



INTERPROCEDURAL STATIC SLICING OF BINARY EXECUTABLES

**AKOS KISS ET AL.
SCAM'03**

0. ABSTRACTION

- 背景
 - 高級言語のプログラムスライス技術は多い
 - バイナリのプログラムスライス技術は少ない
- 提案
 - バイナリでの静的かつInterproceduralなスライス技術
 - ①未解決の関数呼び出しと間接分岐を処理する保守的な手法
 - ②①に加えメモリ関連の改善案
- 実験
 - スライスによって-32~-44%のサイズ削減を達成

1. INTRODUCTION

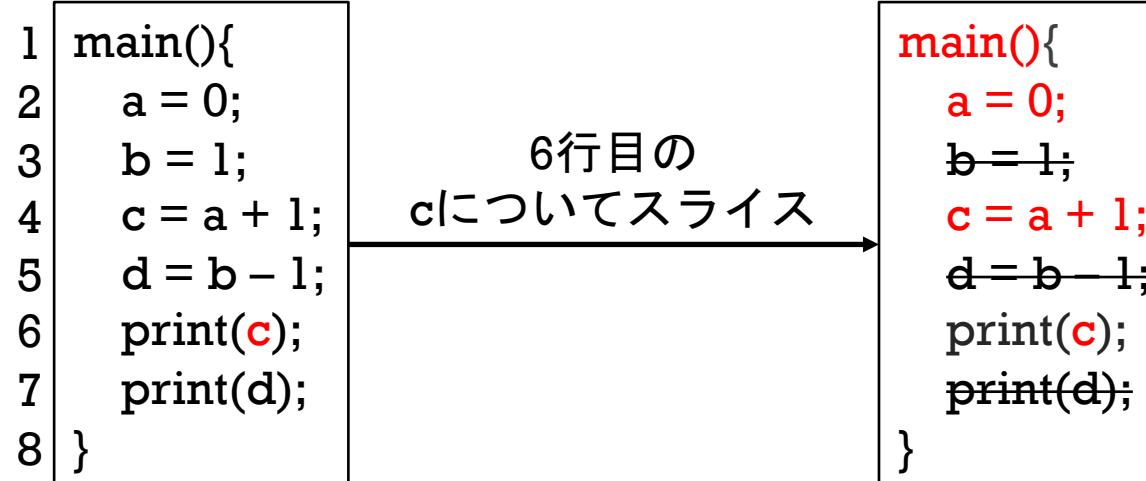
- 高級言語を対象としたスライス技術は多い
- 一方でバイナリスライスの研究は少ない（2003年当時）
 - [8]: intraprocedural
 - [4]: interproceduralの提案のみ（未実装）
- バイナリならではの問題点がある（後述）
- 用途は多い
 - {レガシーコード, アセンブリ, ウィルス}の{コード理解, 復元, バグ修正}など

[8] intraprocedural static slicing of binary executables[ICSM'97]

[4] Static analysis of binary code to isolate malicious behaviors[IEEE IWSE'99]

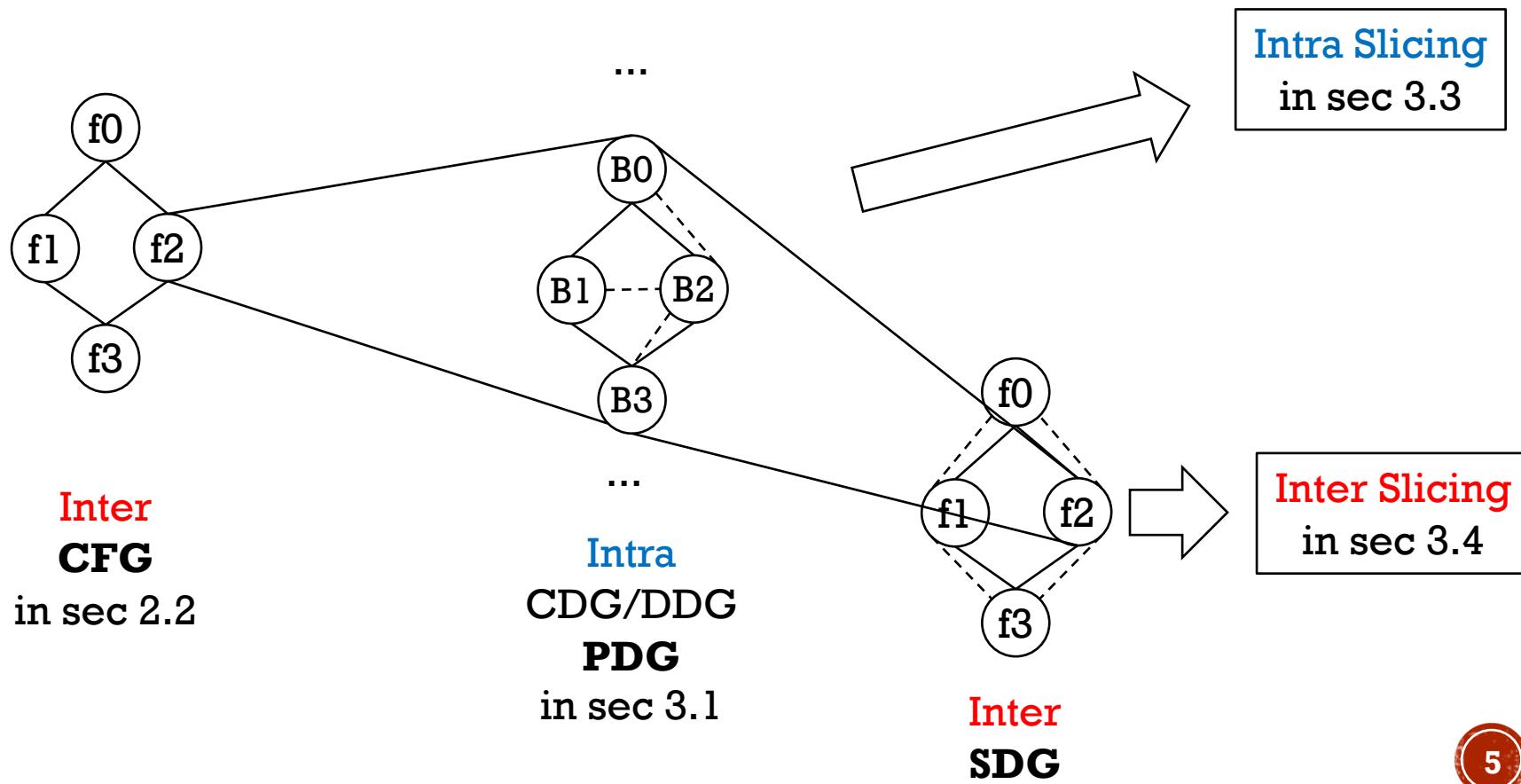
1. INTRODUCTION

- What is 「slicing」 ?
- **slicing**: 関心事に関連する要素だけを抽出すること
 - Path Slicing (Trace Slicing):
 - ex.) 前々回のWoodPecker: トレース内のイベント関連の分岐だけを抽出
 - Program Slicing:
 - プログラム全体から関心事に関係する(最小限の)命令列を抽出
 - →制御依存、データ依存の情報から計算
 - 今回の論文はこっち



2. APPROACH OVERVIEW

- バイナリスライス生成までの大まかな流れ



2. APPROACH

2.1 PROBLEMS

- バイナリのCFGを作りにあたっての問題点
 - 命令境界の判定（特にCISC、今回はRISC）
 - 命令種類が多い
→ソースコードと比較して制御構造が複雑化
 - 関数の境界判定
 - 関数境界を超えるジャンプ（cross-jump）はどうするのか
 - 間接呼び出しの候補
 - 一般にソースコードよりも候補が増加
(→再配置情報などがあれば話は別)
- 基本的な緩和策はシンボル情報、再配置情報の活用
 - &経験的な
コンパイラ、ファイルフォーマット、アーキテクチャの各仕様情報

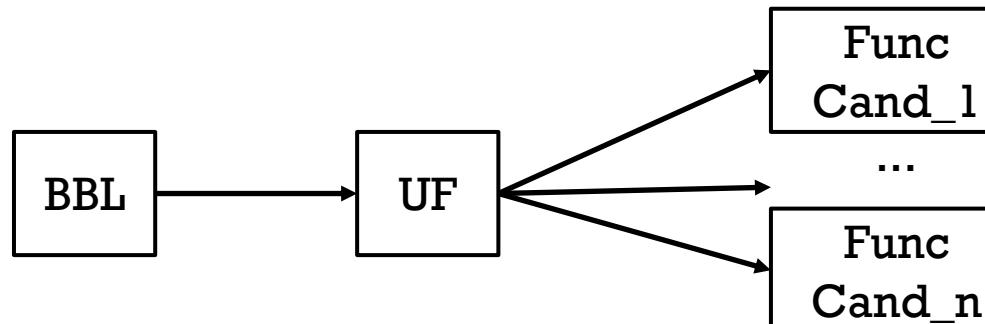
2. APPROACH

2.2 BUILDING CFG

- バイナリCFGの構築

- ①BBL単位でノードを生成
- ②BBLを関数単位でグループ化
- ③BBL間にエッジを生成
- ④関数呼び出しによるCall/Retエッジを生成
- ⑤間接呼び出しについては「Unknown Function」ノードから Calleeになりえる全関数へエッジを生成
- ⑥間接ジャンプについては「Unknown Block」ノードから ジャンプ先になりえる全ブロックへエッジを生成
- ⑦オーバーラップ/クロスジャンプについてはRetエッジを補う

論文内で
具体例なし



2. APPROACH

2.3 EXAMPLE

00002ECC	1808	add:	
00002ECE	46F7	ADD R0, R1, R0 MOV PC, LR	B1
		mul:	
00002ED0	B530	PUSH {R4,R5,LR}	B2
00002ED2	1C0C	ADD R4, R1, #0	
00002ED4	1C05	ADD R5, R0, #0	
00002ED6	2300	MOV R3, #0	
00002ED8	2201	MOV R2, #1	
00002EDA	42A2	CMP R2, R4	
00002EDC	DC0B	BGT 0x00002EF6	
00002EDE	1C18	ADD R0, R3, #0	B3
00002EE0	1C29	ADD R1, R5, #0	
00002EE2	F7FFFFFF3	BL 0x00002ECC (add)	
00002EE6	1C03	ADD R3, R0, #0	B4
00002EE8	1C10	ADD R0, R2, #0	
00002EEA	2101	MOV R1, #1	
00002EEC	F7FFFFFFE	BL 0x00002ECC (add)	
00002EF0	1C02	ADD R2, R0, #0	B5
00002EF2	42A2	CMP R2, R4	
00002EF4	DDF3	BLE 0x00002EDE	
00002EF6	1C18	ADD R0, R3, #0	B6
00002EF8	BD30	POP {R4,R5,PC}	

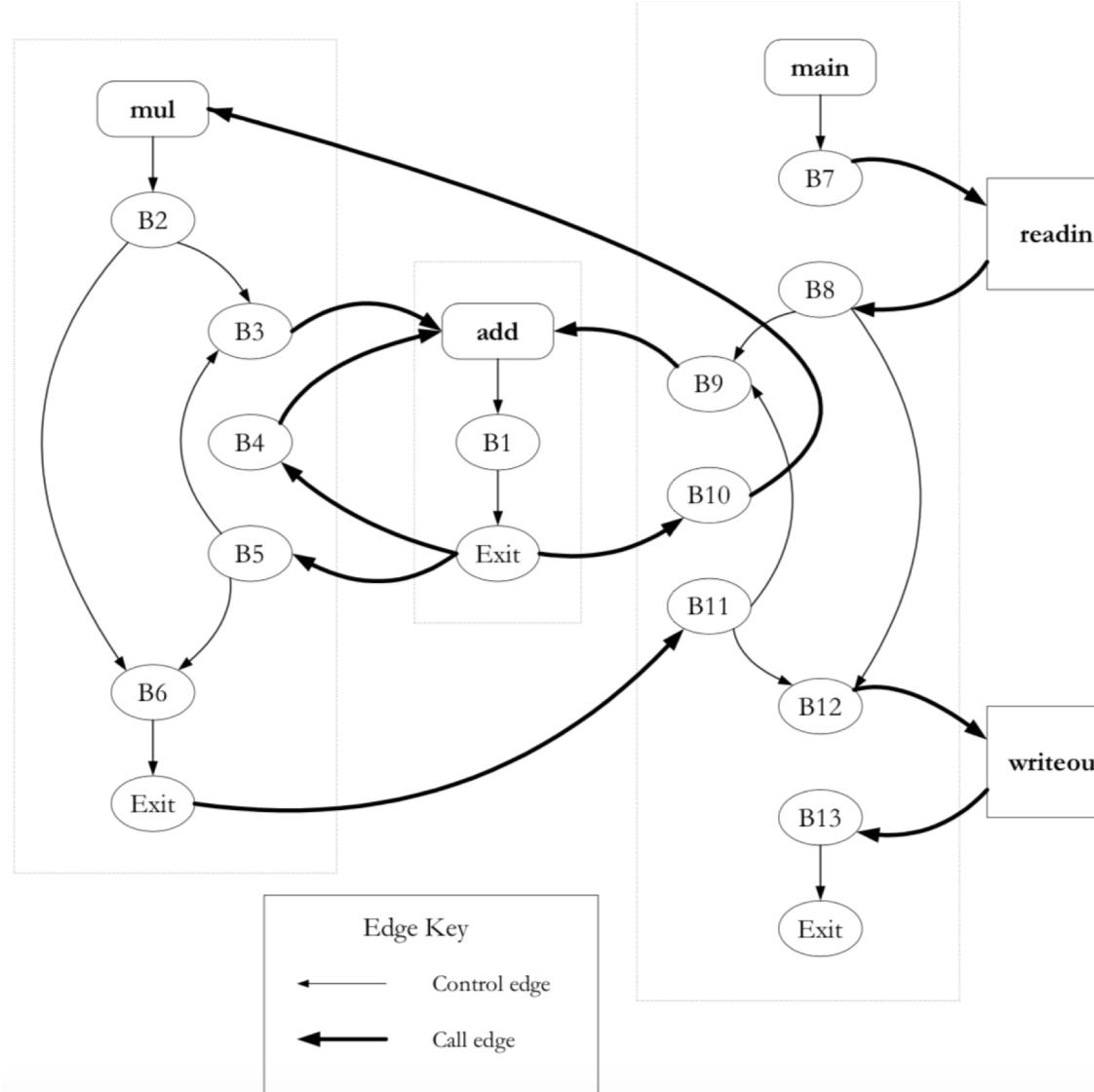
2. APPROACH

2.3 EXAMPLE

main:			
00002EFA	B530	PUSH {R4,R5,LR}	B7
00002EFC	B082	ADD SP, #-8	
00002EFE	F7FFFFFFD5	BL 0x00002EAC (readin)	
00002F02	1C05	ADD R5, R0, #0	B8
00002F04	2000	MOV R0, #0	
00002F06	9000	STR R0, [SP, #0]	
00002F08	2001	MOV R0, #1	
00002F0A	9001	STR R0, [SP, #4]	
00002F0C	2401	MOV R4, #1	
00002F0E	42AC	CMP R4, R5	
00002F10	DA0B	BGE 0x00002F2A	
00002F12	9800	LDR R0, [SP, #0]	B9
00002F14	1C21	ADD R1, R4, #0	
00002F16	F7FFFFFFD9	BL 0x00002ECC (add)	
00002F1A	9000	STR R0, [SP, #0]	B10
00002F1C	9801	LDR R0, [SP, #4]	
00002F1E	F7FFFFFFD7	BL 0x00002ED0 (mul)	
00002F22	9001	STR R0, [SP, #4]	B11
00002F24	3401	ADD R4, #1	
00002F26	42AC	CMP R4, R5	
00002F28	DBF3	BLT 0x00002F12	
00002F2A	9800	LDR R0, [SP, #0]	B12
00002F2C	9901	LDR R1, [SP, #4]	
00002F2E	F7FFFFFFC6	BL 0x00002EBE (writeout)	
00002F32	B002	ADD SP, #8	B13
00002F34	BD30	POP {R4,R5,PC}	

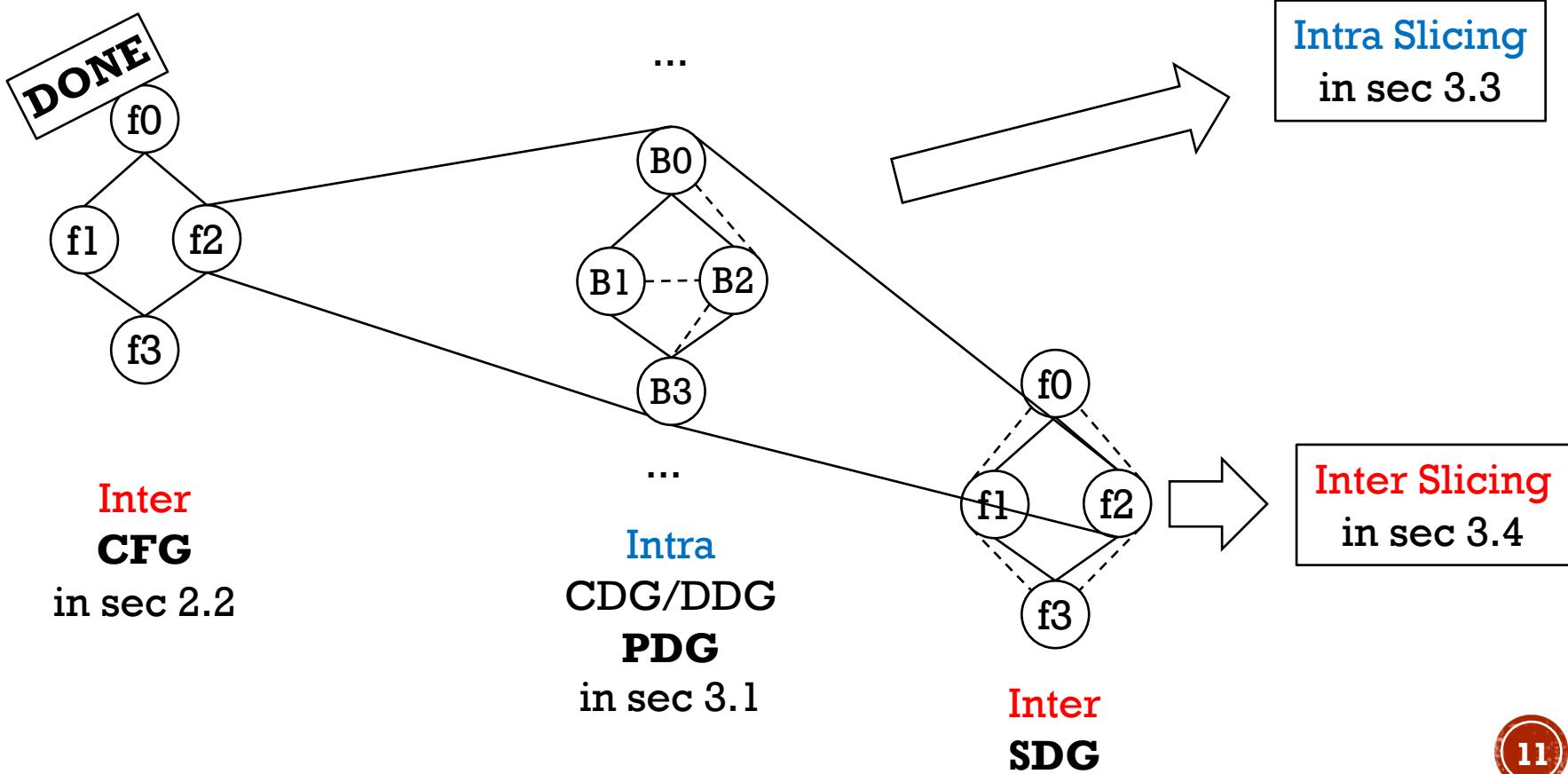
2. APPROACH

2.3 EXAMPLE



3. APPROACH OVERVIEW

- バイナリスライス生成までの大まかな流れ



3. APPROACH

3.1 BUILDING INTRA PDG

- PDG(Program Dependence Graph):
 - 制御依存、データ依存が記述されたグラフ
 - 今回はIntra、すなわち関数単位で生成
 - CDG(Control Dependence Graph)
 - BBL間の制御依存を階層化 (by [10], [16])
 - DDG(Data Dependence Graph)
 - CDGに加えて、データ間の依存関係のエッジを生成
 - ①命令単位のノードと命令引数のノードを追加
 - ②BBL内の命令間のデータ依存関係エッジを追加
 - ③BBL間のデータ依存関係エッジを追加
 - ④BBL間の依存関係エッジを追加
 - ⑤関数呼び出しの引数ノードなどを処理
- PDGを構成
 - 保守的なプログラムスライスが定量時間で計算可能

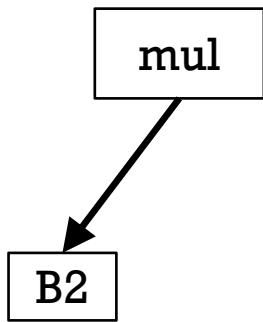
[10] The program dependence graph and its use in optimization [1987]

[16] A fast algorithm for finding dominators in a flowgraph [TPLS'79]

3. APPROACH

3.1 BUILDING INTRA PDG

- CDG: 構造の階層化

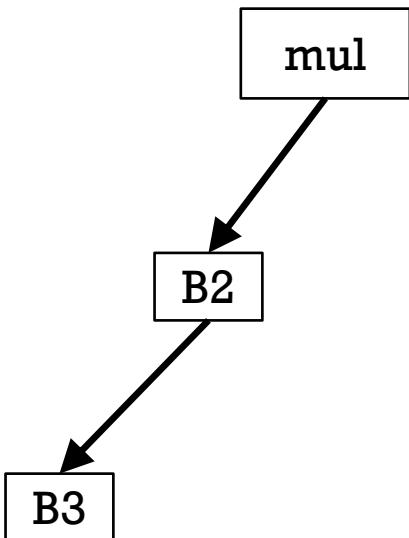


mul:			
00002ED0	B530	PUSH { R4 , R5 , LR }	B2
00002ED2	1C0C	ADD R4, R1, #0	
00002ED4	1C05	ADD R5, R0, #0	
00002ED6	2300	MOV R3, #0	
00002ED8	2201	MOV R2, #1	
00002EDA	42A2	CMP R2, R4	
00002EDC	DC0B	BGT 0x00002EF6	
00002EDE	1C18	ADD R0, R3, #0	B3
00002EE0	1C29	ADD R1, R5, #0	
00002EE2	F7FFFFFF3	BL 0x00002ECC (add)	
00002EE6	1C03	ADD R3, R0, #0	B4
00002EE8	1C10	ADD R0, R2, #0	
00002EEA	2101	MOV R1, #1	
00002EEC	F7FFFFEE	BL 0x00002ECC (add)	
00002EF0	1C02	ADD R2, R0, #0	B5
00002EF2	42A2	CMP R2, R4	
00002EF4	DDF3	BLE 0x00002EDE	
00002EF6	1C18	ADD R0, R3, #0	B6
00002EF8	BD30	POP { R4 , R5 , PC }	

3. APPROACH

3.1 BUILDING INTRA PDG

- CDG: 構造の階層化

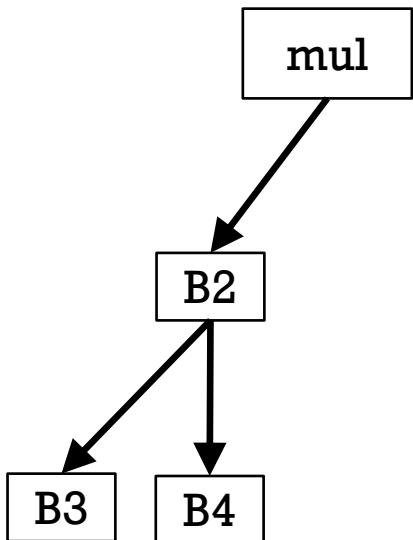


		mul:	
00002ED0	B530	PUSH { R4 , R5 , LR }	B2
00002ED2	1C0C	ADD R4, R1, #0	
00002ED4	1C05	ADD R5, R0, #0	
00002ED6	2300	MOV R3, #0	
00002ED8	2201	MOV R2, #1	
00002EDA	42A2	CMP R2, R4	
00002EDC	DC0B	BGT 0x00002EF6	
00002EDE	1C18	ADD R0, R3, #0	B3
00002EE0	1C29	ADD R1, R5, #0	
00002EE2	F7FFFFFF3	BL 0x00002ECC (add)	
00002EE6	1C03	ADD R3, R0, #0	B4
00002EE8	1C10	ADD R0, R2, #0	
00002EEA	2101	MOV R1, #1	
00002EEC	F7FFFFFFE	BL 0x00002ECC (add)	
00002EF0	1C02	ADD R2, R0, #0	B5
00002EF2	42A2	CMP R2, R4	
00002EF4	DDF3	BLE 0x00002EDE	
00002EF6	1C18	ADD R0, R3, #0	B6
00002EF8	BD30	POP { R4 , R5 , PC }	

3. APPROACH

3.1 BUILDING INTRA PDG

- CDG: 構造の階層化

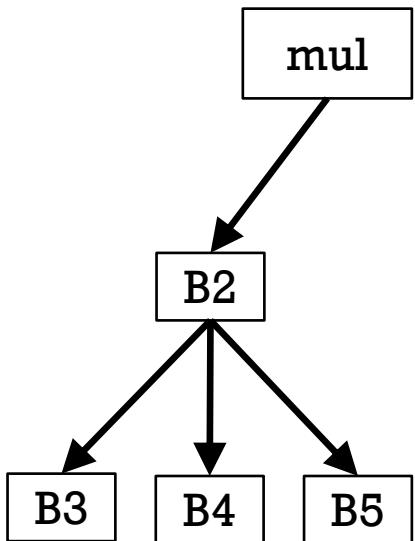


		mul:	
00002ED0	B530	PUSH { R4 , R5 , LR }	B2
00002ED2	1C0C	ADD R4, R1, #0	
00002ED4	1C05	ADD R5, R0, #0	
00002ED6	2300	MOV R3, #0	
00002ED8	2201	MOV R2, #1	
00002EDA	42A2	CMP R2, R4	
00002EDC	DC0B	BGT 0x00002EF6	
00002EDE	1C18	ADD R0, R3, #0	B3
00002EE0	1C29	ADD R1, R5, #0	
00002EE2	F7FFFFFF3	BL 0x00002ECC (add)	
00002EE6	1C03	ADD R3, R0, #0	B4
00002EE8	1C10	ADD R0, R2, #0	
00002EEA	2101	MOV R1, #1	
00002EEC	F7FFFFFFE	BL 0x00002ECC (add)	
00002EF0	1C02	ADD R2, R0, #0	B5
00002EF2	42A2	CMP R2, R4	
00002EF4	DDF3	BLE 0x00002EDE	
00002EF6	1C18	ADD R0, R3, #0	B6
00002EF8	BD30	POP { R4 , R5 , PC }	

3. APPROACH

3.1 BUILDING INTRA PDG

- CDG: 構造の階層化

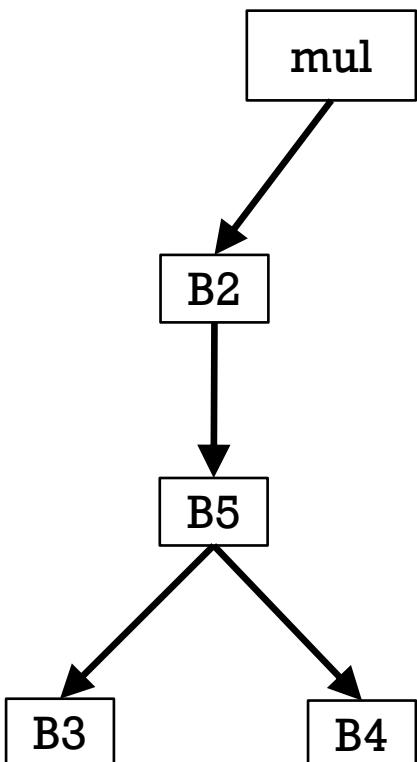


mul:		
00002ED0	B530	PUSH { R4 , R5 , LR }
00002ED2	1C0C	ADD R4, R1, #0
00002ED4	1C05	ADD R5, R0, #0
00002ED6	2300	MOV R3, #0
00002ED8	2201	MOV R2, #1
00002EDA	42A2	CMP R2, R4
00002EDC	DC0B	BGT 0x00002EF6
00002EDE	1C18	ADD R0, R3, #0
00002EE0	1C29	ADD R1, R5, #0
00002EE2	F7FFFFFF3	BL 0x00002ECC (add)
00002EE6	1C03	ADD R3, R0, #0
00002EE8	1C10	ADD R0, R2, #0
00002EEA	2101	MOV R1, #1
00002EEC	F7FFFFFFE	BL 0x00002ECC (add)
00002EF0	1C02	ADD R2, R0, #0
00002EF2	42A2	CMP R2, R4
00002EF4	DDF3	BLE 0x00002EDE
00002EF6	1C18	ADD R0, R3, #0
00002EF8	BD30	POP { R4 , R5 , PC }

3. APPROACH

3.1 BUILDING INTRA PDG

- CDG: 構造の階層化

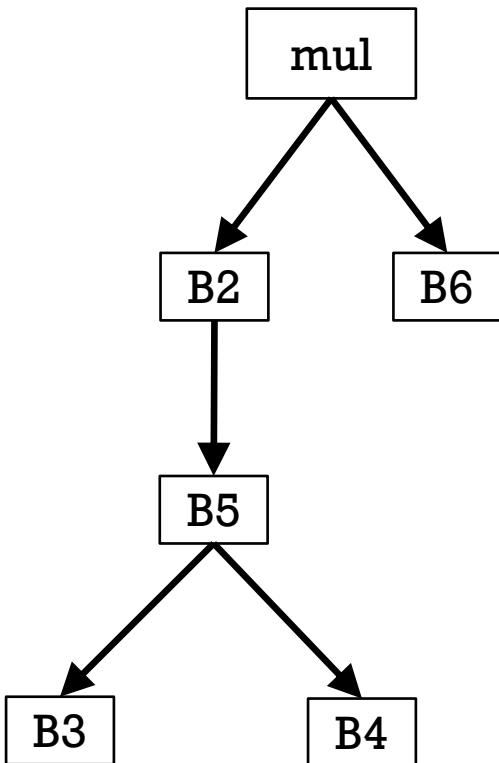


		mul:	
00002ED0	B530	PUSH { R4 , R5 , LR }	B2
00002ED2	1C0C	ADD R4, R1, #0	
00002ED4	1C05	ADD R5, R0, #0	
00002ED6	2300	MOV R3, #0	
00002ED8	2201	MOV R2, #1	
00002EDA	42A2	CMP R2, R4	
00002EDC	DC0B	BGT 0x00002EF6	
00002EDE	1C18	ADD R0, R3, #0	B3
00002EE0	1C29	ADD R1, R5, #0	
00002EE2	F7FFFFFF3	BL 0x00002ECC (add)	
00002EE6	1C03	ADD R3, R0, #0	B4
00002EE8	1C10	ADD R0, R2, #0	
00002EEA	2101	MOV R1, #1	
00002EEC	F7FFFFEE	BL 0x00002ECC (add)	
00002EF0	1C02	ADD R2, R0, #0	B5
00002EF2	42A2	CMP R2, R4	
00002EF4	DDF3	BLE 0x00002EDE	
00002EF6	1C18	ADD R0, R3, #0	B6
00002EF8	BD30	POP { R4 , R5 , PC }	

3. APPROACH

3.1 BUILDING INTRA PDG

- CDG: 構造の階層化



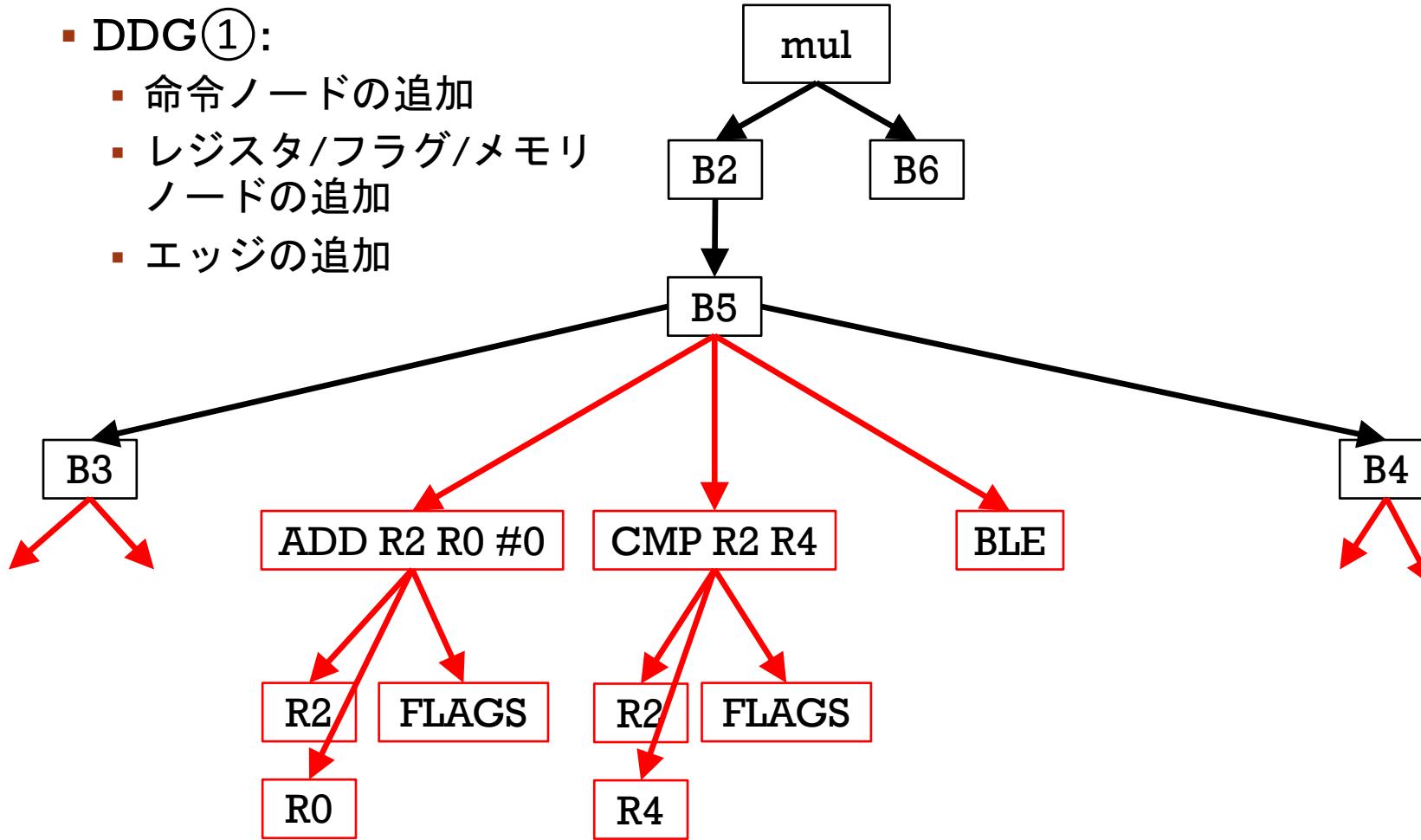
		mul:	
00002ED0	B530	PUSH { R4 , R5 , LR }	B2
00002ED2	1C0C	ADD R4, R1, #0	
00002ED4	1C05	ADD R5, R0, #0	
00002ED6	2300	MOV R3, #0	
00002ED8	2201	MOV R2, #1	
00002EDA	42A2	CMP R2, R4	
00002EDC	DC0B	BGT 0x00002EF6	
00002EDE	1C18	ADD R0, R3, #0	B3
00002EE0	1C29	ADD R1, R5, #0	
00002EE2	F7FFFFFF3	BL 0x00002ECC (add)	
00002EE6	1C03	ADD R3, R0, #0	B4
00002EE8	1C10	ADD R0, R2, #0	
00002EEA	2101	MOV R1, #1	
00002EEC	F7FFFFFFE	BL 0x00002ECC (add)	
00002EF0	1C02	ADD R2, R0, #0	B5
00002EF2	42A2	CMP R2, R4	
00002EF4	DDF3	BLE 0x00002EDE	
00002EF6	1C18	ADD R0, R3, #0	B6
00002EF8	BD30	POP { R4 , R5 , PC }	

3. APPROACH

3.1 BUILDING INTRA PDG

- DDG①:

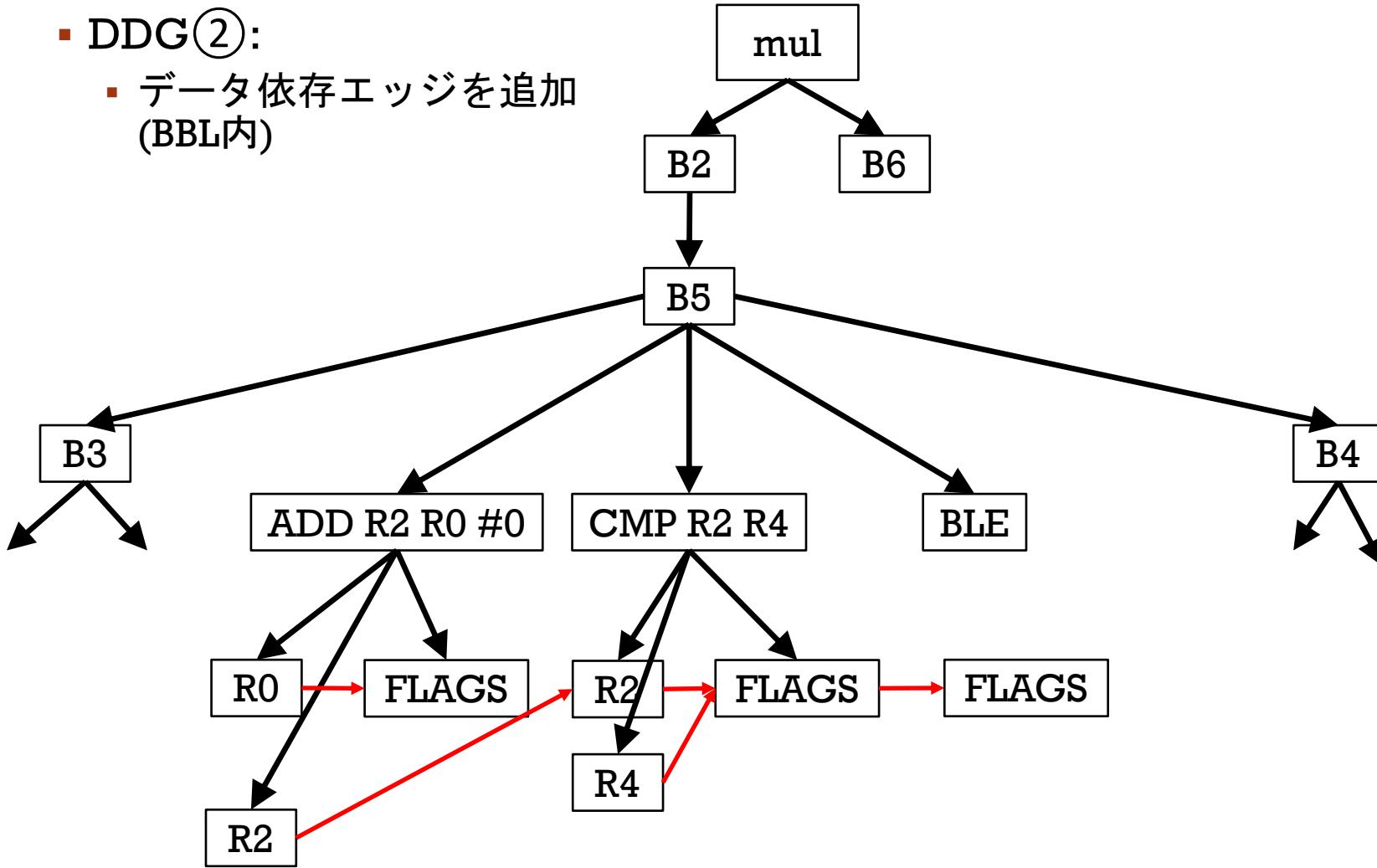
- 命令ノードの追加
- レジスタ/フラグ/メモリノードの追加
- エッジの追加



3. APPROACH

3.1 BUILDING INTRA PDG

- DDG②:
 - データ依存エッジを追加
(BBL内)

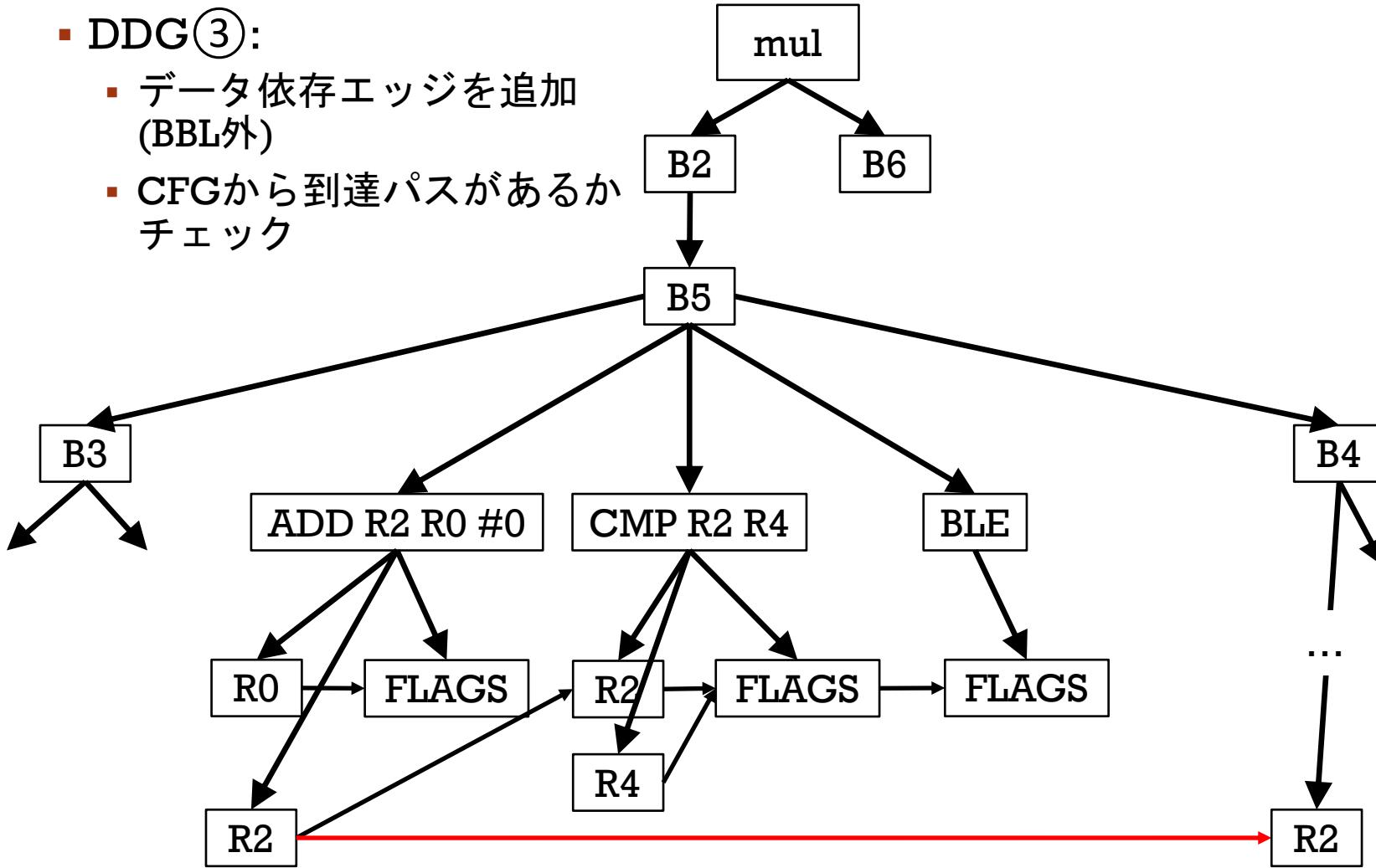


3. APPROACH

3.1 BUILDING INTRA PDG

- DDG③:

- データ依存エッジを追加
(BBL外)
- CFGから到達パスがあるか
チェック

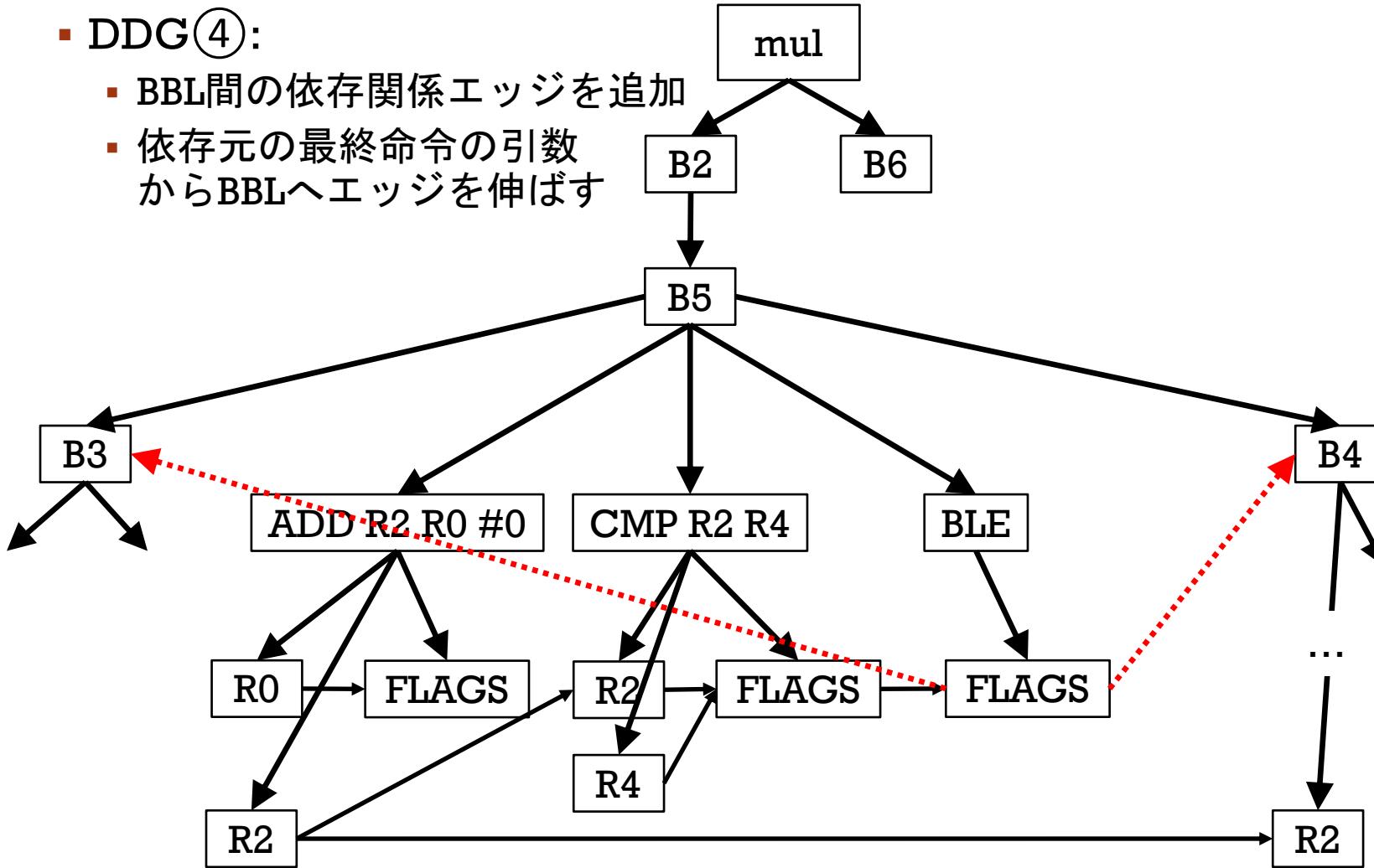


3. APPROACH

3.1 BUILDING INTRA PDG

- DDG④:

- BBL間の依存関係エッジを追加
- 依存元の最終命令の引数からBBLへエッジを伸ばす

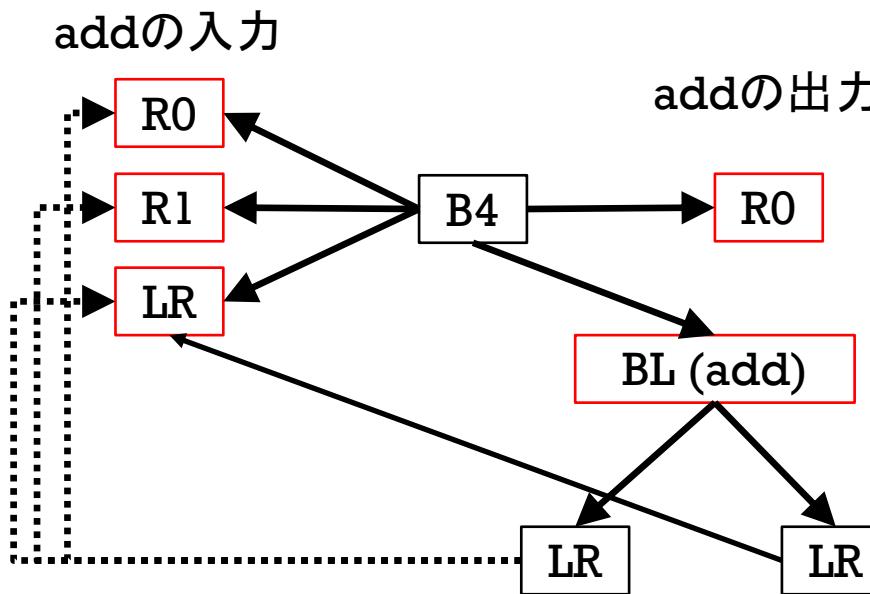


3. APPROACH

3.1 BUILDING INTRA PDG

- DDG⑤:

- 関数呼び出しの場合はLRと**仮入出力ノード**をBBLから追加
 - LR(Link Register): ARMでの関数呼び出しの復帰アドレスの格納先
(スタックに復帰アドレスが積まれた場合は汎用レジスタ)



3. APPROACH

3.1 BUILDING INTRA PDG

- 補足: 関数の引数などはバイナリでは明示的ではない(in ⑤)

- →Used-Def関係を入出力セットとして解析
- 各関数 f について

- U_f : 読み込まれたレジスタ/メモリの集合
- D_f : 書き込まれたレジスタ/メモリの集合
- C_f : f から呼ばれる関数の集合
- I_f : f を構成する命令の集合
- を計算 (Uf/Dfは右の漸化式に従う)

- 最終的に
 D_f は出力パラメータ
 $D_f \cup U_f$ は入力パラメータとして扱う
 (in DDG⑤の仮入出力ノードの決定)

$$\begin{aligned} U_f^{(-1)} &= \emptyset \\ U_f^{(i+1)} &= \bigcup_{j \in I_f} u_j \cup \bigcup_{g \in C_f} U_g^{(i)} \\ U_f &= U_f^{(i)}, \text{ where } U_f^{(i)} = U_f^{(i+1)} \end{aligned}$$

$$\begin{aligned} D_f^{(-1)} &= \emptyset \\ D_f^{(i+1)} &= \bigcup_{j \in I_f} d_j \cup \bigcup_{g \in C_f} D_g^{(i)} \\ D_f &= D_f^{(i)}, \text{ where } D_f^{(i)} = D_f^{(i+1)} \end{aligned}$$

3. APPROACH

3.1 BUILDING INTRA PDG

- 解いてみよう

$$\begin{aligned} U_f^{(-1)} &= \emptyset \\ U_f^{(i+1)} &= \bigcup_{j \in I_f} u_j \cup \bigcup_{g \in C_f} U_g^{(i)} \\ U_f &= U_f^{(i)}, \text{ where } U_f^{(i)} = U_f^{(i+1)} \end{aligned}$$

$$\begin{aligned} D_f^{(-1)} &= \emptyset \\ D_f^{(i+1)} &= \bigcup_{j \in I_f} d_j \cup \bigcup_{g \in C_f} D_g^{(i)} \\ D_f &= D_f^{(i)}, \text{ where } D_f^{(i)} = D_f^{(i+1)} \end{aligned}$$

$$\begin{array}{rcl} C_{\text{add}} & = & \emptyset \\ U_{\text{add}}^{(0)} & = & \boxed{\textcircled{1}} \qquad \qquad D_{\text{add}}^{(0)} & = & \boxed{\textcircled{2}} \end{array}$$

00002ECC	1808	add:
00002ECE	46F7	ADD R0, R1, R0 MOV PC, LR

B1

3. APPROACH

3.1 BUILDING INTRA PDG

- 解いてみよう

$$\begin{aligned} U_f^{(-1)} &= \emptyset \\ U_f^{(i+1)} &= \bigcup_{j \in I_f} u_j \cup \bigcup_{g \in C_f} U_g^{(i)} \\ U_f &= U_f^{(i)}, \text{ where } U_f^{(i)} = U_f^{(i+1)} \end{aligned}$$

$$\begin{aligned} D_f^{(-1)} &= \emptyset \\ D_f^{(i+1)} &= \bigcup_{j \in I_f} d_j \cup \bigcup_{g \in C_f} D_g^{(i)} \\ D_f &= D_f^{(i)}, \text{ where } D_f^{(i)} = D_f^{(i+1)} \end{aligned}$$

$$\begin{array}{rcl} C_{\text{add}} & = & \emptyset \\ U_{\text{add}}^{(0)} & = & \{\text{R0, R1, LR}\} \end{array} \qquad \qquad \begin{array}{rcl} D_{\text{add}}^{(0)} & = & \boxed{\textcircled{2}} \end{array}$$

00002ECC

1808

add:

00002ECE

46F7

ADD R0, R1, R0

B1

MOV PC, LR

3. APPROACH

3.1 BUILDING INTRA PDG

- 解いてみよう

$$\begin{aligned} U_f^{(-1)} &= \emptyset \\ U_f^{(i+1)} &= \bigcup_{j \in I_f} u_j \cup \bigcup_{g \in C_f} U_g^{(i)} \\ U_f &= U_f^{(i)}, \text{ where } U_f^{(i)} = U_f^{(i+1)} \end{aligned}$$

$$\begin{aligned} D_f^{(-1)} &= \emptyset \\ D_f^{(i+1)} &= \bigcup_{j \in I_f} d_j \cup \bigcup_{g \in C_f} D_g^{(i)} \\ D_f &= D_f^{(i)}, \text{ where } D_f^{(i)} = D_f^{(i+1)} \end{aligned}$$

$$\begin{array}{rcl} C_{\text{add}} & = & \emptyset \\ U_{\text{add}}^{(0)} & = & \{\text{R0, R1, LR}\} \end{array} \qquad \qquad \begin{array}{rcl} D_{\text{add}}^{(0)} & = & \{\text{R0}\} \end{array}$$

00002ECC

1808

add:

00002ECE

46F7

ADD R0, R1, R0

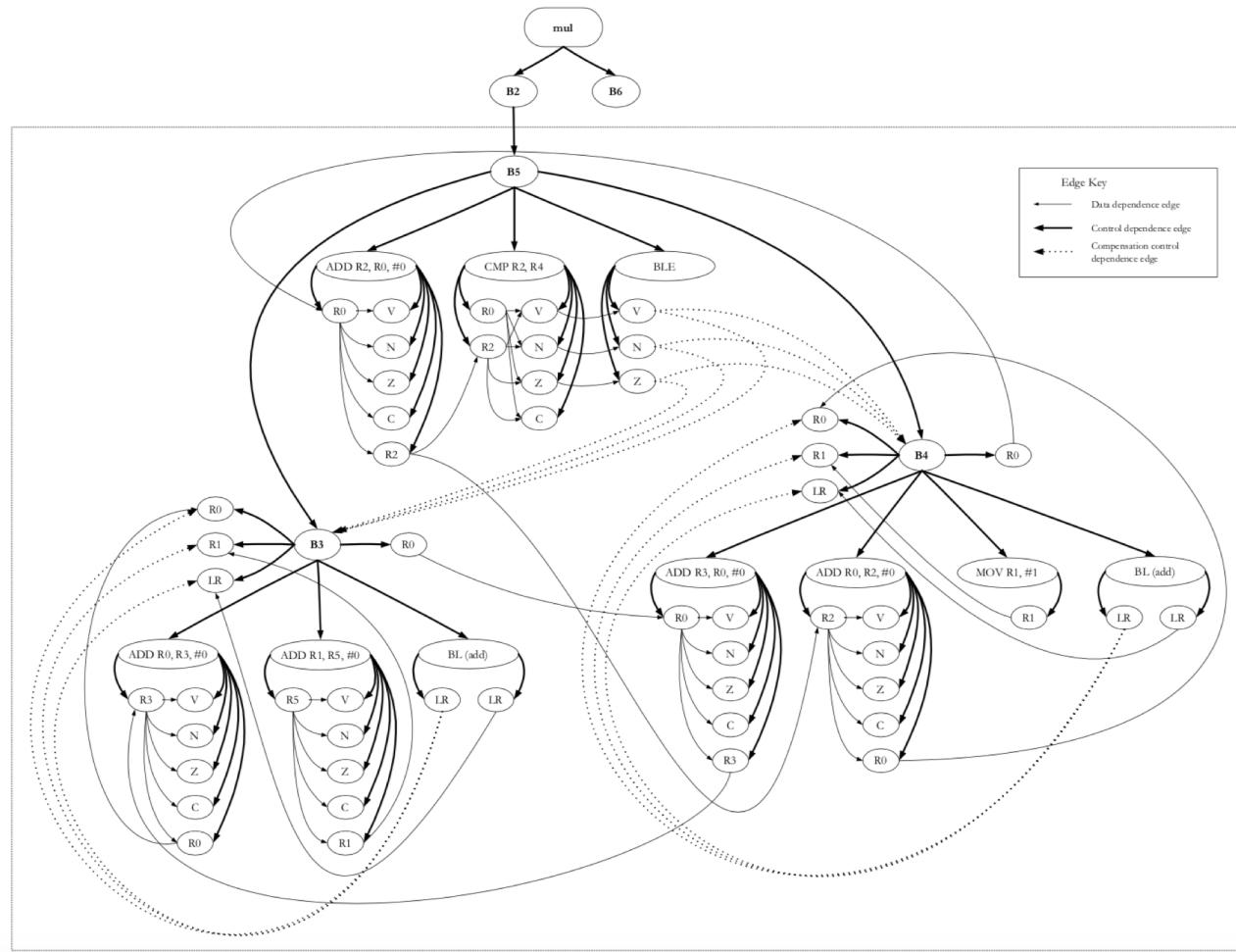
B1

MOV PC, LR

3. APPROACH

3.1 BUILDING INTRA PDG

- そんなこんなでこんな感じのIntra PDGが完成 (例: mul)



3. APPROACH

3.2 IMPROVE PDG

- 3.1までのPDGは安全だが過度に保守的
 - 改善案①:
 - 関数呼び出し時のレジスタ保存/復元によるヒューリスティクス
 - 改善案②:
 - メモリアクセスの解析

3. APPROACH

3.2 IMPROVE PDG

- 3.1までのPDGは安全だが過度に保守的
 - 改善案①:
 - 関数呼び出し時のレジスタ保存/復元によるヒューリスティクス
 - 改善案①:
 - 保存/復元レジスタのセットがわかれば、出力パラメータを $D_f \setminus S_f$ で再定義可能
 - S_f : 関数 f 呼び出し時に保存され、リターン時に復元されるレジスタ集合

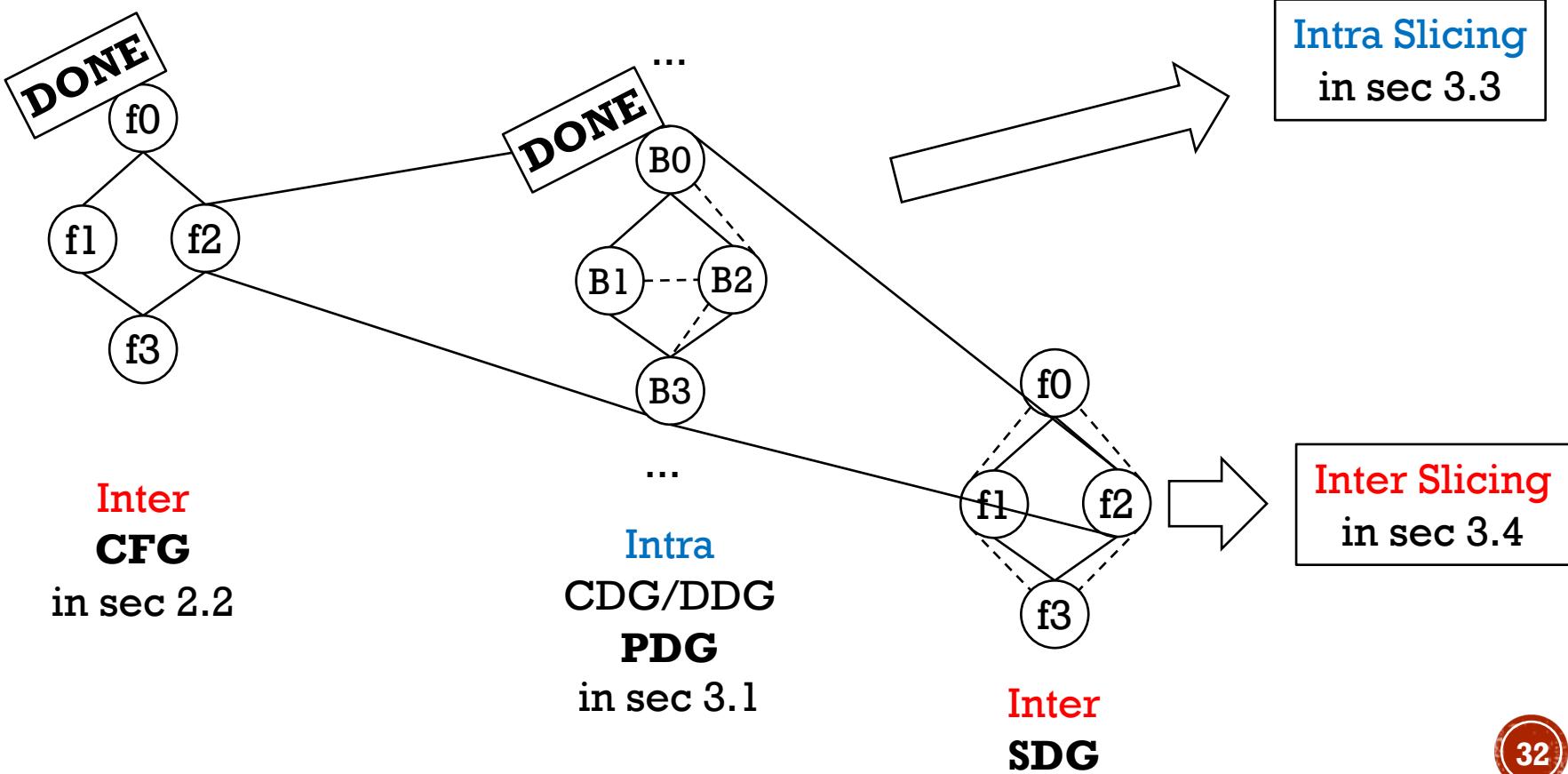
3. APPROACH

3.2 IMPROVE PDG

- 3.1までのPDGは安全だが過度に保守的
 - 改善案②:
 - メモリアクセスの解析
- 改善案②
 - 3.1までの段階ではメモリは単にメモリノード一つで表現
 - →安全だが大雑把すぎ
 - (特にスタック)メモリでの依存解析を行うことで細かい表現を期待
- 各命令の前後で全レジスタがメモリを指すか/どのメモリを指し得るかといった情報を収集
 - →(スタック)メモリノードを細粒化
- 例題のバイナリでは削減がうまく機能（省略）

3. APPROACH OVERVIEW

- バイナリスライス生成までの大まかな流れ



3. APPROACH

3.3 INTRAPROCEDURAL SLICING

- 3.1/3.2で得られたIntra PDGを用いることでスライスを計算可能
 - 任意の命令引数について計算
 - ノードから逆方向にグラフをトラバースすることで得られる
 - 出力は命令ノードの集合
- 関数呼び出しは入出力パラメータを持つ命令として処理

3. APPROACH

3.4 INTERPROCEDURAL SLICING

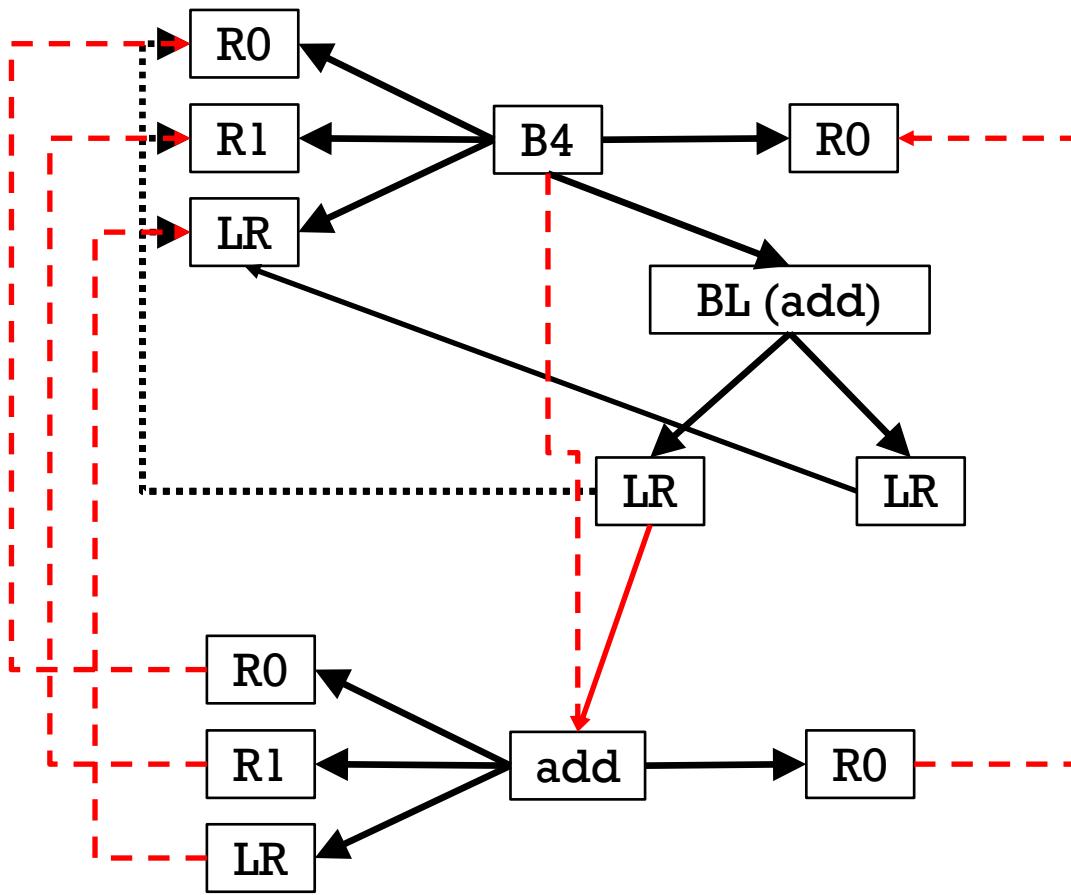
- 3.1/3.2までに生成したIntra PDGからSDGを構築
 - SDG: System Dependence Graph
 - 3.3同様構成された**SDG**を元に逆方向にトラバース
- SDGの構築
 - Intra PDGでの関数呼び出しでの入出力ノードを接続

3. APPROACH

3.4 INTERPROCEDURAL SLICING

- SDGの構築

- Intra PDGでの関数呼び出しでの入出力ノードを接続



4. EXPERIMENT

- 実験環境:
 - 検体: SPEC CINT2000[21], MediaBench benchmark suits[15]
 - ISA: ARM7T
 - コンパイラ:TMS470R1x Optimizing C Compiler ver. 1.27e
 - その他環境への言及なし

- 対象コードサイズ: 11~89 KBytes
 - 400KBytesのコードは計算が終了せず（何時間で打ち切ったのかは不明）

- 各検体についてはCFGを構築
 - PDGは3.1/3.2の両方を生成、比較

[21] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2000 benchmarks.

[15] MediaBench: A tool for evaluating and synthesizing multimedia and communications systems [ISM'97]

4. EXPERIMENT

- 検体情報
 - Code: バイナリサイズ
 - Instructions: 静的命令数
 - Source Lines: LOC

Table 1. Benchmark program size summary

Program	Code	Instructions	Source lines
ansi2knr	11,698	5,835	693
decode	14,594	7,283	1,593
bzip2	27,896	13,934	4,247
toast	32,464	16,218	5,997
sed	38,116	19,044	12,241
cjpeg	88,970	44,468	29,957

4. EXPERIMENT

- Conservative(3.1) vs Improve(3.2)
 - SDG構築時の比較
 - 各カラム: エッジ数

Table 3. SDG edge summary**Table 2. CFG edge summary**

Program	Control	Call	Control dependence	Data dependence	
				conservative	improved
ansi2knr	1,889	572	1964	66,424	51,470
decode	2,085	580	2029	71,216	56,086
bzip2	3,782	1,414	4129	203,308	161,161
toast	3,344	1,222	3459	190,204	135,583
sed	6,691	1,678	7192	795,928	470,836
cjpeg	9,407	3,628	10307	720,381	555,406

- 結果(conservativeを100%とした場合):
 - データ依存エッジは平均27%の削減
 - 最大削減はsedの41%の削減

4. EXPERIMENT

- Intra Slicing

- 各関数から5つの命令を選択 (cjpegのみ3つ)
 - 命令引数についてスライスを計算

Table 4. Intraprocedural slicing summary

Program	Criteria	Average size of functions (instructions)	Average size of slices (instructions)		
			conservative	improved	
ansi2knr	936	55	20	19	
decode	1,066	58	18	16	
bzip2	1,549	85	35	34	
toast	1,702	75	36	36	
sed	2,037	90	43	42	
cjpeg	1,361	88	51	50	

- 結果(スライスの平均命令数/関数の平均命令数) :
 - 保守: **31~58%のサイズを達成**
 - 改善: 保守からさらに-0~-4%程度の削減を達成
 - 3.2の改善案はスライスの削減には効果薄
 - 考察: スタック以外のメモリの保守性が阻害

4. EXPERIMENT

- Inter Slicing

- Intraと同じ条件でスライスを計算

Table 5. Interprocedural slicing summary

Program	Criteria	Size of program (instructions)	Average size of slices (instructions)	
			conservative	improved
ansi2knr	936	5,835	3,604	3,413
decode	1,066	7,283	4,121	3,874
bzip2	1,549	13,934	7,741	7,494
toast	1,702	16,218	9,019	8,718
sed	2,037	19,044	11,580	11,410
cjpeg	1,361	44,468	30,186	29,896

- 結果(スライスの平均命令数/プログラムの命令数):
 - 保守: **56~68%のサイズを達成**
 - 改善: 保守と比べて-1~-4%程度の削減を達成
 - 変わらずスライスには効果薄
 - 考察: メモリの保守性と未解決の関数呼び出しが阻害

CONCLUSION

- 2003年当時では初の(?) Interprocedural Binary Slicing
- 提案:
 - 未解決呼び出し/分岐を保守的に解決するCFGの構築法
 - Intra PDGの構築によるIntra SlicingとSDG構築によるInter Slicingに対応
 - 改善案: 関数呼び出し時のレジスタ保存/復元のヒューリスティクス
 - 改善案: スタックメモリの依存解析によるデータ依存エッジの削減
- 実験
 - 改善案によるデータ依存エッジの削減は効果的
 - サイズ比56~68%のInter Sliceを構築（保守的）
 - 改善案ではスライスのサイズ削減には効果薄

感想

- バイナリ対象のスライスの問題点を列挙はありがたし
 - x86で再現は苦労しそう (Pinである程度補える)
- 細かいグラフ構築のアルゴリズムを他論文にぶん投げ
 - →関連研究読まないと何故そのグラフになるのか分からん
- 3.2節は理解しきれていない
 - 結構な幅とっているくせに具体例がないのが残念
- 実験がなんだかなあ
 - 環境がわからないので解析がどの程度スケールするのか不透明
 - マシンスペックと解析にかかった時間も欲しい