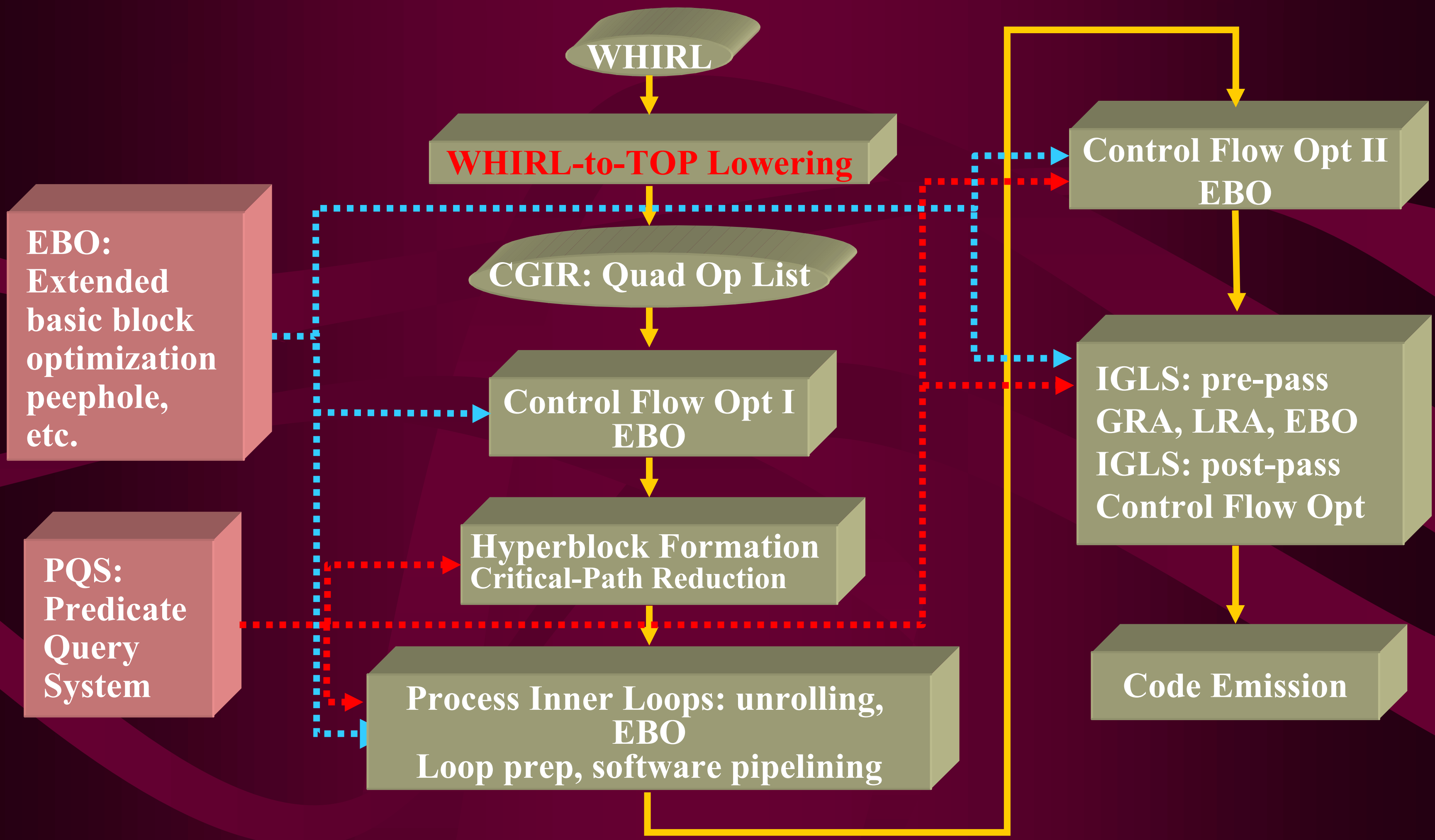
PART II

Overview of The Pro64 Code Generator

Outline

- Code generator flow diagram
- WHIRL/CGIR and TARG-INFO
- Hyperblock formation and predication (HBF)
- Predicate Query System (PQS)
- Loop preparation (CGPREP) and software pipelining
- Global and local instruction scheduling (IGLS)
- Global and local register allocation (GRA, LRA)

Flowchart of Code Generator


WHIRL
WHIRL-to-TOP Lowering
CGIR: Quad Op List
Control Flow Opt I EBO
Hyperblock Formation Critical-Path Reduction
Process Inner Loops: unrolling, EBO Loop prep, software pipelining
Control Flow Opt II EBO
IGLS: pre-pass GRA, LRA, EBO IGLS: post-pass Control Flow Opt
Code Emission
EBO: Extended basic block optimization peephole, etc.
PQS: Predicate Query System

WHIRL

- Abstract syntax tree based
- Symbol table links, map annotations
- Base representation is simple and efficient
- Used through several phases with lowering
- Designed for multiple target architectures

Code Generation Intermediate Representation (CGIR)

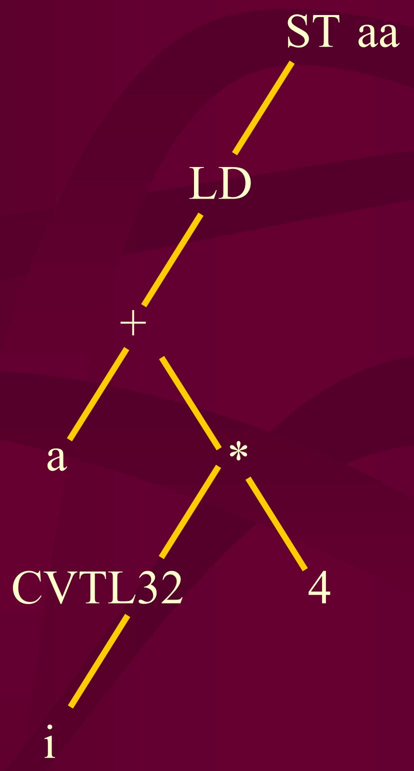
- TOPs (Target Operations) are “quads”
- Operands/results are TNs
- Basic block nodes in control flow graph
- Load/store architecture
- Supports predication
- Flags on TOPs (copy ops, integer add, load, etc.)
- Flags on operands (TNs)

From WHIRL to CGIR

An Example

```
int    *a;
int     i;
int     aa;
aa  =  a[i];
```

(a) Source



(b) WHIRL

$T_1 = sp + \&a;$

$T_2 = ld \quad T_1$

$T_3 = sp + \&i;$

$T_4 = ld \quad T_3$

$T_5 = sxt \quad T_4$

$T_6 = T_5 << 2$

$T_7 = T_6$

$T_8 = T_2 + T_7$

$T_9 = ld \quad T_8$

$T_{10} = sp + \&aa$

$:= st \; T_{10} \; T_9$

(c) CGIR

From WHIRL to CGIR Cont'd

- Information passed
  - alias information
  - loop information
  - symbol table and maps

The Target Information Table (TARG_INFO)

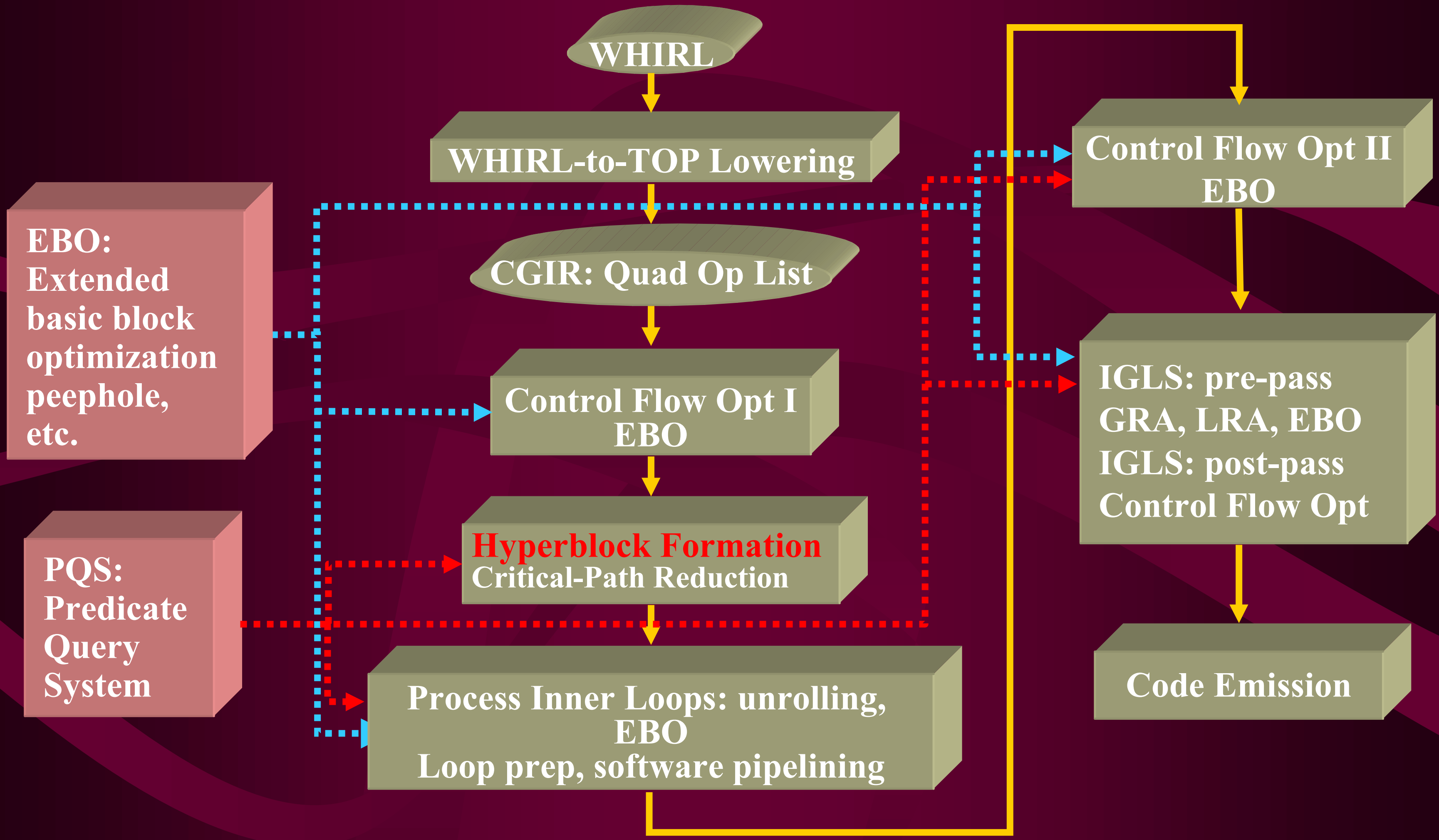
Objective:

- Parameterized description of a target machine and system architecture
- Separates architecture details from the compiler’s algorithms
- Minimizes compiler changes when targeting a new architecture

The Target Information Table (TARG_INFO) Cont'd

- Based on an extension of Cydra tables, with major improvements
- Architecture models have already targeted:
  - Whole MIPS family
  - IA-64
  - IA-32
  - SGI graphics processors (earlier version)

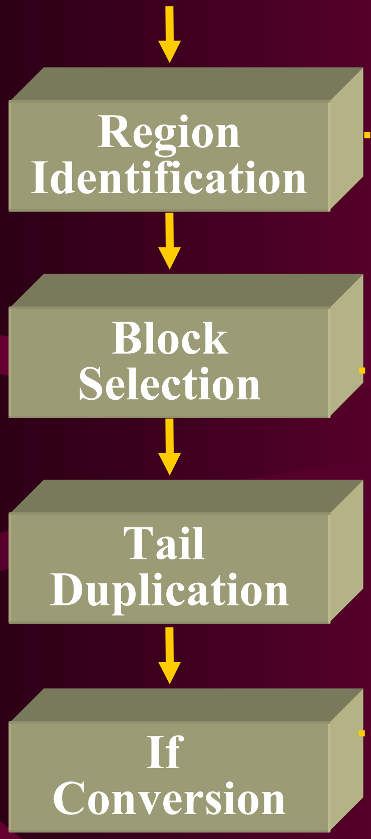
Flowchart of Code Generator



Hyperblock Formation and Predicated Execution

- Hyperblock single-entry multiple-exit control-flow region:
  - loop body, hammock region, etc.
- Hyperblock formation algorithm
  - Based on Scott Mahlke's method *[Mahlke96]*
  - But, less aggressive tail duplication

Hyperblock Formation Algorithm



- Hammock regions
- Innermost loops
- General regions (path based)

- Paths sorted by priorities (freq., size, length, etc.)
- Inclusion of a path is guided by its impact on resources, scheduling height, and priority level

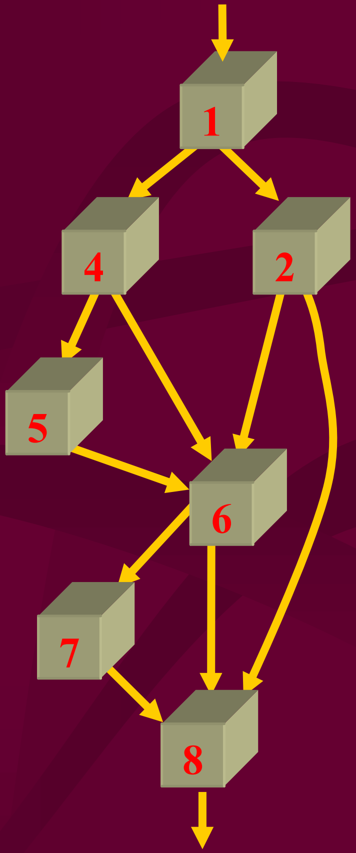
- Internal branches are removed via predication
- Predicate reuse

Objective: Keep the scheduling height close to that of the highest priority path.

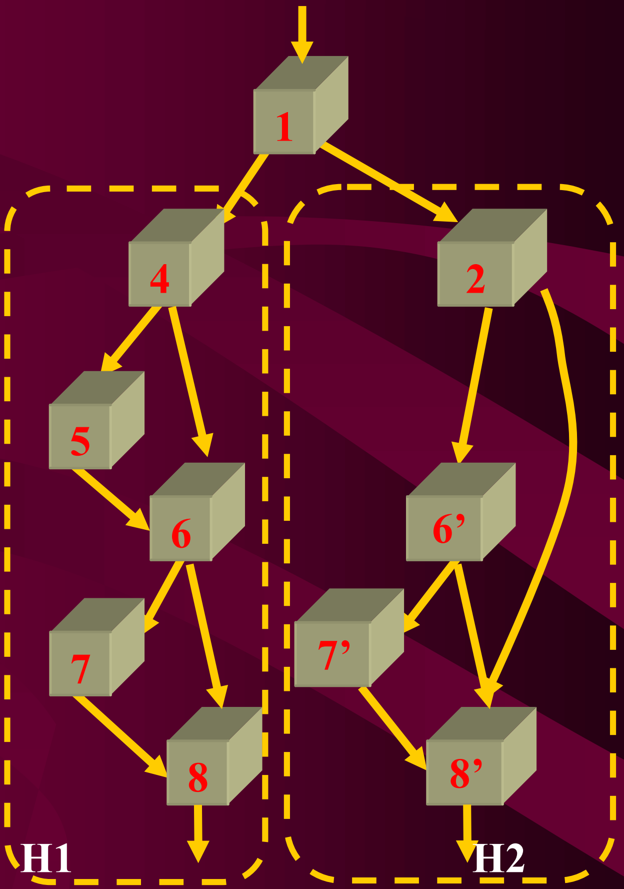
Hyperblock Formation - An Example

```
1    { aa = a[i];
       bb = b[i];
       switch (aa)  {
       case 1:
4,5  {     if (aa < tabsiz)
              aa = tab[aa];
2    { case 2:
6,7  {     if (bb < tabsiz)
              bb = tab[bb];
8    { default:
           ans = aa + bb;
```

(a) Source

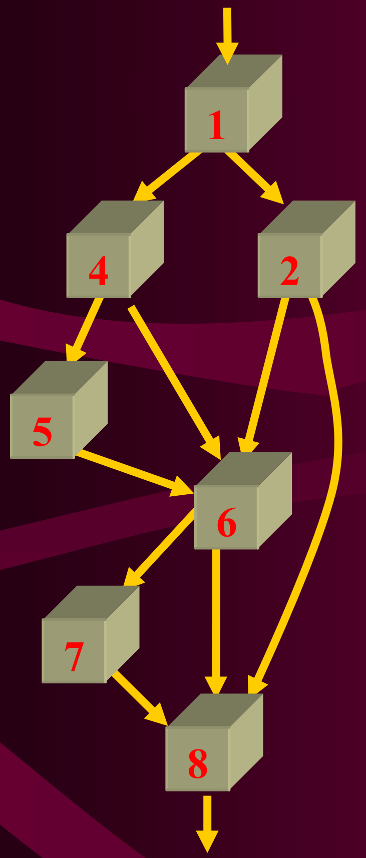


(b) CFG

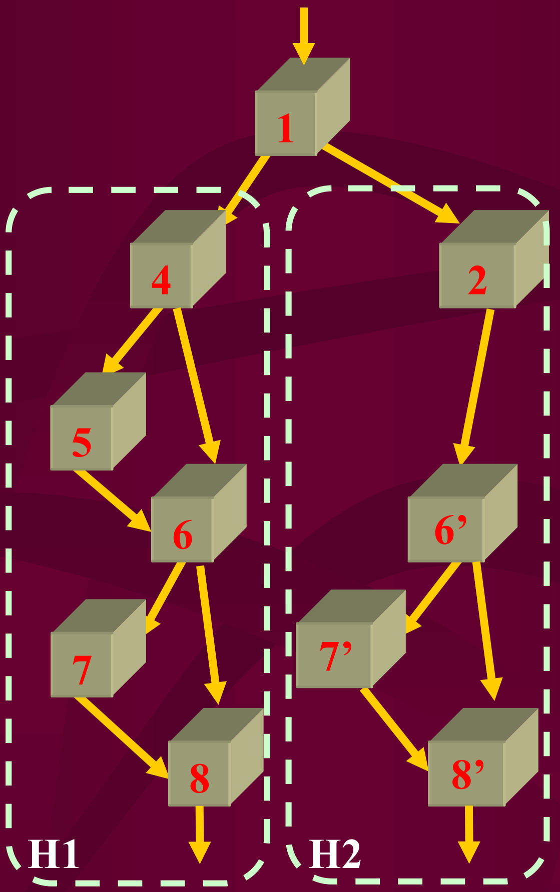


(c) Hyperblock formation with aggressive tail duplication

Hyperblock Formation - An Example

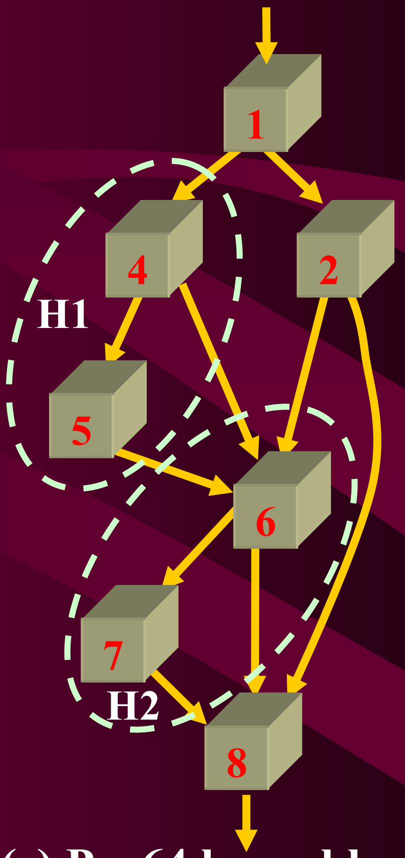


(a) CFG



(b) Hyperblock formation with aggressive tail duplication

Cont'd



(c) Pro64 hyperblock formation

Features of the Pro64 Hyperblock Formation (HBF) Algorithm

- Form “good” vs. “maximal” hyperblocks
- Avoid unnecessary duplication
- No reverse if-conversion
- Hyperblocks are not a barrier to global code motion later in IGLS

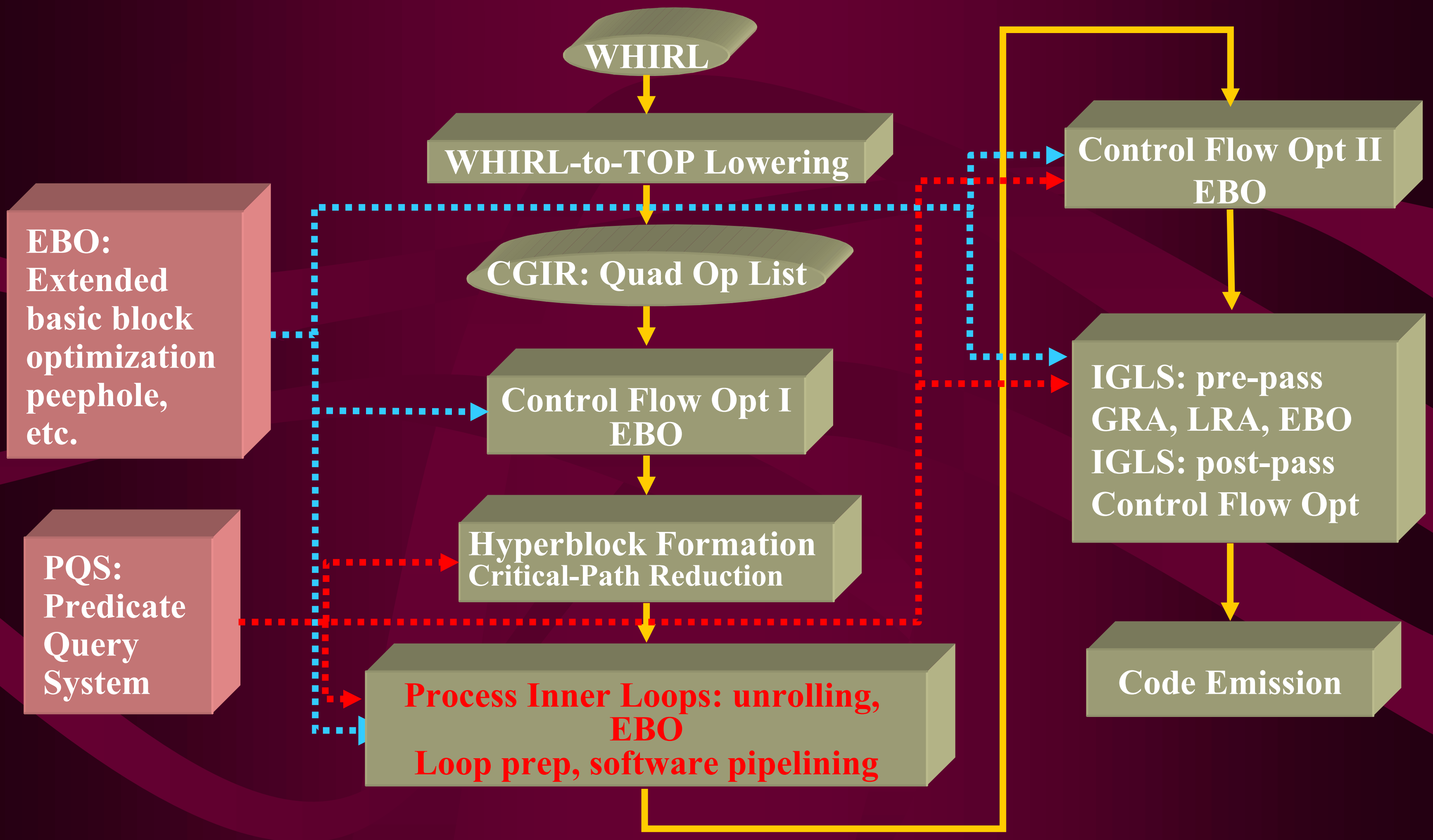
Predicate Query System (PQS)

- Purpose: gather information and provide interfaces allowing other phases to make queries regarding the relationships among predicate values
- PQS functions (examples)

  BOOL PQSCG_is_disjoint (PQS_TN tn_1, PQS_TN tn_2)

  BOOL PQSCG_is_subset (PQS_TN_SET& tns_1, PQS_TN_SET& tns_2)

Flowchart of Code Generator



Loop Preparation and Optimization for Software Pipelining

- Loop canonicalization for SWP
- Read/Write removal *(register aware)*
- Loop unrolling *(resource aware)*
- Recurrence removal or extension
- Prefetch
- Forced if-conversion

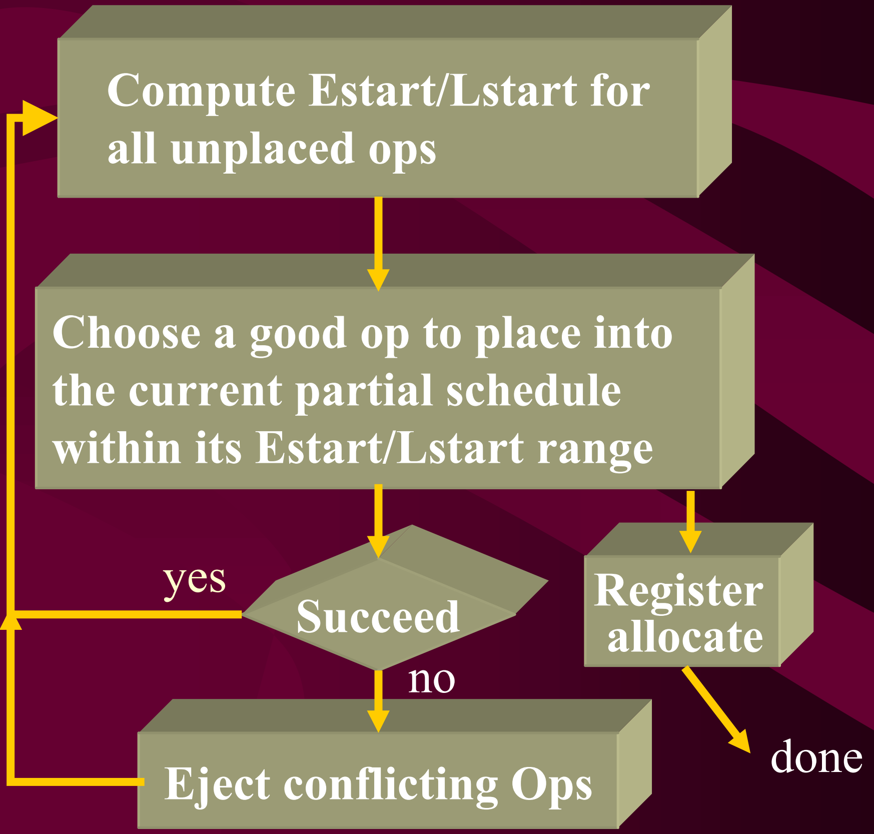
Pro64 Software Pipelining Method Overview

- Test for SWP-amenable loops
- Extensive loop preparation and optimization before application *[DeTo93]*
- Use lifetime sensitive SWP algorithm *[Huff93]*
- Register allocation after scheduling based on Cydra 5 *[RLTS92, DeTo93]*
- Handle both while and do loops
- Smooth switching to normal scheduling if not successful.

Pro64 Lifetime-Sensitive Modulo Scheduling for Software Pipelining

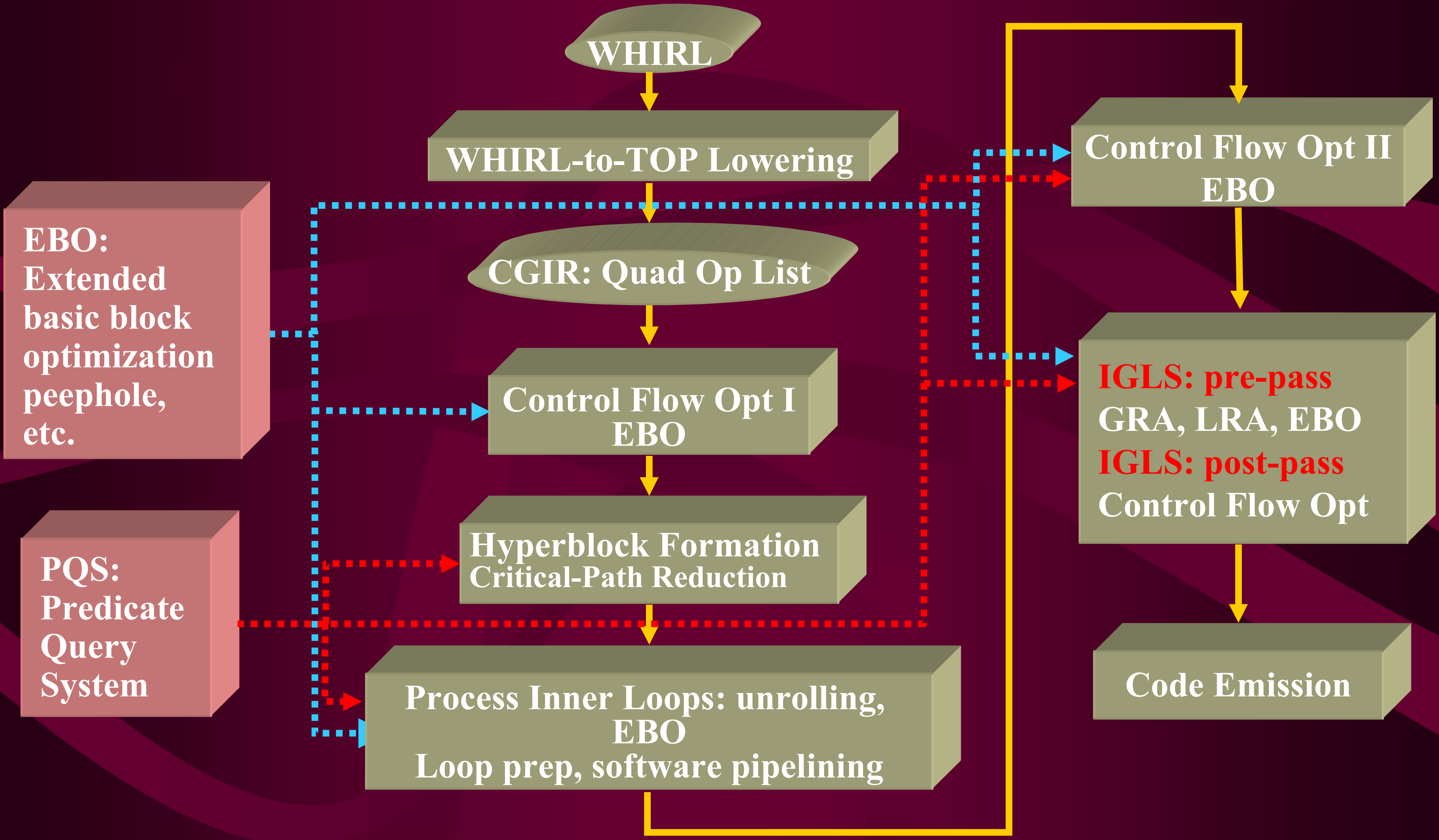
Features

- Try to place an op ASAP or ALAP to minimize register pressure
- Slack scheduling
- Limited backtracking
- Operation-driven scheduling framework



Compute Estart/Lstart for all unplaced ops
Choose a good op to place into the current partial schedule within its Estart/Lstart range
Succeed
yes
no
Register allocate
Eject conflicting Ops
done

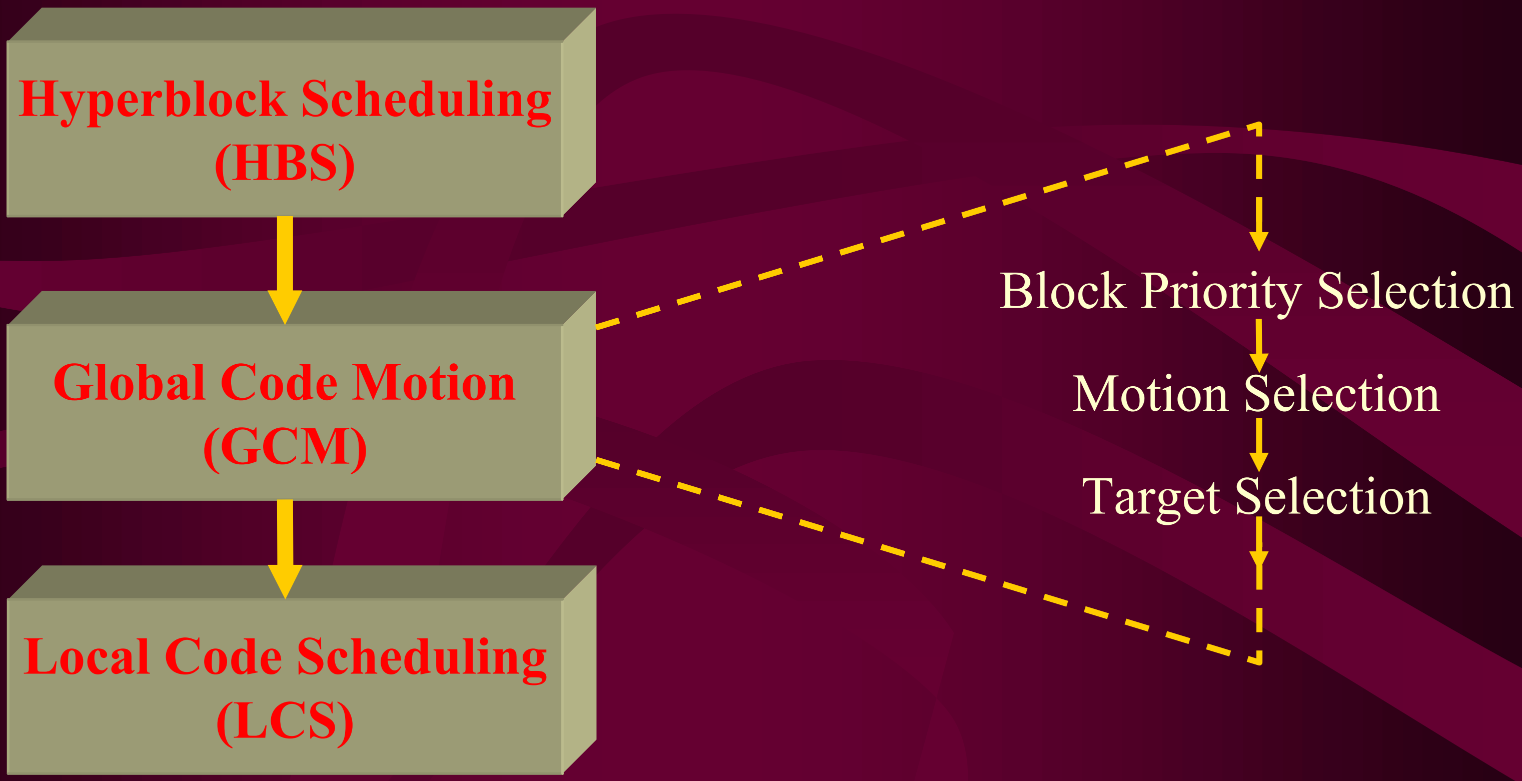
Flowchart of Code Generator



Integrated Global Local Scheduling (IGLS) Method

- The basic IGLS framework integrates global code motion (GCM) with local scheduling *[MaJD98]*
- IGLS extended to hyperblock scheduling
- Performs profitable code motion between hyperblock regions and normal regions

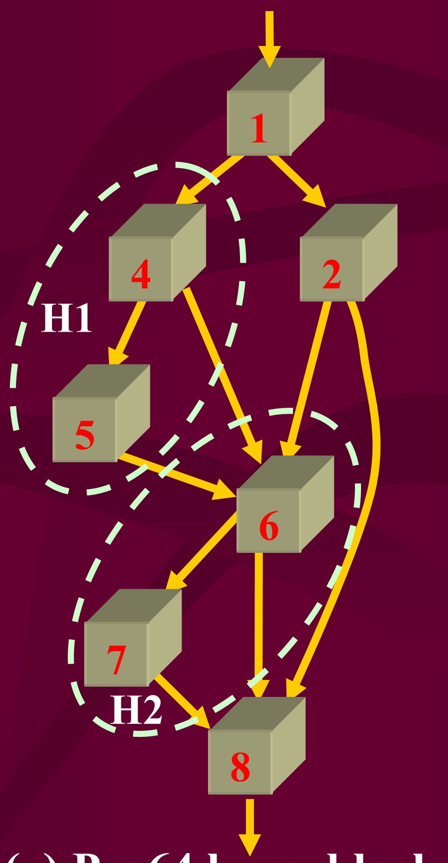
IGLS Phase Flow Diagram



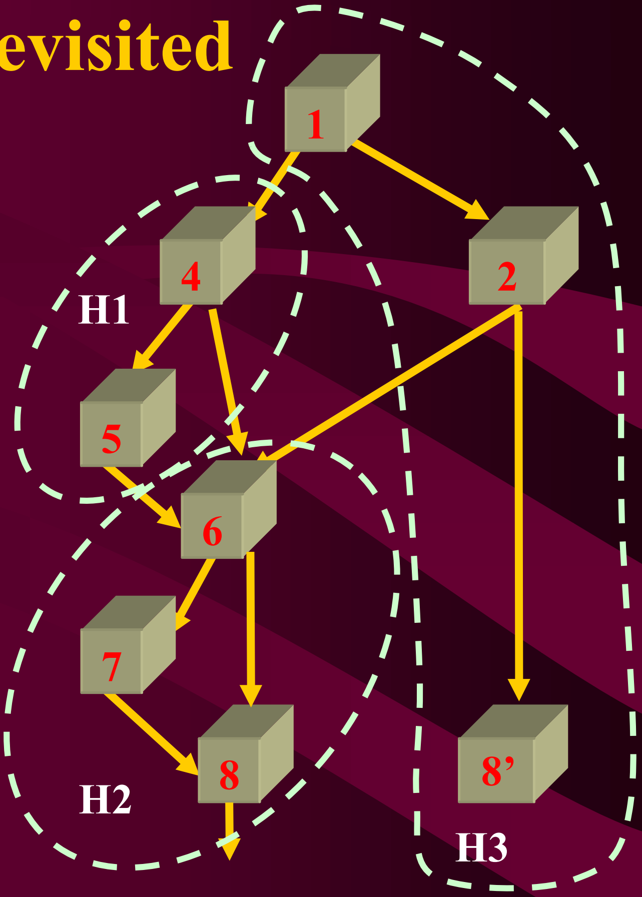
Advantages of the Extended IGLS

Method - The Example Revisited

- **Advantages**:
  - No rigid boundaries between hyperblocks and non-hyperblocks
  - GCM moves code into and out of a hyperblock according to profitability

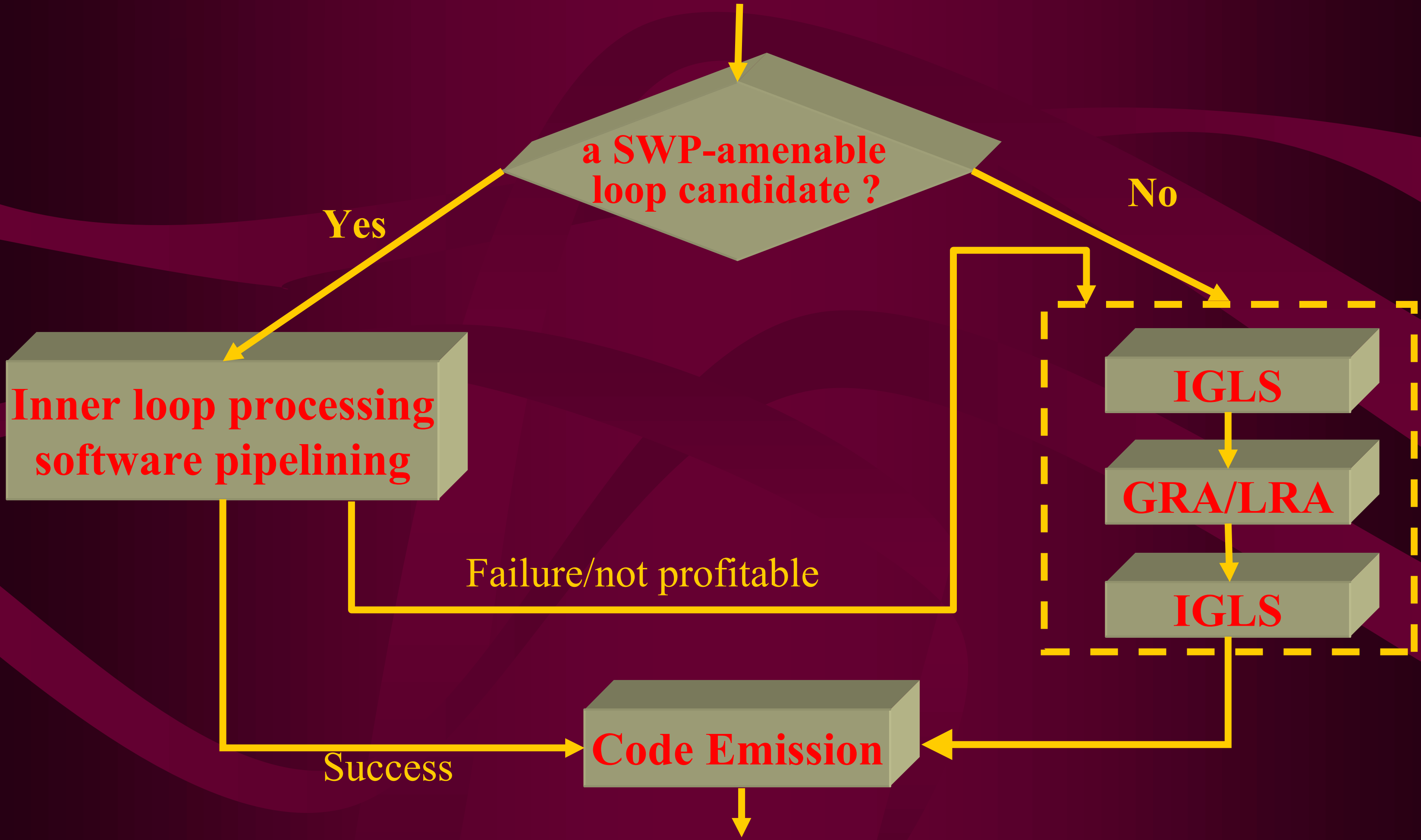


(a) Pro64 hyperblock

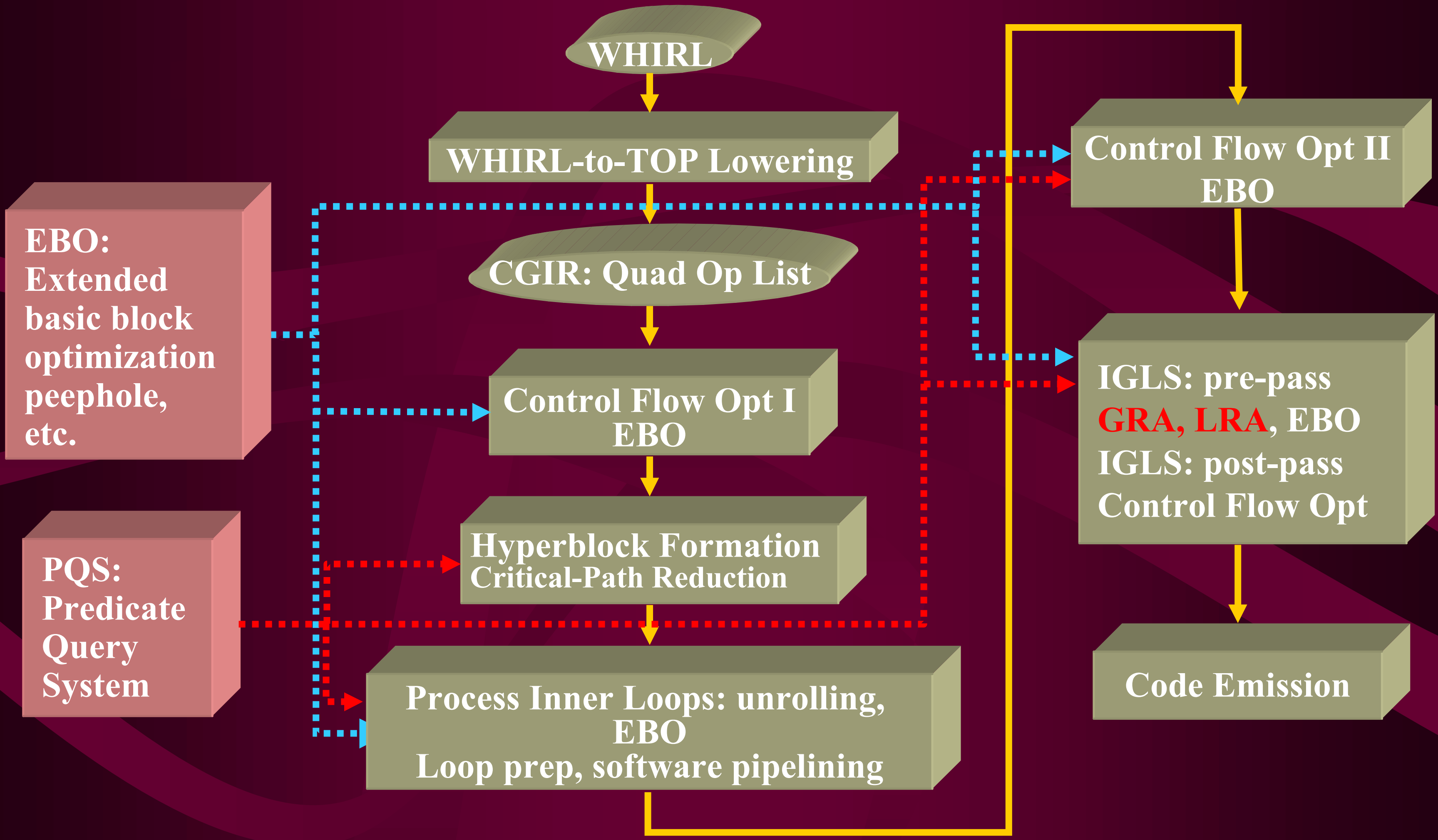


(b) Profitable duplication

Software Pipelining vs Normal Scheduling

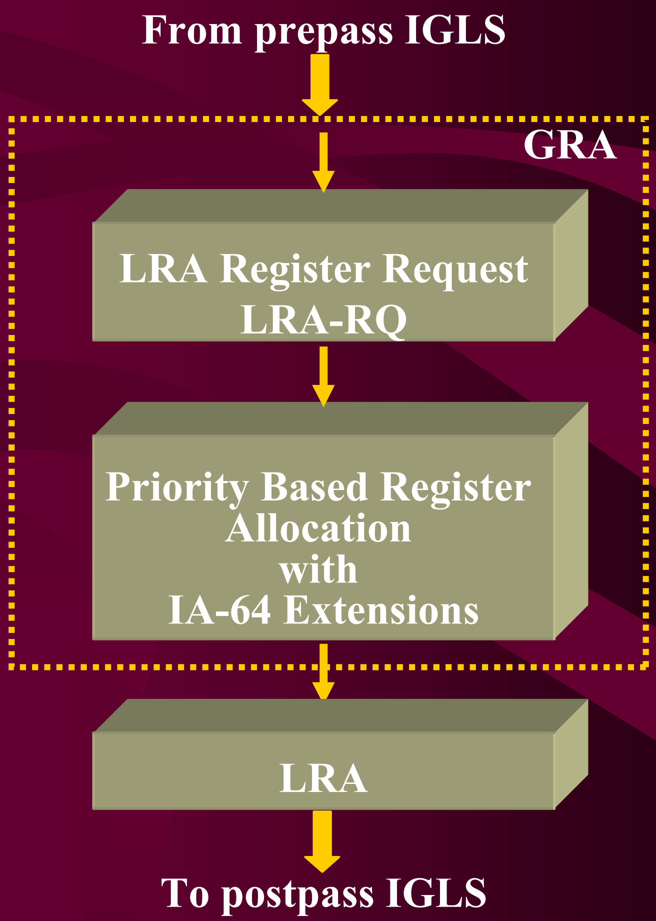

a SWP-amenable loop candidate ?
Yes
No
Inner loop processing software pipelining
IGLS
GRA/LRA
IGLS
Failure/not profitable
Success
Code Emission

Flowchart of Code Generator



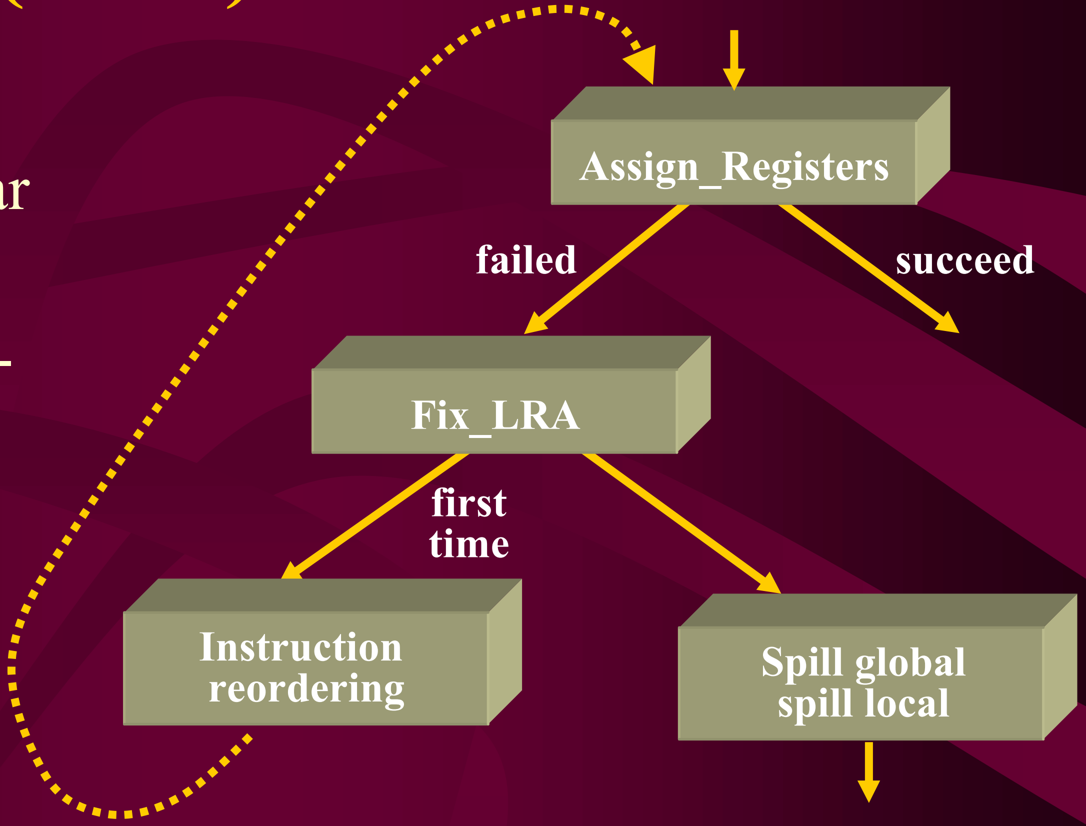
Global and Local Register Allocation (GRA/LRA)

- LRA-RQ provides an estimate of local register requirements
- Allocates global variables using a priority-based register allocator [ChowHennessy90,Chow83, Briggs92]
- Incorporates IA-64 specific extensions, e.g. register stack usage



Local Register Allocation (LRA)

- Assign_registers using reverse linear scan
- Reordering: depth-first ordering on the DDG



Future Research Topics for Pro64 Code Generator

- Hyperblock formation
- Predicate query system
- Enhanced speculation support