Shakti-T : A RISC-V Processor with Light Weight Security Extensions

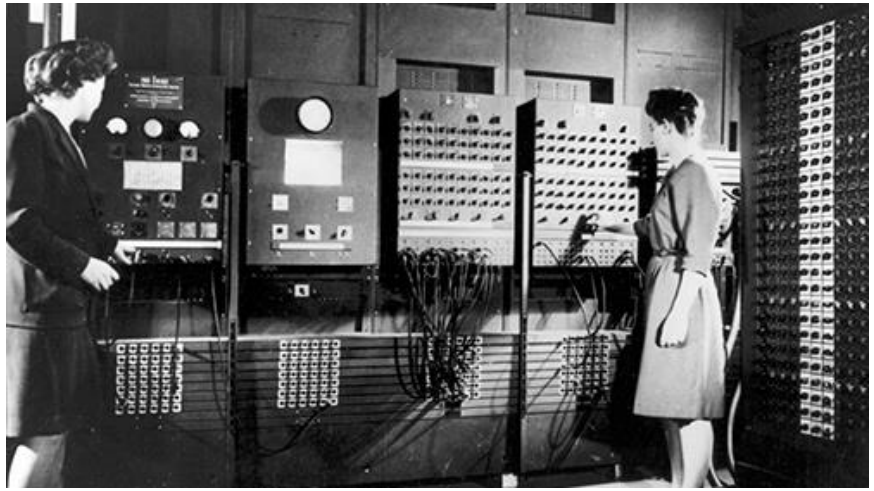
Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan

Department of Computer Science and Engineering
Indian Institute of Technology Madras, India

Why Security?

2000: Pentium 4

1946: ENIAC



Present

Memory-based attacks

- Spatial (Buffer overflow)
  - Stack Smashing
  - Return oriented programming
  - Format string
- Temporal
  - Use-after-free
  - Double-free

Existing Solutions

- Non-executable stack
- Stack Canaries
- Address Space Layout Randomization (ASLR)
- Control Flow Integrity
- Fat pointers

Fat Pointers

- Typical structure:

Value	Base	Bound

- Various implementations
  - SoftBounds (S/W) [Nagarakatte et al., PLDI 2009]
  - HardBound (H/W) [Devietti et al., ACM SIGARCH 2008]
  - Watchdog (H/W) [Nagarakatte et al., ISCA 2012]
  - WatchdogLite (S/W) [Nagarakatte et al., CGO 2014]

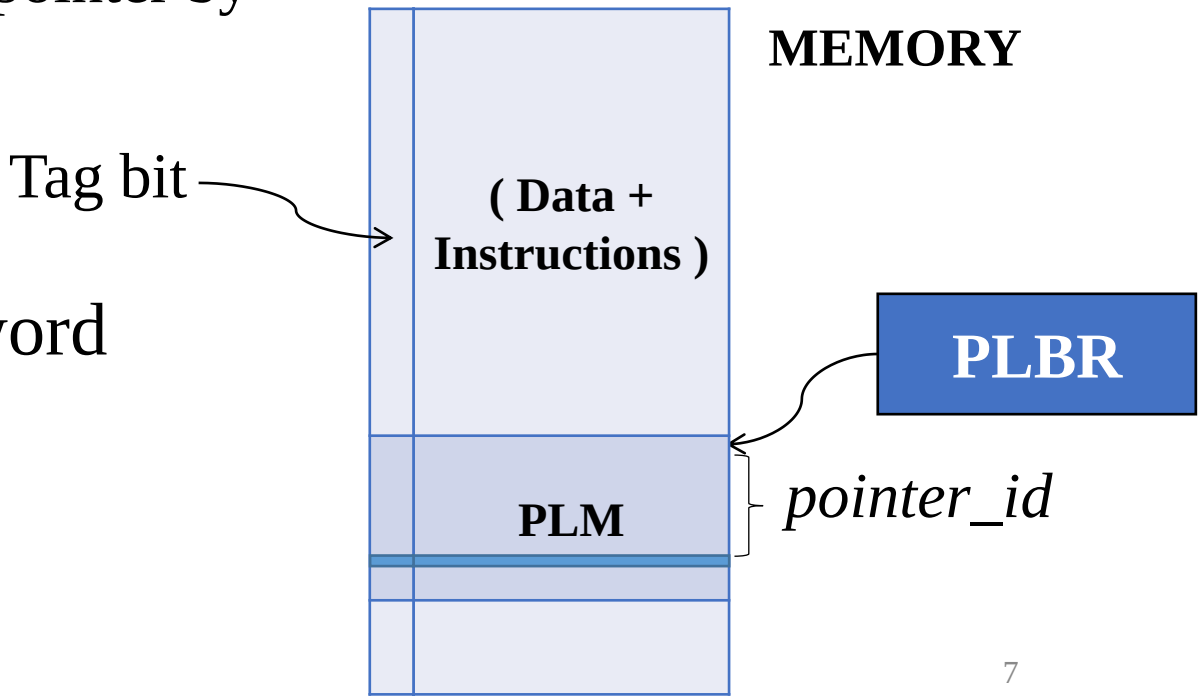
Existing Hardware Solutions

- One of the common design decision is to store the base and bound values (in shadow registers) in the register file alongside the value
- The decision 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

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*)

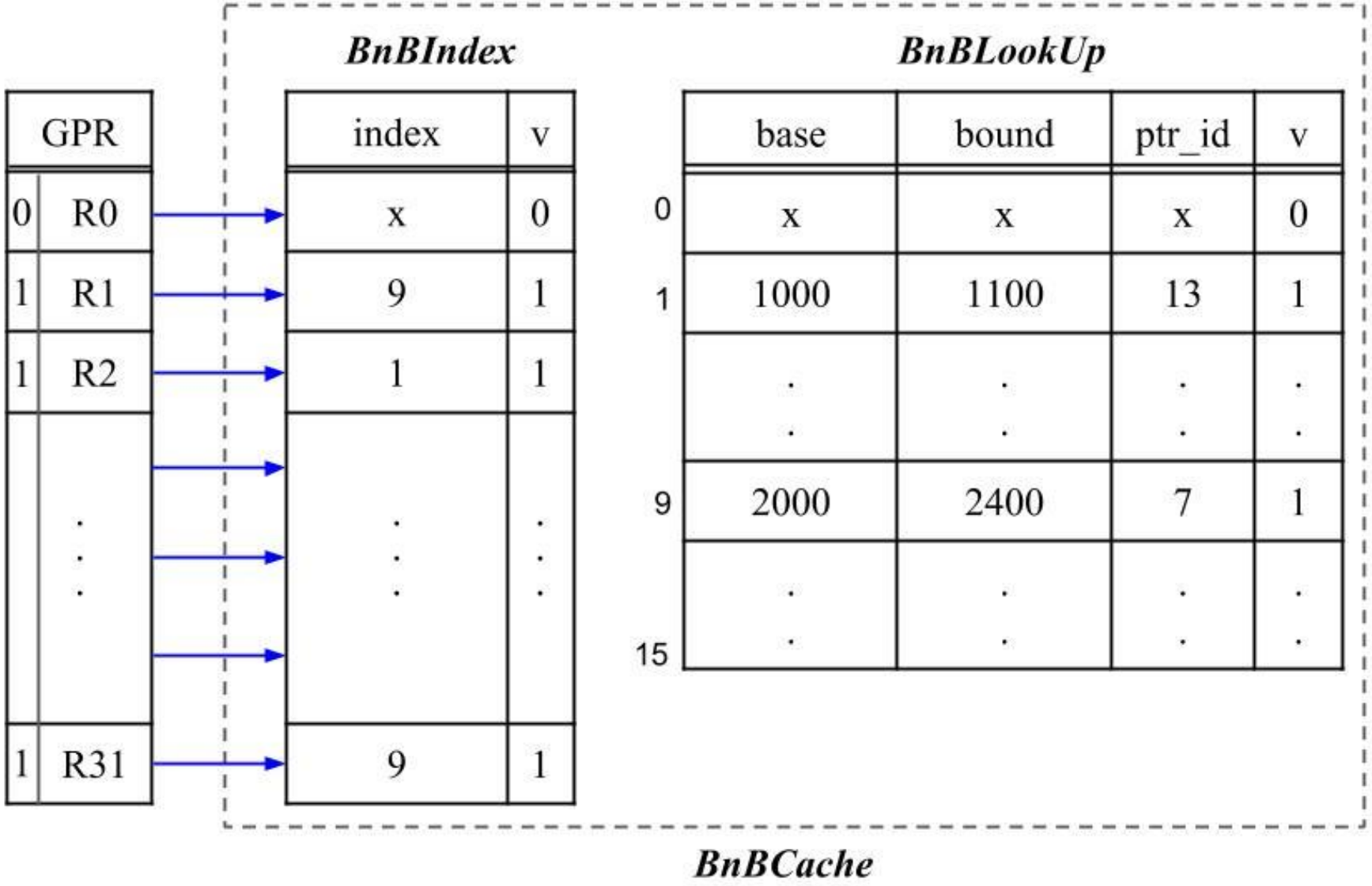
2. Add a 1-bit tag to every memory word
   - 0: Data/Instruction
   - 1: Pointer



Proposed solution (contd…)

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

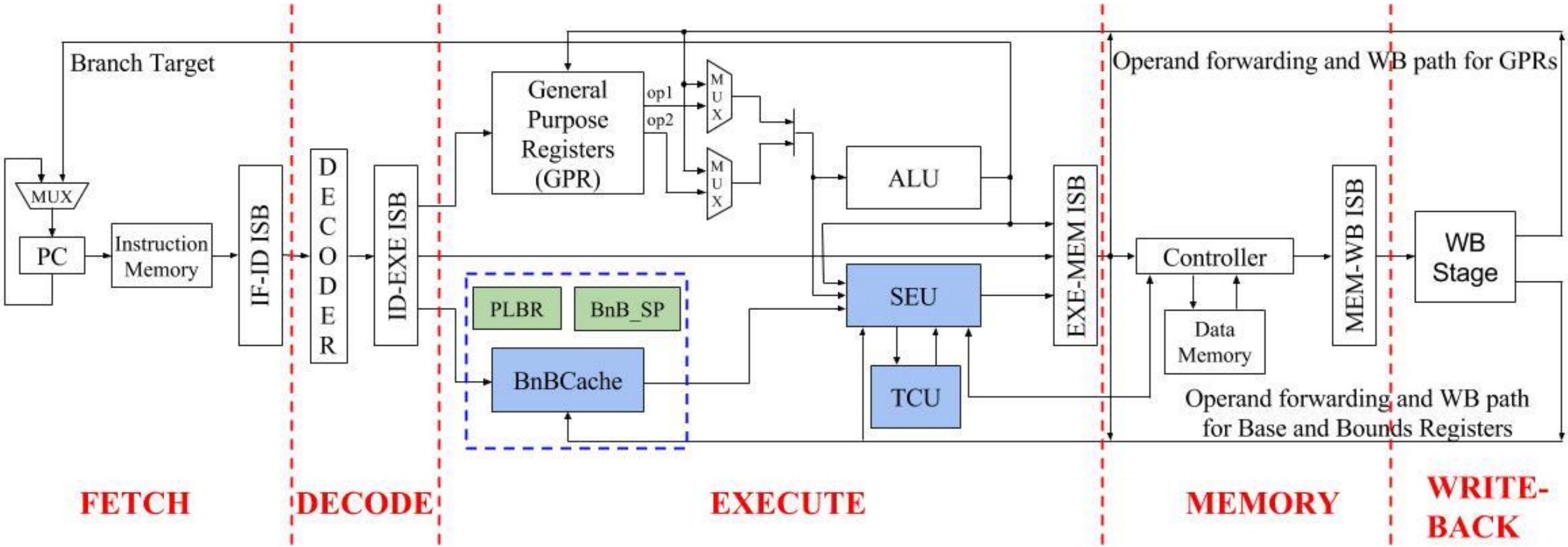
Proposed solution (contd…)



New Instructions

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

The pipeline



Example programs

- Accessing an array
  1. The value of base and bounds is stored in the PLM (using the *wrplm* instruction) when an array is declared
  2. When an array is accessed and the base address is loaded to a GPR, *ldbnb* instruction is also issued to load the base and bounds to the BnBCache

```
char a[10];
char c= a[4];
```

Example programs

- Dynamic memory allocation

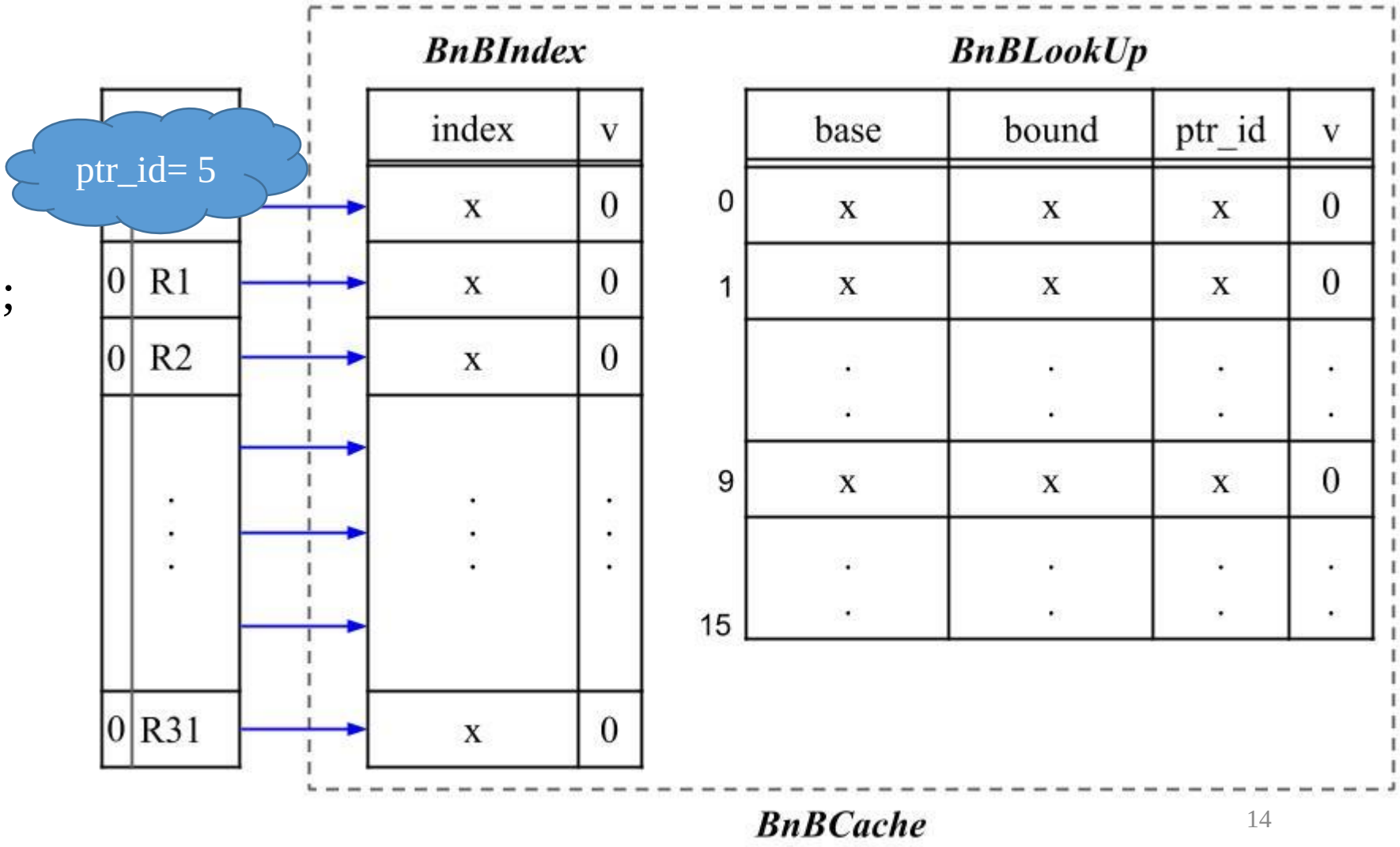
char **ptr* = malloc(n);

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

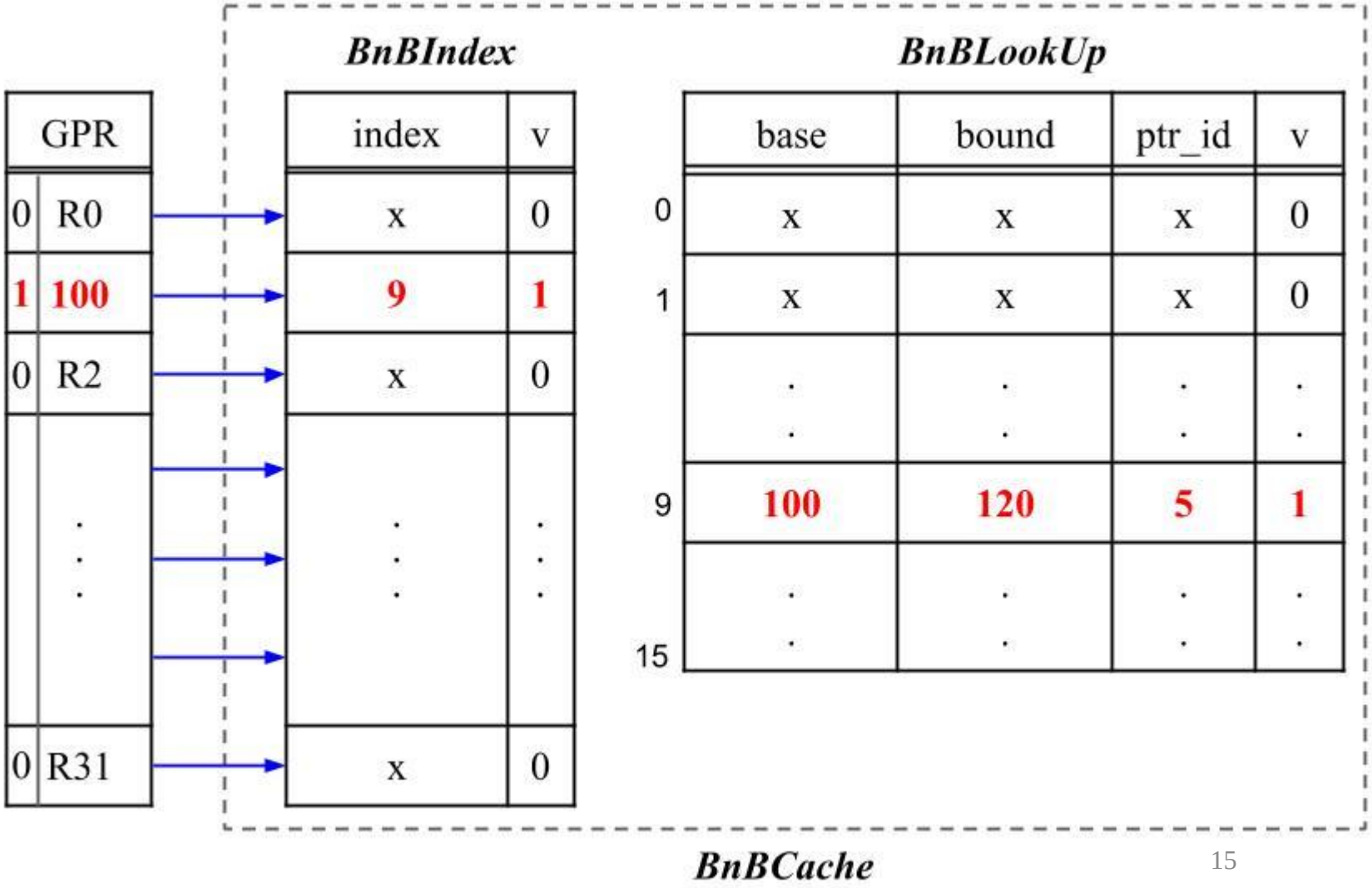
```
function foo( ) {
    char *ptr5;
    ptr5= malloc(20);
    …
    bar( );
    …
}
```



Example programs

- A function call

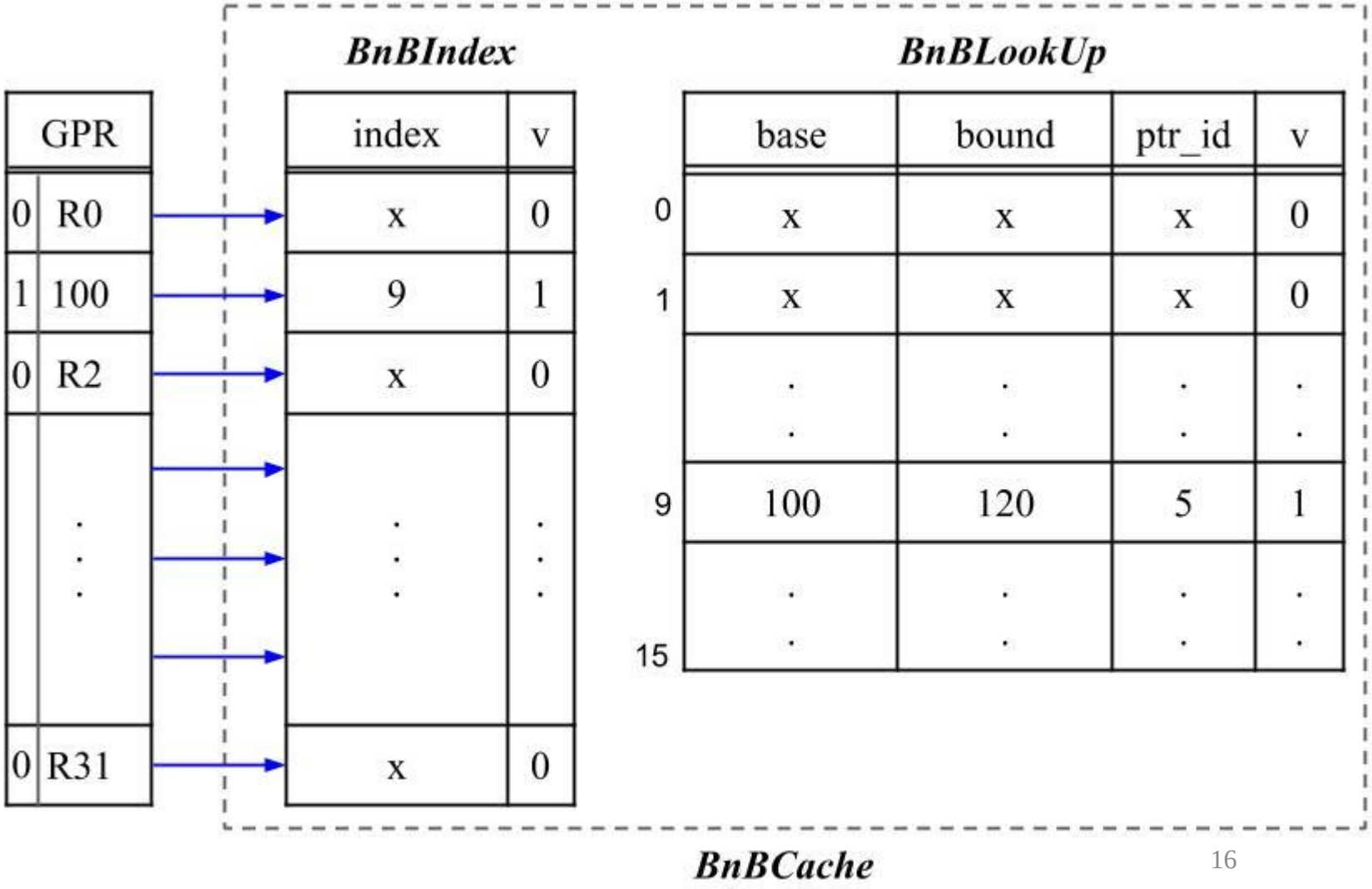
```
function foo( ) {
    char *ptr5;
    ptr5= malloc(20);
    …
    bar( );
    …
}
```



Example programs

- A function call

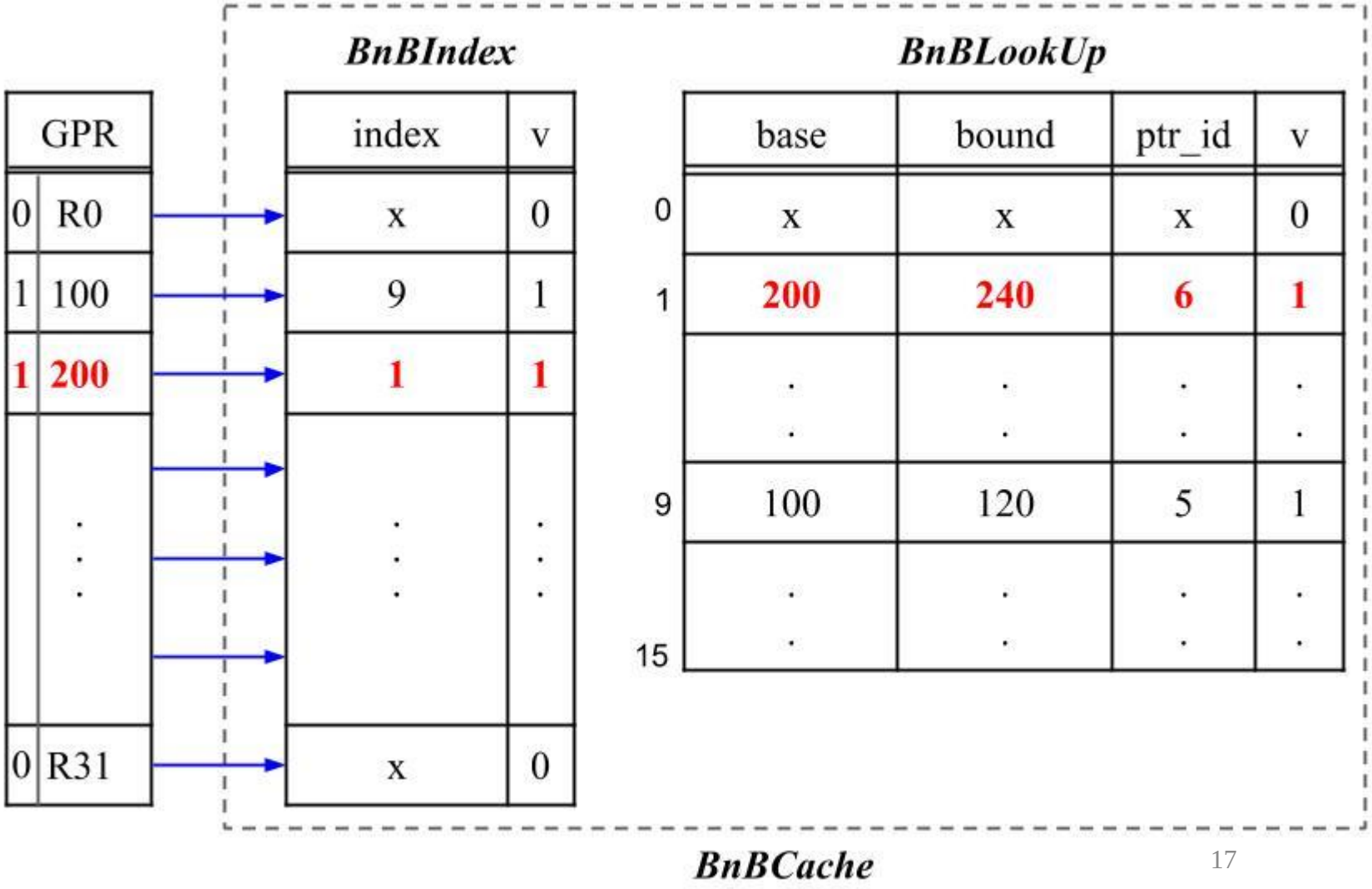
```
function foo( ) {
    char *ptr5;
    ptr5= malloc(20);
    …
    bar( );
    …
}
```



Example programs

- A function call

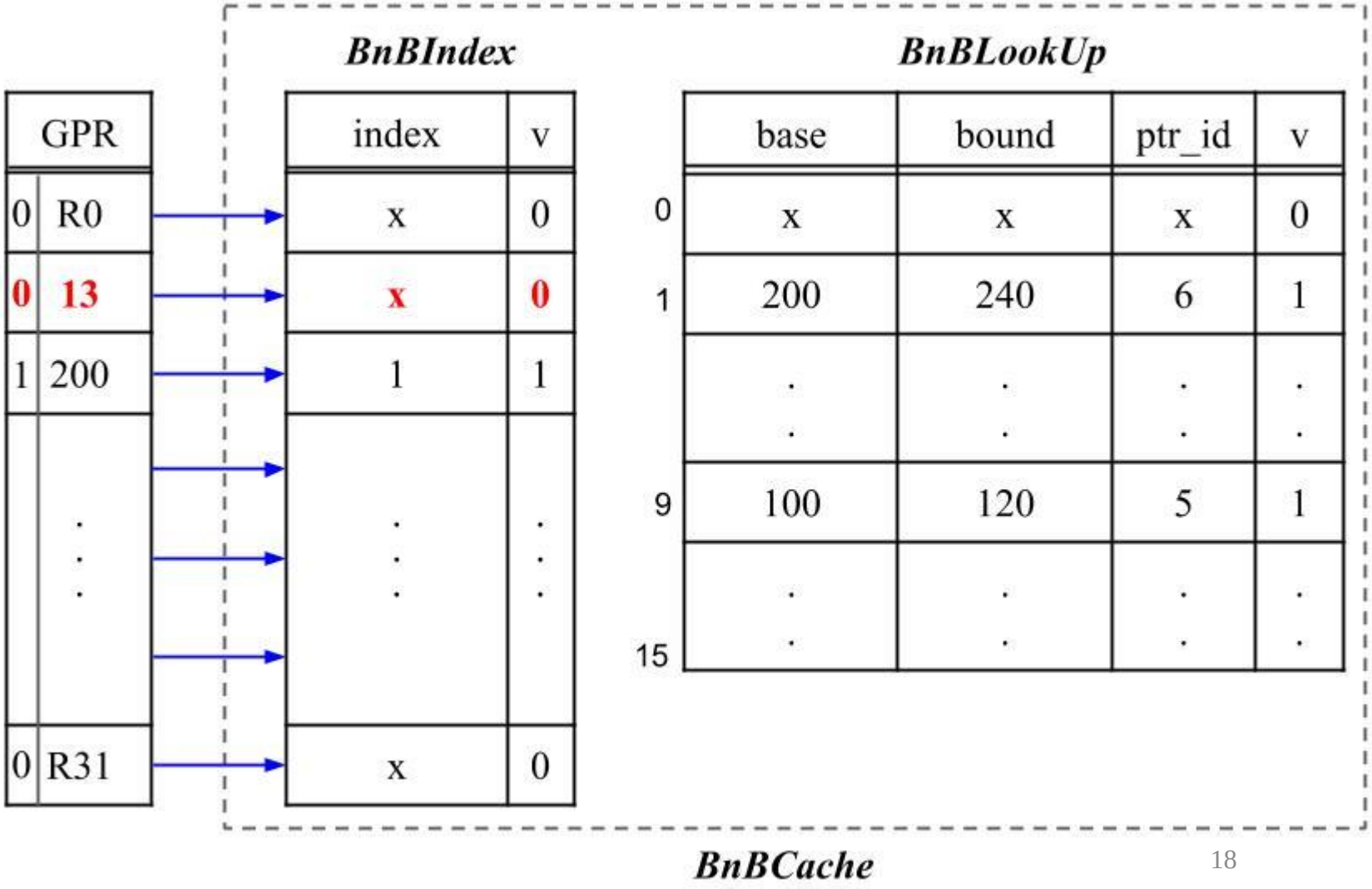
```
function bar( ) {
    char *ptr6;
    ptr6= malloc(40);
    …
    int c= 4+5;
    …
    free(ptr6);
    return;
}
```



Example programs

- A function call

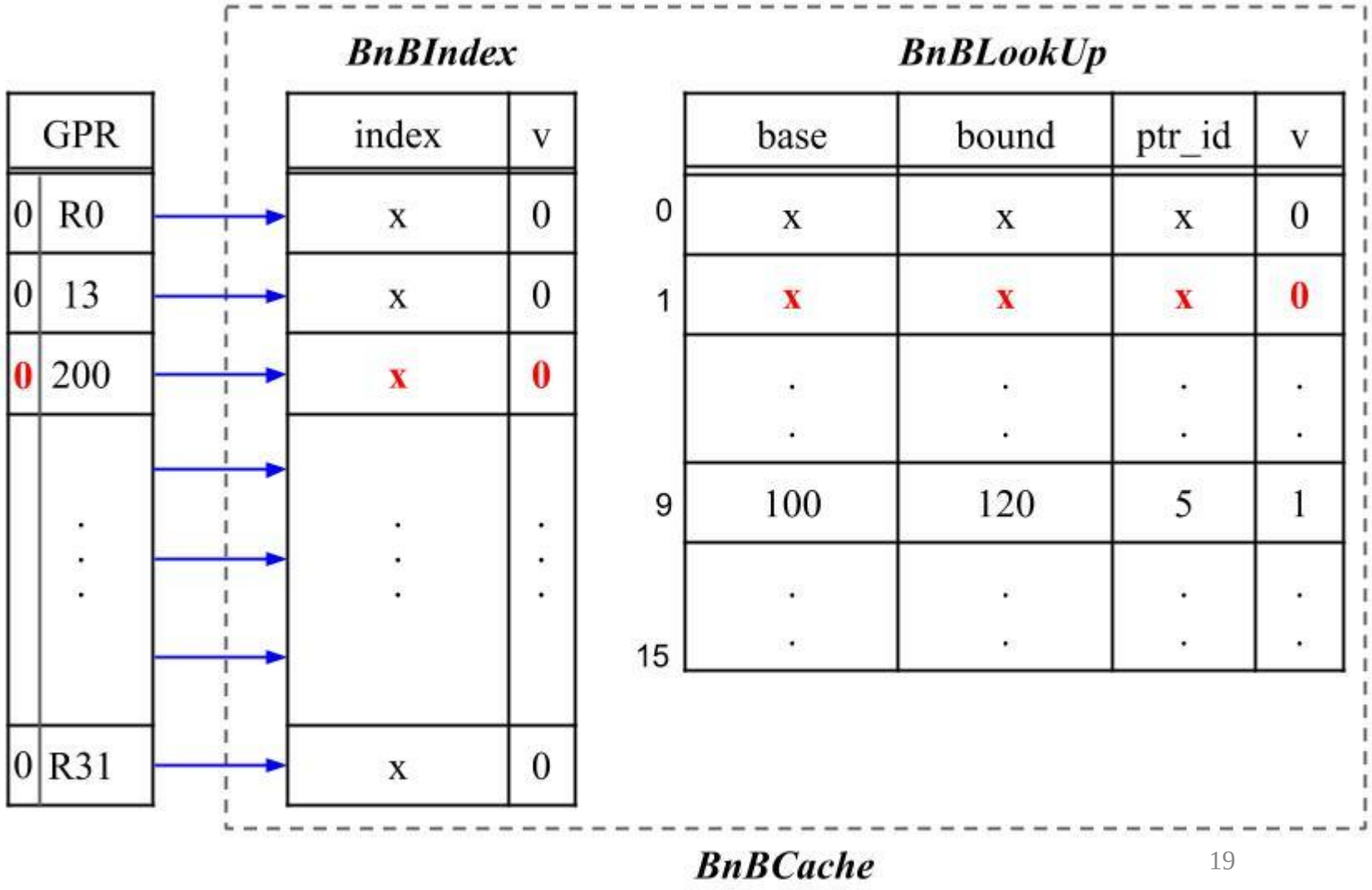
```
function bar( ) {
    char *ptr6;
    ptr6= malloc(40);
    …
    int c= 10+3;
    …
    free(ptr6);
    return;
}
```



Example programs

- A function call

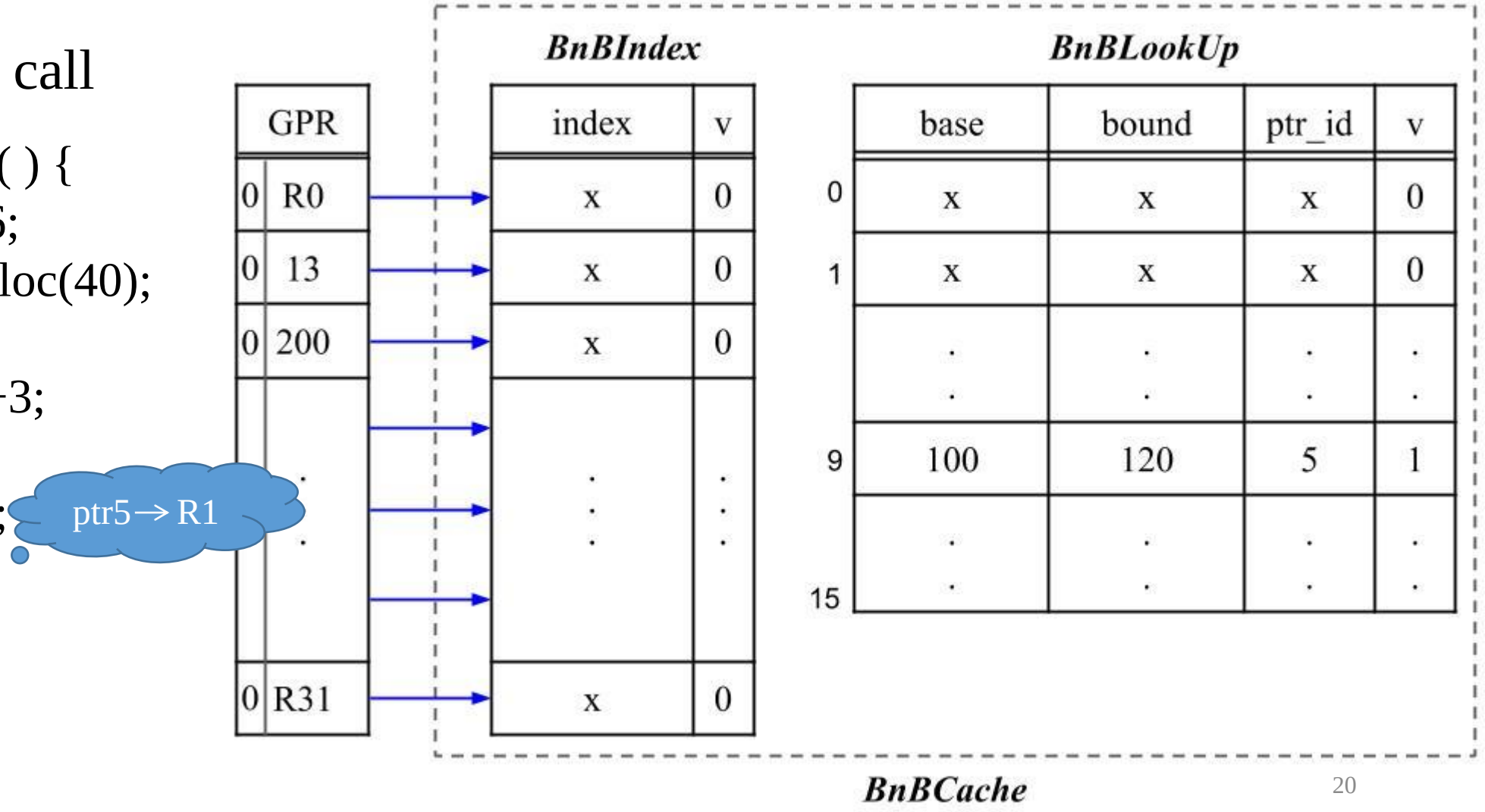
```
function bar( ) {
    char *ptr6;
    ptr6= malloc(40);
    …
    int c= 10+3;
    …
    free(ptr6);
    return;
}
```



Example programs

- A function call

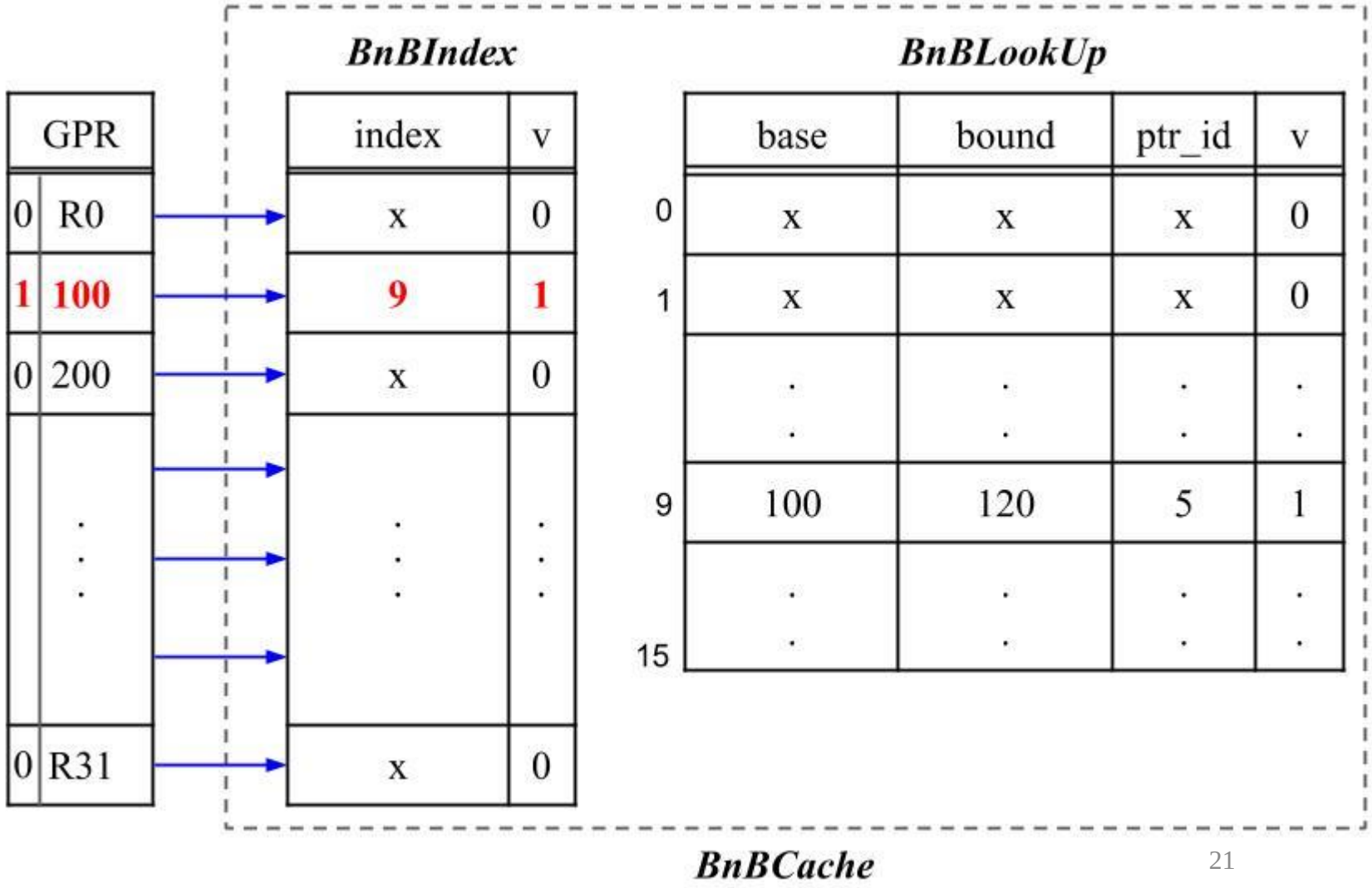
```
function bar( ) {
    char *ptr6;
    ptr6= malloc(40);
    …
    int c= 10+3;
    …
    free(ptr6);
    return;
}
```



Example programs

- A function call

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



Comparison with existing solutions

	Safety checking	Instrumentation methodology	Metadata size for *n* aliased pointers	Memory fragmentation	Performance overhead (delay)
Intel MPX [1]	Spatial	Compiler	128 x *n*	No	N/A
HardBound [2]	Spatial	Hardware	128 x *n*	No	HW: N/A SW: 10%
Low-fat Pointer [3]	Spatial	Hardware	0	Yes	HW: 5%
Watchdog [4]	Spatial & Temporal	Compiler + Hardware	(256 x *n*) + 64	No	HW: N/A SW: 25%
WatchdogLite [5]	Spatial & Temporal	Compiler	(256 x *n*) + 64	No	SW: 29%
Shakti-T	Spatial & Temporal	Hardware	(64 x *n*) + 128	No	HW: 1.5%[+]

Conclusion

- Shakti-T uses the concept of fat pointers to eliminate spatial and temporal memory attacks.
- It uses a common memory region to store the base and bounds.
- The base and bounds are cached at the register level using a dedicated register file, and are accessed using a one-level indexing.
- The additional computations are done in parallel with the ALU's computation and thus, it does not affect the clock period.

Future Work

- Incorporating the necessary changes in the compiler and measuring the actual increase in program execution time by running the modified code on the actual hardware.
- Extending the tagged architecture to enforce fine-grained access control and information flow control.


RISE
Reconfigurable and Intelligent Systems Engineering


INDIAN INSTITUTE OF TECHNOLOGY MADRAS
सिद्धिर्भवति कर्मजा

Thank You!

References

[1] Intel Corporation, “Intel MPX Explained.” https://intel-mpx.github.io/design/

[2] Devietti, Joe, et al. "Hardbound: architectural support for spatial safety of the C programming language." *ACM SIGARCH Computer Architecture News*. Vol. 36. No. 1. ACM, 2008.

[3] 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.

[4] Nagarakatte, Santosh, et al. “Watchdog: Hardware for safe and secure manual memory management and full memory safety.”, *ISCA 2012*.

[5] Nagarakatte, Santosh, et al. "Watchdoglite: Hardware-accelerated compiler-based pointer checking." *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014.