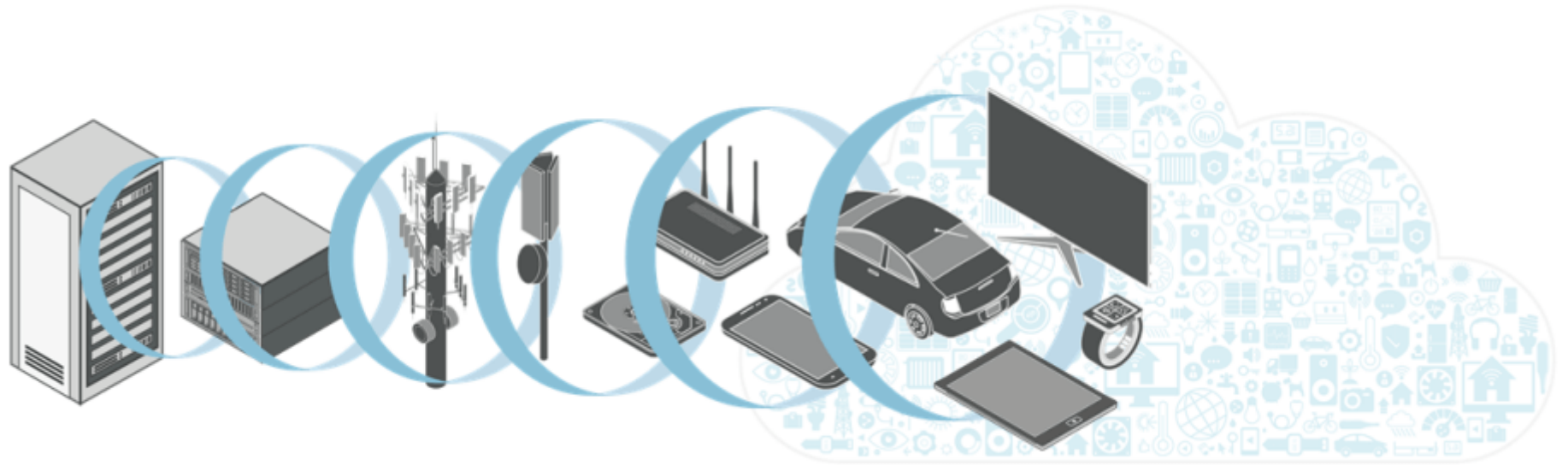


Optimization Goals



Performance ← Primary Goal → Code Size

Motivating Example

```
void quantum_cond_phase(
int control, int target, quantum_reg *reg){
    int i;
    COMPLEX_FLOAT z;
    if(quantum_objcode_put(COND_PHASE, control, target))
        return;
    z = quantum_cexp(pi / (1 << (control - target)));
    for(i=0; i<reg->size; i++) {
        if(reg->node[i].state & (1 << control)) {
            if(reg->node[i].state & (1 << target))
                reg->node[i].amplitude *= z;
        }
    }
    quantum_decohere(reg);
}
```

```
void quantum_cond_phase_inv(
int control, int target, quantum_reg *reg){
    int i;
    COMPLEX_FLOAT z;

    z = quantum_cexp(-pi / (1 << (control - target)));
    for(i=0; i<reg->size; i++) {
        if(reg->node[i].state & (1 << control)) {
            if(reg->node[i].state & (1 << target))
                reg->node[i].amplitude *= z;
        }
    }
    quantum_decohere(reg);
}
```

Motivating Example

```
void quantum_cond_phase(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
    if(quantum_objcode_put(COND_PHASE, control, target))  
        return;  
    z = quantum_cexp(pi / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

Match

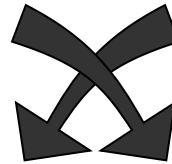
```
void quantum_cond_phase_inv(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
  
    z = quantum_cexp(-pi / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

Motivating Example

```
void quantum_cond_phase(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
    if(quantum_objcode_put(COND_PHASE, control, target))  
        return;  
    z = quantum_cexp(pi / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

```
void quantum_cond_phase_inv(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
  
    z = quantum_cexp(-pi / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

Match



```
void merged(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;
```

```
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

Motivating Example

```
void quantum_cond_phase(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
    if(quantum_objcode_put(COND_PHASE, control, target))  
        return;  
    z = quantum_cexp(pi / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

```
void quantum_cond_phase_inv(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
  
    z = quantum_cexp(-pi / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

Mismatch



```
void merged(bool func_id,  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
    if(func_id)  
        if(quantum_objcode_put(COND_PHASE, control, target))  
            return;  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

Motivating Example

```
void quantum_cond_phase(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
    if(quantum_objcode_put(COND_PHASE, control, target))  
        return;  
    z = quantum_cexp(pi / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

```
void quantum_cond_phase_inv(  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
  
    z = quantum_cexp(-pi / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

Mismatch



```
void merged(bool func_id,  
int control, int target, quantum_reg *reg){  
    int i;  
    COMPLEX_FLOAT z;  
    if(func_id)  
        if(quantum_objcode_put(COND_PHASE, control, target))  
            return;  
    float var = (func_id)?pi:(-pi);  
    z = quantum_cexp(var / (1 << (control - target)));  
    for(i=0; i<reg->size; i++) {  
        if(reg->node[i].state & (1 << control)) {  
            if(reg->node[i].state & (1 << target))  
                reg->node[i].amplitude *= z;  
        }  
    }  
    quantum_decohere(reg);  
}
```

Motivating Example

```
void quantum_cond_phase(
int control, int target, quantum_reg *reg){
    int i;
    COMPLEX_FLOAT z;
    if(quantum_objcode_put(COND_PHASE, control, target))
        return;
    z = quantum_cexp(pi / (1 << (control - target)));
    for(i=0; i<reg->size; i++) {
        if(reg->node[i].state & (1 << control)) {
            if(reg->node[i].state & (1 << target))
                reg->node[i].amplitude *= z;
        }
    }
    quantum_decohere(reg);
}
```

```
void quantum_cond_phase_inv(
int control, int target, quantum_reg *reg){
    int i;
    COMPLEX_FLOAT z;

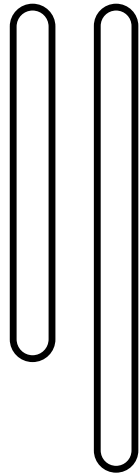
    z = quantum_cexp(-pi / (1 << (control - target)));
    for(i=0; i<reg->size; i++) {
        if(reg->node[i].state & (1 << control)) {
            if(reg->node[i].state & (1 << target))
                reg->node[i].amplitude *= z;
        }
    }
    quantum_decohere(reg);
}
```



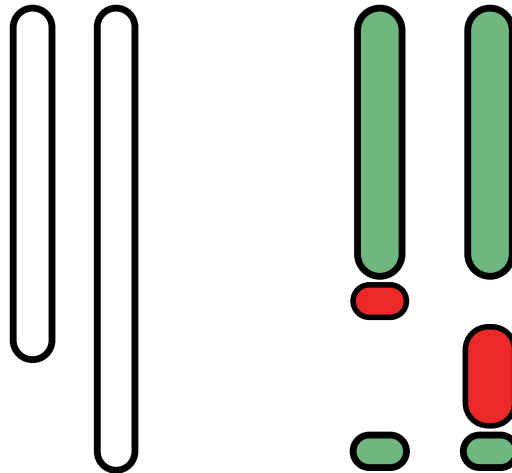
```
void merged(bool func_id,
int control, int target, quantum_reg *reg){
    int i;
    COMPLEX_FLOAT z;
    if(func_id)
        if(quantum_objcode_put(COND_PHASE, control, target))
            return;
    float var = (func_id)?pi:(-pi);
    z = quantum_cexp(var / (1 << (control - target)));
    for(i=0; i<reg->size; i++) {
        if(reg->node[i].state & (1 << control)) {
            if(reg->node[i].state & (1 << target))
                reg->node[i].amplitude *= z;
        }
    }
    quantum_decohere(reg);
}
```

Key Insight

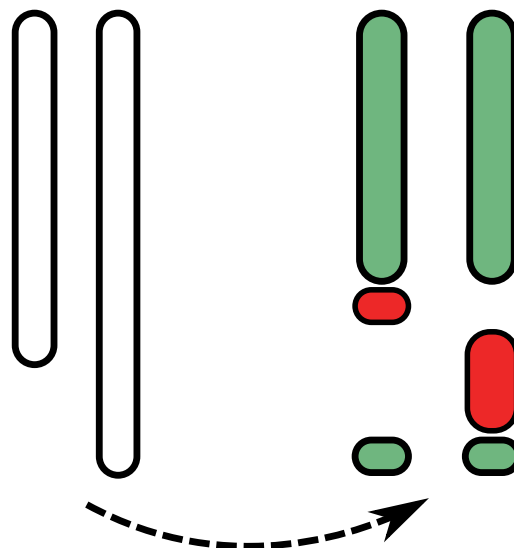
Key Insight



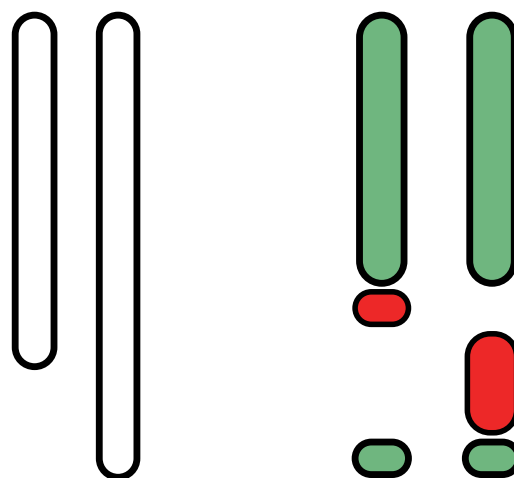
Key Insight



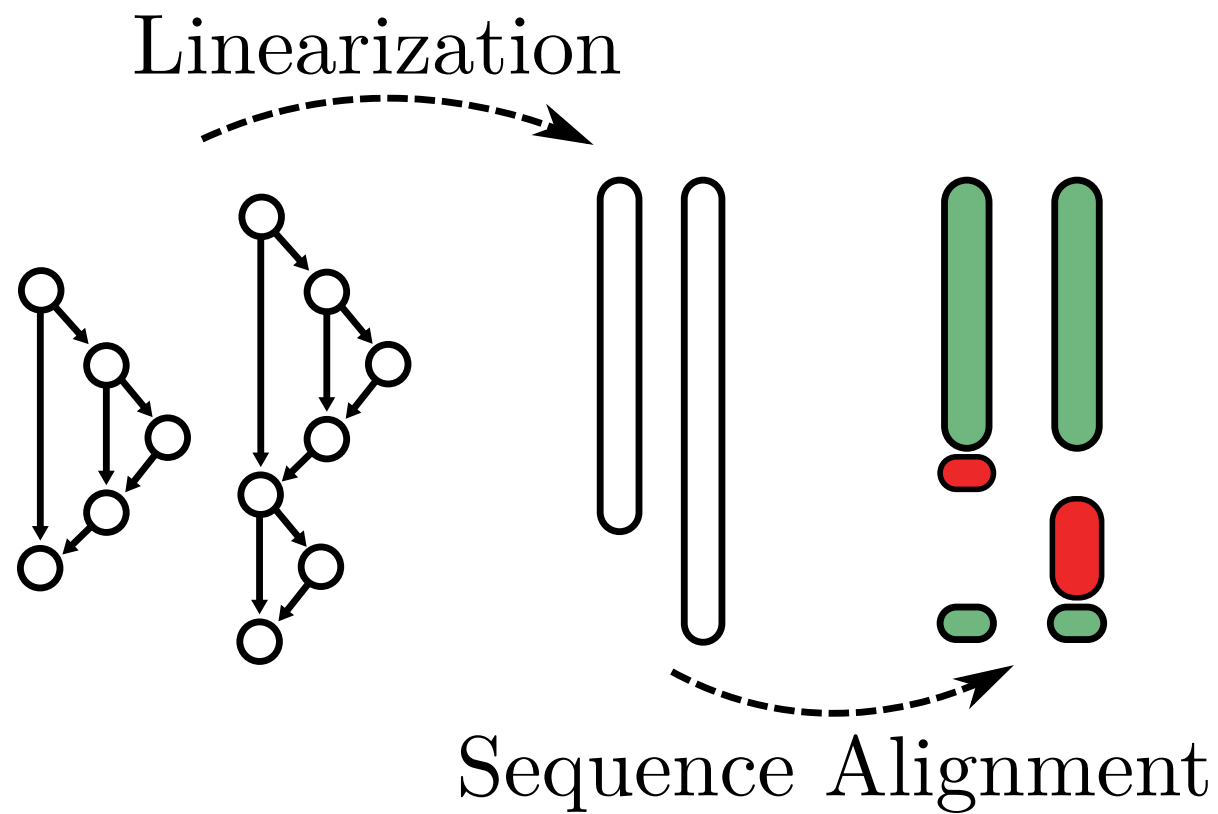
Key Insight

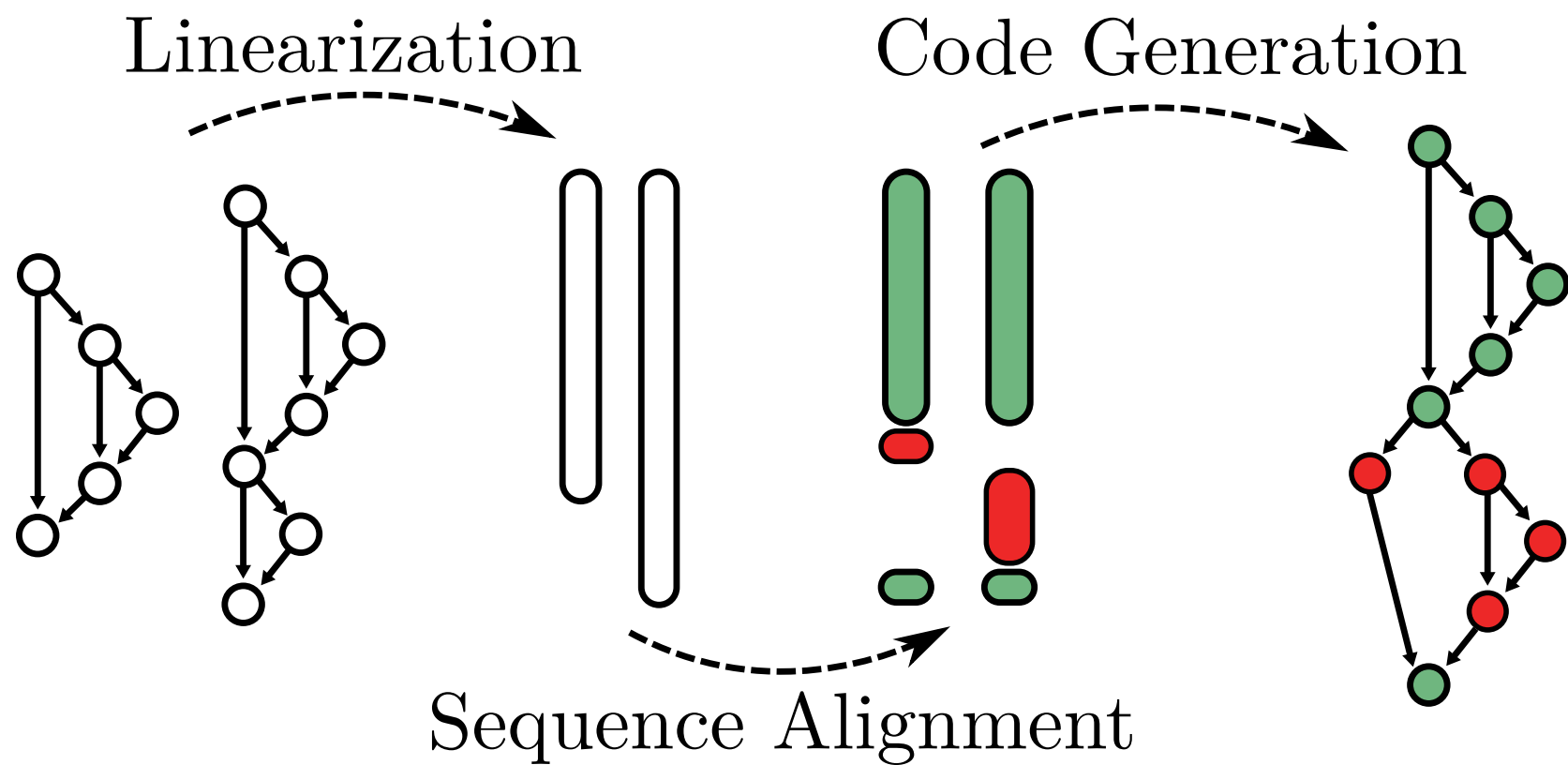


Sequence Alignment

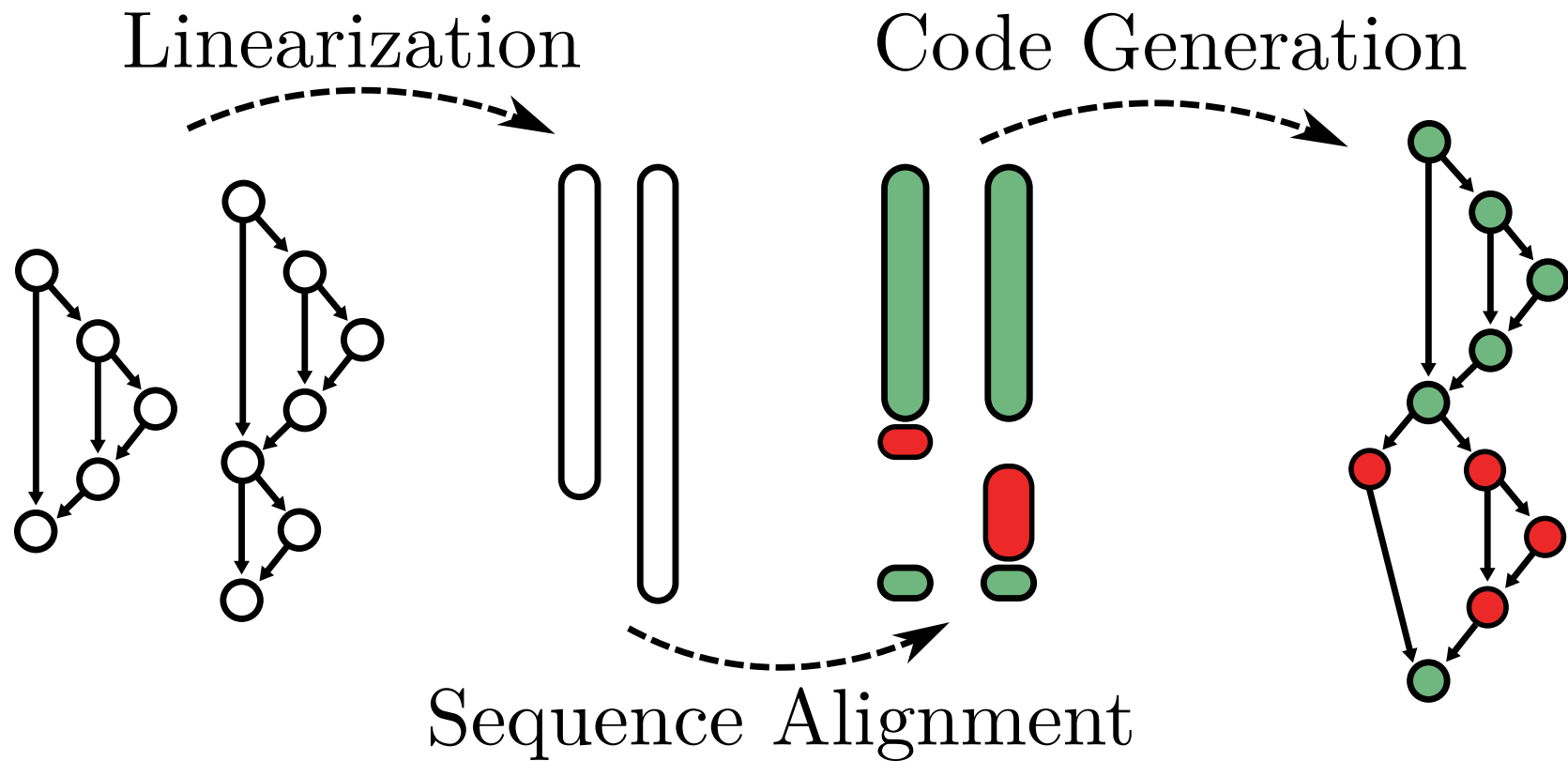


Sequence Alignment





Our Technique



Our Technique

Our Technique

```
double ExponentialRandom(void) {  
    double x;  
    do {  
        x = sre_random();  
    } while (x == 0.0);  
    return -log(x);  
}
```

Our Technique

```
double ExponentialRandom(void) {  
    double x;  
    do {  
        x = sre_random();  
    } while (x == 0.0);  
    return -log(x);  
}
```

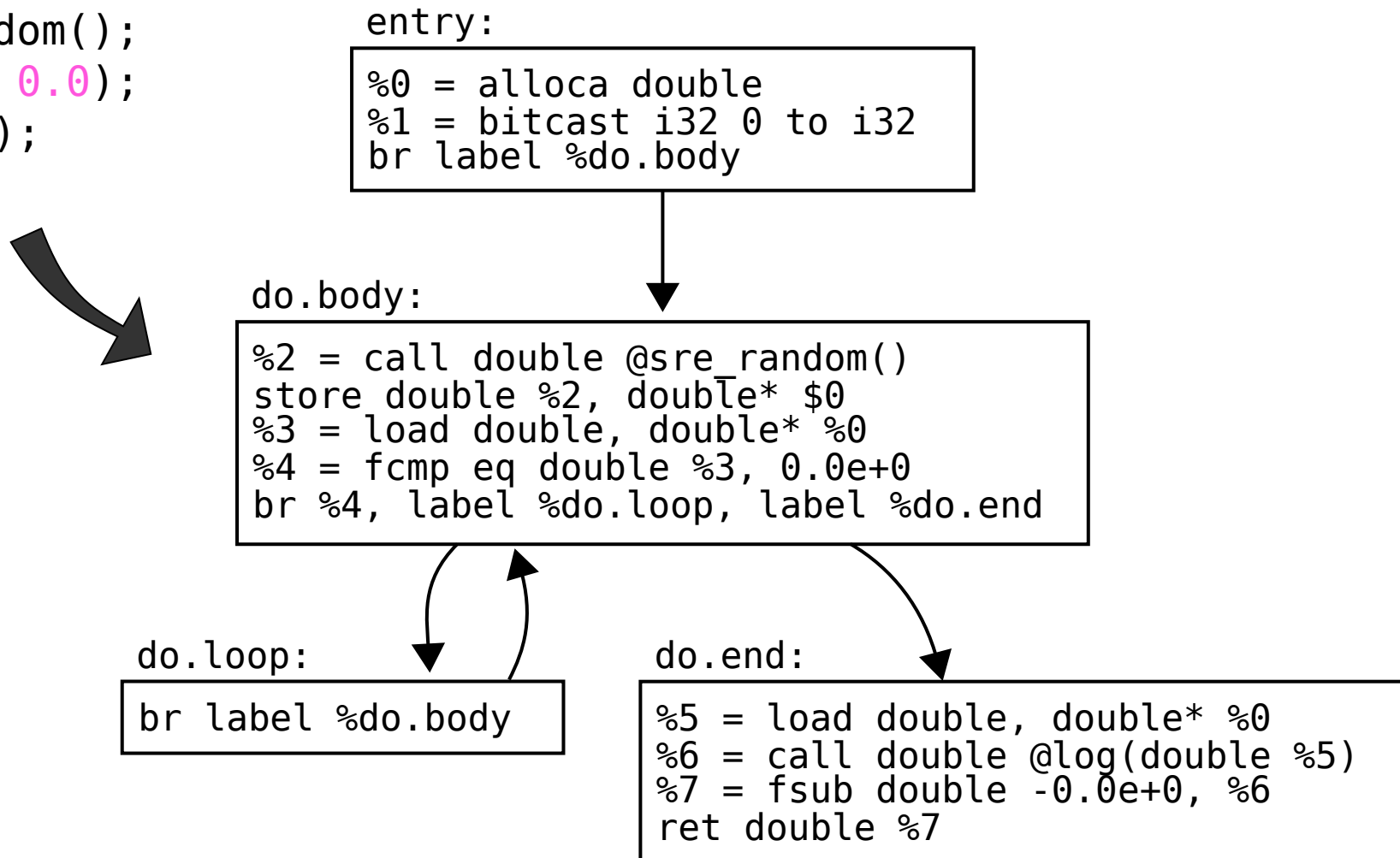
```
double sre_random_positive(void) {  
    double x;  
    do {  
        x = sre_random();  
    } while (x == 0.0);  
    return x;  
}
```

Our Technique

```
double ExponentialRandom(void) {  
    double x;  
    do {  
        x = sre_random();  
    } while (x == 0.0);  
    return -log(x);  
}
```

Our Technique

```
double ExponentialRandom(void) {  
    double x;  
    do {  
        x = sre_random();  
    } while (x == 0.0);  
    return -log(x);  
}
```



Our Technique

entry:

```
%0 = alloca double  
%1 = bitcast i32 0 to i32  
br label %do.body
```

do.body:

```
%2 = call double @sre_random()  
store double %2, double* %0  
%3 = load double, double* %0  
%4 = fcmp eq double %3, 0.0e+0  
br %4, label %do.loop, label %do.end
```

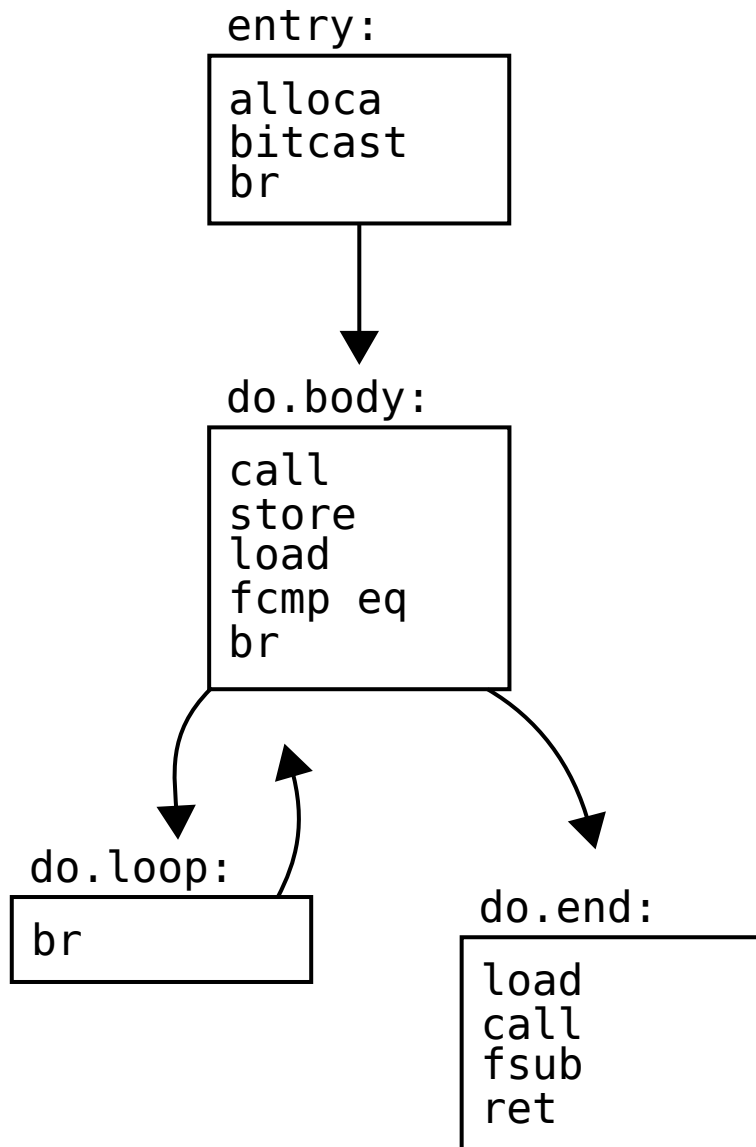
do.loop:

```
br label %do.body
```

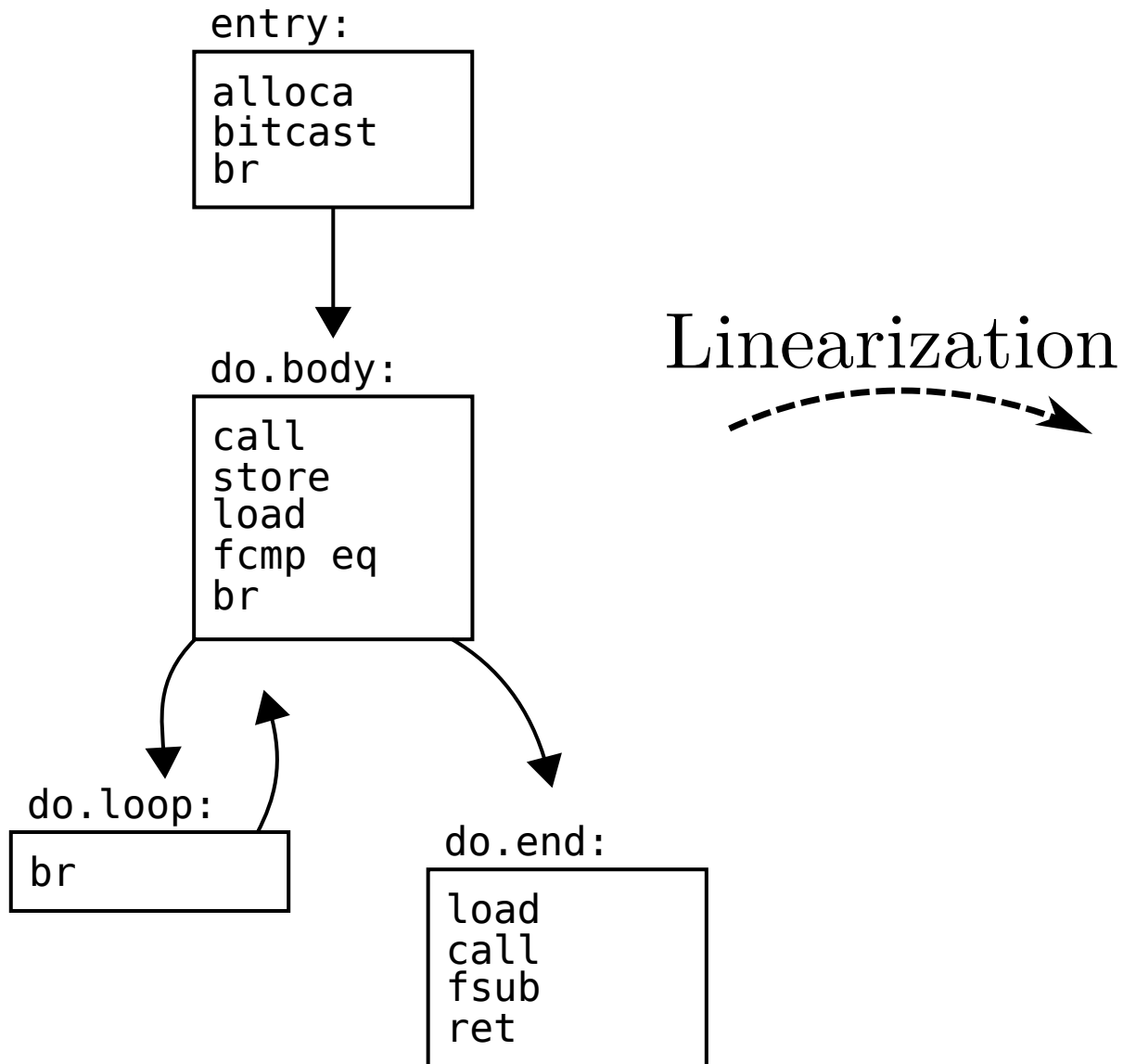
do.end:

```
%5 = load double, double* %0  
%6 = call double @log(double %5)  
%7 = fsub double -0.0e+0, %6  
ret double %7
```

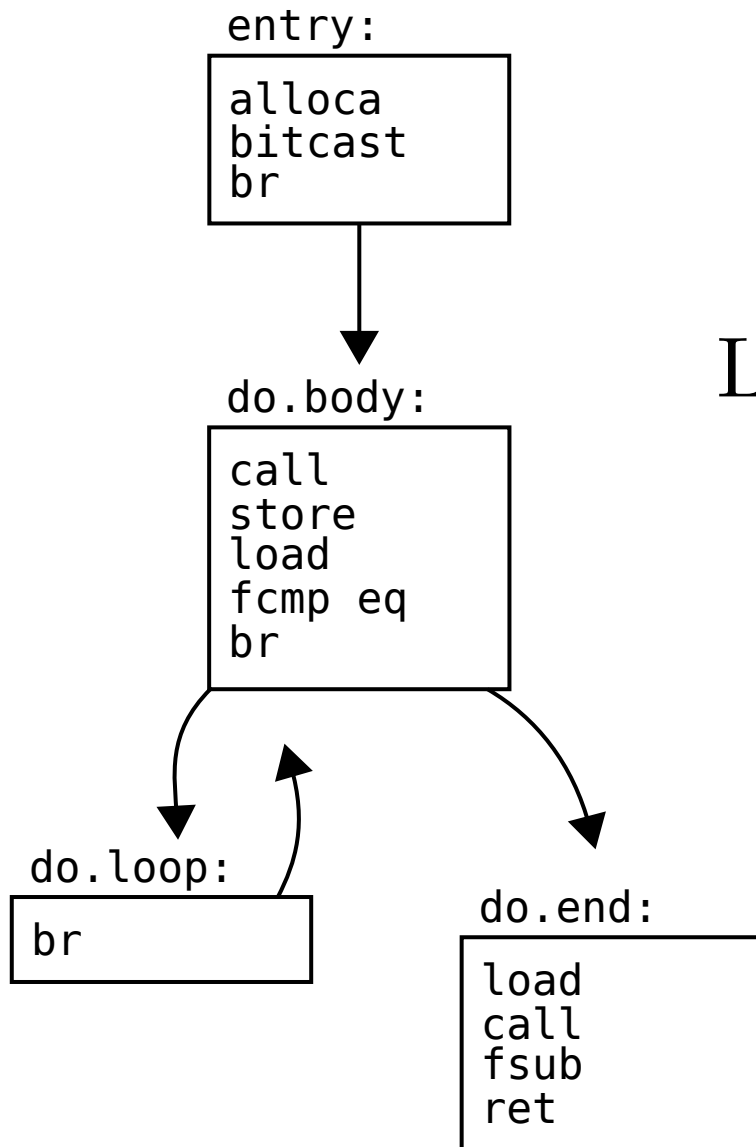
Our Technique



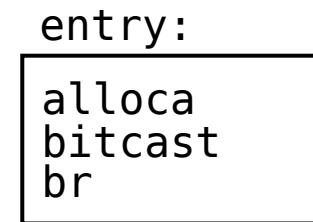
Our Technique



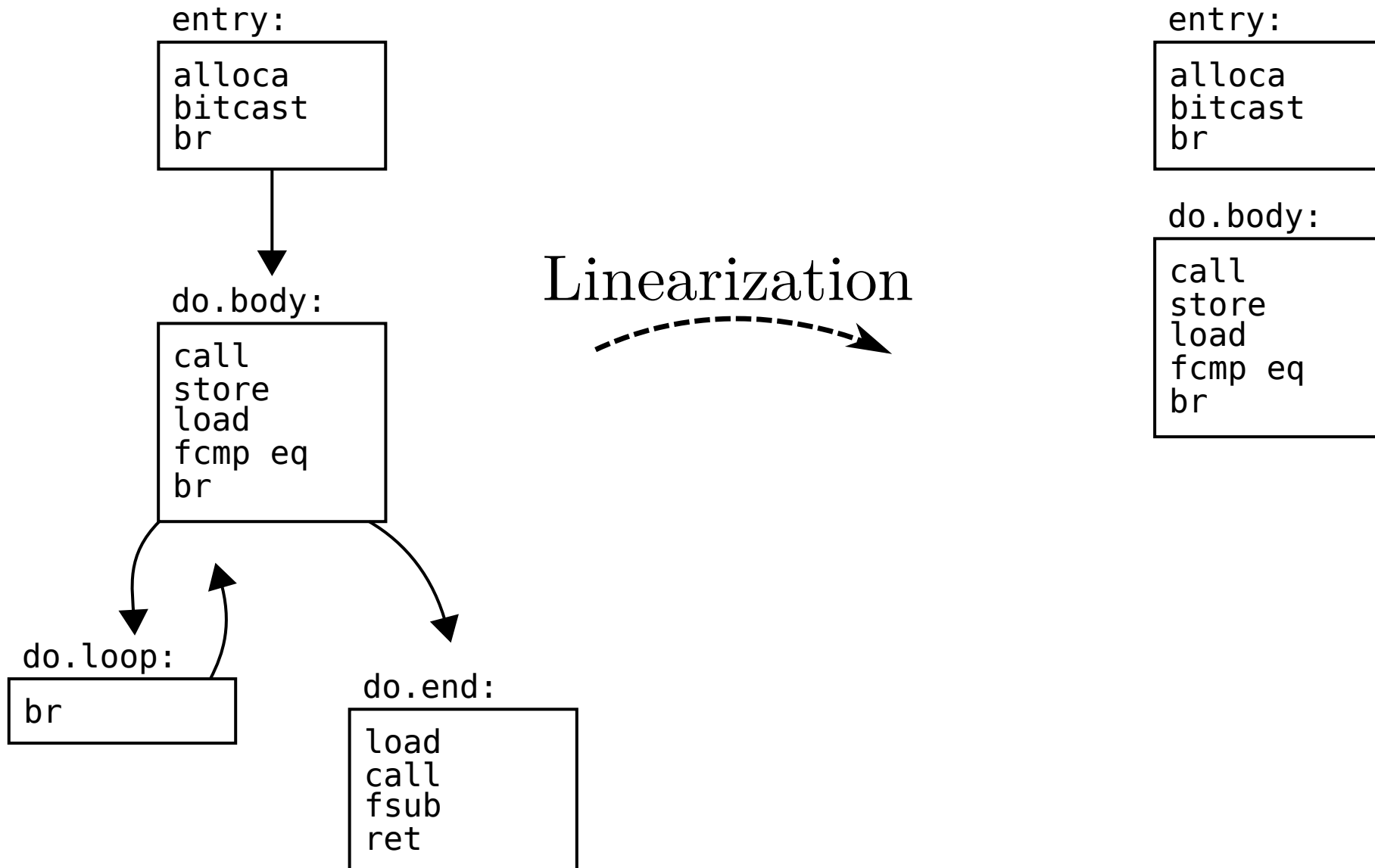
Our Technique



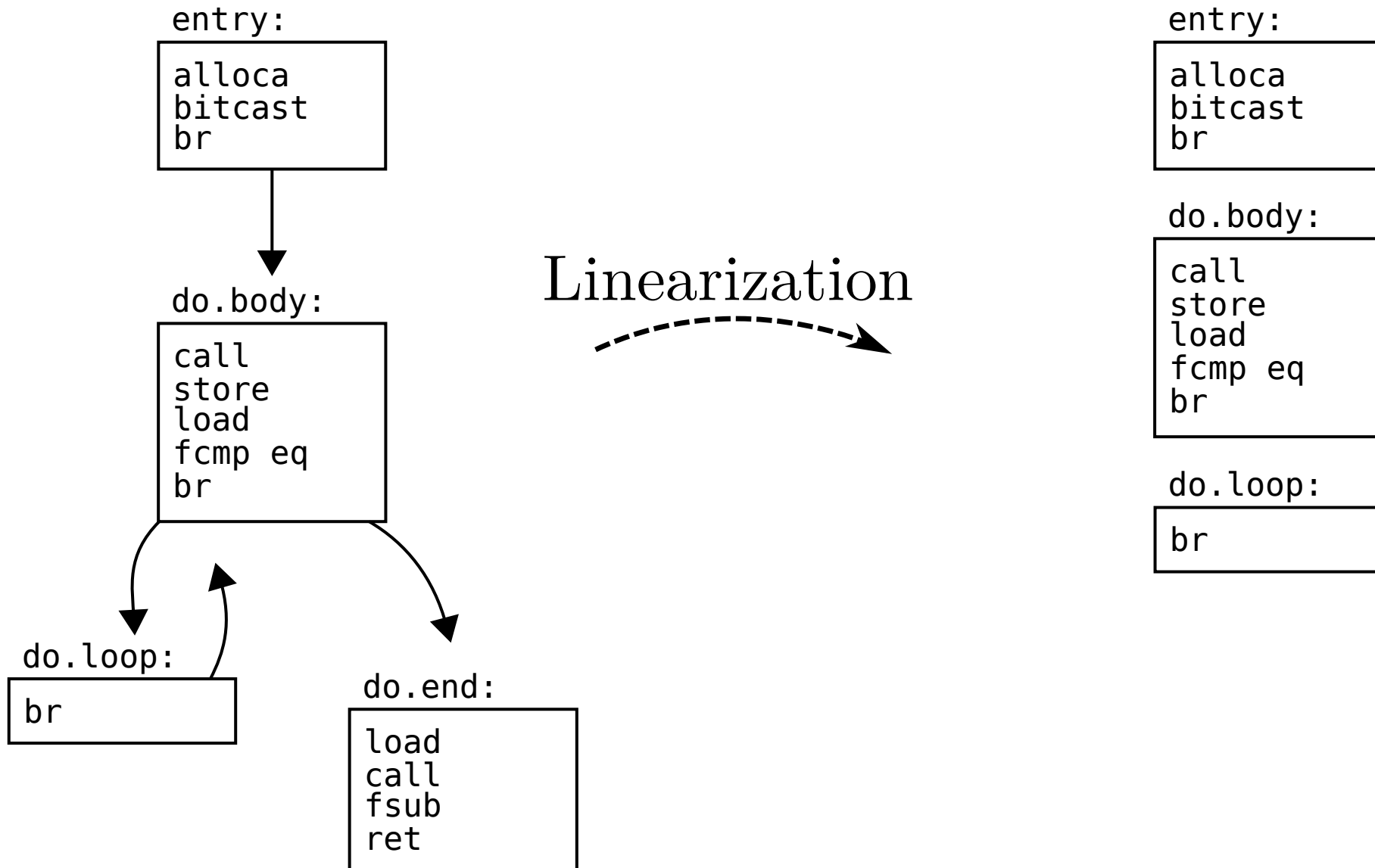
Linearization



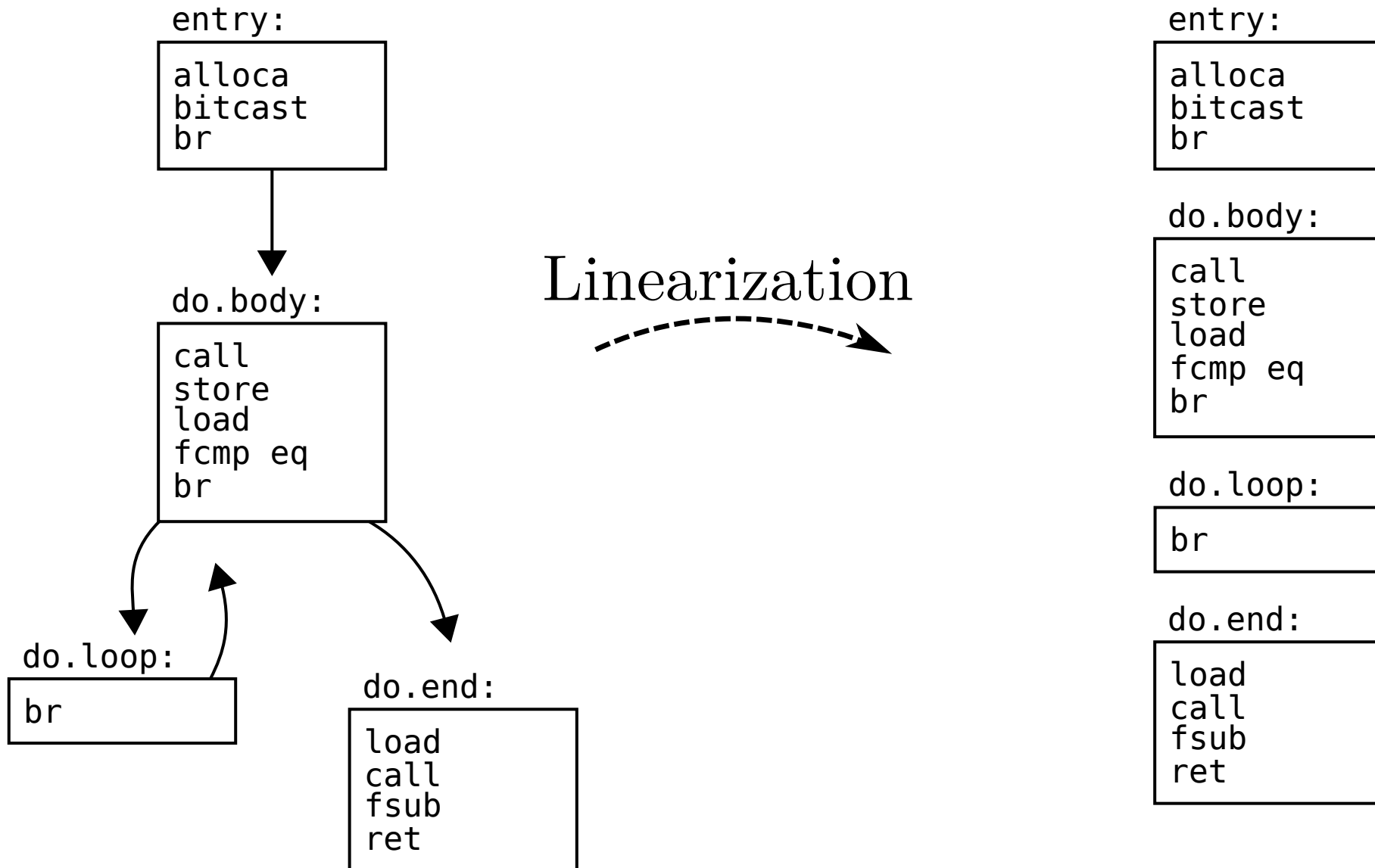
Our Technique



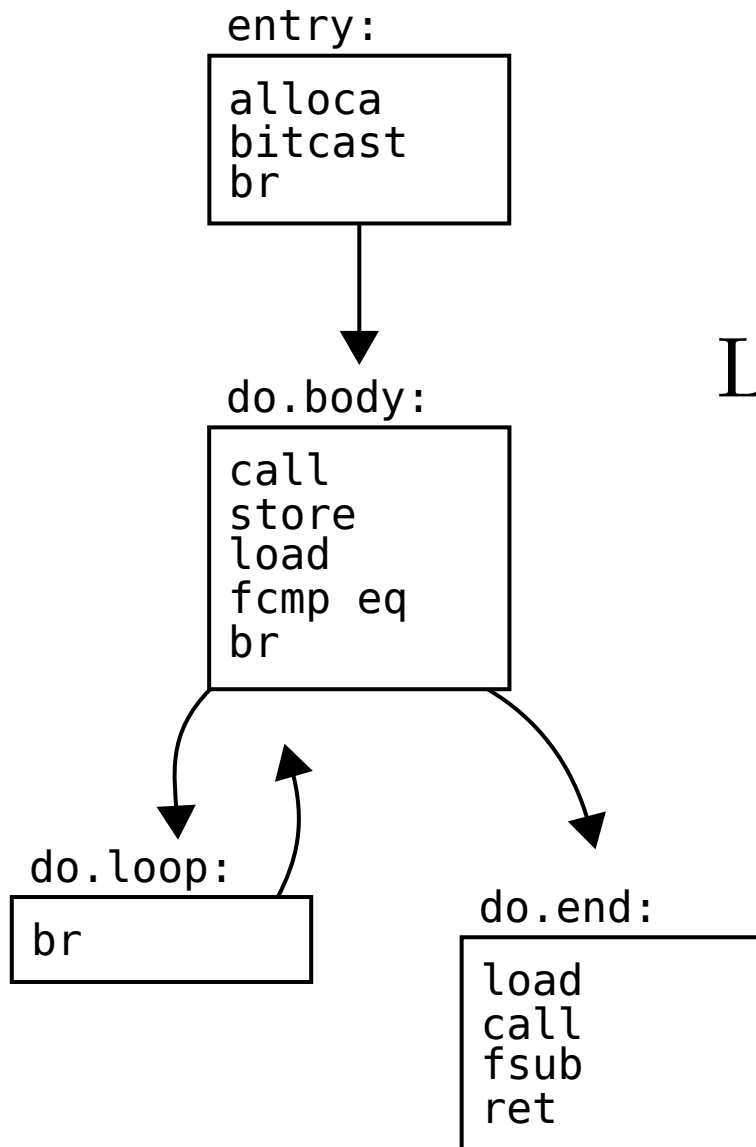
Our Technique



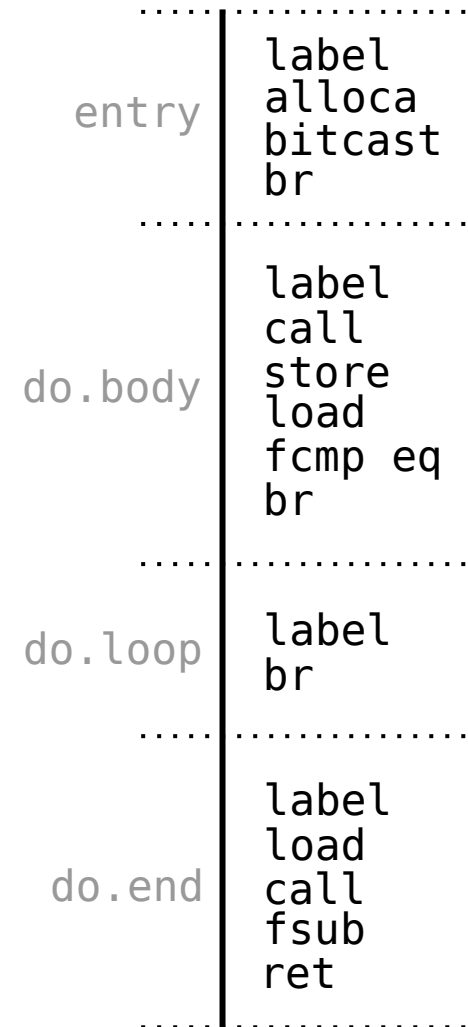
Our Technique



Our Technique



Linearization



Our Technique

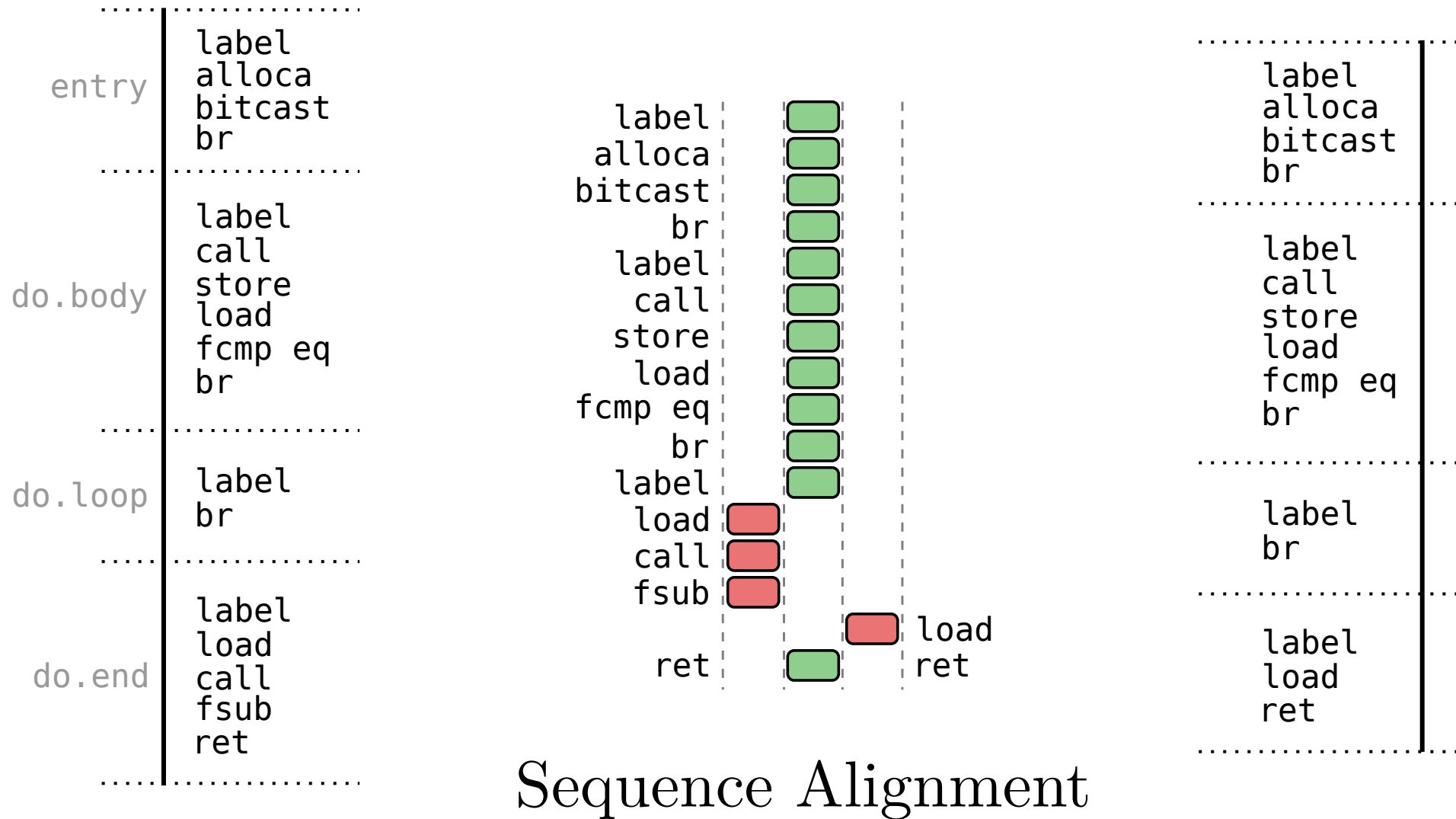
entry	label alloca bitcast br
do.body	label call store load fcmp eq br
do.loop	label br
do.end	label load call fsub ret

Our Technique

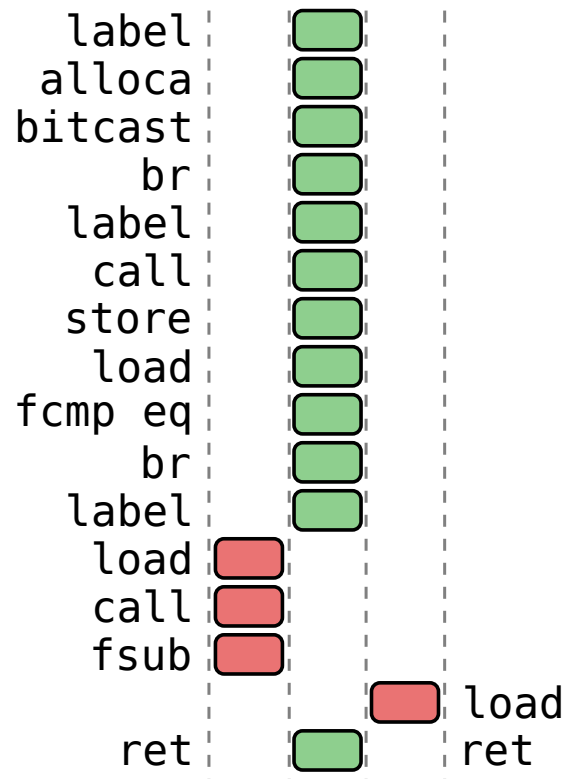
entry	label alloca bitcast br
do.body	label call store load fcmp eq br
do.loop	label br
do.end	label load call fsub ret

label alloca bitcast br
label call store load fcmp eq br
label br
label load ret

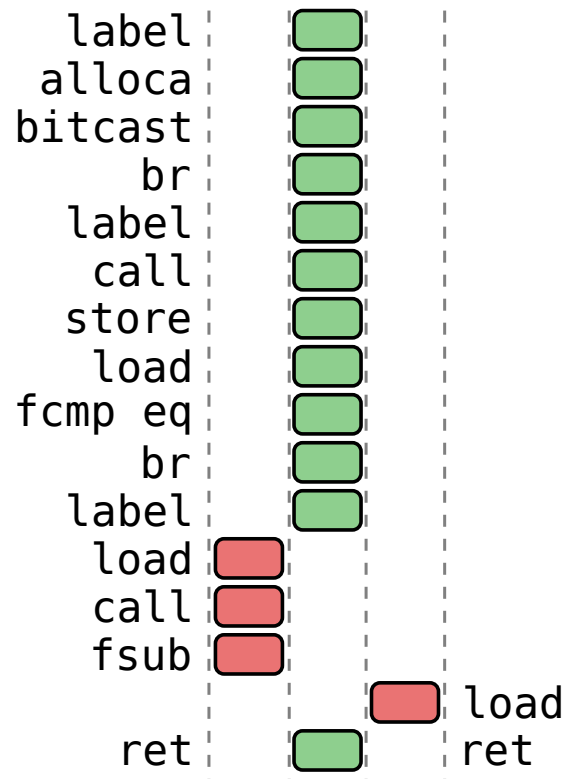
Our Technique



Our Technique



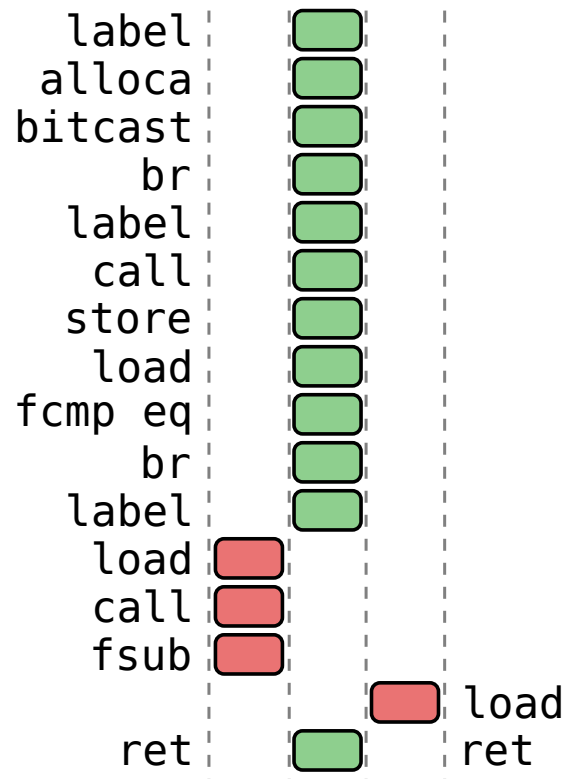
Our Technique



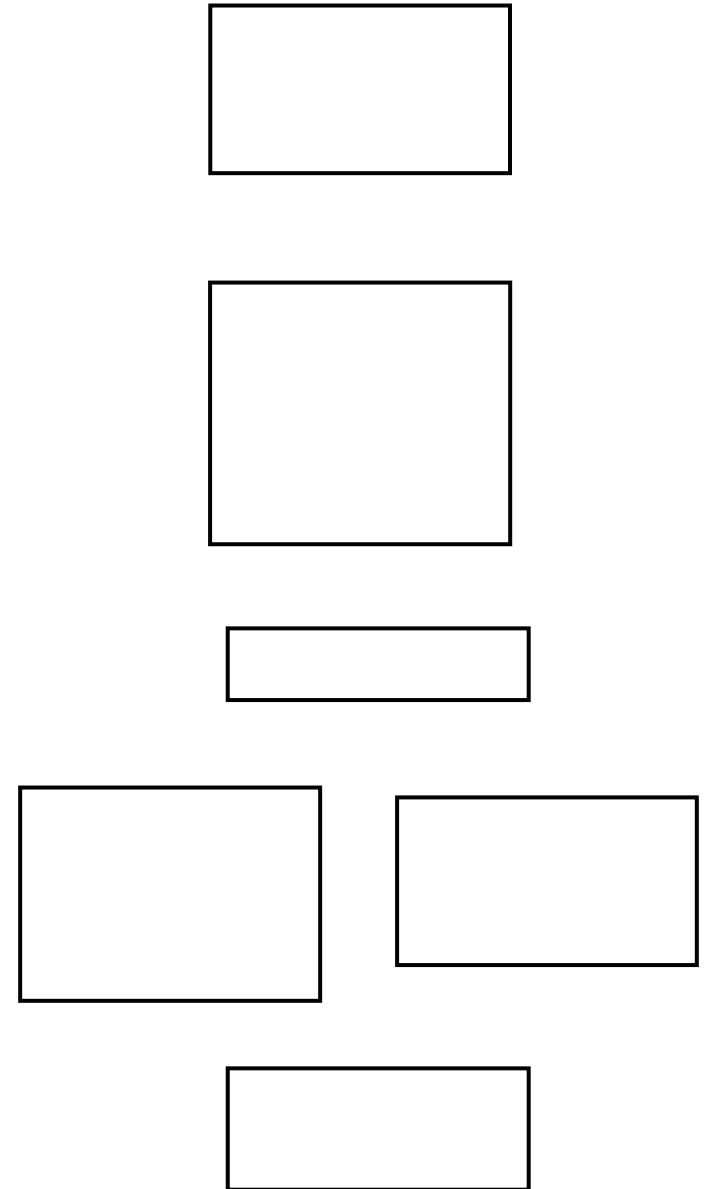
Code Generation



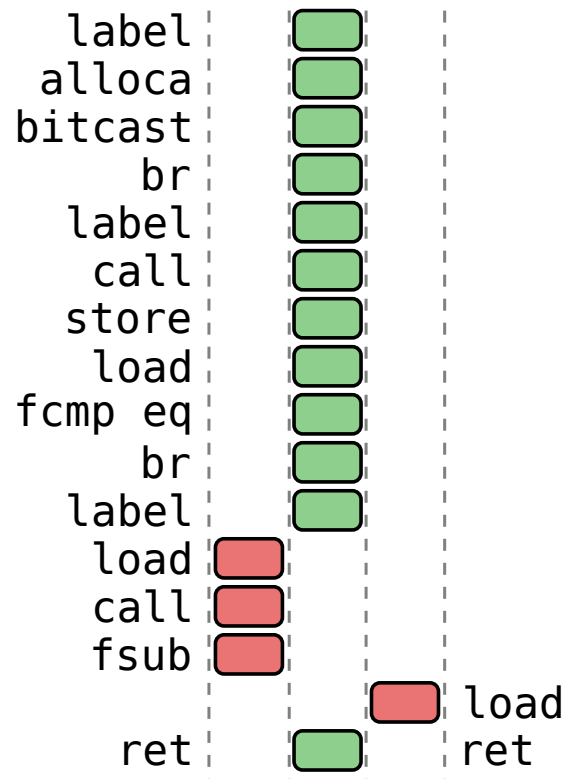
Our Technique



Code Generation



Our Technique



Code Generation



```
alloca
bitcast
br
```

```
call
store
load
fcmp eq
br
```

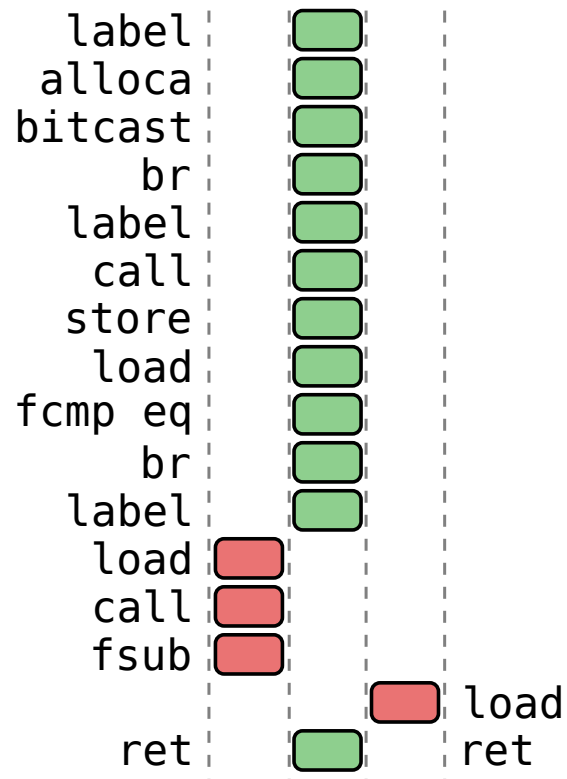
br

```
load
call
fsub
br
```

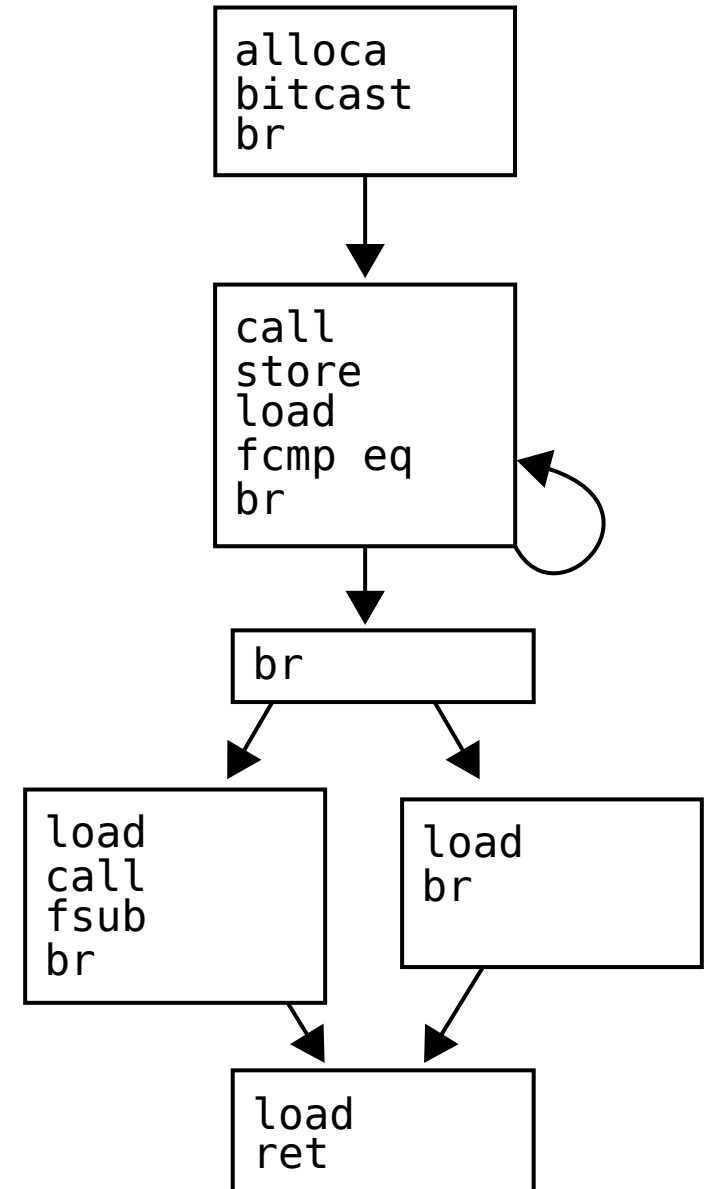
```
load
br
```

```
load
ret
```

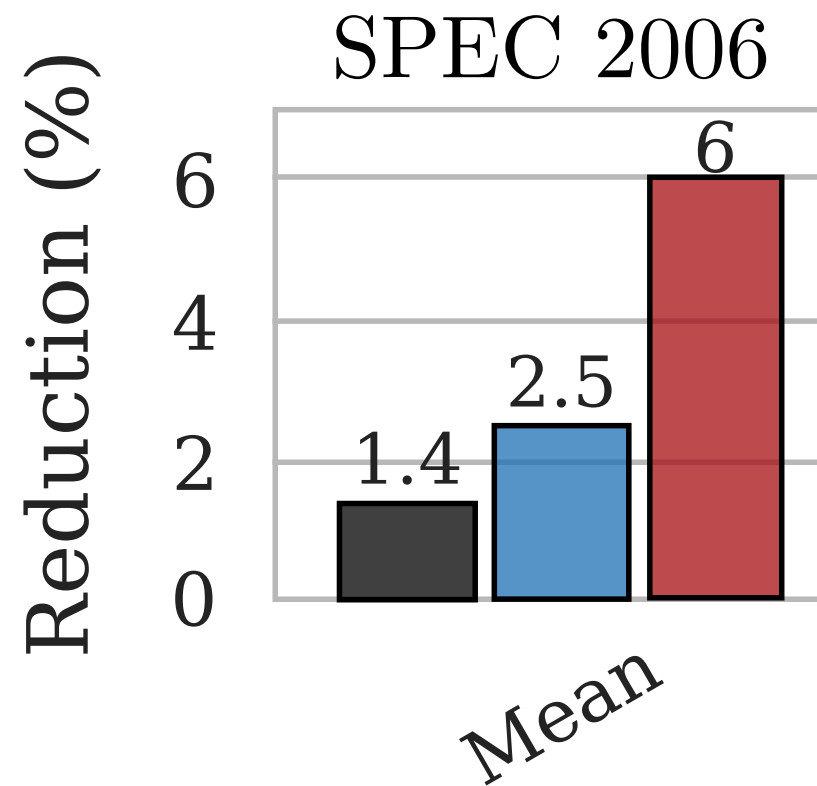
Our Technique



Code Generation



Evaluation



Identical

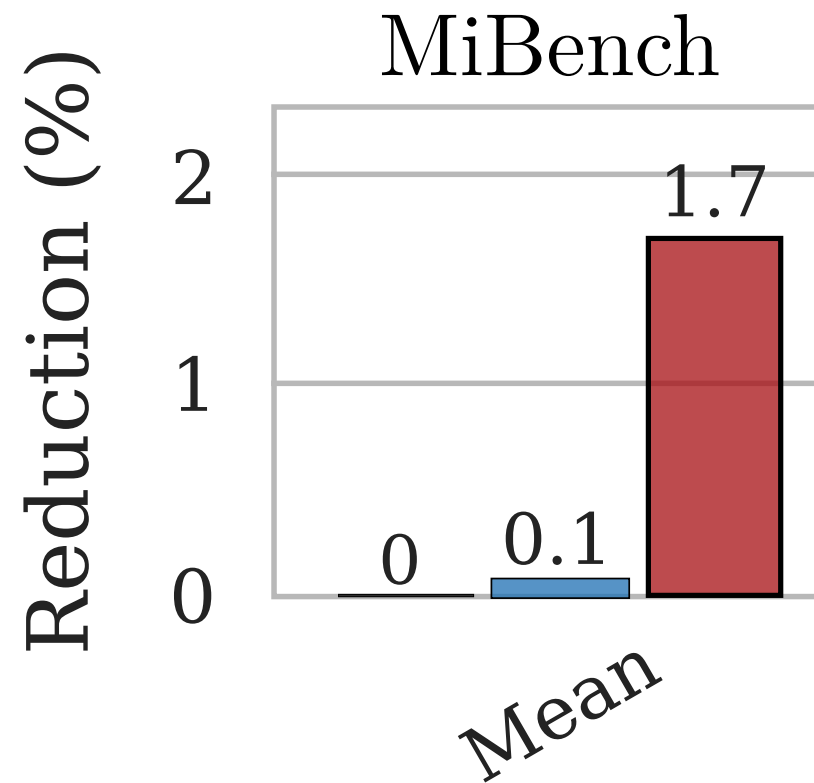


SOA



FMMSA

Evaluation



Identical



SOA



FMMSA