Instruction Selection for Compilers that Target Architectures with Echo Instructions

Philip Brisk
philip@cs.ucla.edu

Ani Nahapetian
ani@cs.ucla.edu

Majid Sarrafzadeh
majid@cs.ucla.edu

Embedded and Reconfigurable Systems Lab
Computer Science Department
University of California, Los Angeles

Outline

- Code Compression
- LZ77 Compression
- Echo Instructions
- Compiler Framework
- Experimental Results
- Summary

Code Compression

Why Compress a Program?

- Silicon Requirements for on-chip ROM
- Power Consumption
- Cost to Fabricate
- Cost Paid by the Consumer

Are there Costs to Decompression?

- Performance Overhead 
- Dedicated Hardware (or… Software)
- Longer CPU Pipelines
  - Increased Branch Misprediction Penalty

LZ77 Compression

To Compress a String, Identify Repeated Substrings and Replace Each with a Pointer

(Offset, Length of Sequence)

Length

ABCDBCABCDBACABCDBADAABCDBDC

28

ABCDBCABCDBACABCDBADAABCDBDC

one character

ABCDBC(6, 5)AC(9, 5)ADA(12, 5)DC

Echo Instructions

Decompresses LZ77-compressed Programs with Minimal Hardware Requirements

- **2 Dedicated Registers: R_1, R_2**
- **1 Decrementer with = 0 Test (NOR)**

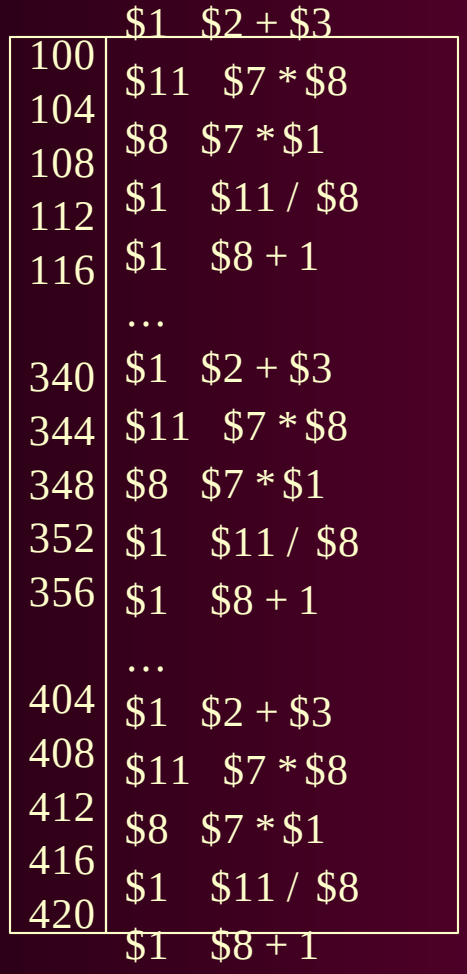
Echo(Offset, N)

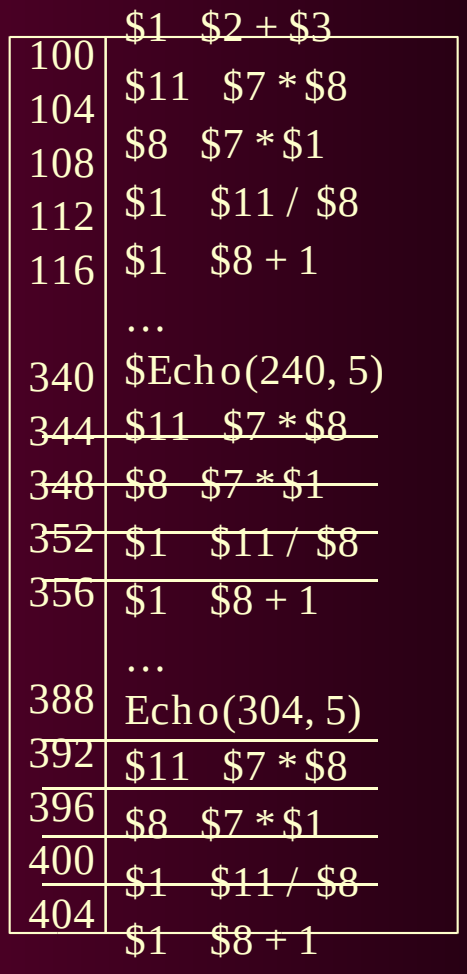
1. **Save PC and N in R_1 and R_2**
2. **Branch to PC – Offset**
3. **Execute the next N instructions**
4. **Return to the Call Point**
5. **Restore PC from R_1**

(Fraser, Microsoft '02)

(Lau, CASES '03)

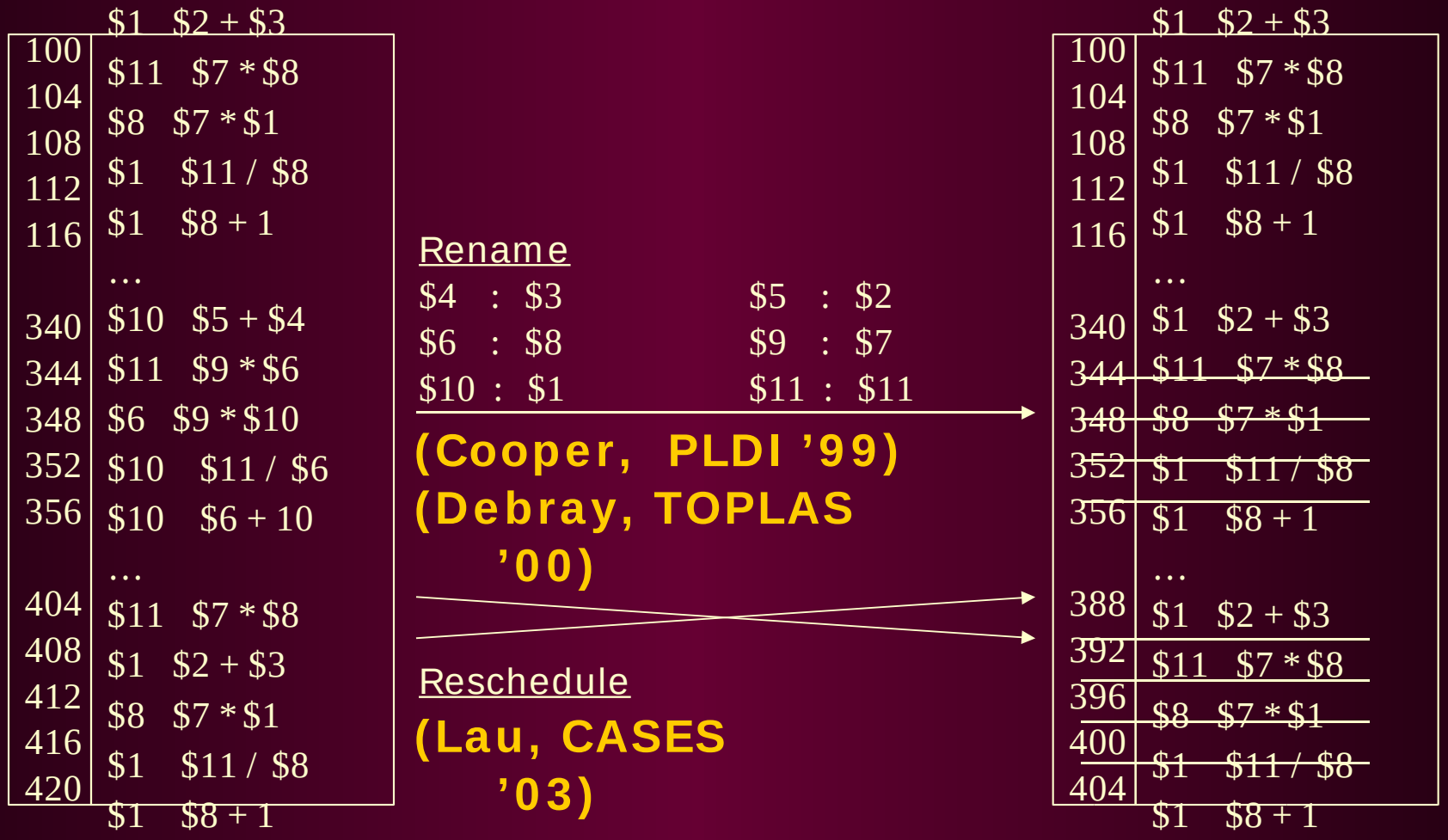
Substring Matching





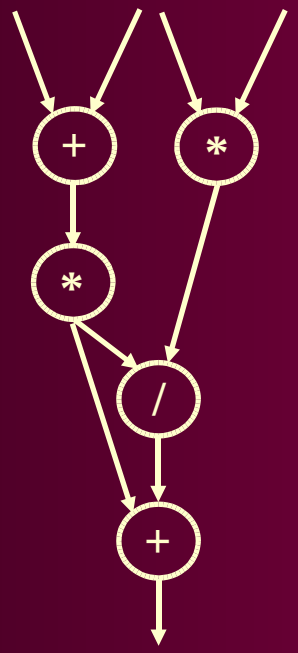
(Fraser, SCC '84)

Reschedule/Rename



DFG Isomorphism

```
        $1   $2 + $3
100     $11   $7 * $8
104     $8   $7 * $1
108     $1    $11 / $8
112     $1    $8 + 1
116     …
340     $10   $5 + $4
344     $11   $9 * $6
348     $6   $9 * $10
352     $10    $11 / $6
356     $10    $6 + 10
        …
404     $11   $7 * $8
408     $1   $2 + $3
412     $8   $7 * $1
416     $1    $11 / $8
420     $1    $8 + 1
```



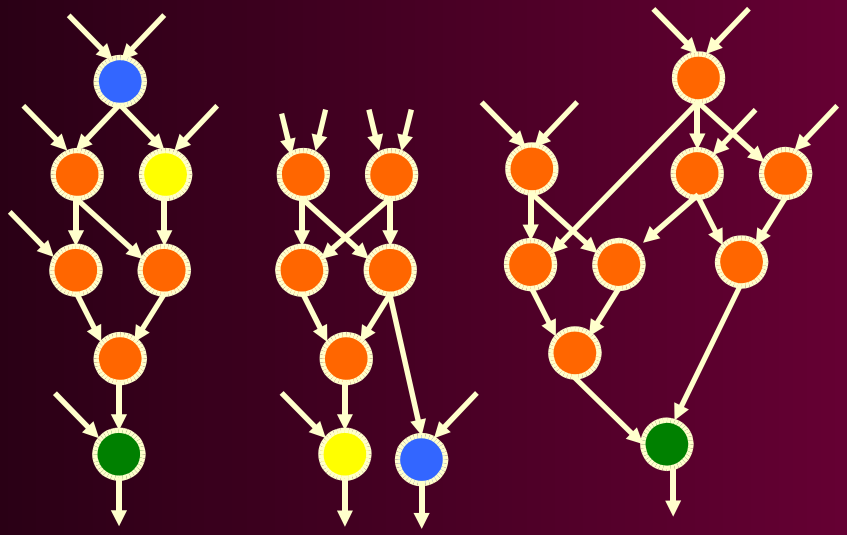
Our Approach

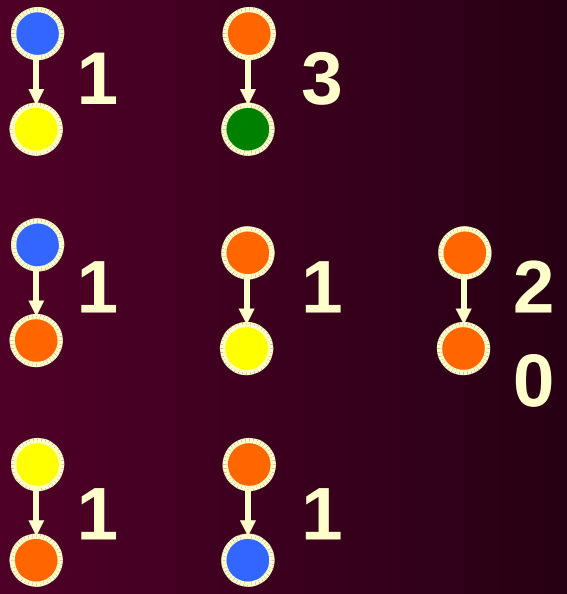
- Represent Sequences as Data Flow Graphs
- Identify Repeated Isomorphic Subgraphs
- Replace Subgraphs with Echo Instructions

Isomorphic Subgraph Identification

Edge Contraction (Kastner, ICCAD '01)

- **Consider a New Subgraph for Each DFG Edge**

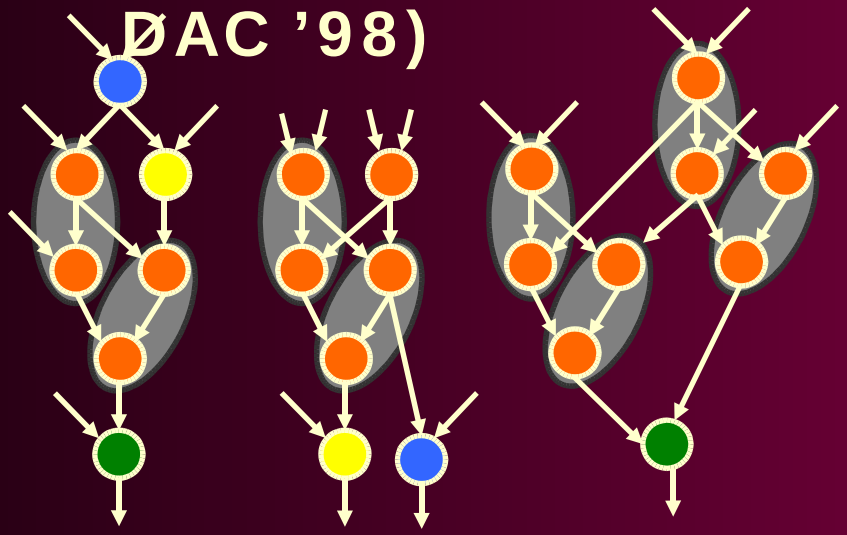


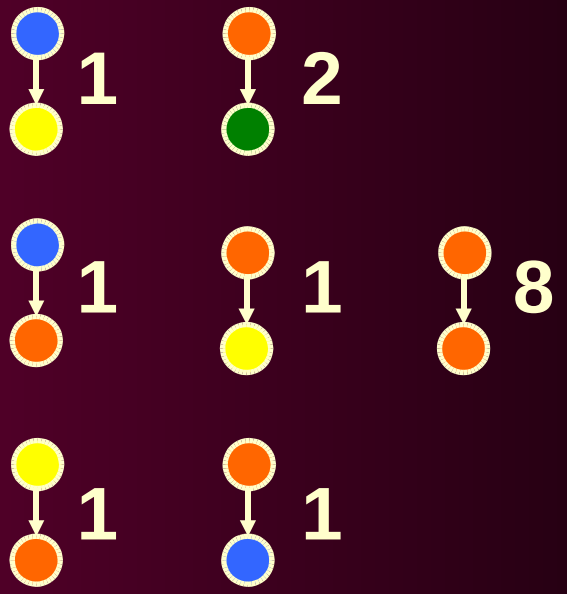


Isomorphic Subgraph Identification

Compute an Independent Set for Each Edge Type

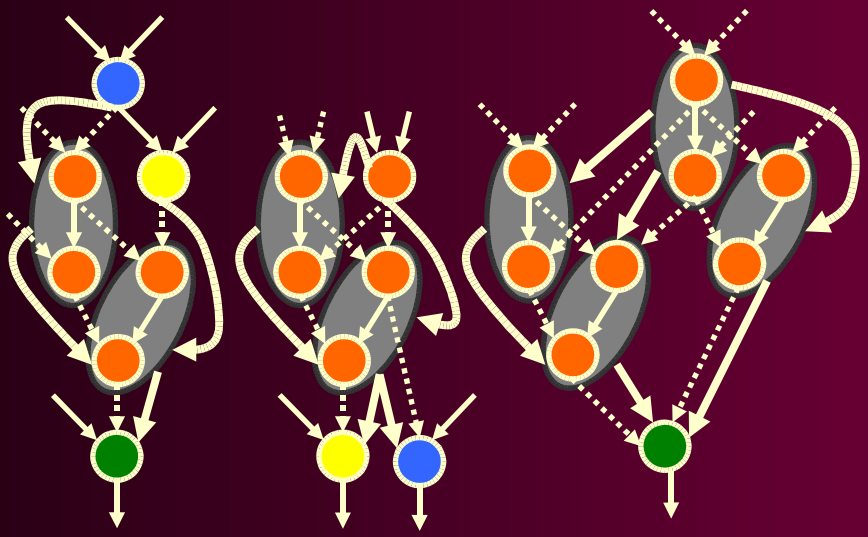
- **NP-Complete Problem**
- **Iterative Improvement Algorithm - (Kirovski, DAC '98)**





Isomorphic Subgraph Identification

Replace Most Frequently Occurring Pattern with a Template



Original DFG Edge

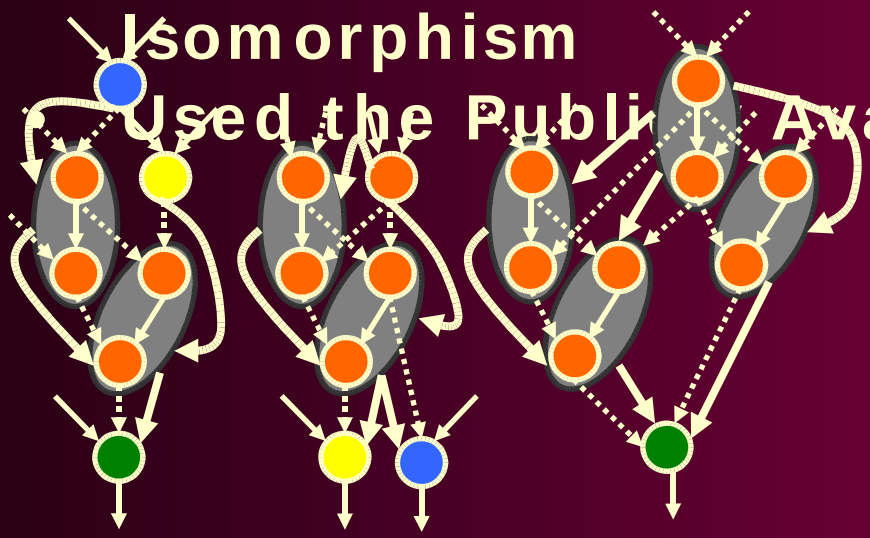
Data Dependencies Incident on Templates

Data Dependencies that Cross Template Boundaries

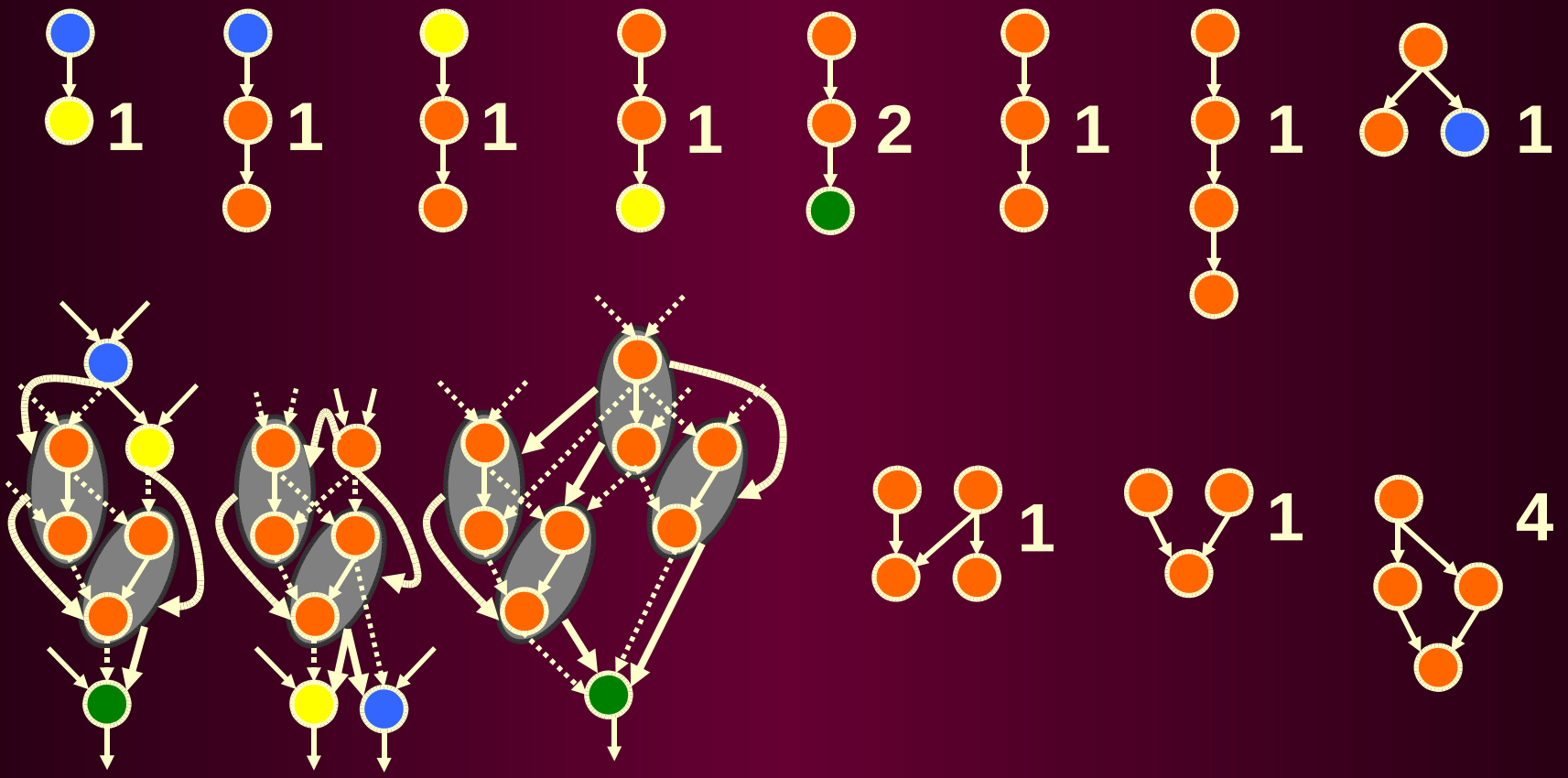
Isomorphic Subgraph Identification

Edge Contraction in the Presence of Templates

- **Generate New Templates Along Bold Edges**
- **Test for Template Equivalence is DAG Isomorphism**
  - **Used the Publicly Available VF2 Algorithm**

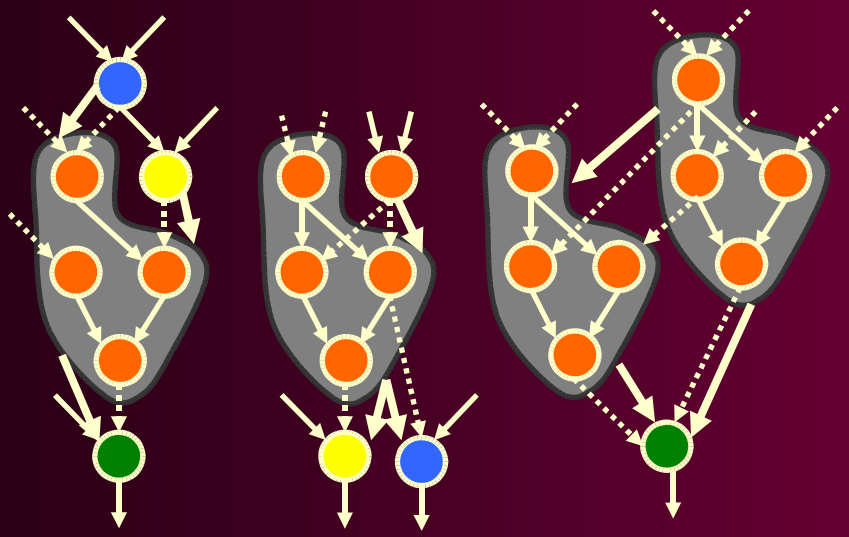


Isomorphic Subgraph Identification



Isomorphic Subgraph Identification

Replace Most Frequently Occurring Pattern with a Template



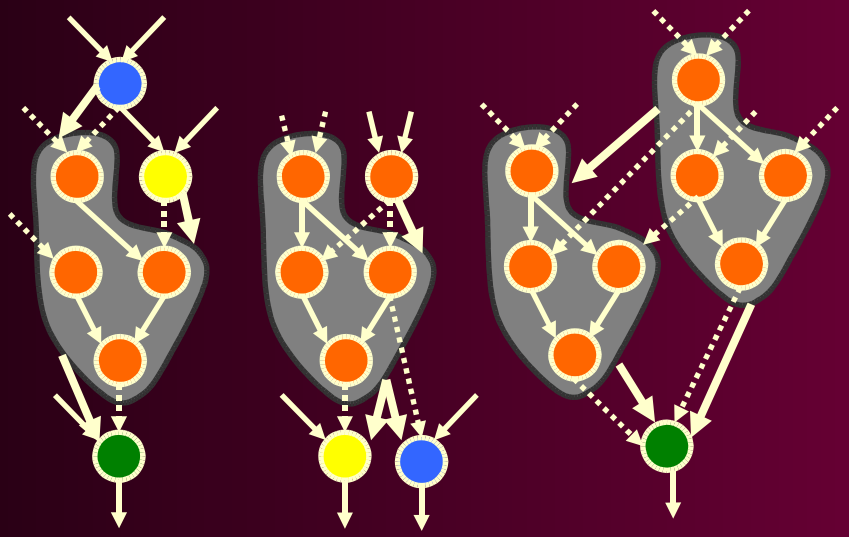
Original DFG Edge

Data Dependencies Incident on Templates

Data Dependencies that Cross Template Boundaries

Isomorphic Subgraph Identification

Replace Most Frequently Occurring Pattern with a Template



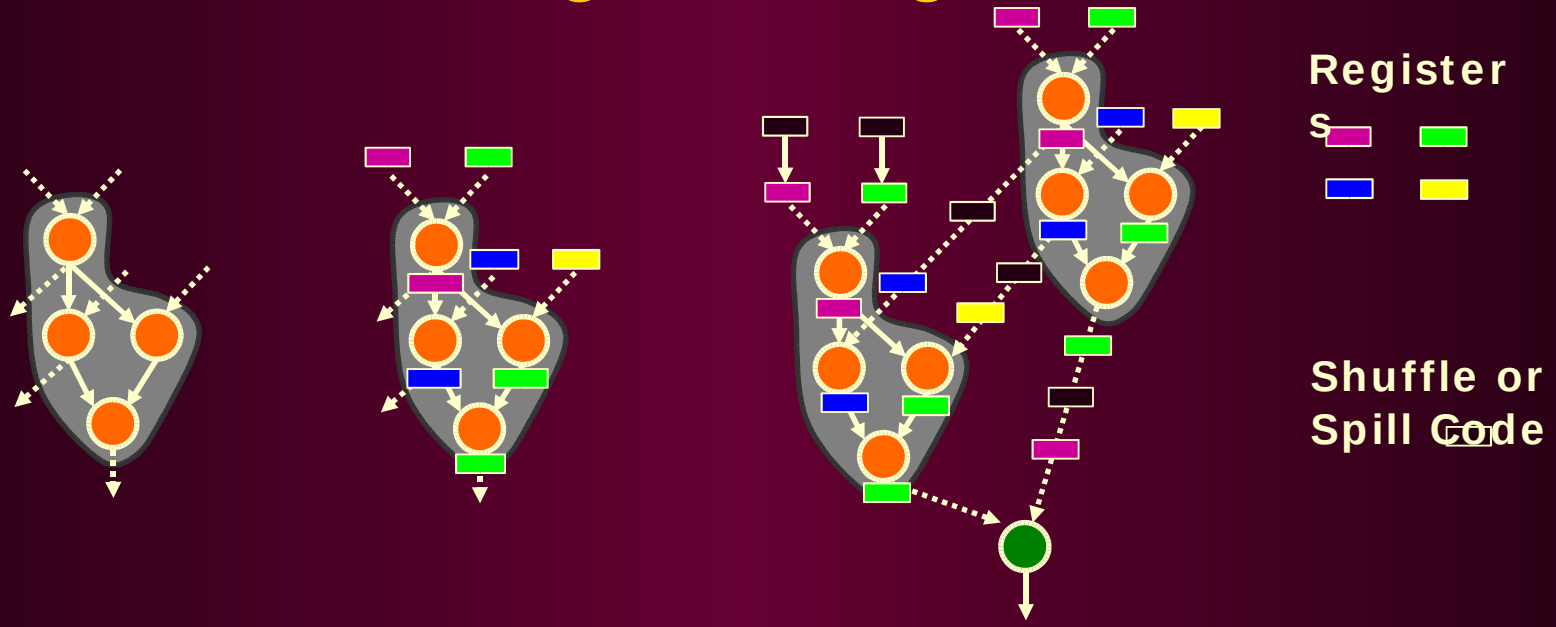
Original DFG Edge

Data Dependencies Incident on Templates

Data Dependencies that Cross Template Boundaries

Register Allocation

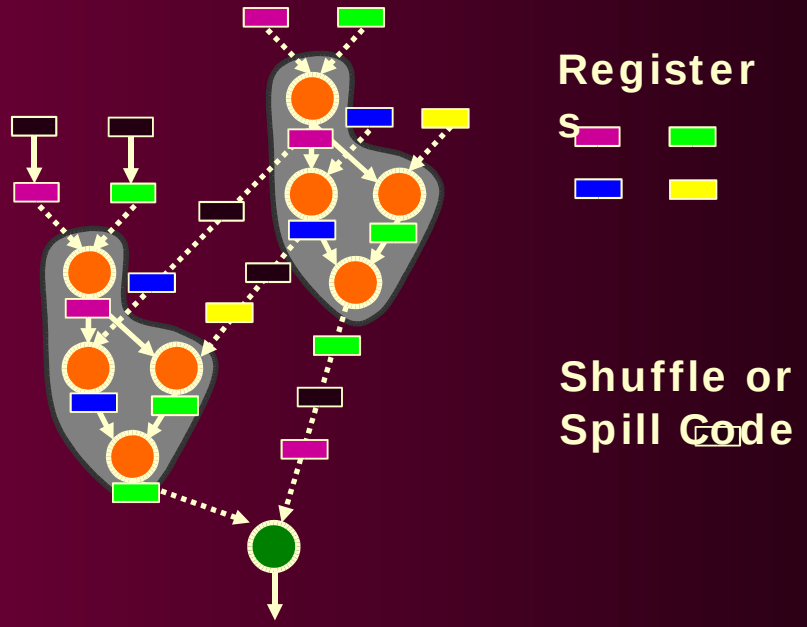
Isomorphic Templates Must Have Identical Usage of Registers



Register Allocation

Code Reuse Constraints May Work Against Code Size

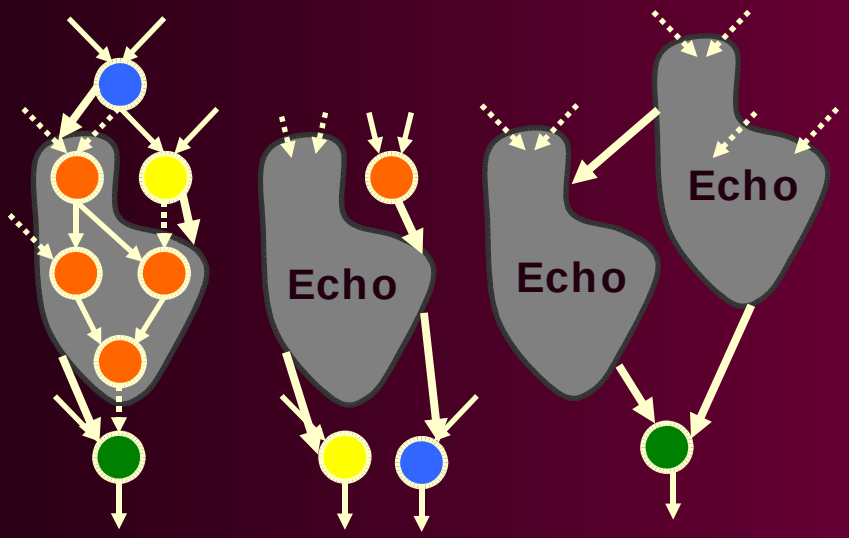
Each Template Eliminates 3 Instrs.
5 Shuffle/Spill Ops. are Required
The General Problem is Very Complicated
Existing Allocation Techniques Are Not Applicable



Present Status: The Allocator is a Work-in-

Isomorphic Subgraph Identification

After Register Allocation, Replace Subgraphs with Echo Instructions



Experimental Framework

Built Subgraph Identification into the Machine-SUIF Framework

- **Pass Placed Between Instruction Selection and Register Allocation**
- **Current Implementation Supports Alpha as Target**
  - **Allows for Future Integration with SimpleScalar Simulator**

Our Goal is to Evaluate the Effectiveness of Subgraph Identification

Experimental Methodology

Without Allocation in Place, We Cannot:

- **Estimate Where Shuffle/Spill Code Will be Inserted at Template Boundaries**
- **Determine Which Copy Instructions Will be Coalesced**

But We Can:

- **Make Assumptions Regarding the Starting Point for Register Allocation**

Two Approaches to Coalescing

Pessimistic Coalescing (Most Allocators)

- **Begin with All Copy Instructions in Place**
- **Coalesce Copies When Safe**

Optimistic Coalescing (Park & Moon, PACT ’98):

- **Initially Coalesce ALL Copy Instructions**
- **Re-Introduce Coalesced Copies to Avoid Spilling Live Ranges Whenever Possible**

Assumptions and Measurement

Pessimistic Assumption

- No Copy Instructions are Coalesced

Optimistic Assumption

- ALL Copy Instructions are Coalesced

Compute the Number of DFG Operations Before and After Compression Step

PU: Pessimistic, Uncompressed OU: Optimistic, Uncompressed

PC: Pessimistic, Compressed OC: Optimistic, Compressed

Benchmarks

Taken from the MediaBench and MiBench Application Suites

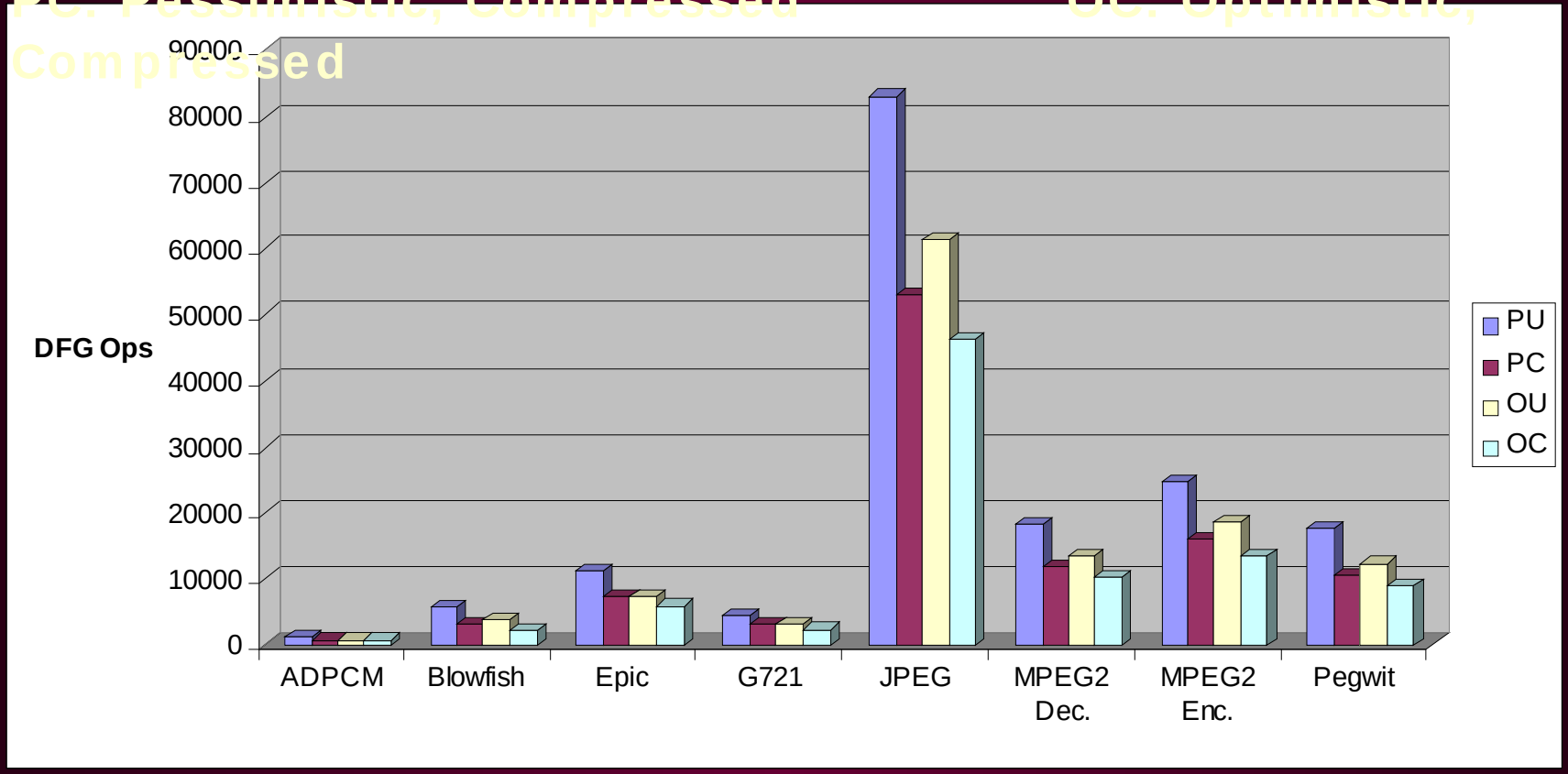
Benchmar

k	Description
ADPCM	Adaptive Differential Pulse Code Modulation
lowfish	Symmetric Block Cipher with Variable Key Length
Epic	Image Data Compression Utility
G721	Voice Compression
JPEG	Image Compression and Decompression
MPEG2 Dec	MPEG2 Decoder
MPEG2 Enc	MPEG2 Encoder
Pegwit	Public Key Encryption and Authentication

Experimental Results

PU: Pessimistic, Uncompressed OU: Optimistic, Uncompressed

PC: Pessimistic, Compressed OC: Optimistic, Compressed



Runtime Considerations

Algorithm Ran Efficiently (a few seconds) for Most Benchmarks

Several Notable Exceptions

- Four Common Features
  - Large DFGs
  - User-Defined Macros
  - Unrolled Loops
  - Cyclic Shifting of Parameters

- sha1.c (Pegwit) – One DFG
  - Compilation Time Was in Excess of 3 Hrs

Runtime Considerations

sha1.c

```
#define R0(v, w, x, y , z, i) {    z += ...;      w = ... }

void SHA1Transform( unsigned long state[5], ... ) {

    unsigned long  a = state[0], b = state[1], c = state[2],
                   d = state[3], e = state[4];

    R0(a, b, c, d, e, 0);
    R0(e, a, b, c, d, 1);
    R0(d, e, a, b, c, 2);
    R0(c, d, e, a, b, 3);
    R0(b, c, d, e, a, 4);
    R0(a, b, c, d, e, 5);
    ...
    R0(a, b, c, d, e, 15);
}
```

Runtime Considerations

sha1.c

```
#define R0(v, w, x, y , z, i) {    z += ...;      w = ... }

void SHA1Transform( unsigned long state[5], ... ) {

    unsigned long  a = state[0], b = state[1], c = state[2],
                   d = state[3], e = state[4], tmp;

    for( unsigned long i = 0; i < 16; i++) {
        R0(a, b, c, d, e, i);
        tmp = e; e = d; d = c; c = b; b = a; a = tmp;
    }
}
```

Compilation Time Was Reduced to Seconds

Conclusion

Echo Instructions

- Compression at a Minimal Hardware Cost
- Performance Overhead is Two Branches per Echo

Compiler Optimization

- Identify Redundancy via Subgraph Isomorphism
- New Challenges for Register Allocation

Experiments

- Significant Redundancy Observed in Compiler's IR