PART III:
Using Pro64 in Compiler Research and Development

Case Studies

Outline

- General Remarks
- ***Case Study I***: Integration of new instruction reordering algorithm to minimize register pressure
  [Govind,Yang,Amaral,Gao2000]
- ***Case Study II***: Design and evaluation of an induction pointer prefetching algorithm
  [Stouchinin,Douillet,Amaral,Gao2000]

Case I

- Introduction of the Minimum Register Instruction Sequence (MRIS) problem and a proposed solution
  - Problem formulation
  - The proposed algorithm
- Pro64 porting experience
  - Where to start
  - How to start
  - Results
- Summary

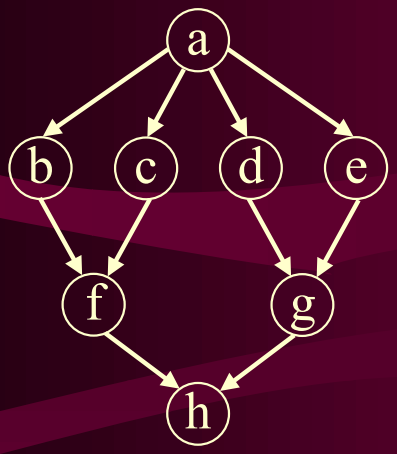
Researchers

- R. Govindarajan *(Indian Inst. Of Science)*
- Hongbo Yang *(Univ. of Delaware)*
- Chihong Zhang *(Conexant)*
- José Nelson Amaral *(Univ. of Alberta)*
- Guang R. Gao *(Univ. of Delaware)*

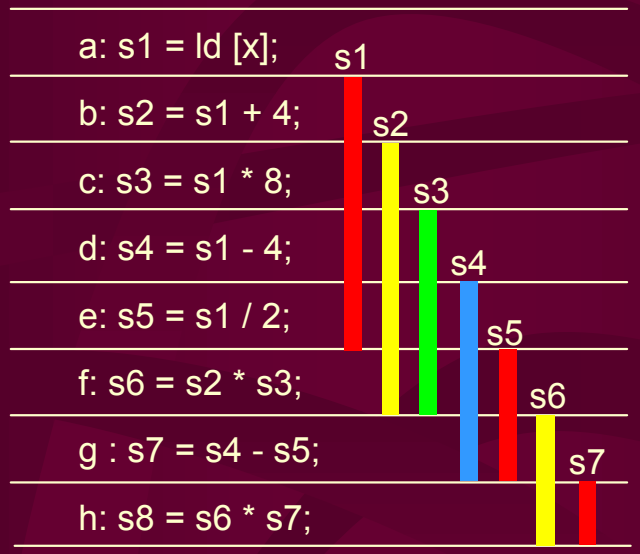
The Minimum Register Instruction Sequence Problem

Given a data dependence graph ***G***,
derive an *instruction sequence* ***S*** for ***G***
that is optimal in the sense that its
register requirement is minimum.

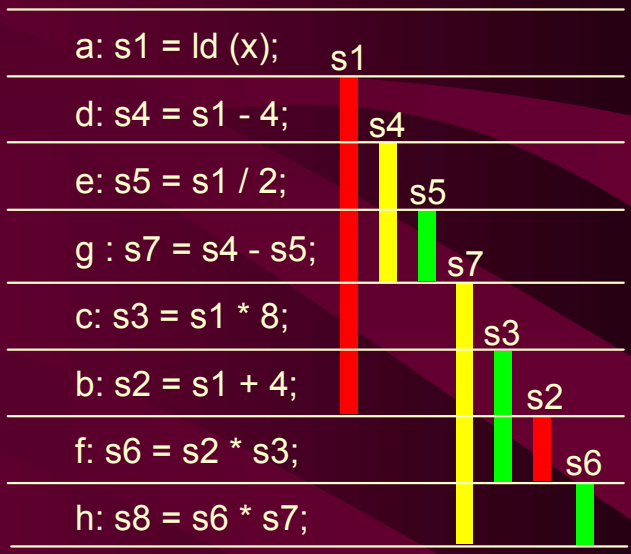
A Motivating Example



(a) DDG



(b) Instruction Sequence 1



(c) Instruction Sequence 2

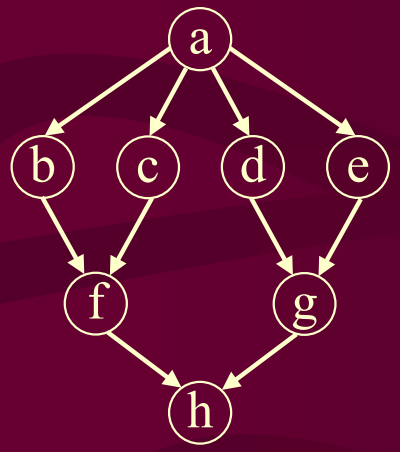
Observation: Register requirements drop 25% from (b) to (c) !

Motivation

- **IA-64 style processors**
  - Reduce spills in local register allocation phase
  - Reduce Local Register Allocation (LRA) requests in Global Register Allocation (GRA) phase
  - Reduce overall register pressure on a per procedure basis
- **Out-of-order issue processor**
  - Instruction reordering buffer
  - Register renaming

How to Solve the MRIS Problem?

- Register lineages
- Live range of lineages
- Lineage interference



L1 = (a, b, f, h);
L2 = (c, f);
L3 = (e, g, h);
L4 = (d, g);

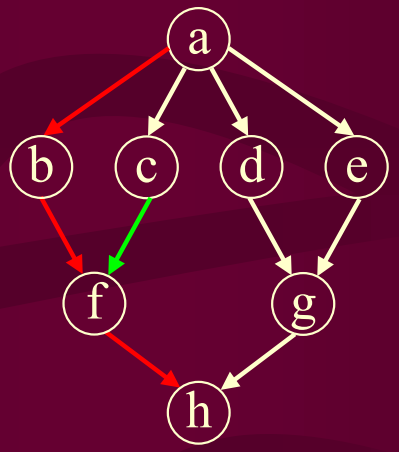
(a) Concepts

(b) DDG

(c) Lineages

How to Solve the MRIS Problem?

- Register lineages
- Live range of lineages
- Lineage interference



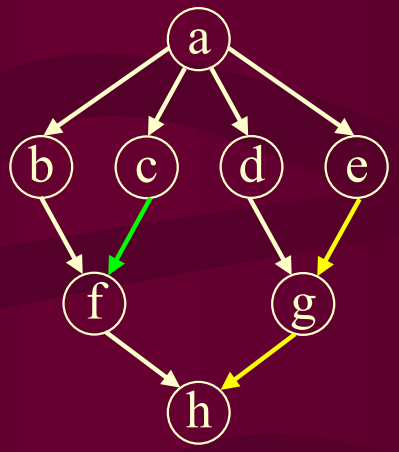
L1 = (a, b, f, h);
L2 = (c, f);
L3 = (e, g, h);
L4 = (d, g);

(a) Concepts (b) DDG (c) Lineages

Questions: Can **L1** and **L2** share the same register?

How to Solve the MRIS Problem?

- Register lineages
- Live range of lineages
- Lineage interference



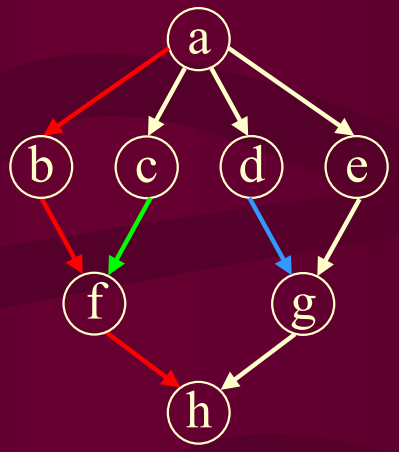
L1 = (a, b, f, h);
L2 = (c, f);
L3 = (e, g, h);
L4 = (d, g);

(a) Concepts (b) DDG (c) Lineages

Questions: Can L1 and L2 share the same register?
Can **L2** and **L3** share the same register?

How to Solve the MRIS Problem?

- Register lineages
- Live range of lineages
- Lineage interference

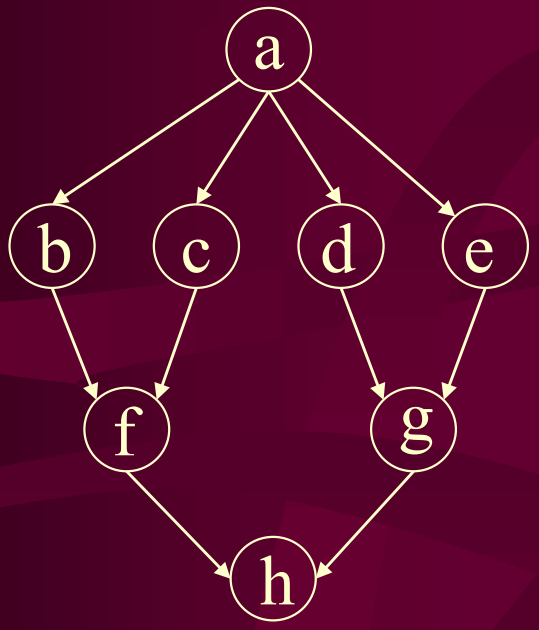


L1 = (a, b, f, h);
L2 = (c, f);
L3 = (e, g, h);
L4 = (d, g);

(a) Concepts (b) DDG (c) Lineages

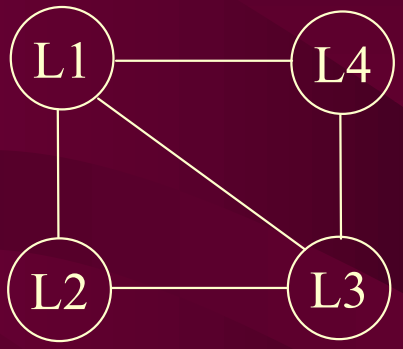
Questions: Can L1 and L2 share the same register?
Can L2 and L3 share the same register?
Can **L1** and **L4** share the same register?
Can **L2** and **L4** share the same register?

Lineage Interference Graph



(a) Original DDG

L1 = (a, b, f, h);
L2 = (c, f);
L3 = (e, g, h);
L4 = (d, g);



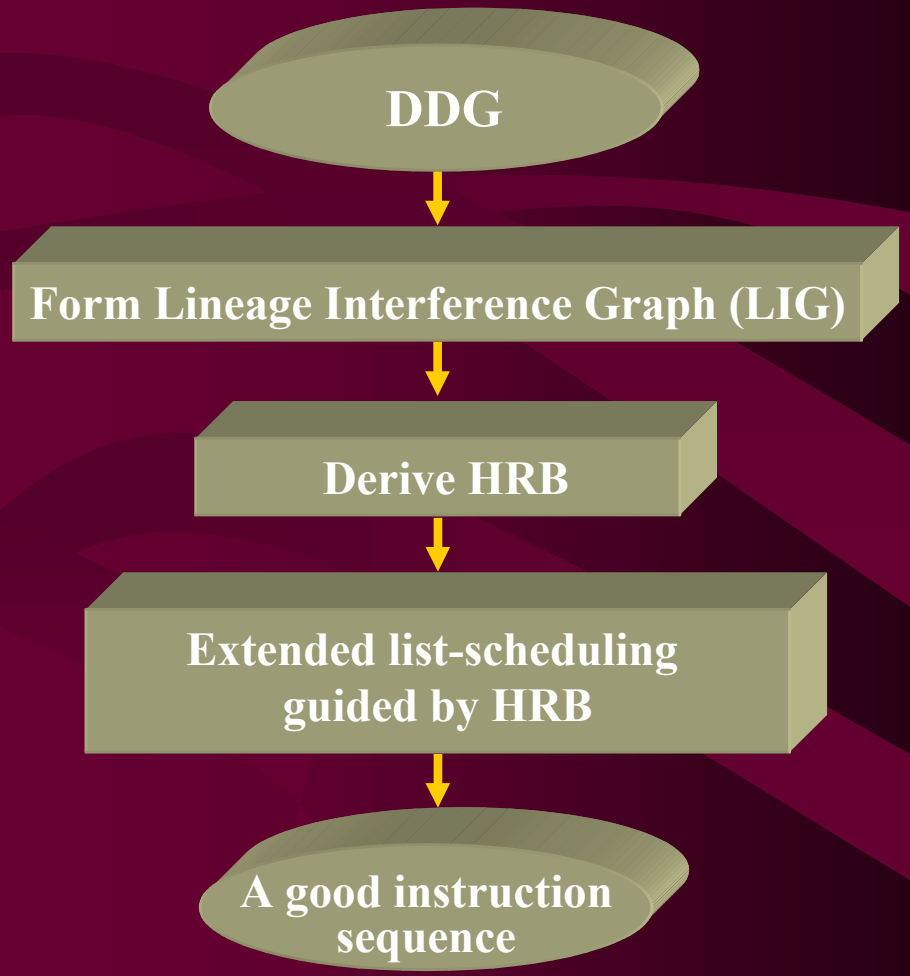
(b) Lineage Interference Graph (LIG)

Question: Is the lower bound of the required registers = 3?

Challenge: Derive a "Heuristic Register Bound" (HRB)!

Our Solution Method

- A "good" construction algorithm for LIG
- An effective heuristic method to calculate the HRB
- An efficient scheduling method (do not backtrack)



Pro64 Porting Experience

- Porting plan and design
- Implementation
- Debugging and validation
- Evaluation

Implementation

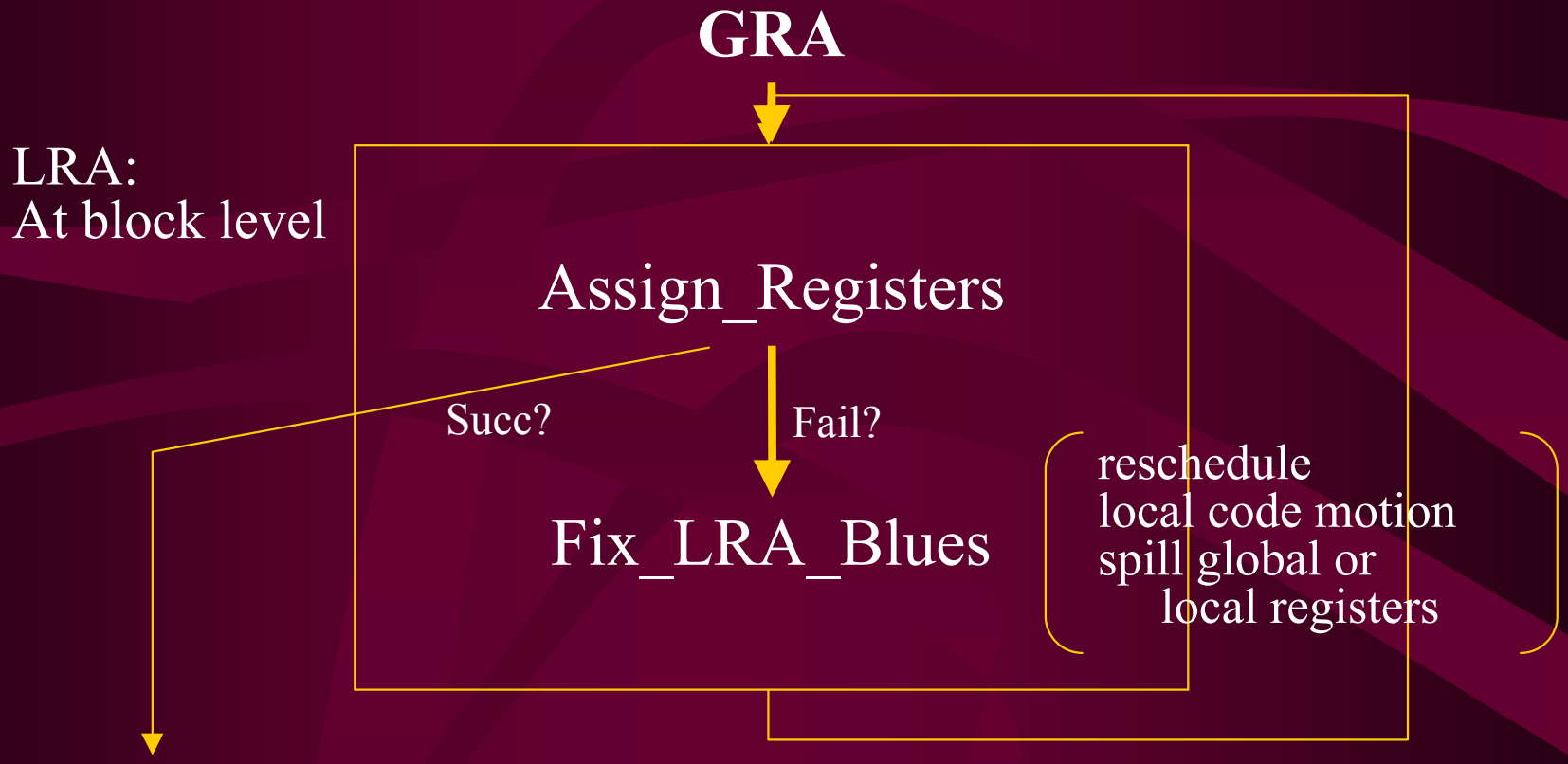
- Dependence graph construction
- LIG formation
- LIG construction and coloring
- The reordering algorithm implementation

Porting Plan and Design

- Understand the compiler infrastructure
- Understand the register model (mainly from targ_info)

  e.g.:
  - register classes: (int, float, predicate, app, control)
  - register save/restore conventions: caller/callee save, return value, argument passing, stack pointer, etc.

Register Allocation



Implementation

- DDG construction: use native service routines: e.g. CG_DEP_Compute_Graph
- LIG coloring: using native support for set package (e.g. bitset.c)
- Scheduler implementation: vector package native support (e.g. cg_vector.cxx)
- Access dependence graph using native service functions ARC_succs, ARC_preds, ARC_kind

Debugging and Validation

- Trace file
  - tt54:0x1. General trace of LRA
  - tt45: 0x4. Dependence graph building
  - tr53. Target Operations (TOP) before LRA
  - tr54. TOP after LRA

Evaluation

- Static measurement
  - Fat point -tt54: 0x40
- Dynamic measurement
  - Hardware counter in R12k and *perfex*

Evaluation

- For the MIPS R12K (SPEC95fp), the lineage-based algorithm reduce the number of loads executed by 12%, the number of stores by 14%, and the execution time by 2.5% over a baseline.
- It is slightly better than the algorithm in the MIPSPro compiler.

Case II

Design and Evaluation of an Induction Pointer Prefetching Algorithm

Researchers

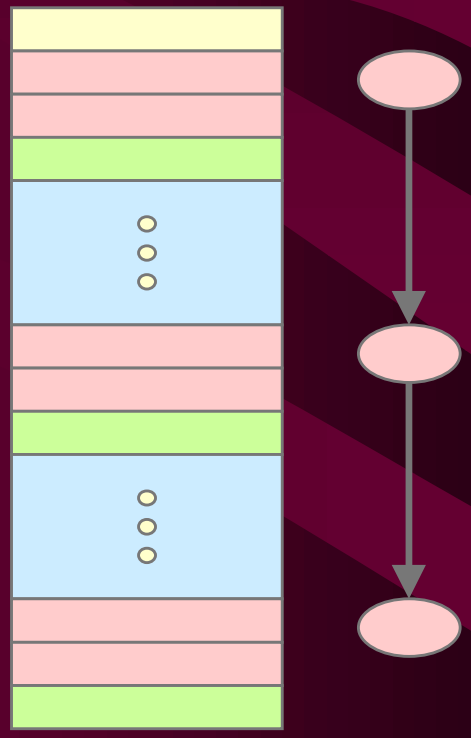
- Artour Stoutchinin *(STMicroelectronics)*
- José Nelson Amaral *(Univ. of Alberta)*
- Guang R. Gao *(Univ. of Delaware)*
- Jim Dehnert *(Silicon Graphics Inc.)*
- Suneel Jain *(Narus Inc.)*
- Alban Douillet *(Univ. of Delaware)*

Motivation

The important loops of many programs are pointer-chasing loops that access recursive data structures through induction pointers.

Example:

```
max = 0;
current = head;
while(current != NULL)
        {
          if(current->key > max)
             max = current->key;
         current = current->next;
        }
```



Problem Statement

How to identify pointer-chasing recurrences?

How to decide whether there are enough processor resources and memory bandwidth to profitably prefetch an induction pointer?

How to efficiently integrate induction pointer prefetching with loop scheduling based on the profitability analysis?

Prefetching Costs

- More instructions to issue
- More memory traffic
- Longer code (disruption in instruction cache)
- Displacement of potentially good data from cache

Before prefetching:

```
t226 = lw 0x34(t228)
```

After prefetching:

```
t226 = lw 0x34(t228)
tmp = subu t226, t226s
tmp = addu tmp, tmp
tmp = addu t226, tmp
pref 0x0(tmp)
t226s = t226
```

What to Prefetch?
When to Prefetch it?

A good optimizing compiler should only prefetch data that will actually be referenced.

It should prefetch far enough in advance to prevent a cache miss when the reference occurs.

But, not too far in advance, because the data might be evicted from the cache before it is used, or might displace data that will be referenced again.

Prefetch Address

In order to prefetch, the compiler must calculate addresses that will be referenced in future iterations of the loop.

For loops that access regular data structures, such as vectors and matrices, compilers can use static analysis of the array indexes to compute the prefetching addresses.

How can we predict future values of induction pointers?

Key Intuition

Recursive data structures are often allocated at regular intervals.

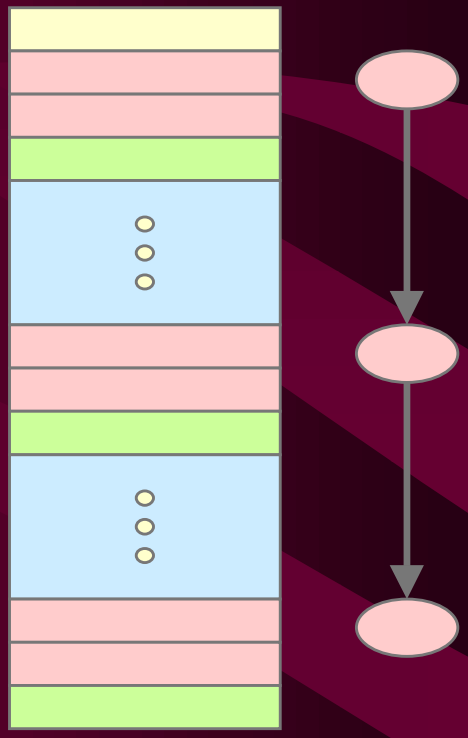
Example:

```
curr = head = (item) malloc(sizeof(item));
while(curr->key = get_key()) != NULL)
        {
          curr->next = curr = (item)malloc(sizeof(item));
          other_memory_allocations();
        }
curr -> next = NULL;
```

Pre-Fetching Technique

Example:

```
max = 0;
current = head;
tmp = current;
while(current != NULL)
      {
       if(current->key > max)
          max = current->key;
      current = current->next;
      stride = current - tmp;
      prefetch(current + stride*k);
      tmp = current;
      }
```



Prefetch Sequence (R10K)

In our implementation, the stride is recomputed in every iteration of the loop, making it tolerant of (infrequent) stride changes.

```
stride = addr - addr.prev
stride = stride * k
addr.pref = addr + stride
addr.prev = addr
pref addr.pref
```

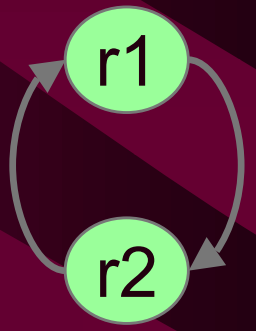
Identification of Pointer-Chasing Recurrences

A surprisingly simple method works well: look in the intermediate code for recurrence circuits containing only loads with constant offsets.

Examples:

```
node = ptr->next;          r1 <- load r2, offset_next
ptr = node->ptr;           r2 <- load r1, offset_ptr

current = current->next;   r2 <- load r1
                           r1 <- load r2, offset_next
```



Profitability Analysis

Goal: Balance the gains and costs of prefetching.

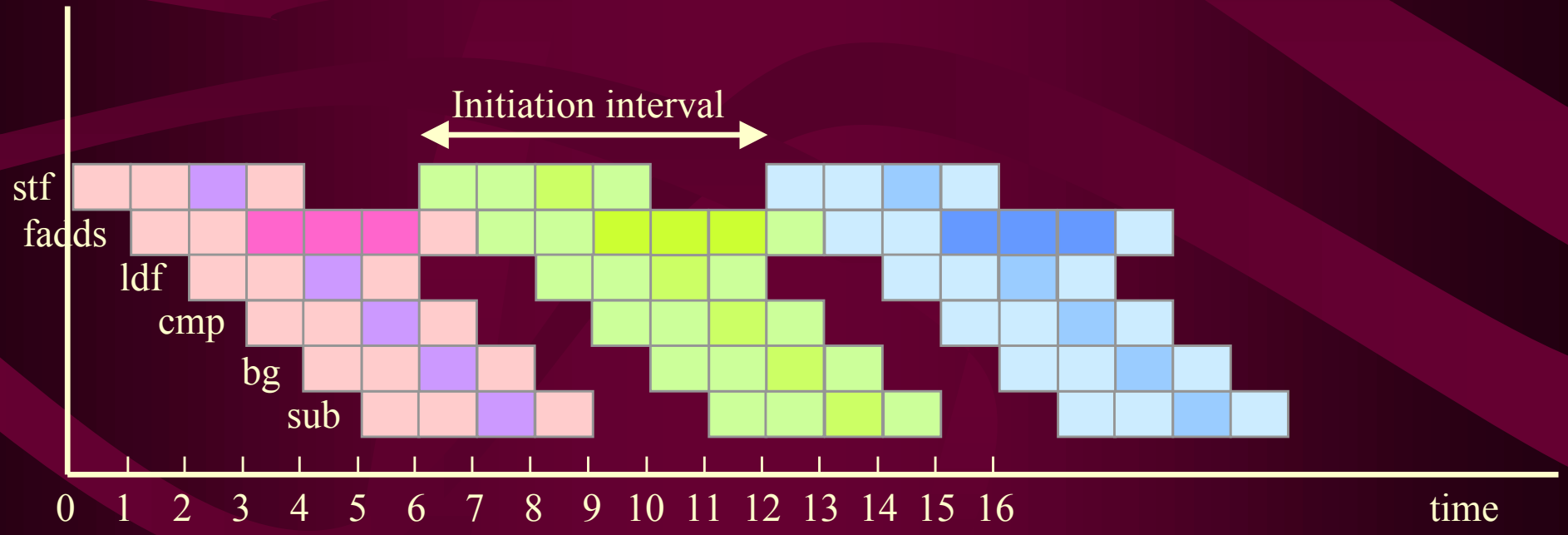
Although we use resource estimates analogous to those done for software pipelining, we consider loop bodies with control flow.

How to estimate the resources available for prefetching in a basic block B that belongs to many data dependence recurrences?

Software Pipelining

What limits the speed of a loop?

- Data dependences: recurrence initiation interval (recMII)
- Processor resources: resource initiation interval (resMII)
- Memory accesses: memory initiation interval (memMII)

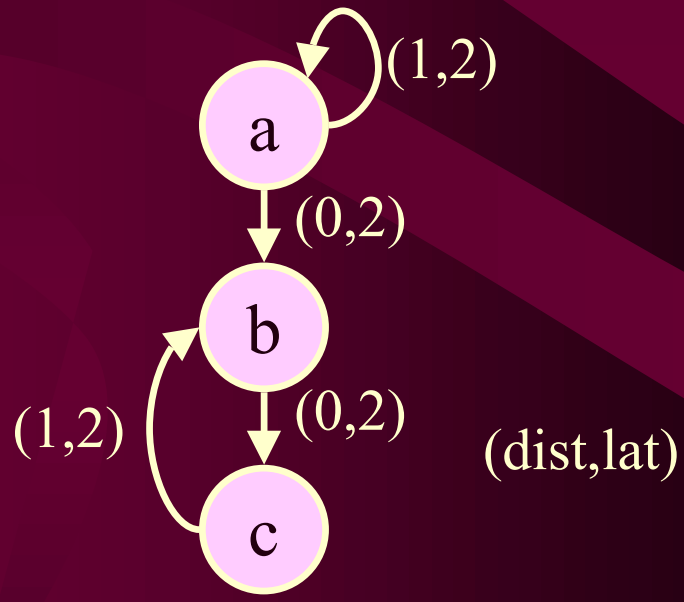


Data Dependences(recMII)

The recurrence minimum initiation interval (recMII) is given by:

$$recMII = \max_{(\forall cycle\,\theta)} \left\lceil \frac{latency(\theta)}{iterationdistance(\theta)} \right\rceil$$

```
        for i = 0 to N - 1 do
a:          X[i] = X[i - 1] + R[i];
b:          Y[i] = X[i] + Z[i - 1];
c:          Z[i] = Y[i] + 1;
        end;
```



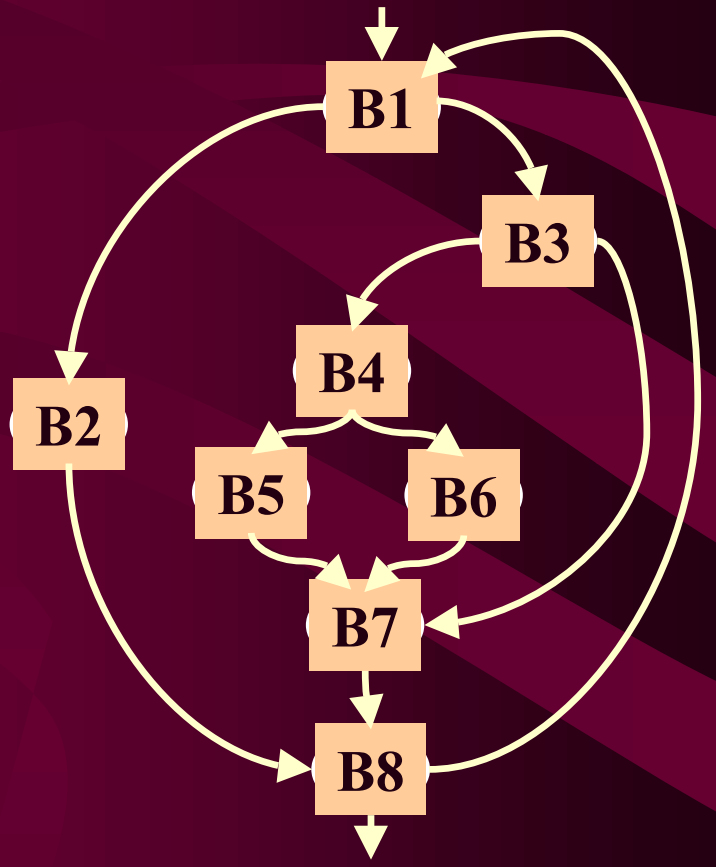
The recMII for Loops with Control Flow

An instruction of a basic block B, can belong to many recurrences (with distinct control paths).

We define the recurrence MII of a load operation L as:

$$recMII(L) = \max_{c|L \in c}[recMII(c)]$$

L ∈ c means that the operation L is part of the recurrence c.

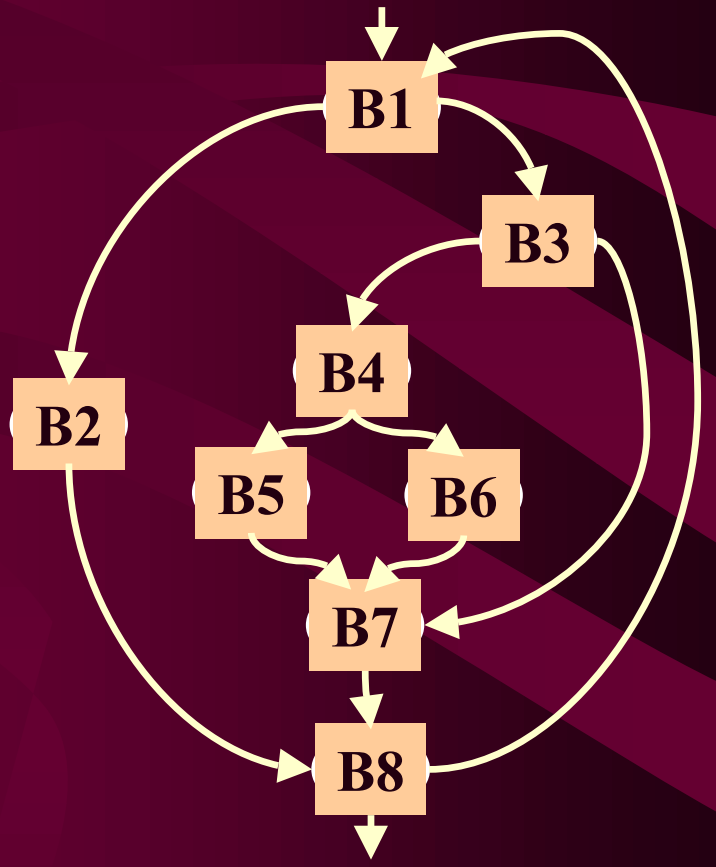


Control Flow Graph

Processor Resources(resMII)

A basic block B may belong to multiple control paths. We define the resource constraint of a basic block B as the maximum over all control paths that execute B.

$$resMII(B) = \max_{p|B \in p}[resMII(p)]$$



Control Flow Graph

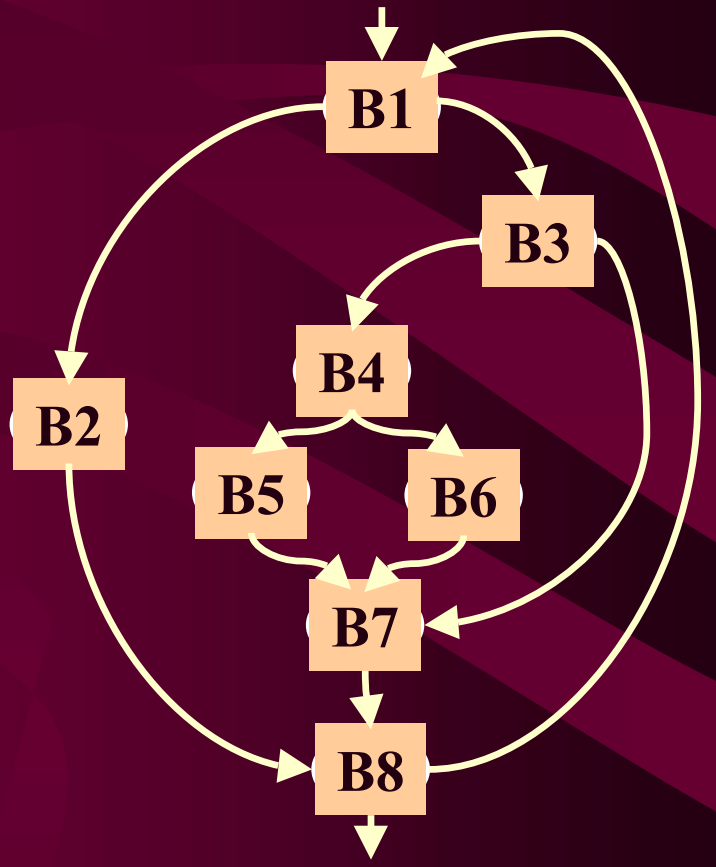
Available Memory Bandwidth

Processors with non-blocking caches can support up to k outstanding cache misses without stalling.

We define the available memory bandwidth of all control paths that execute a basic block B as

$$M(B) = \min_{p:B \in p} \{k - m(p)\}$$

where $m(p)$ is the number of expected cache misses in each control path p.



Control Flow Graph

Profitability Analysis

Adding prefetch code for an induction pointer L in a basic block B is profitable if both:

(1) the mii due to recurrences that contain L is greater than the resMII after prefetch insertion, and

(2) there is enough memory bandwidth to enable another cache miss without causing stalls.

$$resMII^{\mathbf{P}}(B) \leq recMII^{\mathbf{P}}(L) \wedge M(B) > 0$$

Computing Available Memory Bandwidth

To compute the available memory bandwidth of a control path we need to estimate how many cache misses are expected in that control path.

We use a graph coloring technique over a cache miss interference graph to predict which memory references are likely to incur a miss.

The Miss Interference Graph

Two memory references interfere if:

1. They are both expected to miss the cache
2. They can both be issued in the same iteration of the loop
3. They do not fall into the same cache line

Miss Interference Graph assumptions:

1. Loop invariant references are cache hits (global-pointer relative, stack-pointer relative, etc).
2. Memory references on mutually exclusive control paths do not interfere.
3. References relative to the same base address interfere only if their relative offset is larger than the cache line.

Prefetching Algorithm

DoPrefetch(P,V,E)

1. $C \leftarrow$ pointer-chasing recurrences
2. $R \leftarrow$ Prioritized list of induction pointer loads in C
3. $N \leftarrow$ Prioritized list of other loads (not in C)
4. $O \leftarrow$ R + N
5. mark each L in O as a cache miss
6. for each L in O, $L \in B$
7. do if $recMII^P(B) \leq resMII^P(B)$ and S(B)
8. then add prefetch for L to B
9. mark L as cache hit
10. endif
11. endfor

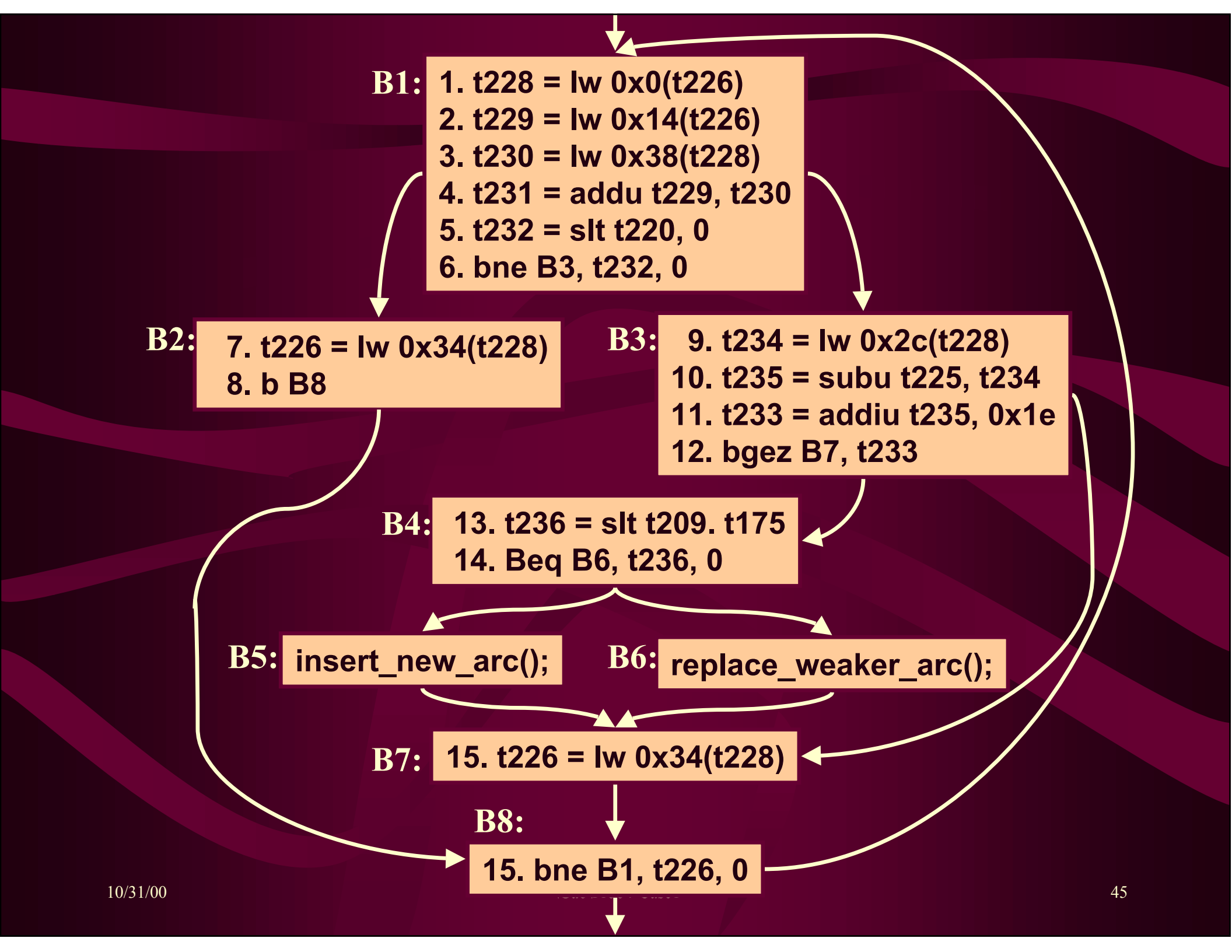
An Example*

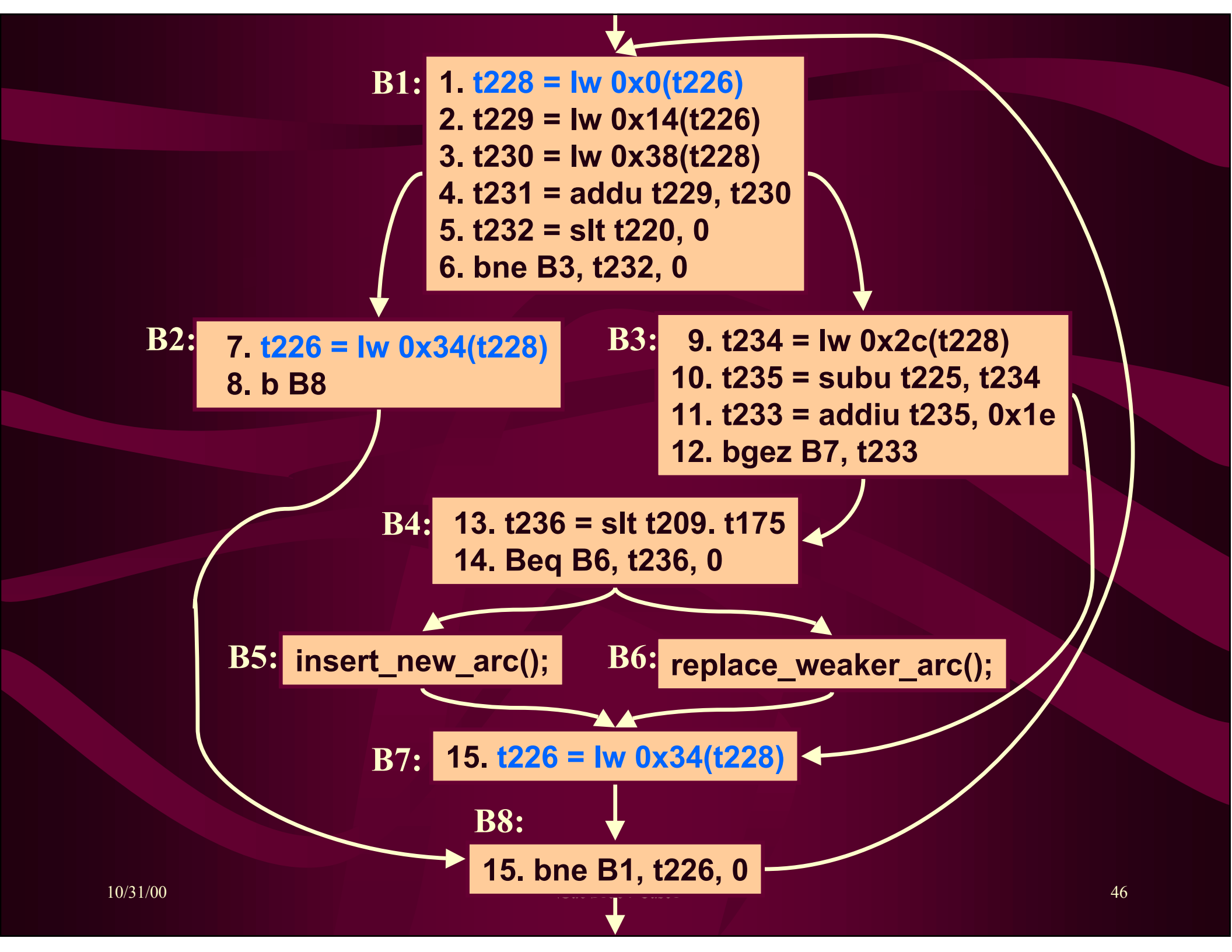
*mcf: minimal cost flow optimizer, (Konrad-Zuse Informatics Center, Berlin)

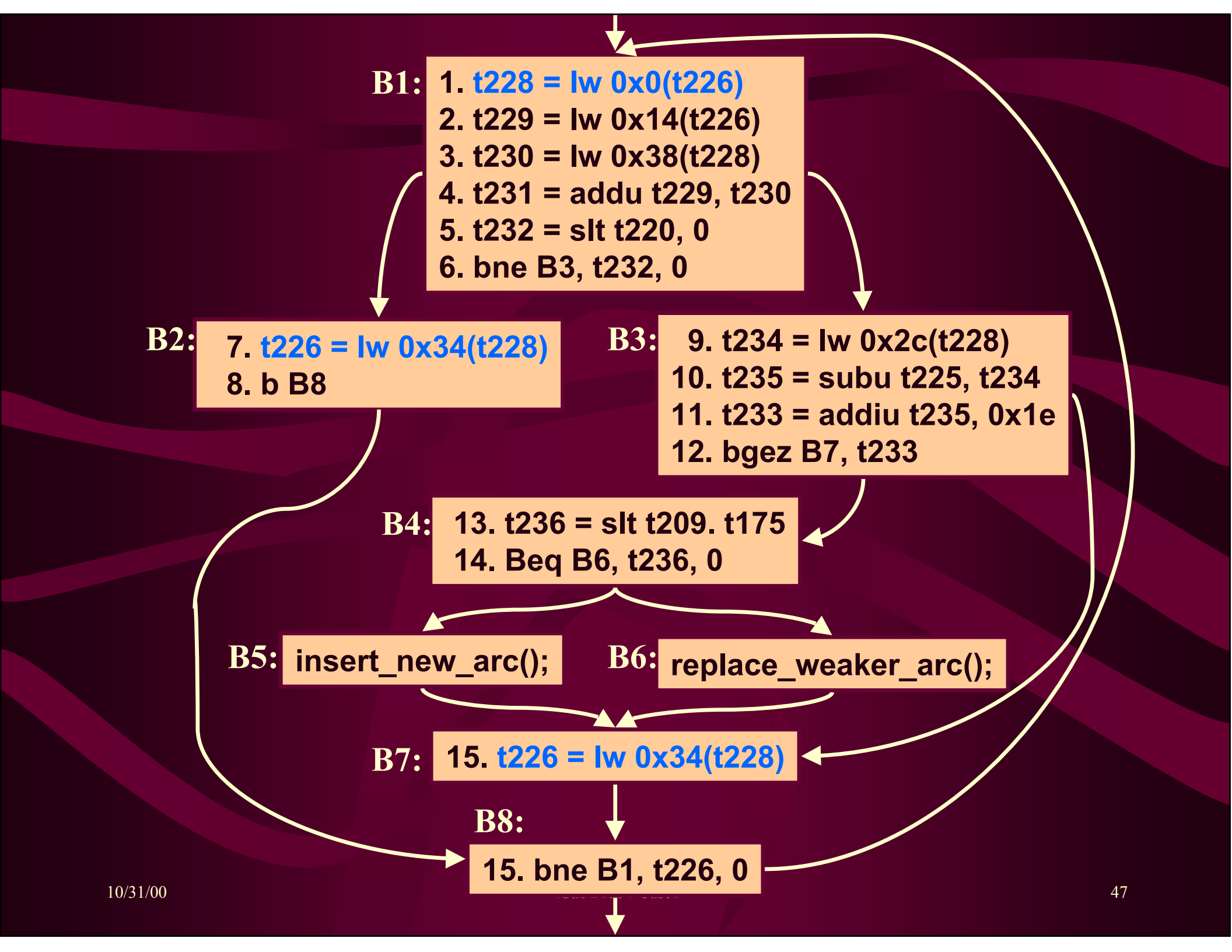
```
1  while (arcin){
2          tail = arcin->tail;
3          if (tail->time + arcin->org_cost > latest){
4             arcin = (arc_t *)tail->mark;
5             continue;
              }
6          arc_cost = tail->potential + head_potential;
7          if (red_cost < 0) {
8             if (new_arcs < MAX_NEW_ARCS){
9                insert_new_arc(arcnew, new_arcs, tail,
                                        head, arc_cost, red_cost);
10               new_arcs++;
                }
11            else if((cost_t)arcnew[0].flow > red_cost)
12                   replace_weaker_arc(arcnew, tail, head,
                                                arc_cost, red_cost);
              }
13         arcin = (arc_t *)tail->mark;
           }
```

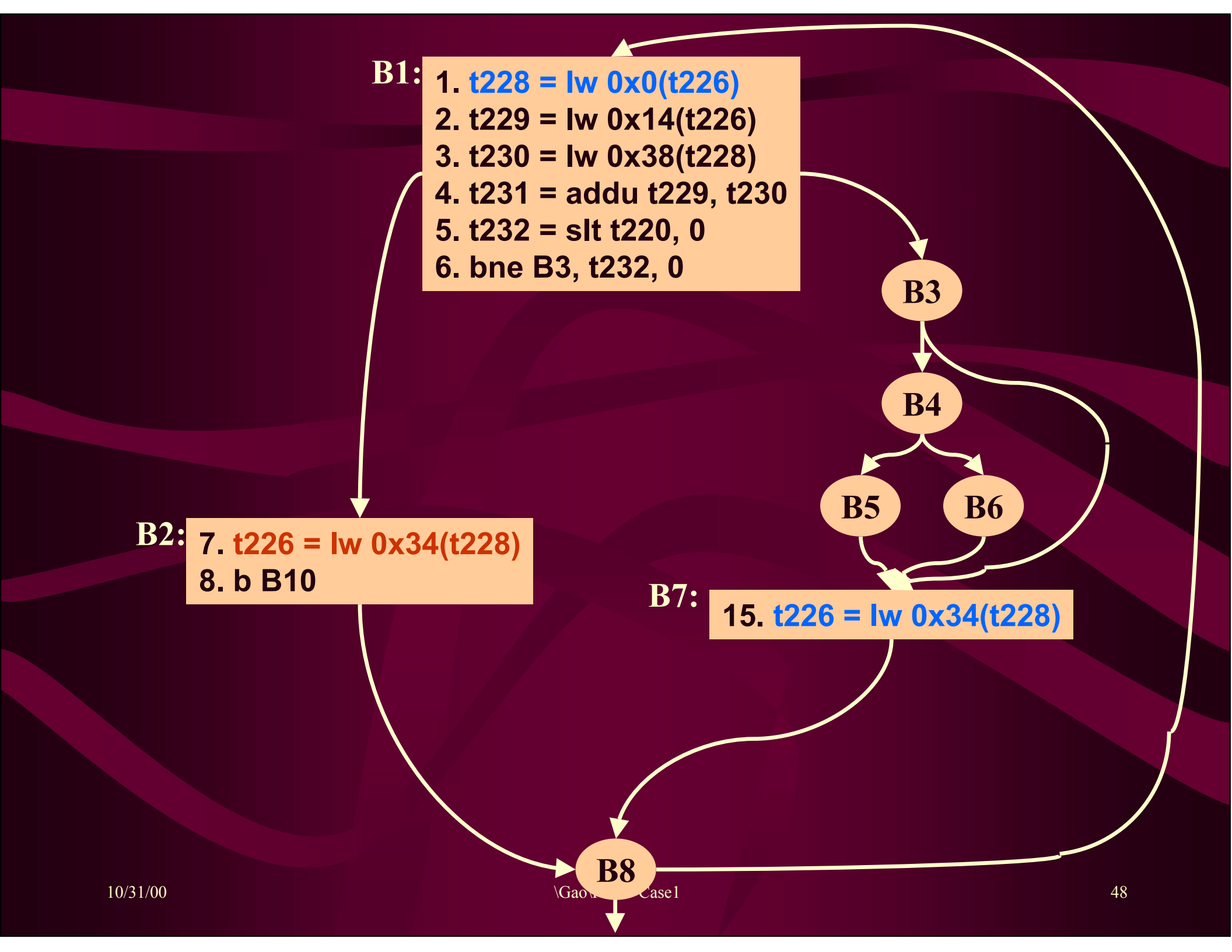
An Example

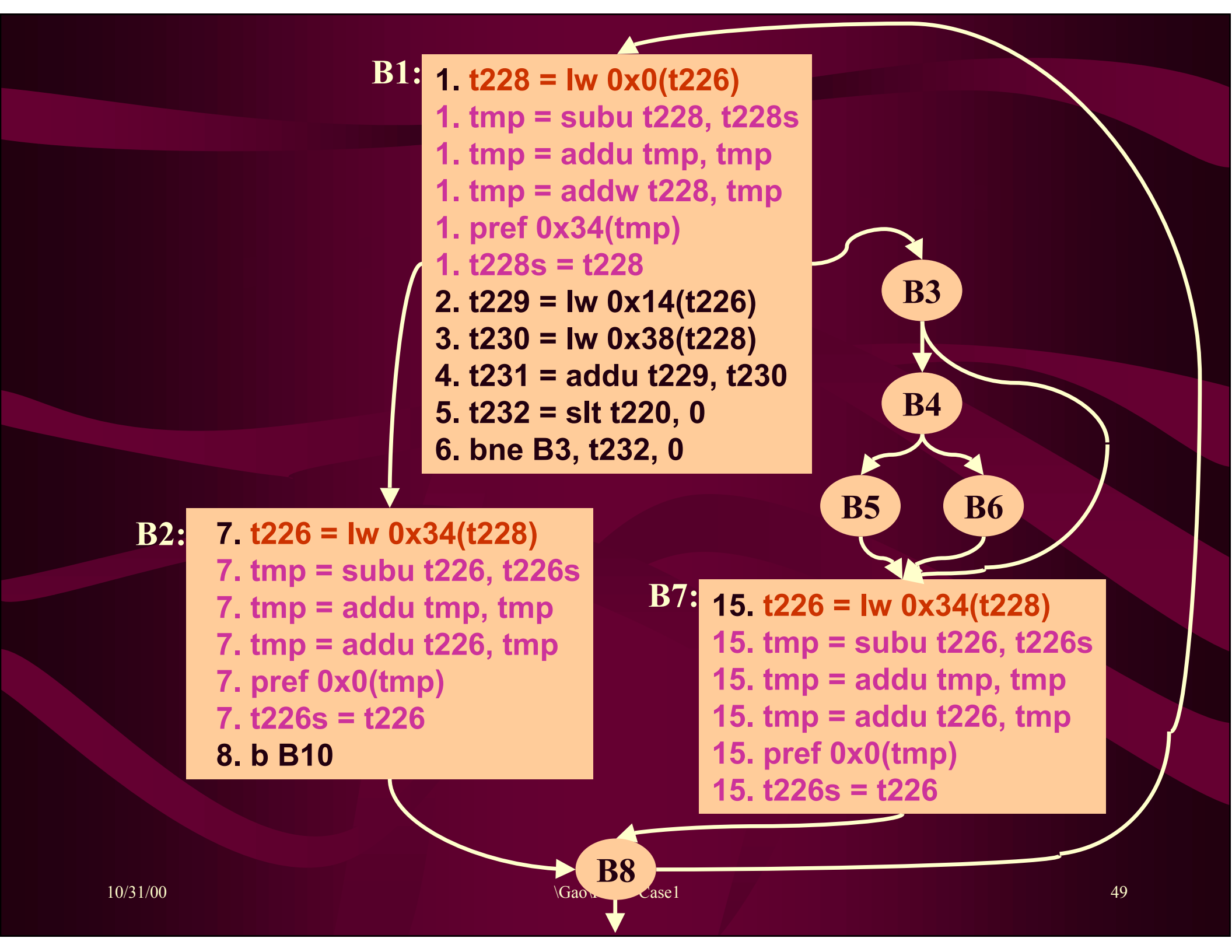
```
1 while (arcin){
2         tail = arcin->tail;
3         if (tail->time + arcin->org_cost > latest){
4           arcin = (arc_t *)tail->mark;
5           continue;
           }
6         arc_cost = tail->potential + head_potential;
7         if (red_cost < 0) {
8           if (new_arcs < MAX_NEW_ARCS){
9             insert_new_arc(arcnew, new_arcs, tail,
                                    head, arc_cost, red_cost);
10            new_arcs++;
             }
11          else if((cost_t)arcnew[0].flow > red_cost)
12                  replace_weaker_arc(arcnew, tail, head,
                                              arc_cost, red_cost);
           }
13        arcin = (arc_t *)tail->mark;
         }
```


B1:
1. t228 = lw 0x0(t226)
2. t229 = lw 0x14(t226)
3. t230 = lw 0x38(t228)
4. t231 = addu t229, t230
5. t232 = slt t220, 0
6. bne B3, t232, 0
B2:
7. t226 = lw 0x34(t228)
8. b B8
B3:
9. t234 = lw 0x2c(t228)
10. t235 = subu t225, t234
11. t233 = addiu t235, 0x1e
12. bgez B7, t233
B4:
13. t236 = slt t209. t175
14. Beq B6, t236, 0
B5:
insert_new_arc();
B6:
replace_weaker_arc();
B7:
15. t226 = lw 0x34(t228)
B8:
15. bne B1, t226, 0


B1:
1. t228 = lw 0x0(t226)
2. t229 = lw 0x14(t226)
3. t230 = lw 0x38(t228)
4. t231 = addu t229, t230
5. t232 = slt t220, 0
6. bne B3, t232, 0
B2:
7. t226 = lw 0x34(t228)
8. b B8
B3:
9. t234 = lw 0x2c(t228)
10. t235 = subu t225, t234
11. t233 = addiu t235, 0x1e
12. bgez B7, t233
B4:
13. t236 = slt t209. t175
14. Beq B6, t236, 0
B5:
insert_new_arc();
B6:
replace_weaker_arc();
B7:
15. t226 = lw 0x34(t228)
B8:
15. bne B1, t226, 0


B1:
1. t228 = lw 0x0(t226)
2. t229 = lw 0x14(t226)
3. t230 = lw 0x38(t228)
4. t231 = addu t229, t230
5. t232 = slt t220, 0
6. bne B3, t232, 0
B2:
7. t226 = lw 0x34(t228)
8. b B8
B3:
9. t234 = lw 0x2c(t228)
10. t235 = subu t225, t234
11. t233 = addiu t235, 0x1e
12. bgez B7, t233
B4:
13. t236 = slt t209. t175
14. Beq B6, t236, 0
B5:
insert_new_arc();
B6:
replace_weaker_arc();
B7:
15. t226 = lw 0x34(t228)
B8:
15. bne B1, t226, 0


B1:
1. t228 = lw 0x0(t226)
2. t229 = lw 0x14(t226)
3. t230 = lw 0x38(t228)
4. t231 = addu t229, t230
5. t232 = slt t220, 0
6. bne B3, t232, 0
B3
B4
B5
B6
B2:
7. t226 = lw 0x34(t228)
8. b B10
B7:
15. t226 = lw 0x34(t228)
B8


B1:
1. t228 = lw 0x0(t226)
1. tmp = subu t228, t228s
1. tmp = addu tmp, tmp
1. tmp = addw t228, tmp
1. pref 0x34(tmp)
1. t228s = t228
2. t229 = lw 0x14(t226)
3. t230 = lw 0x38(t228)
4. t231 = addu t229, t230
5. t232 = slt t220, 0
6. bne B3, t232, 0
B2:
7. t226 = lw 0x34(t228)
7. tmp = subu t226, t226s
7. tmp = addu tmp, tmp
7. tmp = addu t226, tmp
7. pref 0x0(tmp)
7. t226s = t226
8. b B10
B3
B4
B5
B6
B7:
15. t226 = lw 0x34(t228)
15. tmp = subu t226, t226s
15. tmp = addu tmp, tmp
15. tmp = addu t226, tmp
15. pref 0x0(tmp)
15. t226s = t226
B8

When Pointer Prefetch Works

Bench mark	Execution Time			Performance Improvement	
	No Prefetch	Without Analysis	Analysis	Without Analysis	Analysis
mcf	3,396 s	2,854 s	2,699 s	16.0%	20.5%
ft	517 s	436 s	333 s	15.6%	35.5%
mlp	632 s	333 s	538 s	8.3%	14.9%
vpr	1,771 s	-	1,761 s	-	0.6%
twolf	2,540 s	2,657 s	2,531 s	-4.6%	0.4%

When Pointer Prefetch Does Not Help

Bench mark	Execution Time			Performance Improvement	
	No Prefetch	Without Analysis	Analysis	Without Analysis	Analysis
gap	1,174 s	-	1,207 s	-	-2.8%
li	285 s	293 s	292 s	-2.8%	-2.5%
perlbmk	2,062 s	-	2,104 s	-	-2.0%
eon	949 s	-	959 s	-	-1.1%
parser	2,180 s	333 s	538 s	-3.0%	-0.5%
gcc	122 s	123 s	122 s	-0.7%	-0.1%

Summary of Attributes

- Software-only implementation
- Simple candidate identification
- Simple code transformation
- No impact on user data structures
- Simple profitability analysis, local to loop
- Performance degradations are rare, minor

Open Questions

- How often is the speculated stride correct?
- Can instrumentation feedback help?
- How well does the speculative prefetch work with other recursive data structures: trees, graphs, etc?
- How well does this approach work for read/write recursive data structures?

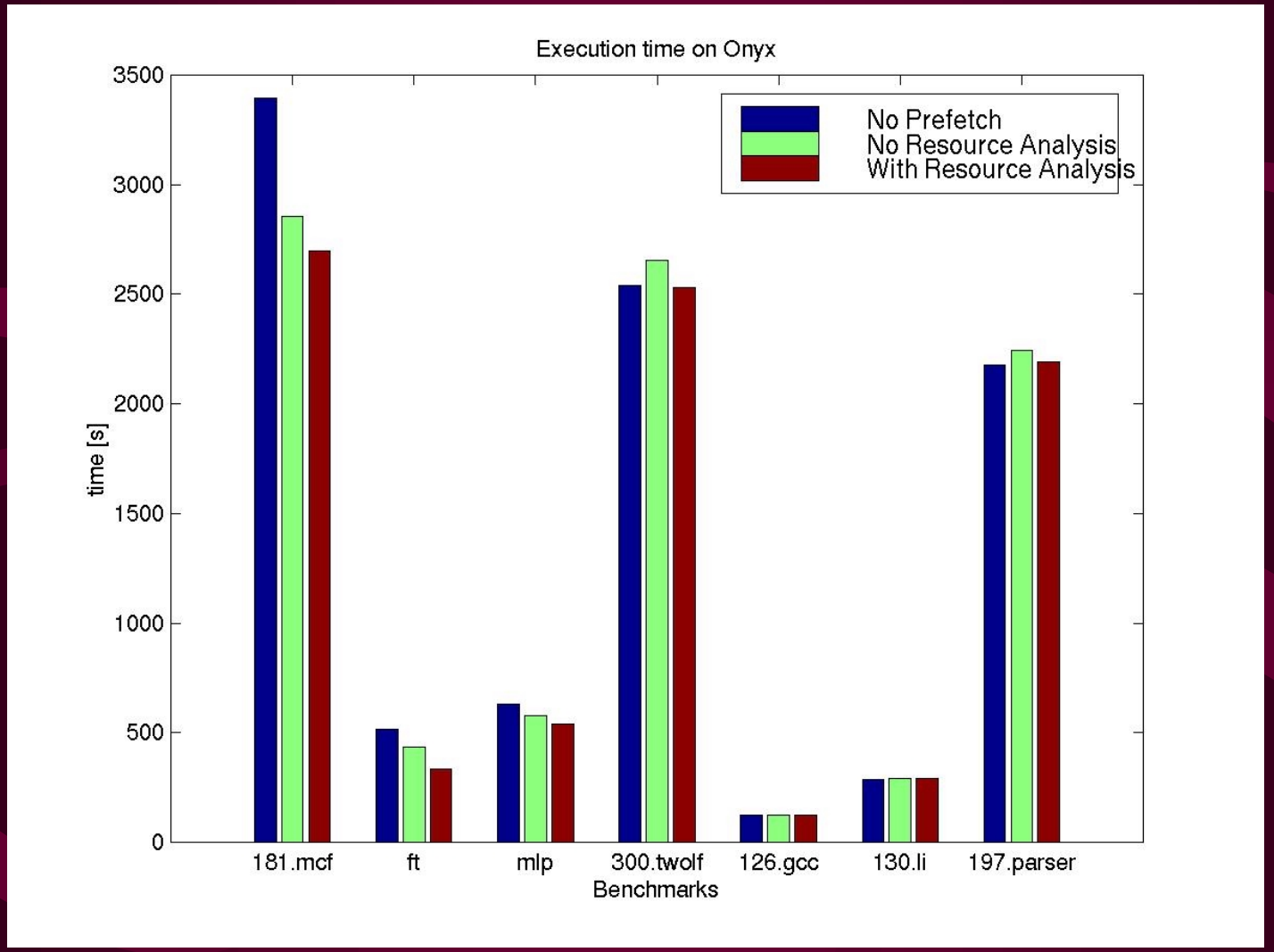
Related Work (Software)

- Luk-Mowry *(ASPLOS-96)*
  - Greedy prefetching; History-Pointer prefetching; Data Linearization Prefetching;
    - Change the data structure storage;
- Lipatsi *et al. (Micro-95)*
  - Prefetching pointers at procedure call sites;
- Liu-Dimitri-Kaeli *(Journal of Syst. Arch.-99)*
  - Maintains a table of offsets for prefetching

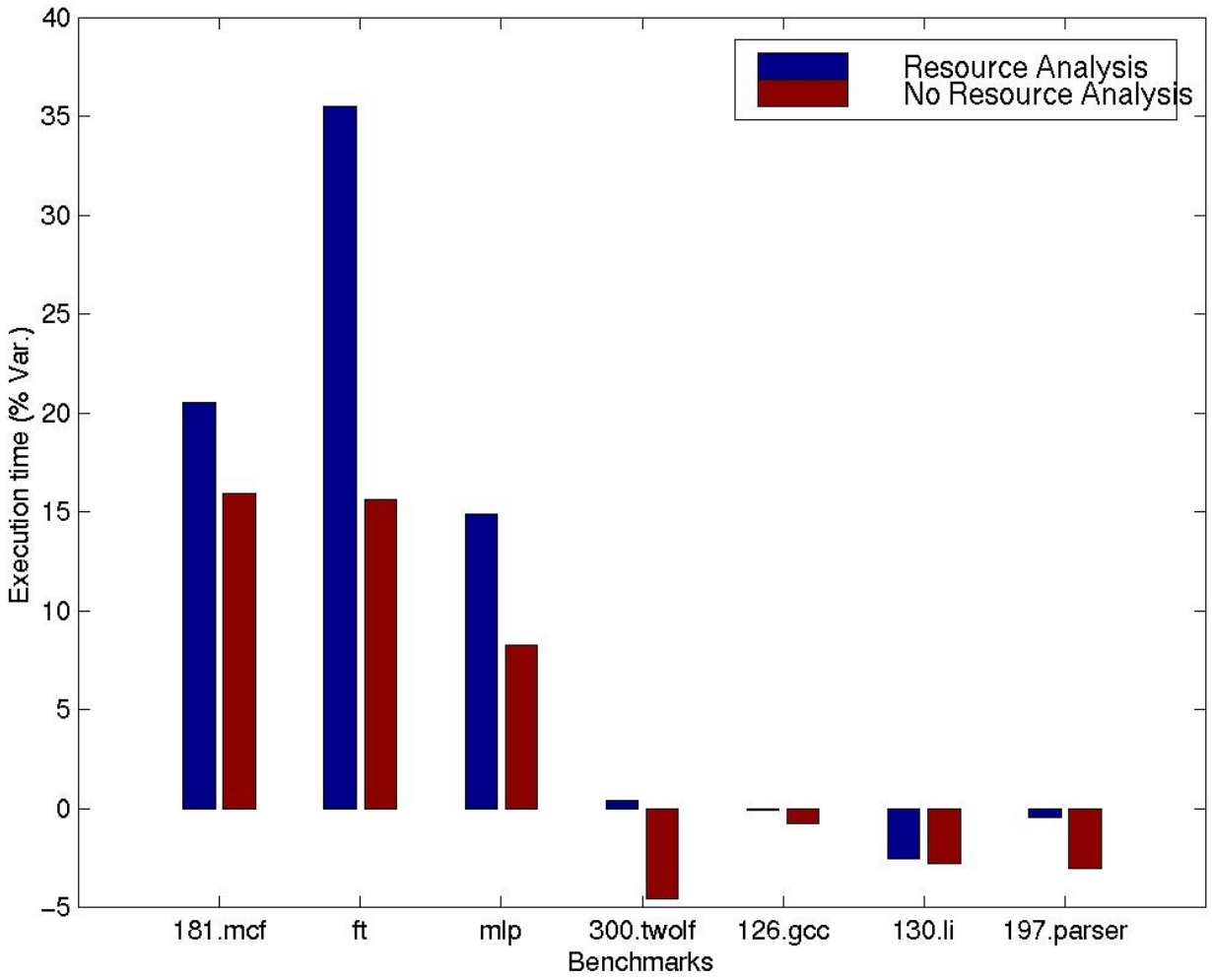
Related Work (Hardware)

- Roth-Moshovos-Sohi *(ASPLOS, 1998)*
- Gonzales-Gonzales (*ICS, 1997)*
- Mehrotra *(Urbana-Champaign, 1996)*
- Chen-Baer *(Trans. Computer, 1995)*
- Charney-Reeves (*Trans. Comp., 1994)*
- Jegou-Teman *(ICS, 1993)*
- Fu-Patel *(Micro, 1992)*

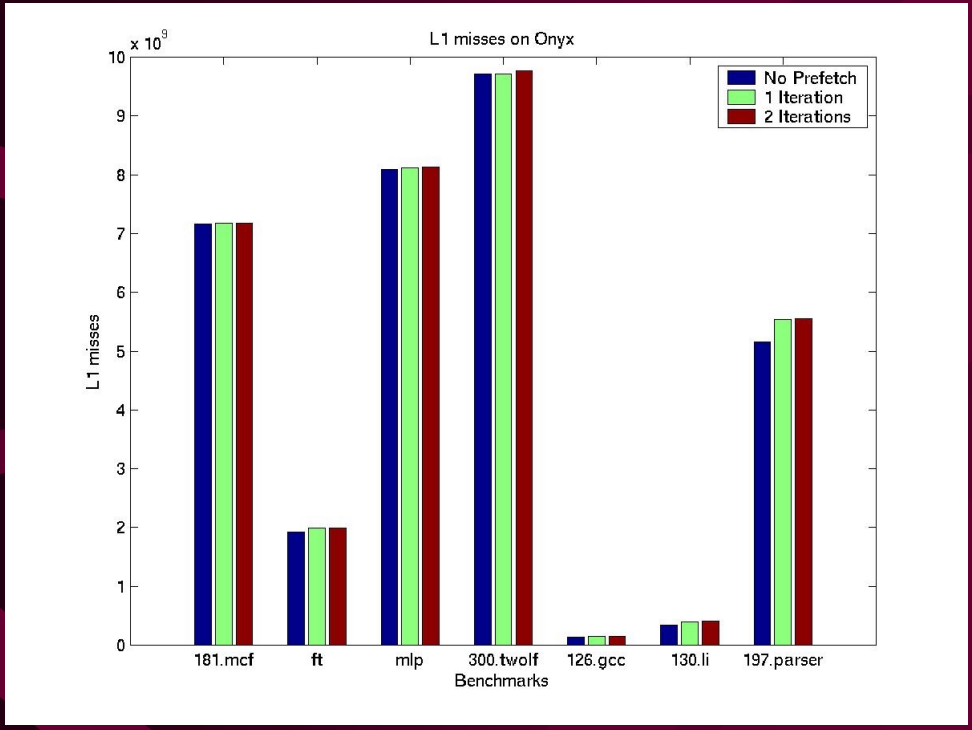
Execution Time Measurements

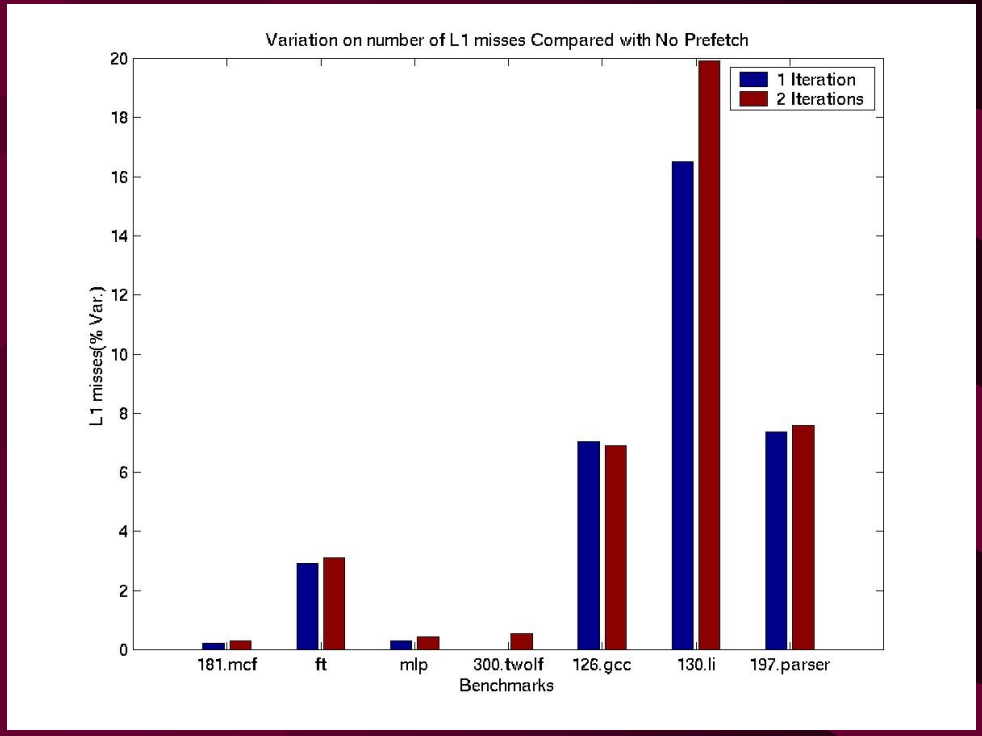


Prefetch Improvement

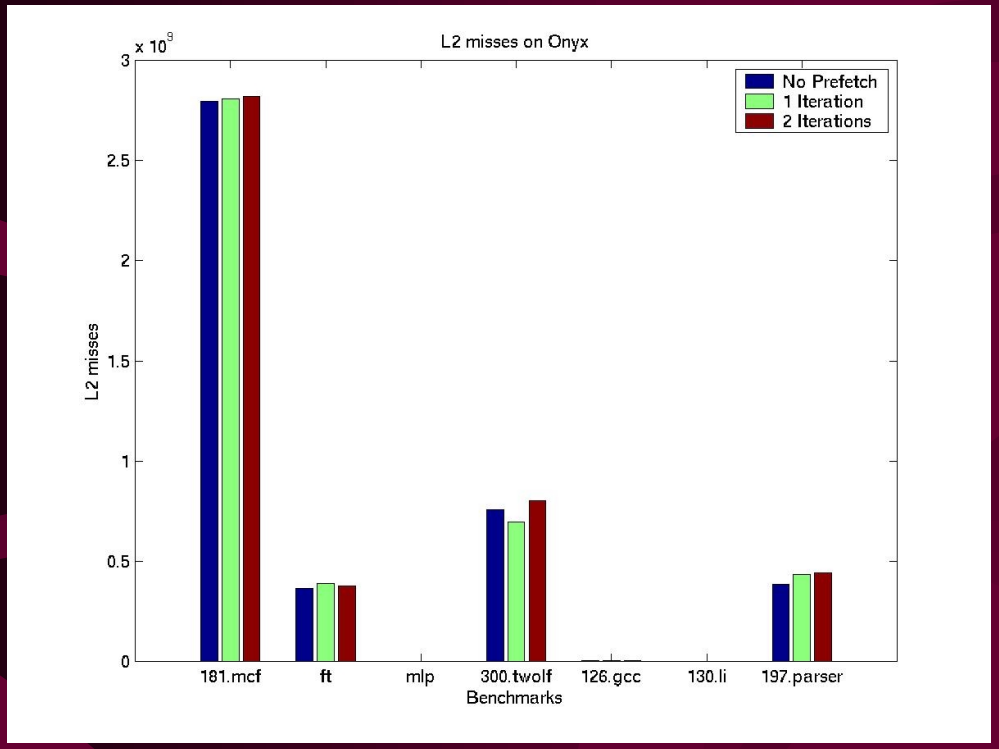


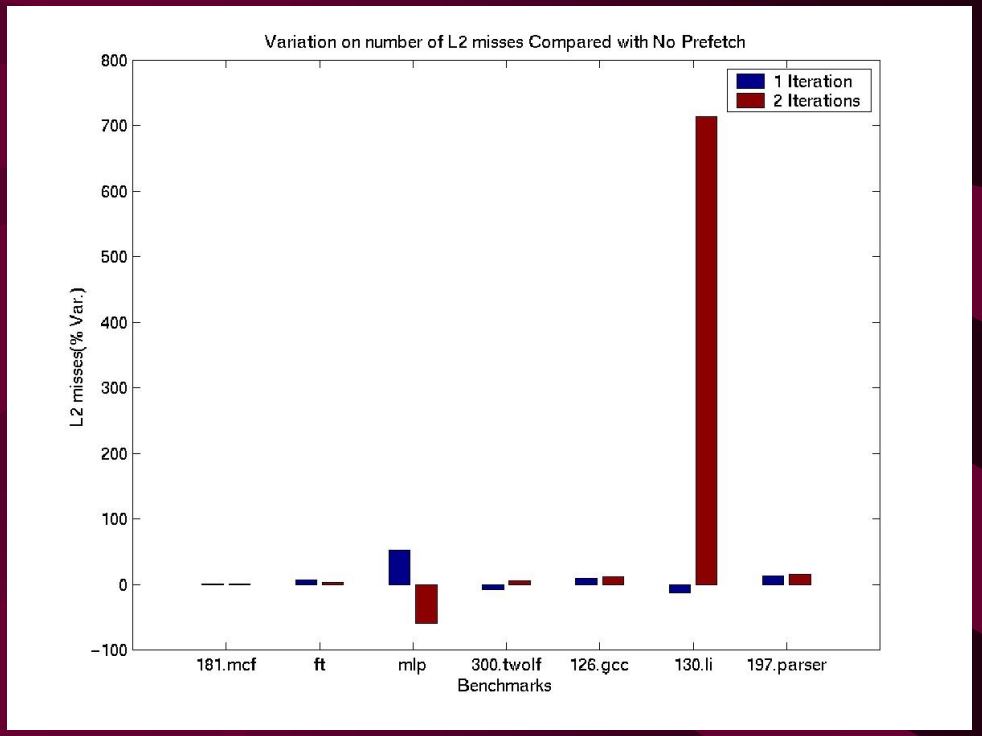
L1 Cache Misses



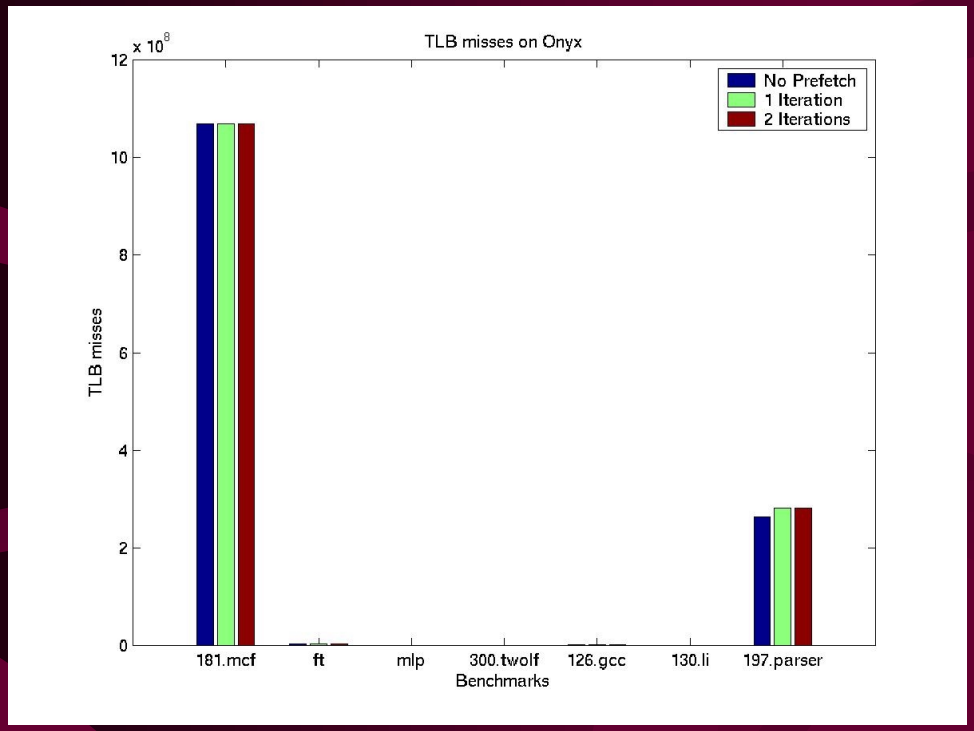


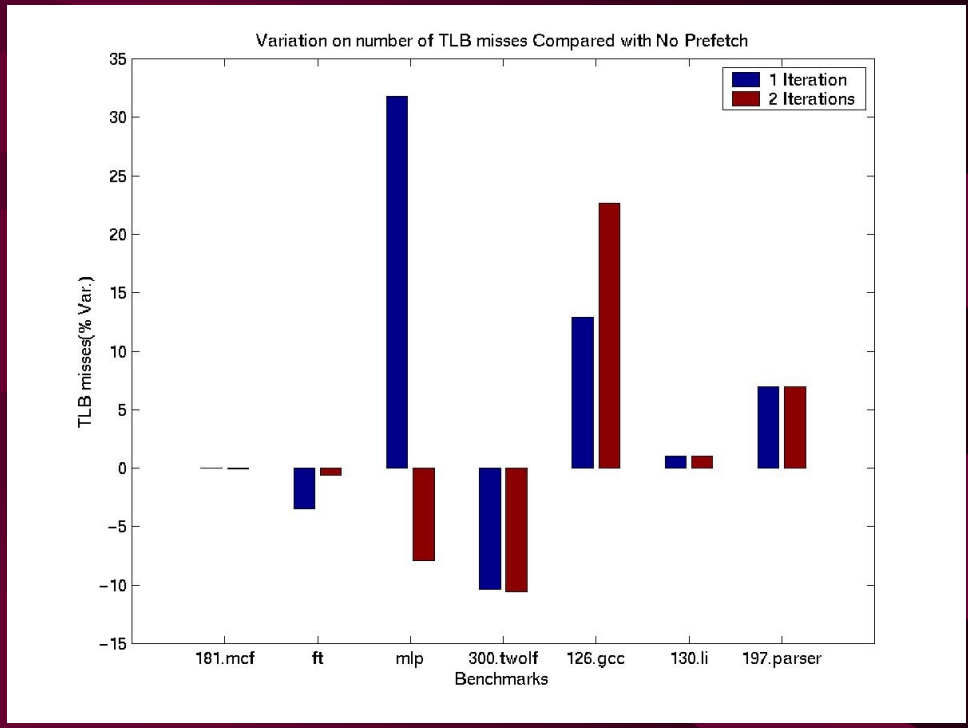
L2 Cache Misses





TLB Misses


TLB misses on Onyx
x 10^8
TLB misses
Benchmarks
No Prefetch
1 Iteration
2 Iterations
181.mcf
ft
mlp
300.twolf
126.gcc
130.li
197.parser


Variation on number of TLB misses Compared with No Prefetch
TLB misses(% Var.)
Benchmarks
1 Iteration
2 Iterations
181.mcf
ft
mlp
300.twolf
126.gcc
130.li
197.parser

Benchmarks

gcc	GNU C compiler
li	Lisp interpreter
mcf	Minimal cost flow solver
parser	Syntactic parser of English
twolf	Place and route simulator
mlp	Multi-layer perceptron simulator
ft	Minimum spanning tree algorithm

Targeting Pro64 to a New Processor

For -O0

- **Create a new targ_info**
- Adjust configuration file for ABI
- Create a new WHIRL-to-CG-lower for instruction selection

Advanced

- Adjust CG utilities (e.g., predication, EBO patterns, SWP stuff, etc.)