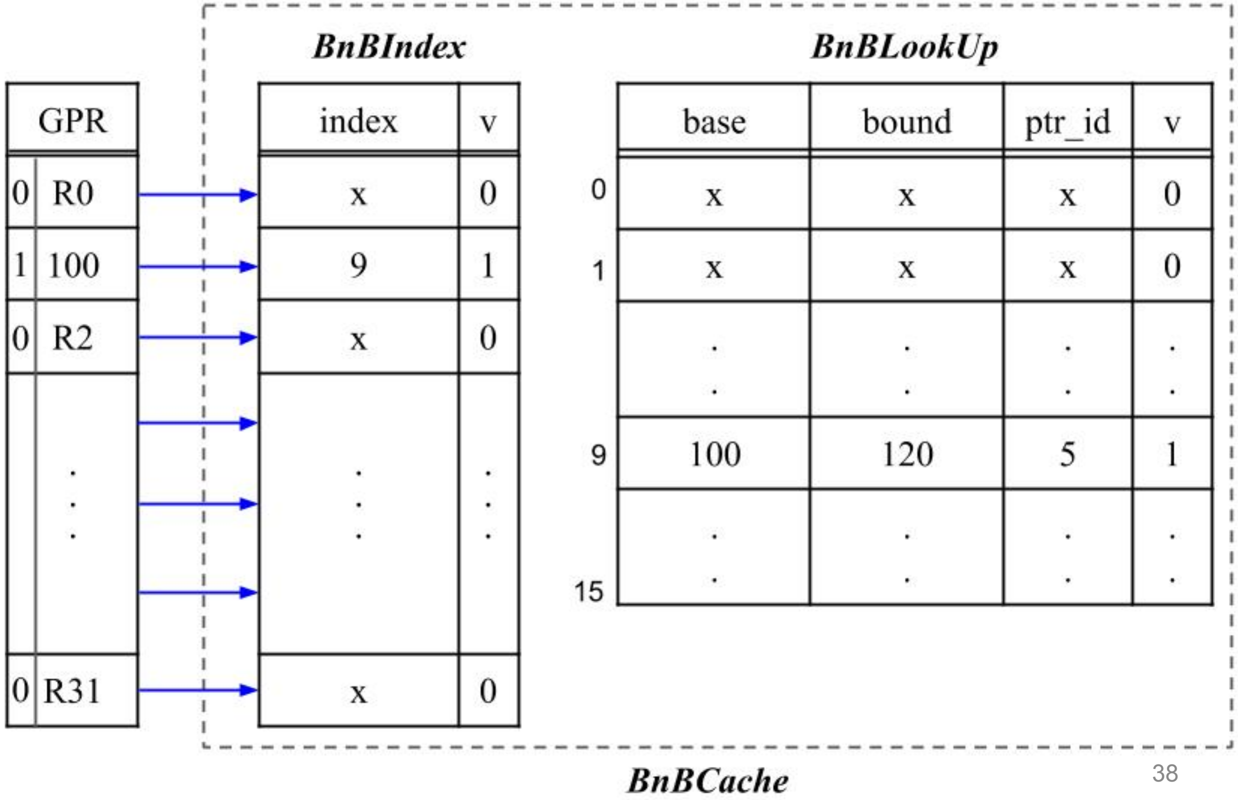
```
function foo( ) {
    char *ptr5;
    ptr5= malloc(20);
    …
    bar( );
    …
}
```



Example programs

- A function call

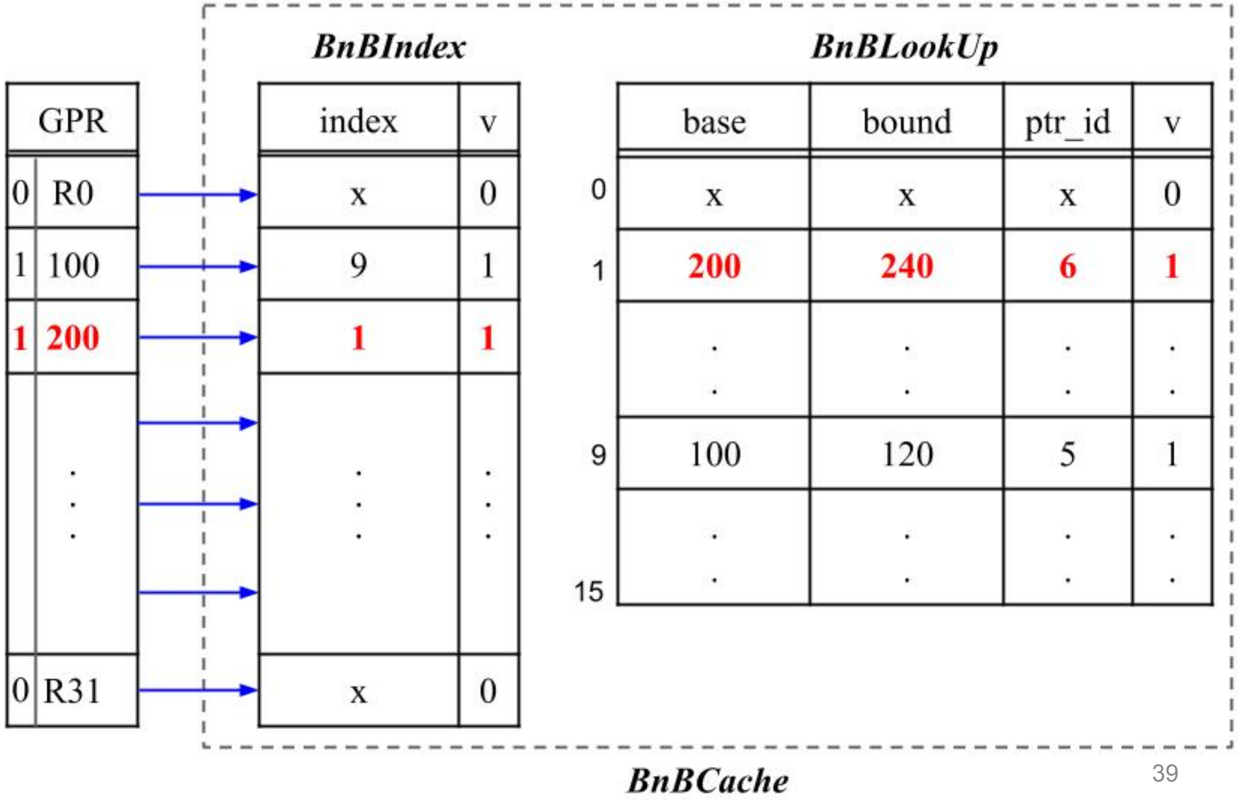
```
function foo( ) {
    char *ptr5;
    ptr5= malloc(20);
    …
    bar( );
    …
}
```



Example programs

- A function call

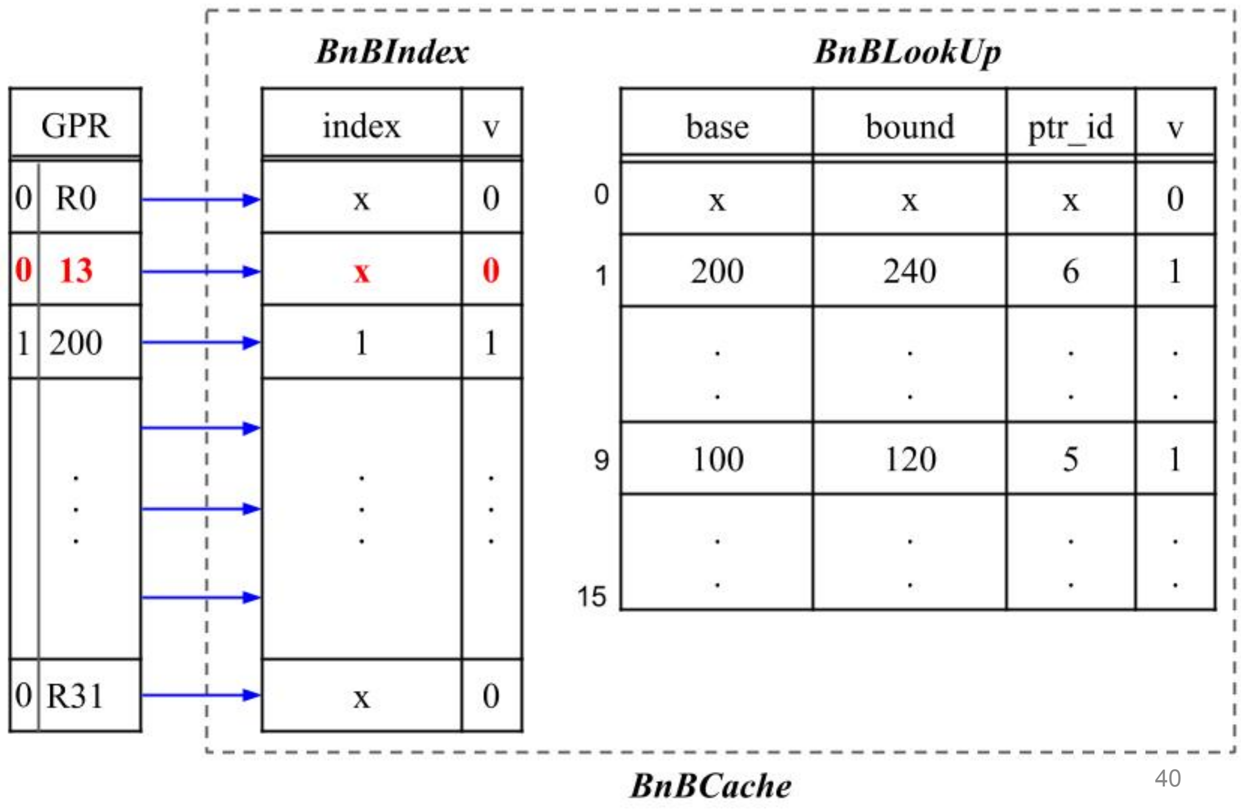
```
function bar( ) {
    char *ptr6;
    ptr6= malloc(40);
    …
    int c= 4+5;
    …
    free(ptr6);
    return;
}
```



Example programs

- A function call

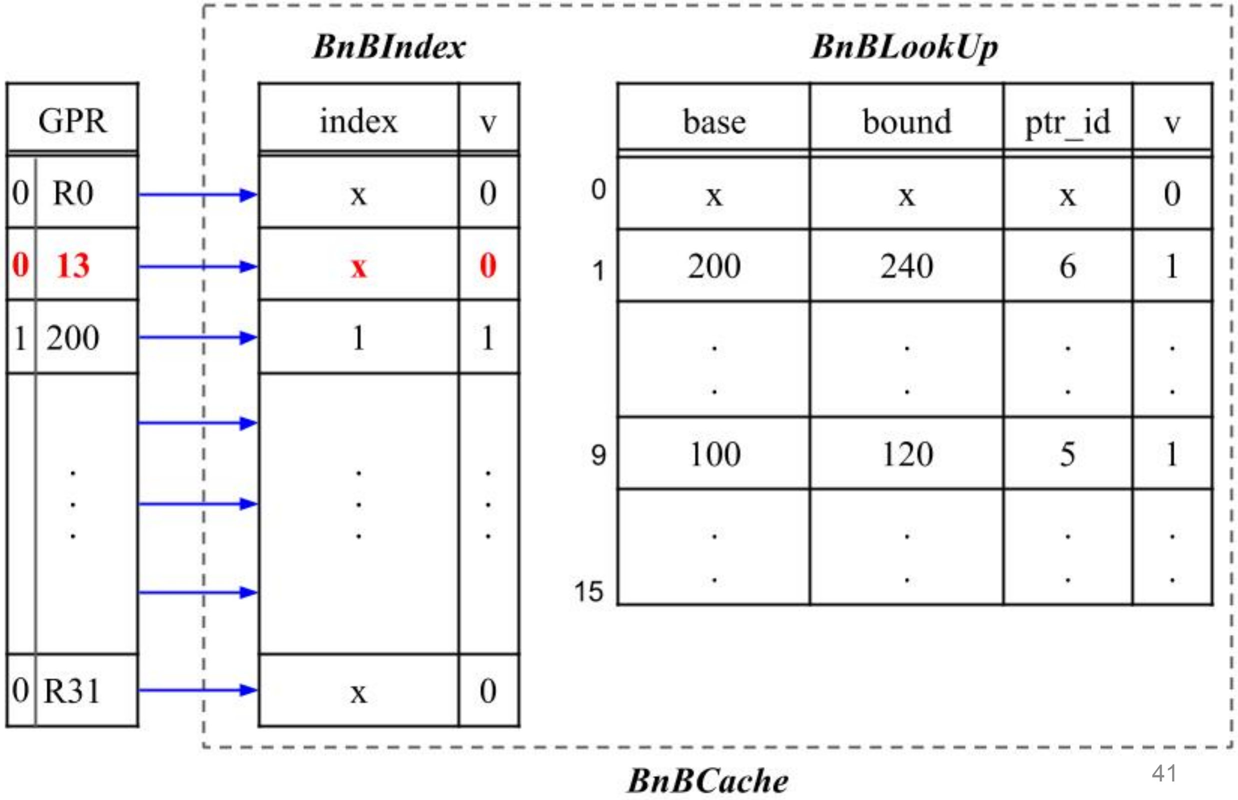
```
function bar( ) {
    char *ptr6;
    ptr6= malloc(40);
    …
    int c= 10+3;
    …
    free(ptr6);
    return;
}
```



Example programs

- A function call

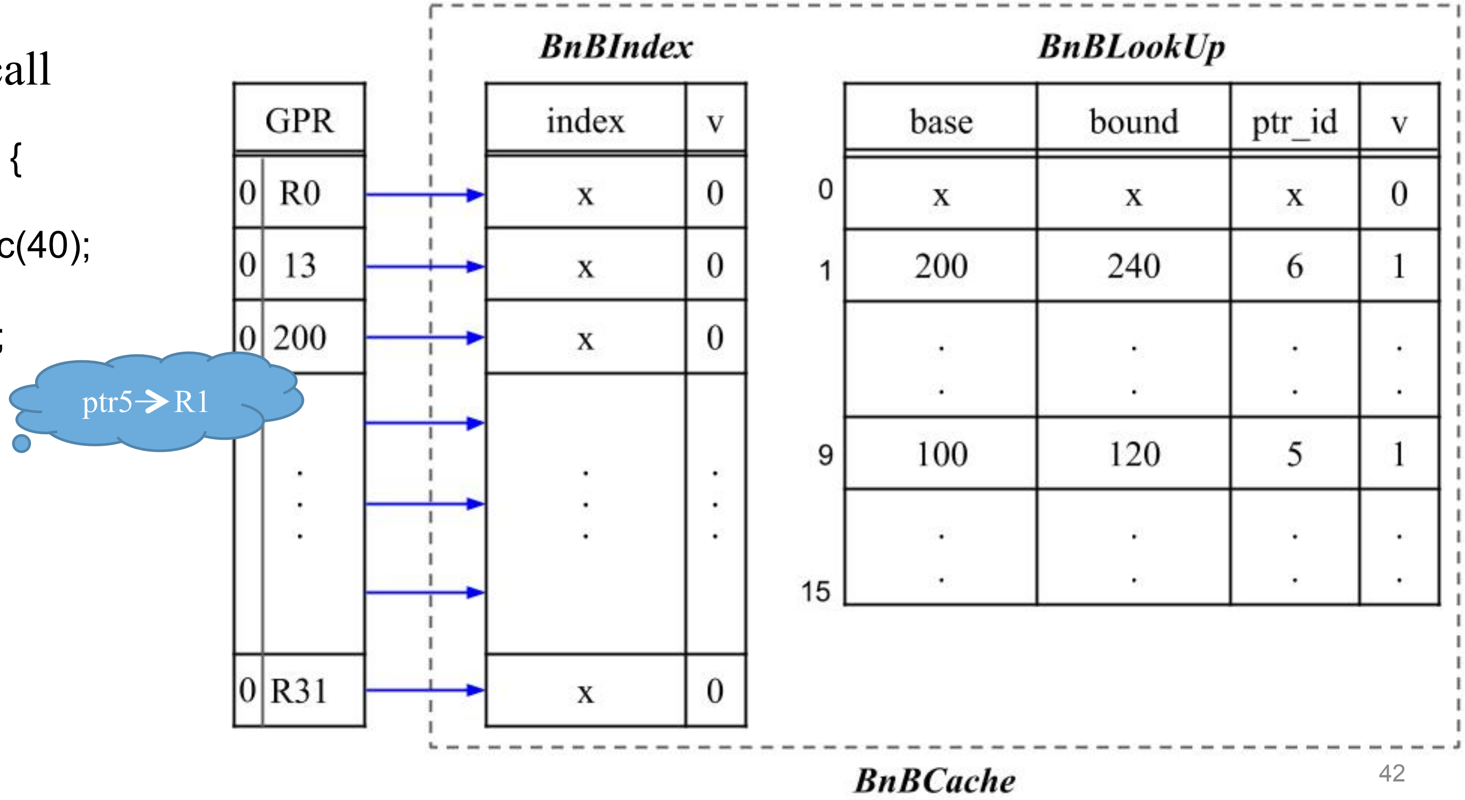
```
function bar( ) {
    char *ptr6;
    ptr6= malloc(40);
    …
    int c= 10+3;
    …
    free(ptr6);
    return;
}
```



Example programs

- A function call

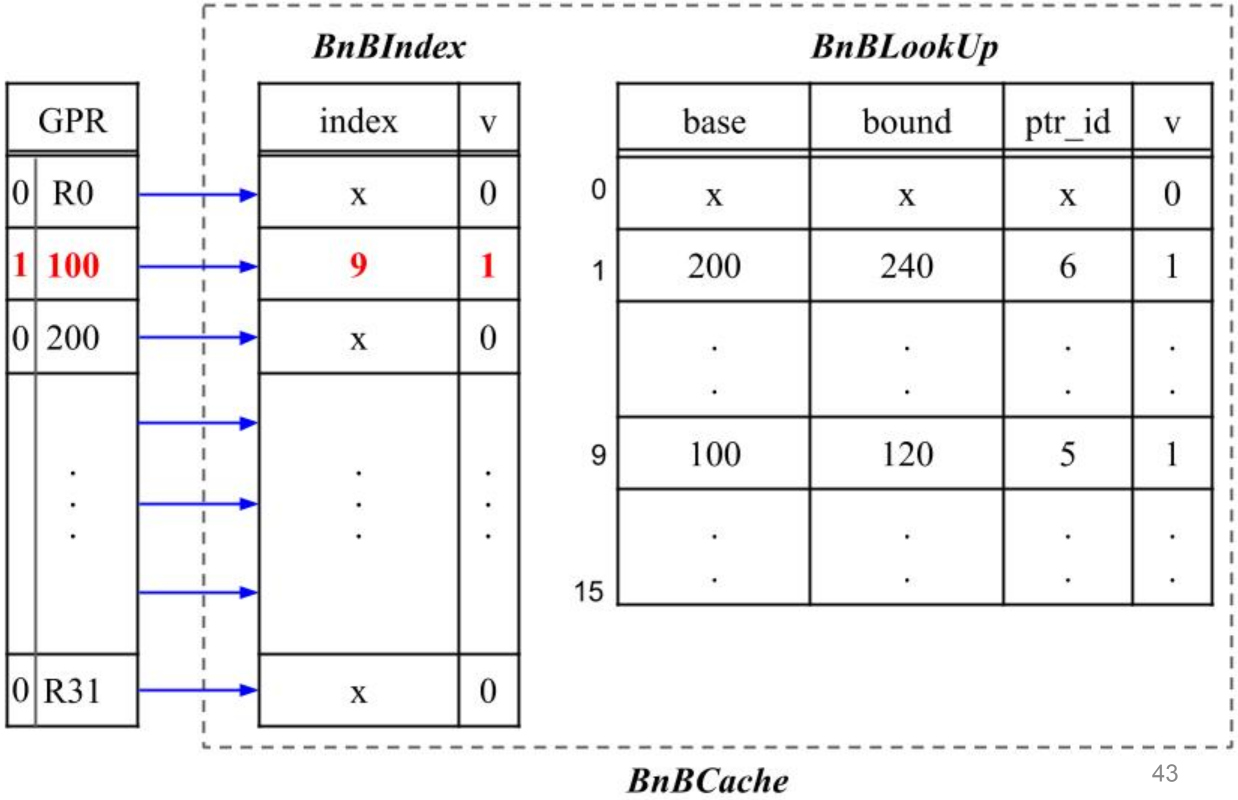
```
function bar( ) {
    char *ptr6;
    ptr6= malloc(40);
    …
    int c= 10+3;
    …
    free(ptr6);
    return;
}
```



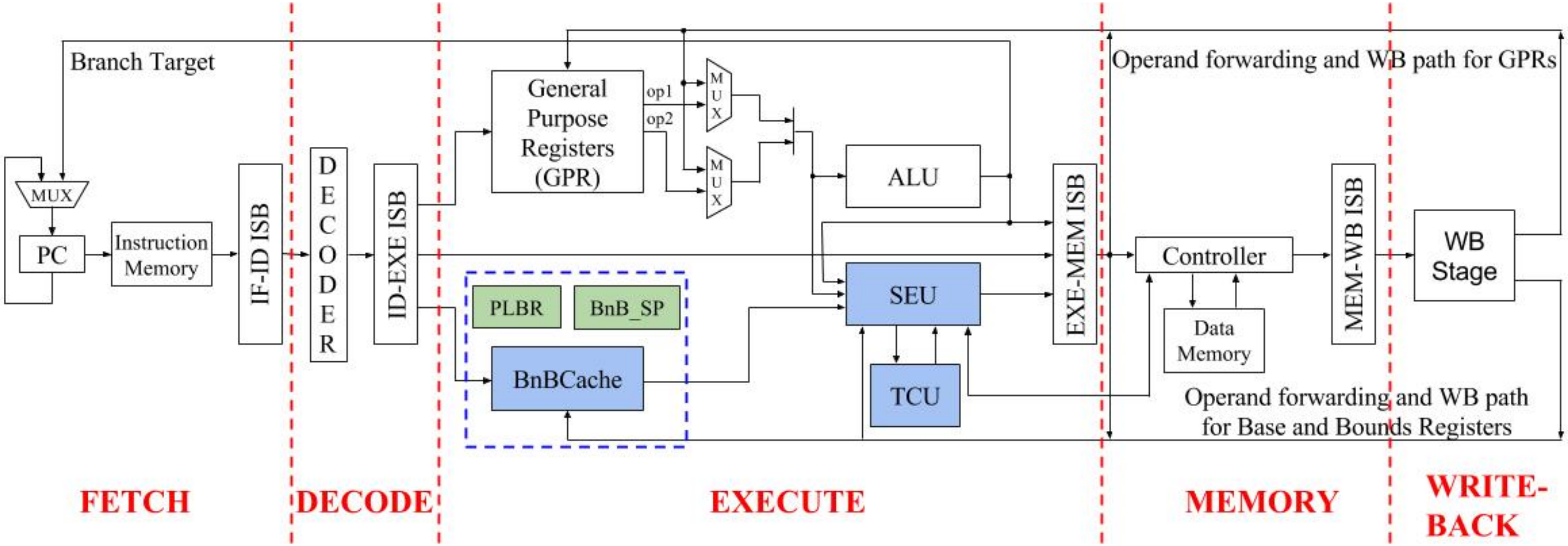
Example programs

- A function call

```
function foo( ) {
    char *ptr5;
    ptr5= malloc(20);
    …
    bar( );
    …
}
```



The pipeline



Comparison with existing solutions

	Safety checking	Instrumentation methodology	Metadata size for *n* aliased pointers	Memory fragmentation	Performance overhead (delay)
Intel MPX	Spatial	Compiler	128 x *n*	No	N/A
HardBound	Spatial	Hardware	128 x *n*	No	HW: N/A SW: 10%
Low-fat Pointer	Spatial	Hardware	0	Yes	HW: 5%
Watchdog	Spatial & Temporal	Compiler + Hardware	(256 x *n*) + 64	No	HW: N/A SW: 25%
WatchdogLite	Spatial & Temporal	Compiler	(256 x *n*) + 64	No	SW: 29%
Shakti-T	Spatial & Temporal	Hardware	(64 x *n*) + 128	No	HW: 1.5%*

References

[1] D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In Proceeding of the 28th International Conference on Software Engineering, May 2006.

[2] F. C. Eigler. Mudflap: Pointer Use Checking for C/C++. In GCC Developer's Summit, 2003.

[3] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Ar- chitecture: A Safe Execution Environment for Commodity Operating Systems. In Proceedings of the 21st ACM Symposium on Operating Systems Principles, Oct. 2007.

[4] Nagarakatte, Santosh, et al. "SoftBound: Highly compatible and complete spatial memory safety for C." *ACM Sigplan Notices* 44.6 (2009): 245-258. Link: http://cis.upenn.edu/acg/papers/pldi09_softbound.pdf

[5] Kwon, Albert, et al. "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security." *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013. Link: http://ic.ese.upenn.edu/pdf/fatptr_ccs2013.pdf

[6] Nagarakatte, Santosh, Milo MK Martin, and Steve Zdancewic. "Watchdog: Hardware for safe and secure manual memory management and full memory safety." *ACM SIGARCH Computer Architecture News*. Vol. 40. No. 3. IEEE Computer Society, 2012.
Link: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1740&context=cis_papers

[7] Menon, Arjun, et al. "Shakti-T: A RISC-V Processor with Light Weight Security Extensions." *Proceedings of the Hardware and Architectural Support for Security and Privacy*. ACM, 2017.

Backup slides

A 5-stage pipelined processor

Microarchitecture (Shakti-C)