DaCapo
: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption

Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, and Ju Min Lee, Yonsei University;
Sunchul Jung and Taekyung Kim, CryptoLab. Inc.;
Dongyoon Lee, Stony Brook University; Hanjun Kim, Yonsei University

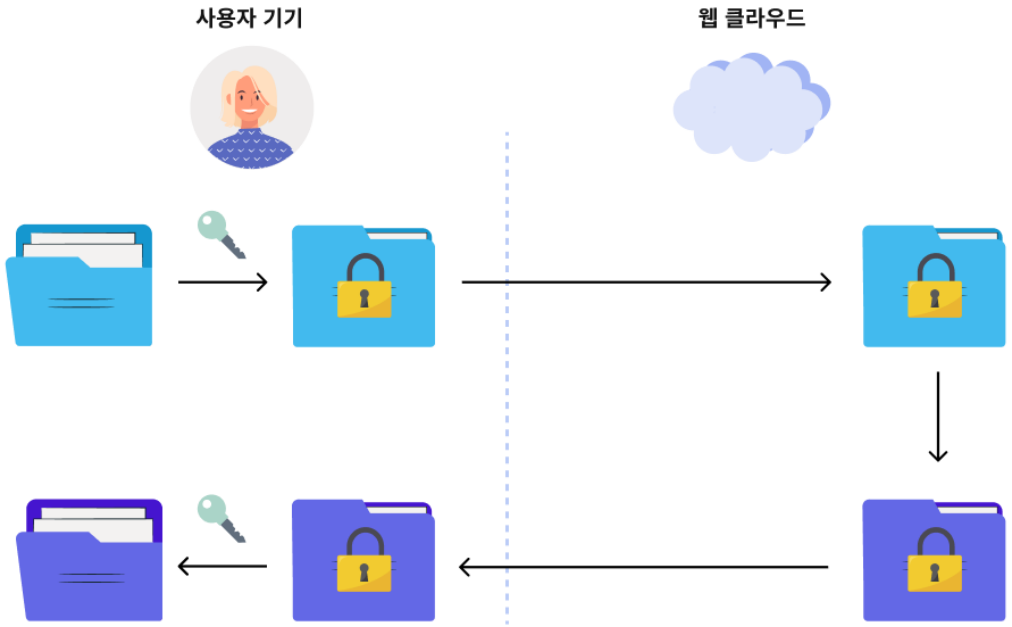
33rd USENIX Security Symposium.
August 14–16, 2024

Introduction

❖ FHE (Fully homomorphic encryption)

- **완전 동형 암호**
  - 암호화 상태에서 **Bootstraping**을 이용하여 **무제한 연산**이 가능하고, 연산의 **결과도 암호화된 상태로 제공**됨

→ 신뢰할 수 없는 환경에서 데이터 프라이버시를 보장하는 연산을 가능하게 하는 암호 기술



[그림 1] 동형 암호화를 사용하는 클라우드 스토리지

([그림1 출처)
https://adguard.com/ko/blog/homomorphic-encryption-era.html

Introduction

Q1. RNS-CKKS가 ML에 적합한 이유로 SIMD도 언급하였는데 원리가 이해가 잘 안갔습니다

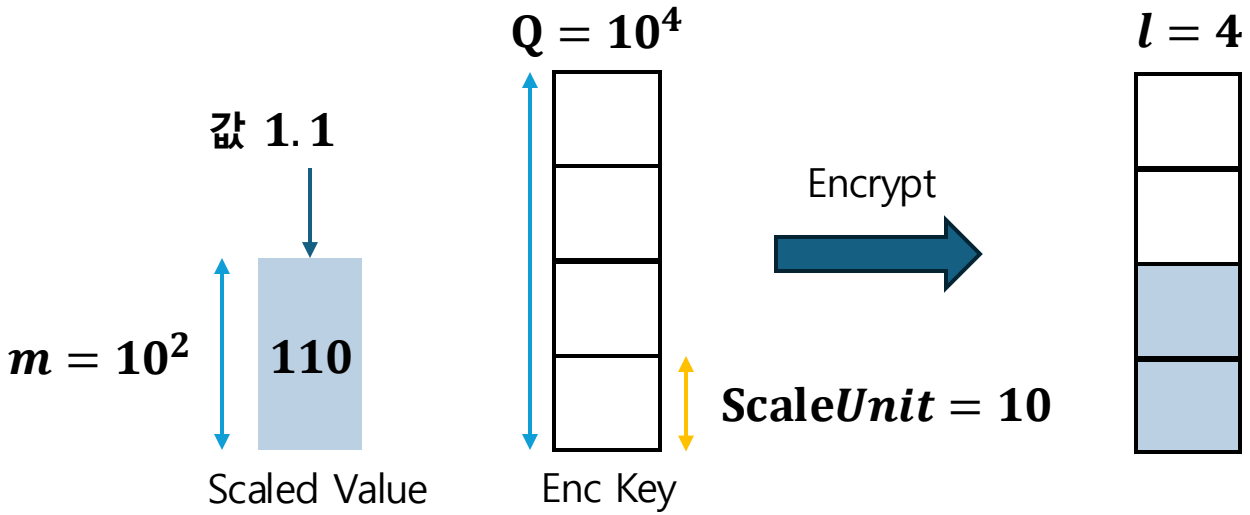
❖ RNS-CKKS

- **기계 학습(ML) 모델에 적합한 동형암호 스킴**
  - **고정소수점(Fixed-Point) 연산** 지원
    - 실수를 직접 암호화할 수 없기 때문에 실수를 정수로 변환(Encoding)한 후 암호화하는 방식을 사용

Introduction

❖ RNS-CKKS

- **Scale (m) :** 실수 값을 정수로 변환하기 위한 요소
- **Coefficient Modulus (Q) :** 스케일링된 값을 저장할 수 있는 최대 용량
  - 즉, 암호화된 값이 저장될 수 있는 공간의 크기
- **Level (l) :** 사용 가능한 스케일링 유닛 수

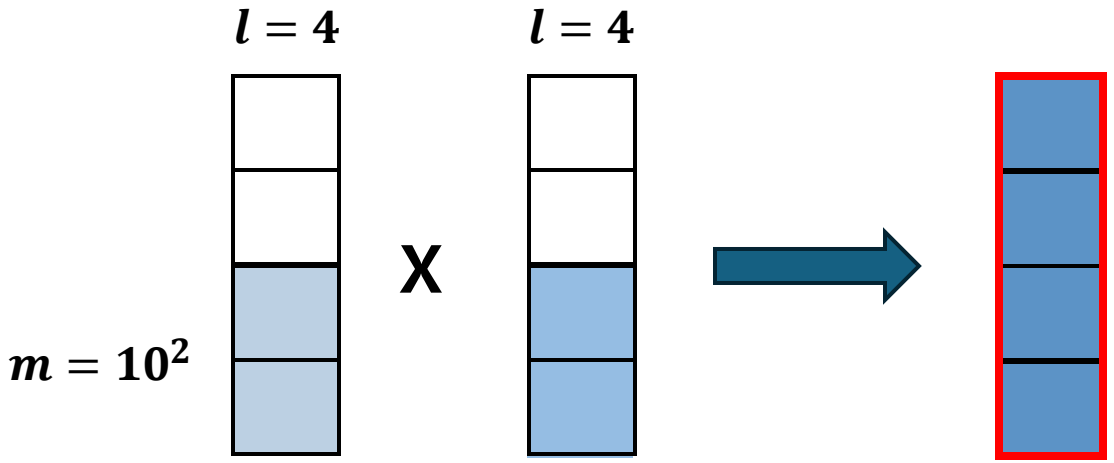


[그림 2] Scale10^2을 이용한 1.1 암호화 예시

Introduction

❖ RNS-CKKS 곱셈 연산

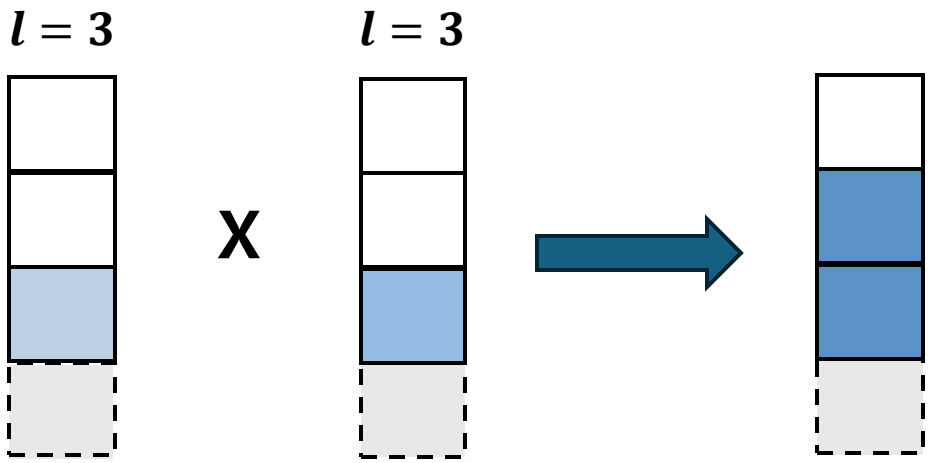
- 곱셈 연산을 수행하려면 두 암호문의 레벨 l이 동일해야 함
- 스케일 m은 곱셈 연산 후 누적됨



[그림 3] Multiplication

❖ RNS-CKKS 리스케일 연산

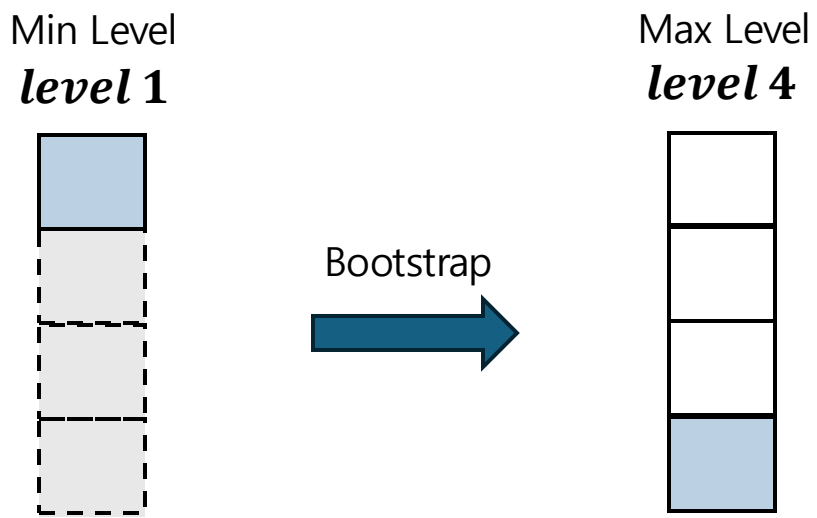
- 스케일이 너무 커지면 Rescale 연산을 수행해야 함



[그림 4] Rescale

❖ RNS-CKKS 부트스트래핑 연산

- 레벨이 소진된 암호문을 **복구**하여 **추가적인 연산**이 가능하도록 만듦
  - CKKS에서 **가장 비용이 많이 드는** 연산이므로, 최적의 시점을 찾아 효율적으로 수행할 필요가 있음

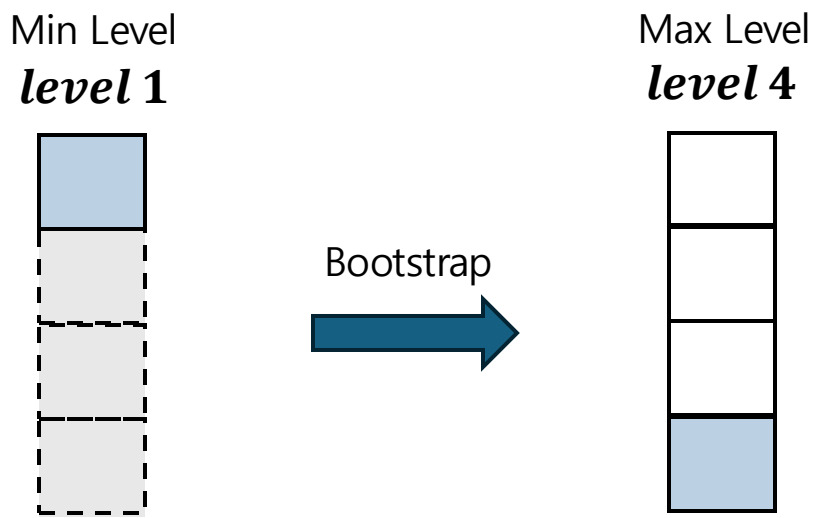


[그림 5] Bootstrapping

❖ RNS-CKKS 부트스트래핑 연산

- 레벨이 소진된 암호문을 **복구**하여 **추가적인 연산**이 가능하도록 만듦
  - CKKS에서 **가장 비용이 많이 드는** 연산이므로, 최적의 시점을 찾아 효율적으로 수행할 필요가 있음

⇒ FHE 환경에서 실용적인 연산을 수행하려면 부트스트래핑을 최소화하면서도, 최적의 타이밍에 수행해야 함



[그림 5] Bootstrapping

Motivation

❖ 기존 RNS-CKKS 컴파일러

- 기존의 RNS-CKKS 컴파일러는 **자동 스케일 관리**를 지원.
  - 하지만 부트스트래핑 자동 삽입을 고려하지 못함.

→ LeNet-5 와 같은 상대적으로 작은 규모의 딥러닝 모델에는 적용 가능하지만, 곱셈 연산 depth가 깊은 복잡한 모델에는 적용하기 어려운 한계를 가짐.

⇒ 이를 해결하기 위해 부트스트래핑을 지원하는 새로운 컴파일러가 필요함

Motivation

❖ 기존 RNS-CKKS 컴파일러 – EVA(Encrypted Vector Arithmetic))[18]

Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2020.

- 입력 프로그램을 받아 고정 소수점 스케일과 출력 스케일을 고려하여 최적화된 FHE 프로그램을 생성

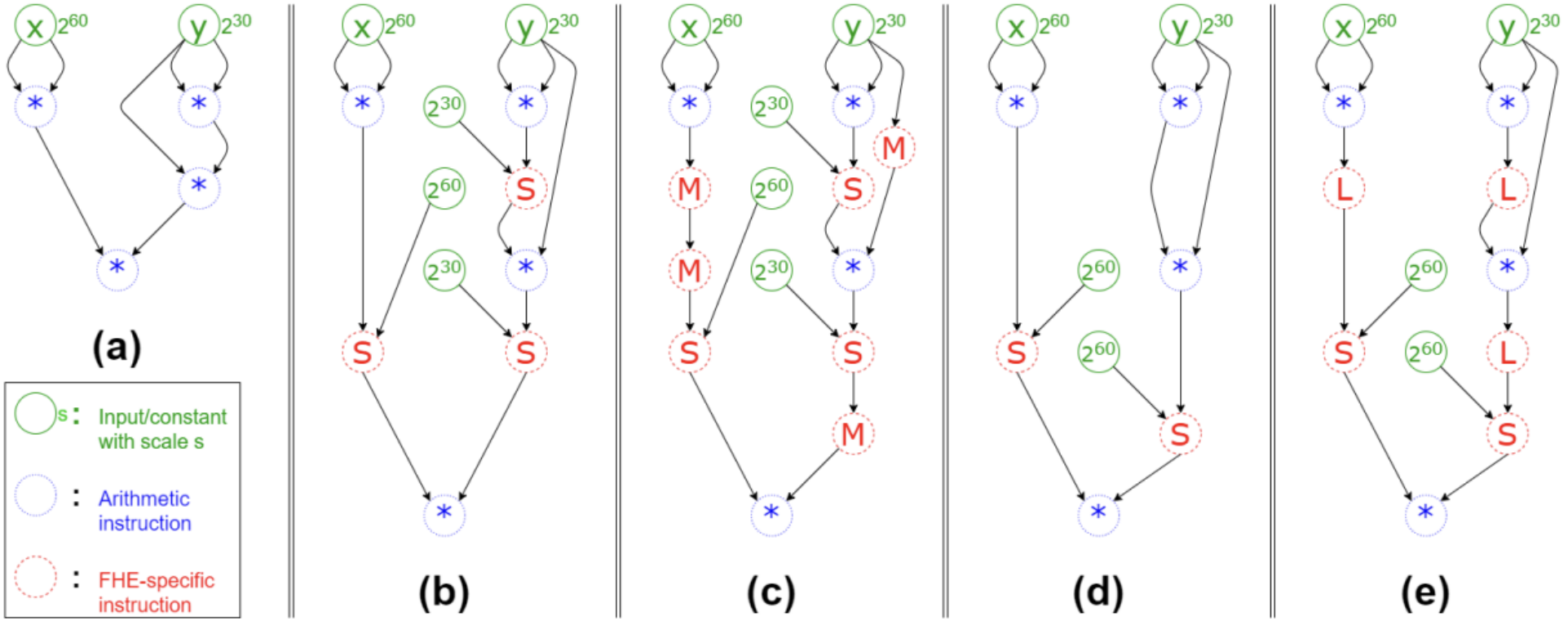


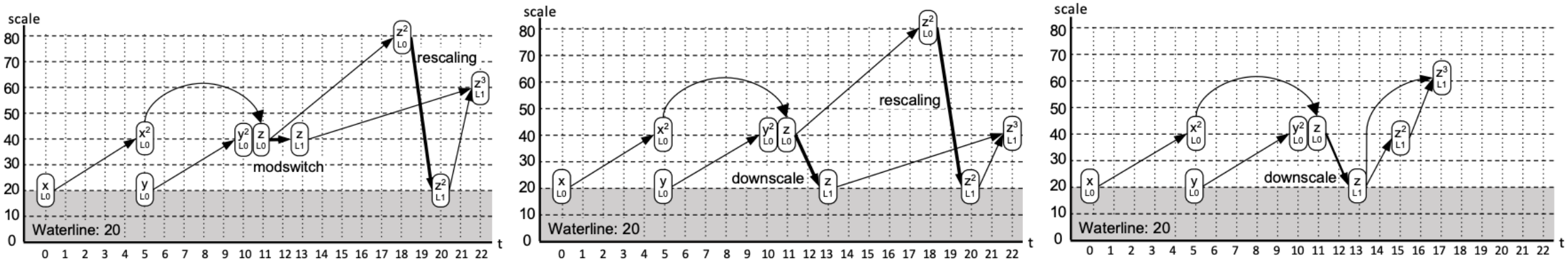
Figure 1. x^2y^3 example in EVA: (a) input; (b) after ALWAYS-RESCALE; (c) after ALWAYS-RESCALE & MODSWITCH; (d) after WATERLINE-RESCALE; (e) after WATERLINE-RESCALE & RELINEARIZE (S: RESCALE, M: MODSWITCH, L: RELINEARIZE).

Motivation

❖ 기존 RNS-CKKS 컴파일러 – HECATE[41]

YongwooLee,SeonyeongHeo,SeonyoungCheon,Shin- nung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. HECATE: Performance-Aware Scale Optimization for Homomoprhic Encryption Com- piler. In 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2022.

- 능동적 스케일 관리와 다운스케일 연산을 도입하여 스케일을 효율적으로 조정



(a) Existing HE compiler (EVA) (b) Proactive rescaling (PARS) (c) Scale management space explorer (SMSE)

Fig. 2: Comparison of the scale management schemes of EVA and HECATE for the example program which calculates $(x^2 + y^2)^3$ which is a part of root mean square. `rescale` reduces the scale by 2^{60}, and increases the level by one. `modswitch` only increases the level, and `downscale` reduces the scale to the waterline while increasing its level.

Motivation

❖ Challenges of Manual Bootstrapping Placement

1) 스케일 오버플로우 방지 및 정확성 보장

- ✔ 부트스트래핑은 누적 스케일이 **최대 용량을 초과하기 전**에 수행되어야 함.
- ✔ 산술 연산뿐만 아니라 rotation, rescale 연산까지 고려해야 하므로 **누적 스케일 추적이 어려움**.

Motivation

❖ Challenges of Manual Bootstrapping Placement

1) 스케일 오버플로우 방지 및 정확성 보장

- ✔ 부트스트래핑은 누적 스케일이 **최대 용량을 초과하기 전**에 수행되어야 함.
- ✔ 산술 연산뿐만 아니라 rotation, rescale 연산까지 고려해야 하므로 **누적 스케일 추적이 어려움**.

2) 성능 최적화 문제

- ✔ 부트스트래핑은 RNS-CKKS에서 가장 비용이 큰 연산으로, **높은 latency**를 초래.
- ✔ 가능한 부트스트래핑을 **최소화**하는 것이 유리함.

⇒ 그러나 특정 경우에는 **오히려 더 많은 부트스트래핑이 성능을 향상**시킬 수도 있기에
최적의 부트스트래핑 횟수와 위치를 결정하는 것은 **복잡한 문제**임.

❖ Challenges of Manual Bootstrapping Placement

3) 부트스트래핑 배치가 미치는 영향

- ✔ 부트스트래핑은 암호문의 스케일과 레벨을 변경하여 **이후 연산에 영향**을 줌.
- ✔ 동일한 부트스트래핑 개수라도 **위치에 따라 성능 차이**가 발생하므로 최적의 배치가 복잡한 문제임.

Motivation

❖ Challenges of Manual Bootstrapping Placement

3) 부트스트래핑 배치가 미치는 영향

- ✔ 부트스트래핑은 암호문의 스케일과 레벨을 변경하여 **이후 연산에 영향**을 줌.
- ✔ 동일한 부트스트래핑 개수라도 **위치에 따라 성능 차이**가 발생하므로 최적의 배치가 복잡한 문제임.

자동 부트스트래핑 지원 컴파일러의 필요성

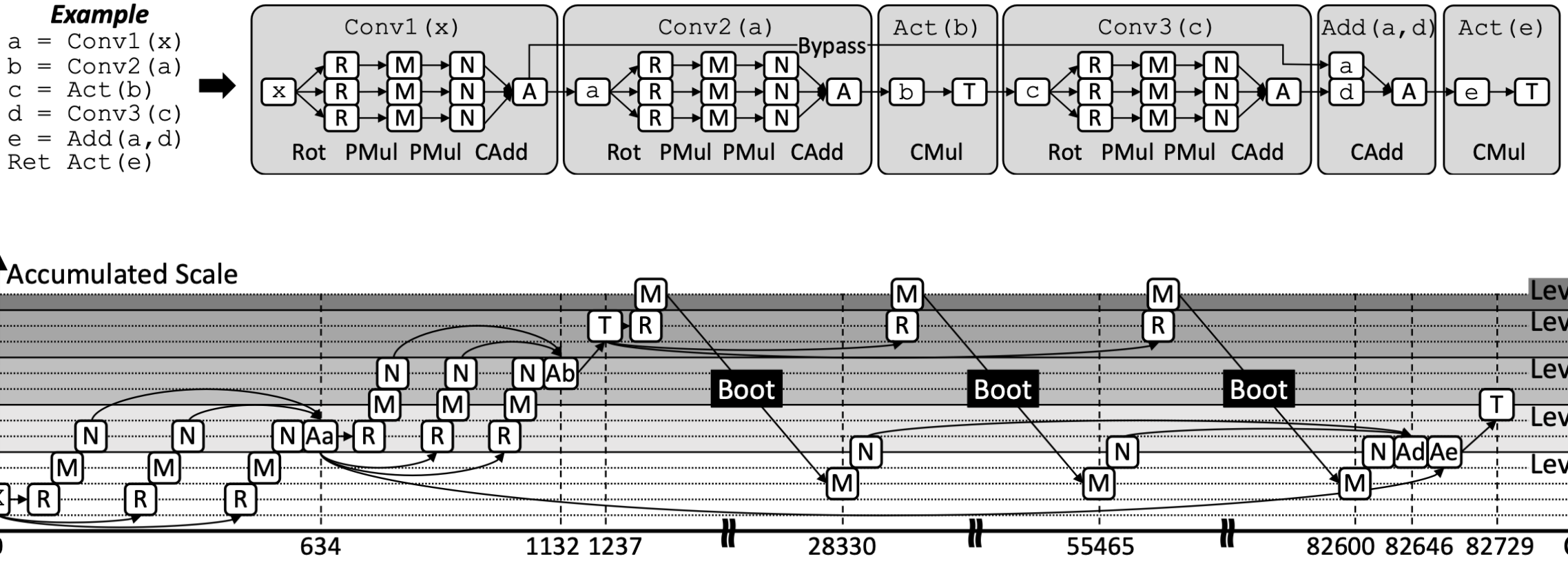
- ✔ 기존 RNS-CKKS 컴파일러는 부트스트래핑을 자동화하지 못해 한계가 있음.
- ✔ 수동 부트스트래핑은 **오류 가능성**이 높고 **최적화가 어려운 문제**를 초래함.
- ✔ 부트스트래핑을 **자동으로 최적 배치할 수 있는 새로운 컴파일러 기술**이 필요함.

❖ DACAPO

- **DACAPO: 제안된 자동 부트스트래핑 관리 컴파일러**
  - 부트스트래핑을 자동으로 수행하는 DACAPO가 성능 최적화를 가능하게 함.
  - **핵심 기능**
    - 자동 부트스트래핑 배치
    - 비용 기반 최적화
  - **성능 평가**
    - 기존 FHE 컴파일러가 지원하지 않는 딥러닝 모델(예: ResNet, AlexNet, VGG 등)을 평가하여, 수동으로 구현된 FHE 프로그램 대비 **평균 1.21배**의 성능 향상을 달성

Bootstrapping 1

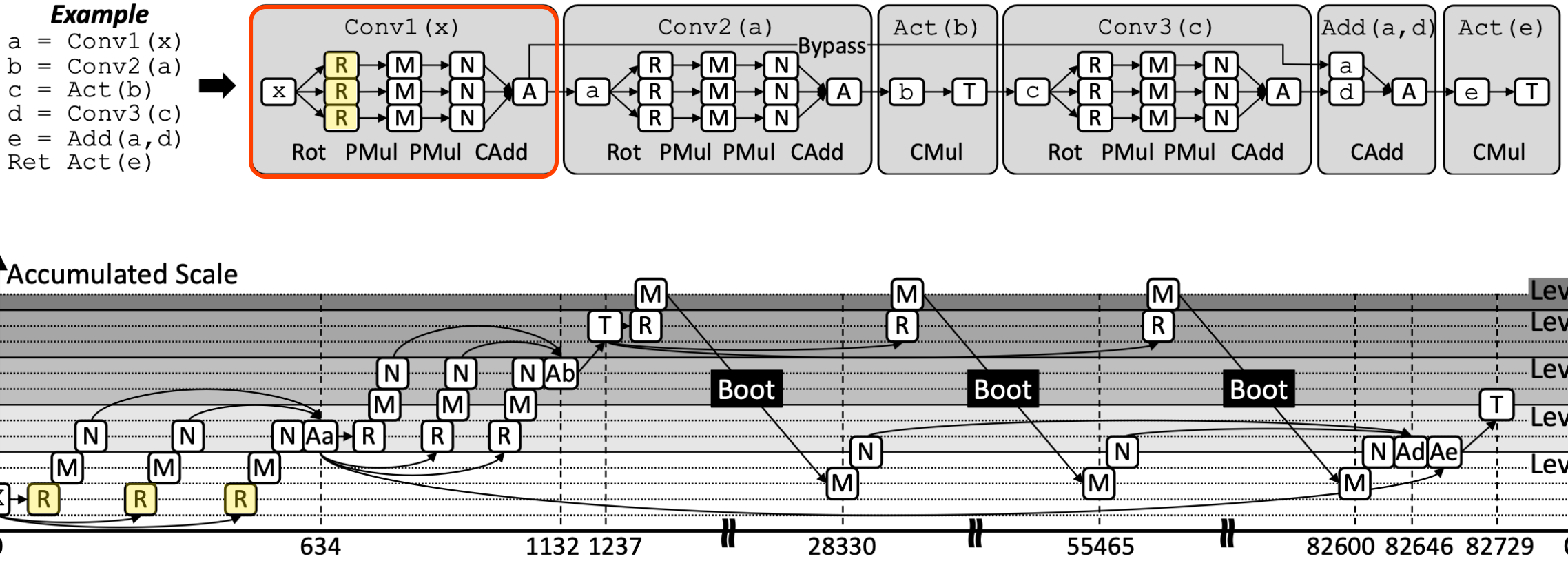
❖ 기본적인 부트스트래핑 접근법



[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 1

❖ 기본적인 부트스트래핑 접근법

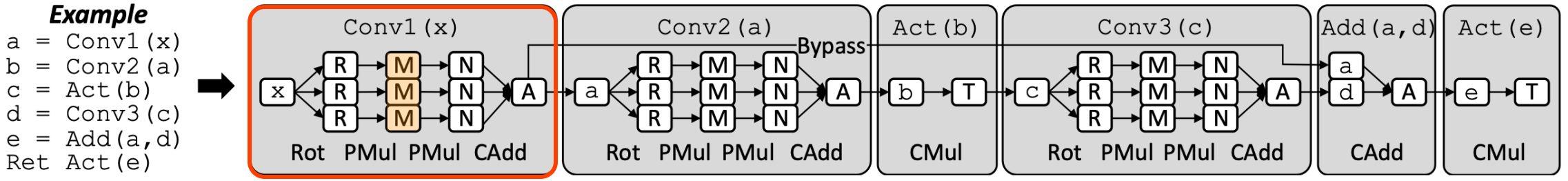


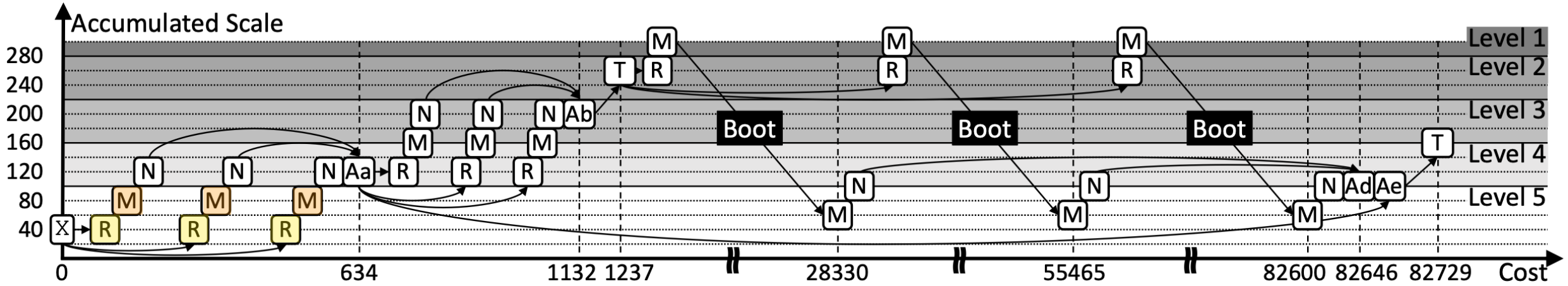
[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 1

Q2. PMul은 암호문*평문 간 곱셈인데 어떻게 두 번 있는지

❖ 기본적인 부트스트래핑 접근법

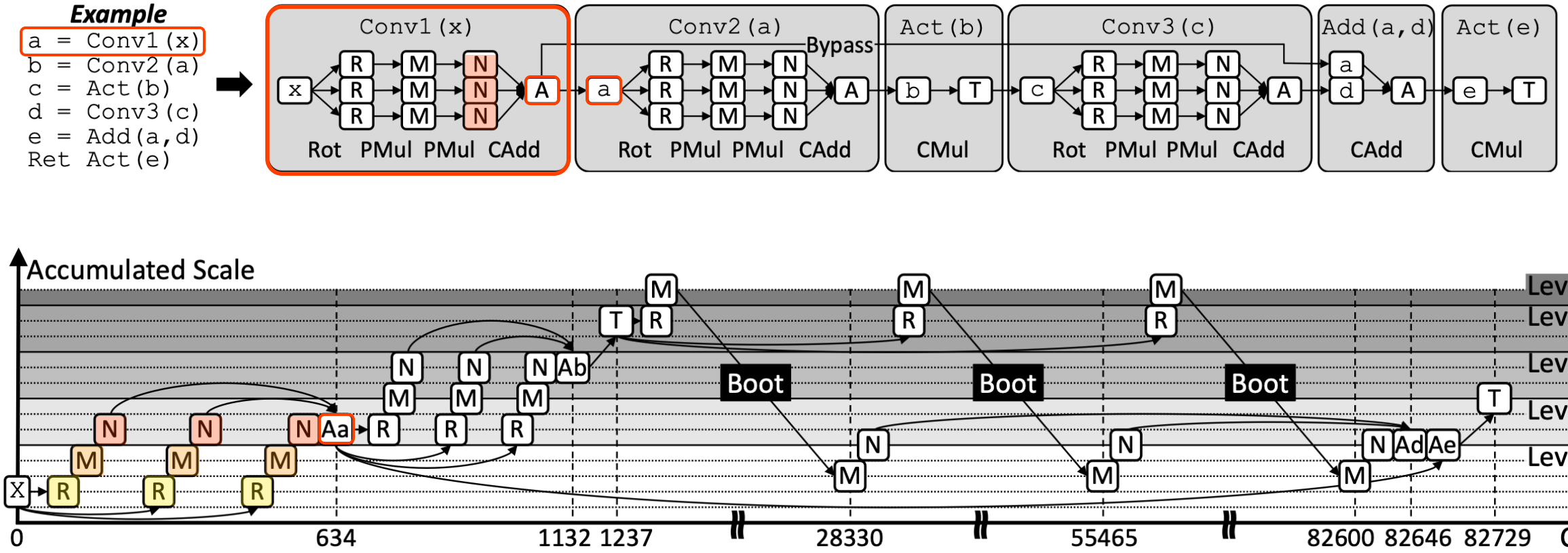




[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 1

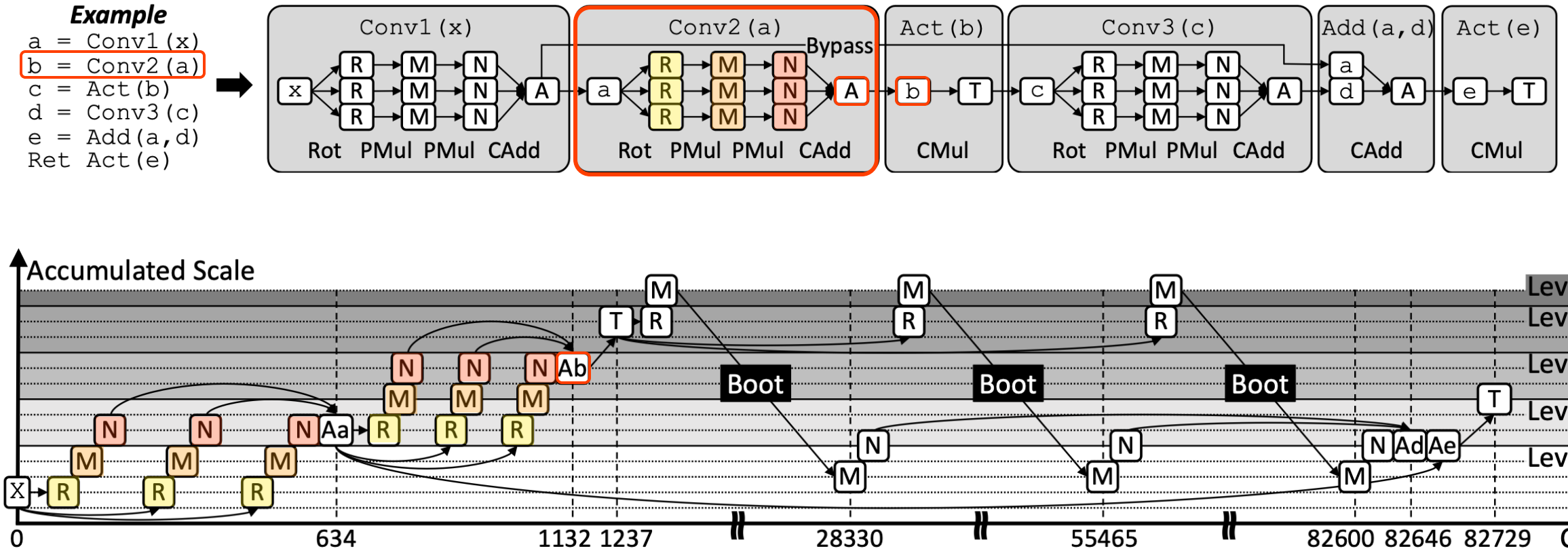
❖ 기본적인 부트스트래핑 접근법



[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 1

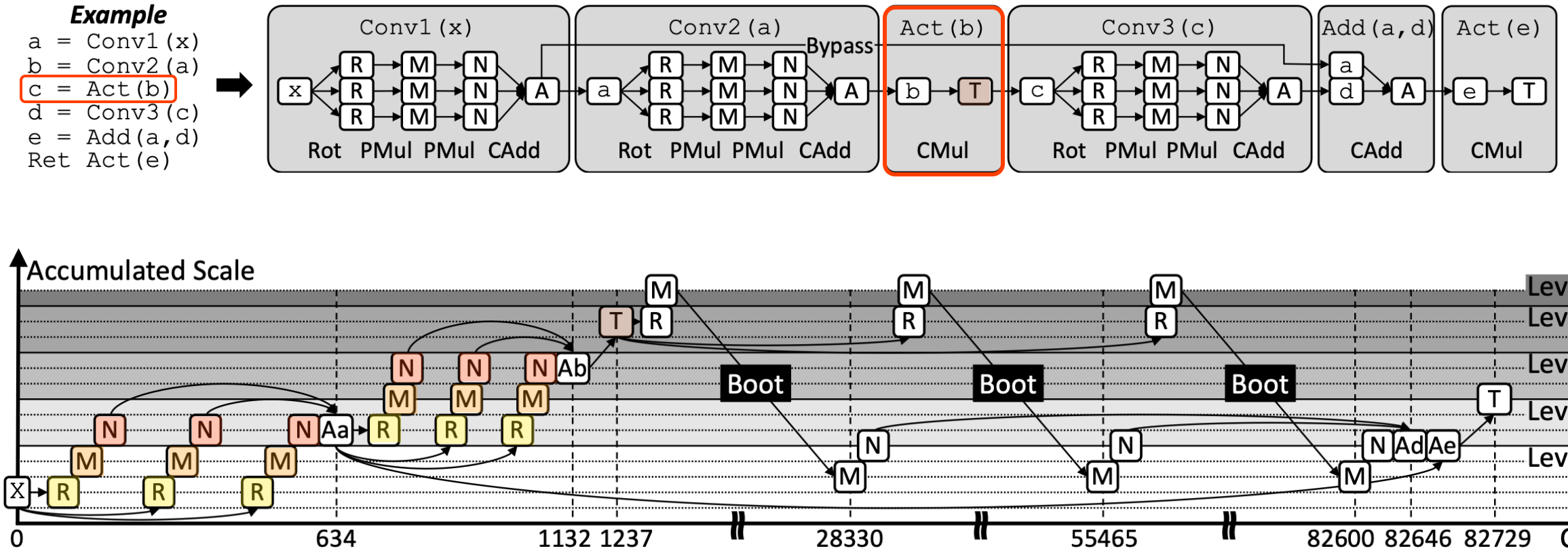
❖ 기본적인 부트스트래핑 접근법



[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 1

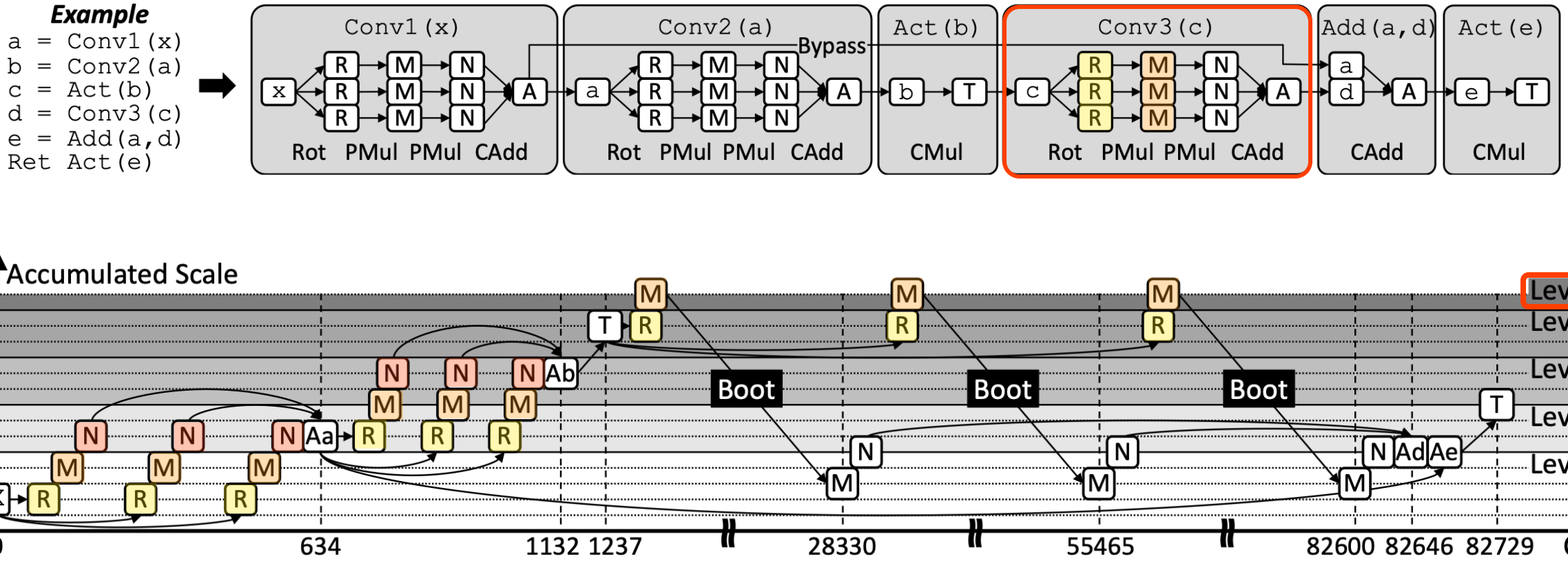
❖ 기본적인 부트스트래핑 접근법



[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 1

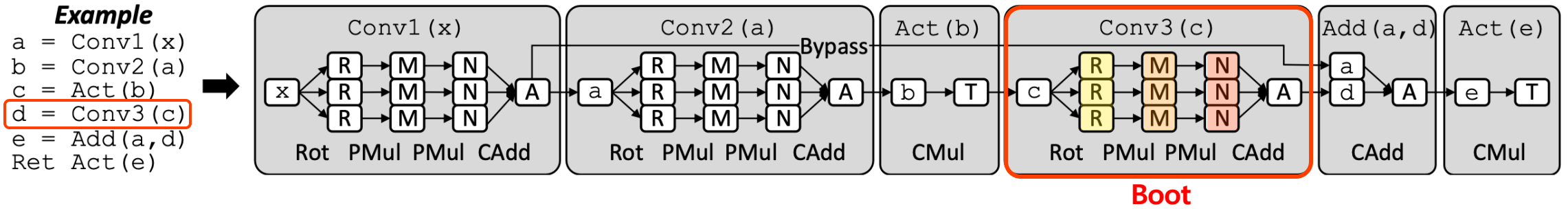
❖ 기본적인 부트스트래핑 접근법

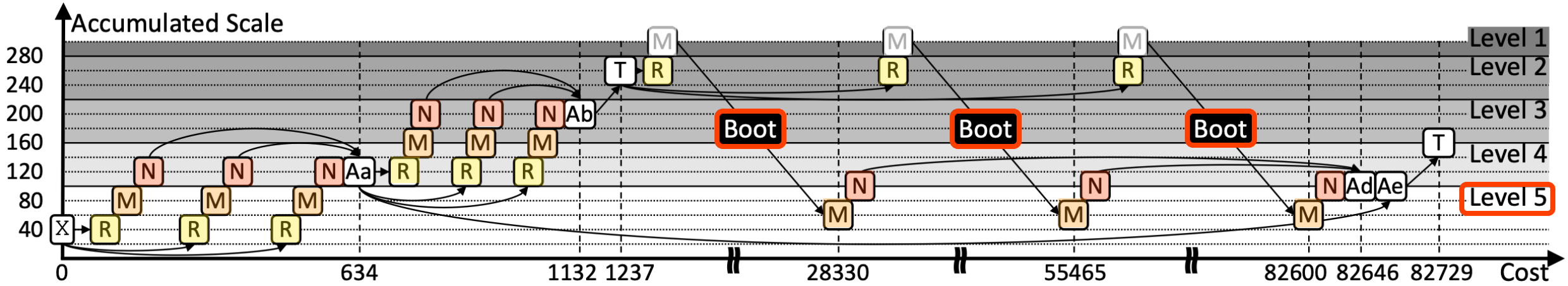


[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 1

❖ 기본적인 부트스트래핑 접근법





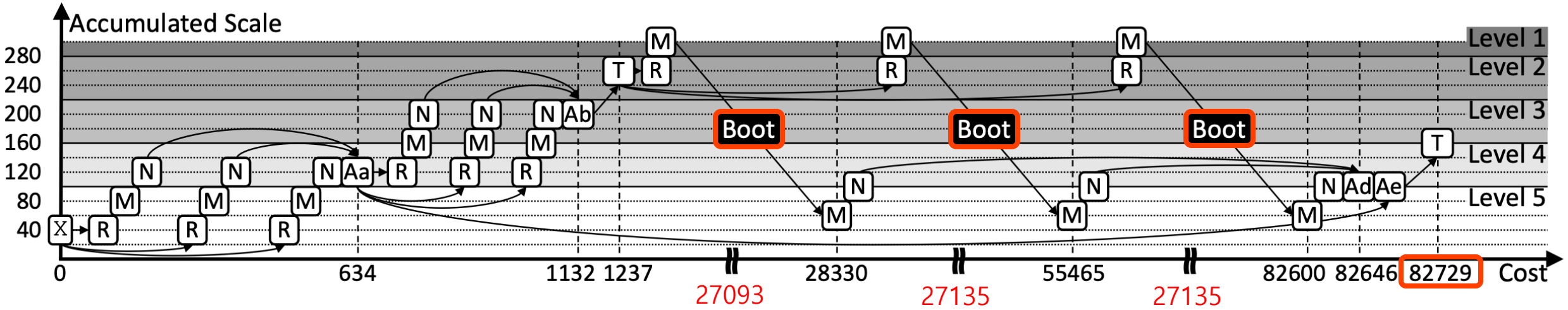
[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 1

Q2. 3채널에 대한 연산을 동시에(?) 수행하는 것인지
여기서 BootStrapping 마다 Cost가 왜 약간씩 다른지

❖ 기본적인 부트스트래핑 접근법

- **데이터 흐름이 분기**되면서 **여러 번**의 부트스트래핑이 필요해짐.
  - 3번의 Bootstrapping → 불필요한 부트스트래핑으로 인해 높은 비용 발생.



[그림 6] 기본적인 부트스트래핑 방식 동작 과정

Bootstrapping 2

❖ Liveness 기반 Bootstrapping 접근법

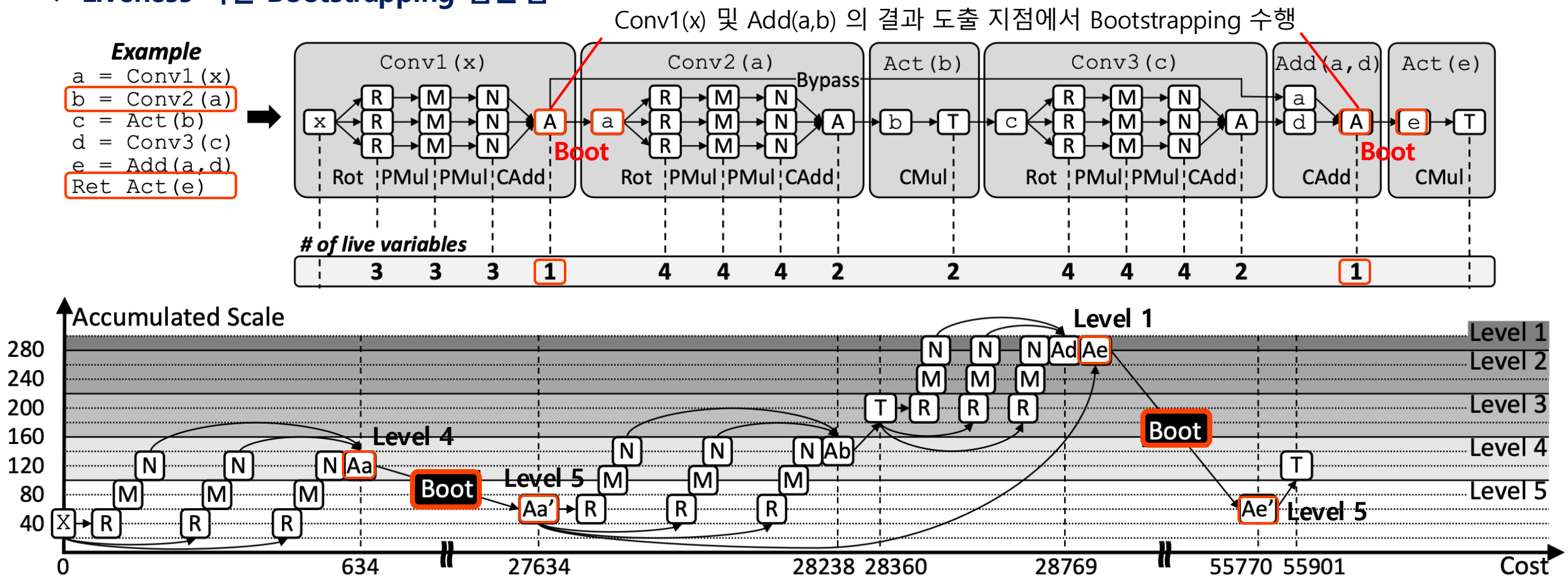
- 기존 방법: 데이터 흐름이 분기되면서 여러 암호문이 사용될 경우, 각 분기된 암호문에 대해 부트스트래핑이 필요해져 비효율적
- 해결책: **liveness analysis**을 통해 각 프로그램 지점에서 **live-out 암호문**의 개수를 계산하고, **가장 적은 live-out을 가지는 지점**에서 부트스트래핑을 수행

- **Liveness Analysis**: 프로그램의 각 지점에서 어떤 변수가 살아있을지(live)를 분석하는 것
- **live-out 암호문**: 특정 프로그램 지점에서 아직 사용되지 않았지만, 이후에 사용될 암호문

Bootstrapping 2

Q3. 왜 Add(a,d)를 했는지
Q4. Conv1(x)의 A에서 live variable 수가 1인 점

❖ Liveness 기반 Bootstrapping 접근법

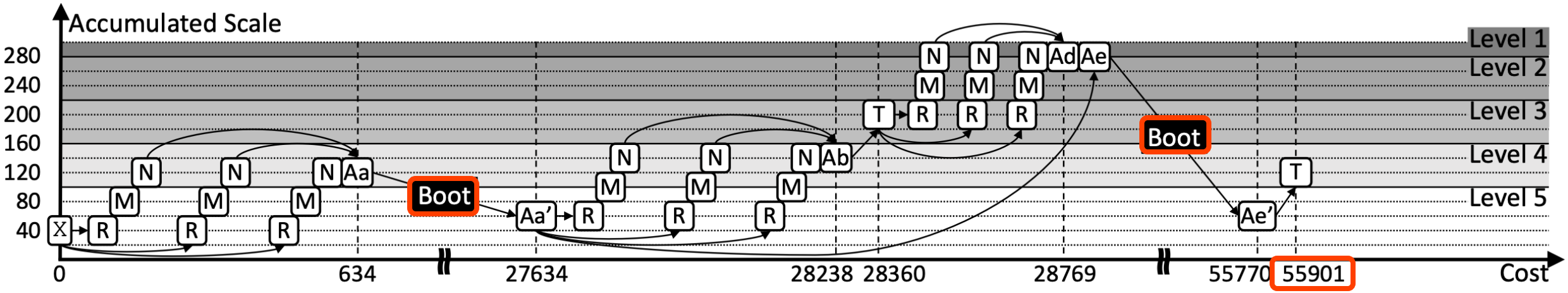


[그림 7] Liveness 기반 Bootstrapping 방식 동작 과정

Bootstrapping 2

❖ Liveness 기반 Bootstrapping 접근법

- **Live-out 개수가 최소화**되는 지점에서 Bootstrapping을 수행하여 **불필요한 연산을 제거**
- 2번의 Bootstrapping



[그림 7] Liveness 기반 Bootstrapping 방식 동작 과정

Bootstrapping 3

❖ Bypassed Liveness-aware Bootstrapping 접근법

- 기존 Live-out 기반 Bootstrapping 방식에서는 **모든 암호문이 이후 연산에서 사용될 가능성이 있는지 여부**를 고려하지 않음
  - 장기간 사용되지 않더라도 나중에 한 번이라도 사용되면 이를 Live-out 변수로 포함하여 Bootstrapping을 수행함

    ⇒ 불필요한 연산 낭비 발생

- **Bypassed 암호문**: **일정 기간 동안 사용되지 않다가** 나중에 다시 사용되는 암호문
  - 데이터 흐름 상 한참 동안 연산에 참여하지 않는 것

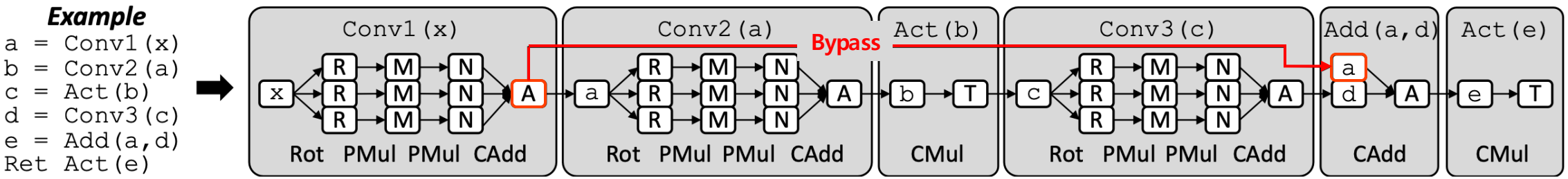
Bootstrapping 3

❖ Bypassed Liveness-aware Bootstrapping 접근법

- 기존 Live-out 기반 Bootstrapping 방식에서는 **모든 암호문이 이후 연산에서 사용될 가능성이 있는지 여부**를 고려하지 않음
  - 장기간 사용되지 않더라도 나중에 한 번이라도 사용되면 이를 Live-out 변수로 포함하여 Bootstrapping을 수행함

    ⇒ 불필요한 연산 낭비 발생

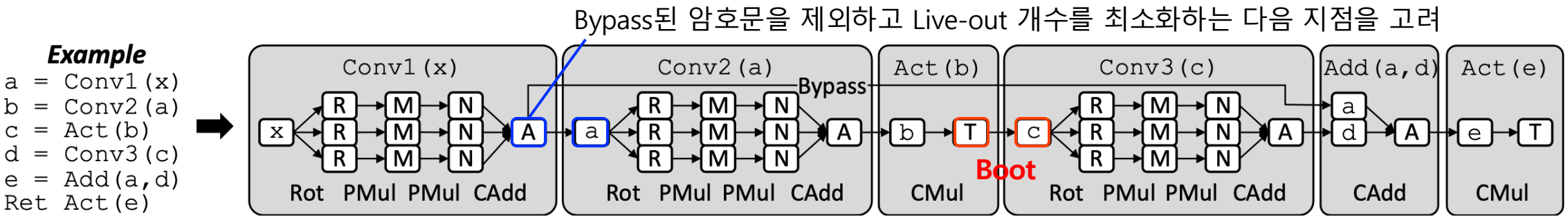
- **Bypassed 암호문**: **일정 기간 동안 사용되지 않다가** 나중에 다시 사용되는 암호문
  - 데이터 흐름 상 한참 동안 연산에 참여하지 않는 것

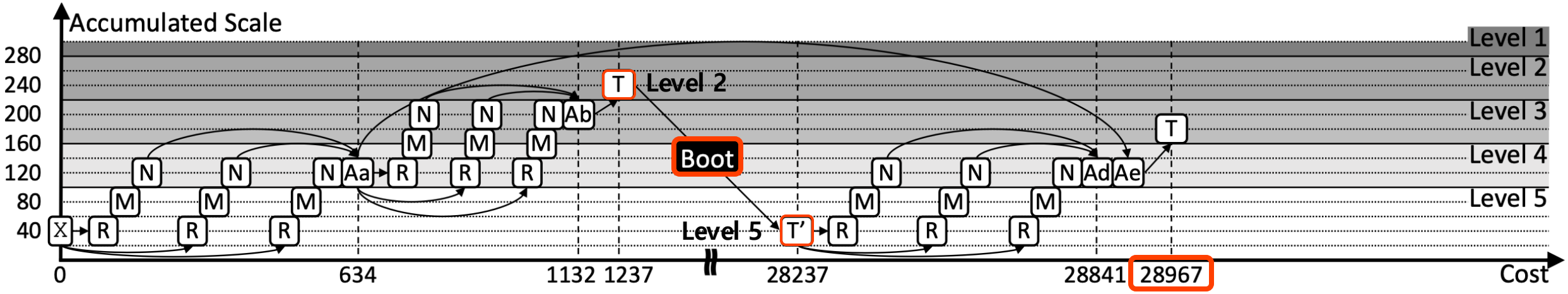


[그림 8] Bypassed Liveness-aware Bootstrapping 방식 동작 과정

Bootstrapping 3

❖ Bypassed Liveness-aware Bootstrapping 접근법



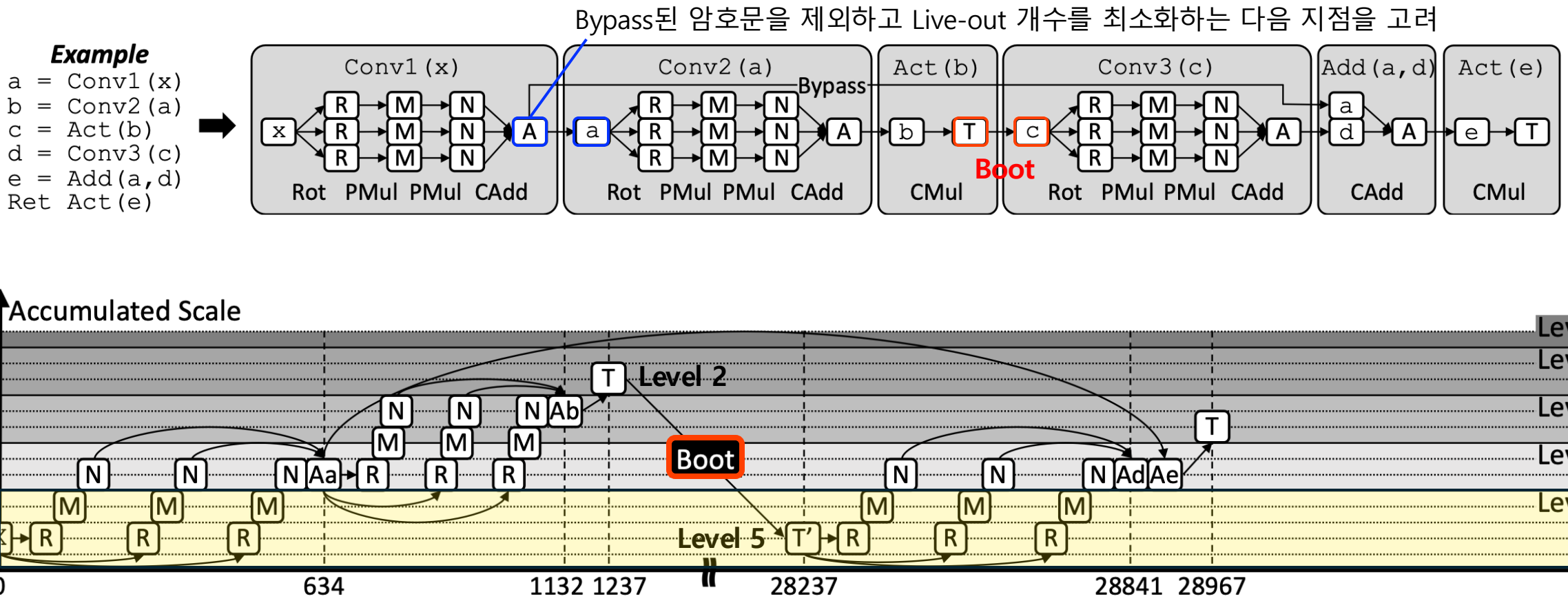


기존 방법 대비 약 64.99% 감소

[그림 8] Bypassed를 고려한 Bootstrapping 방식 동작 과정

Bootstrapping 3

❖ Bypassed Liveness-aware Bootstrapping 접근법

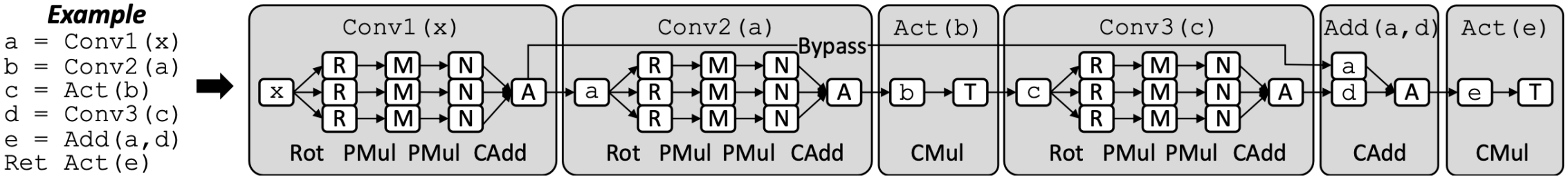


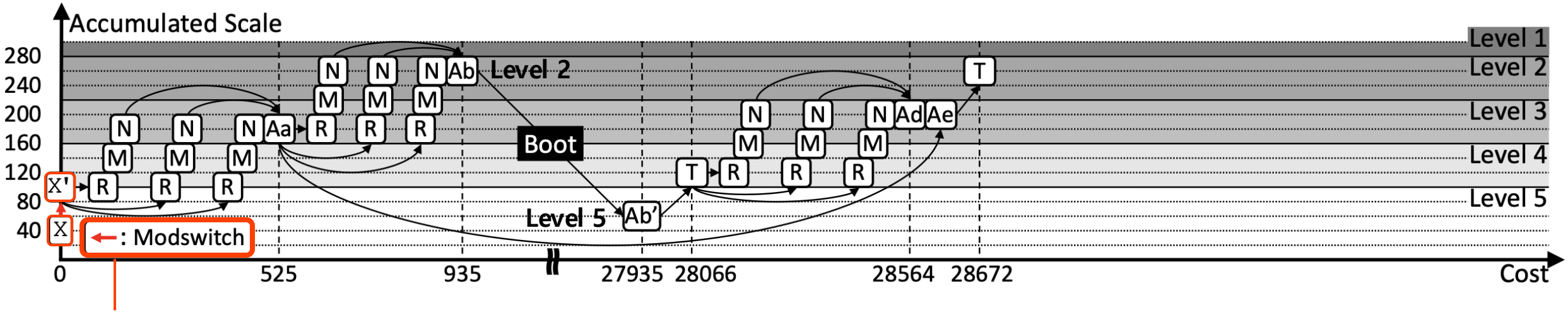
[그림 8] Bypassed를 고려한 Bootstrapping 방식 동작 과정

Bootstrapping - DACAPO

Q5. 맨 처음 Modswitch는 Cost에 포함되지 않는지

❖ DACAPO의 비용을 고려한 Bootstrapping 접근법



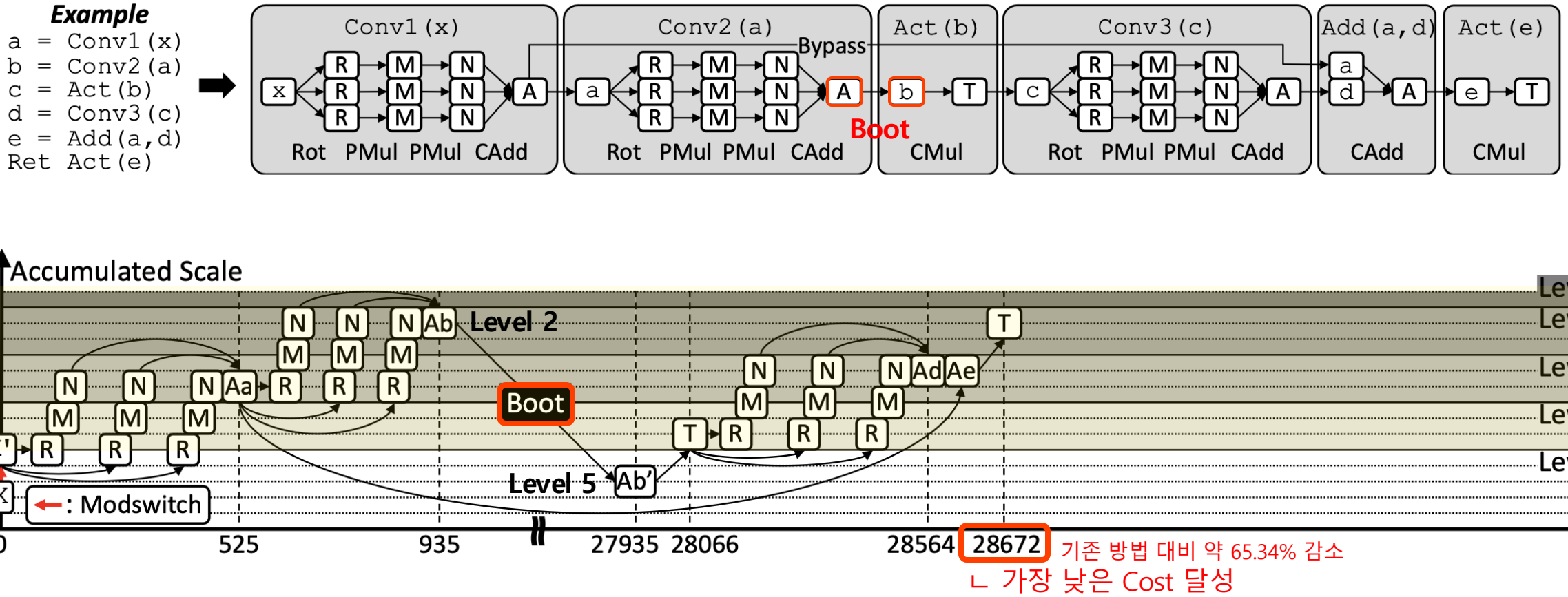


초기에 **ModSwitch**를 하여 이후 수행될 연산이 **더 낮은 레벨에서 실행**될 수 있도록 함
(Level 5 → 4)

[그림 9] DACAPO의 비용을 고려한 Bootstrapping 방식 동작 과정

Bootstrapping - DACAPO

❖ DACAPO의 비용을 고려한 Bootstrapping 접근법



[그림 9] DACAPO의 비용을 고려한 Bootstrapping 방식 동작 과정

❖ DACAPO 개요

- **부트스트래핑(bootstrapping)** 연산을 자동으로 배치하는 최초의 **FHE 컴파일러**
- 입력 프로그램을 RNS-CKKS 연산으로 변환하고, 부트스트래핑을 포함한 **스케일 관리 연산**을 적용하여 GPU 가속 HEaaN 라이브러리를 호출하는 **LLVM IR** 코드 생성
- **비용 인식 기반(cost-aware)** 부트스트래핑 배치 지원

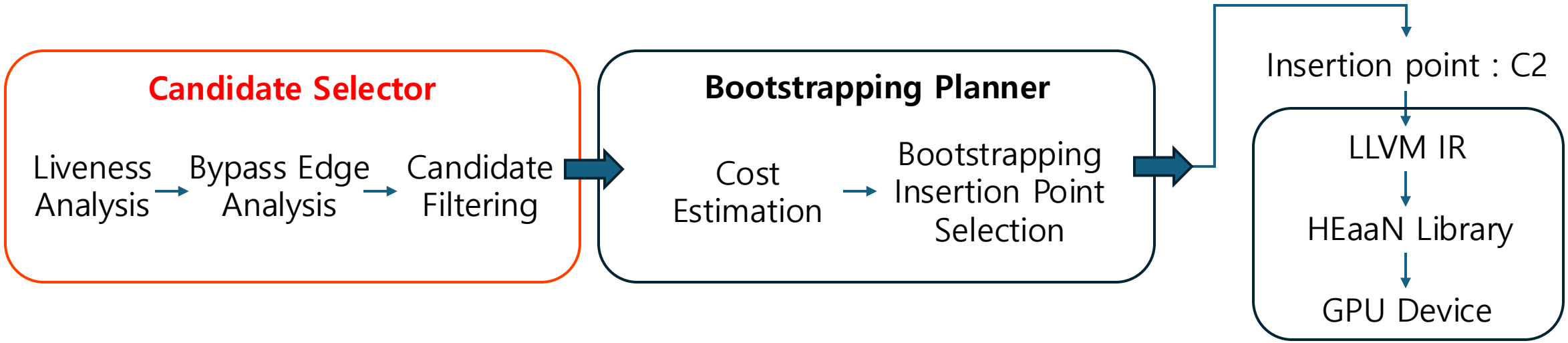
핵심 구성요소

1. **후보 선택기(Candidate Selector)**
2. **부트스트래핑 계획 수립기(Bootstrapping Planner)**

DACAPO - Candidate Selector

❖ 후보 선택기 (Candidate Selector)

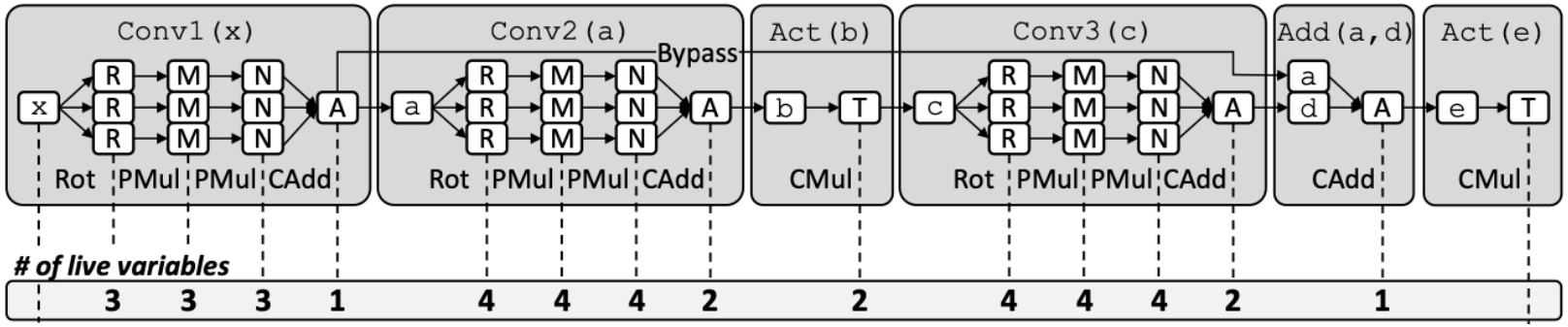
- 목적: 프로그램 내에서 **부트스트래핑을 삽입**할 수 있는 **후보 지점(candidate points)을 식별**
  - 세 단계로 구성
    1) 생존성 분석 (Liveness Analysis)
    2) 오랫동안 사용되지 않다가 나중에 다시 사용되는 암호문 탐지 (Bypass Edge Analysis)
    3) 부트스트래핑 후보 지점 필터링 (Candidate Filtering)

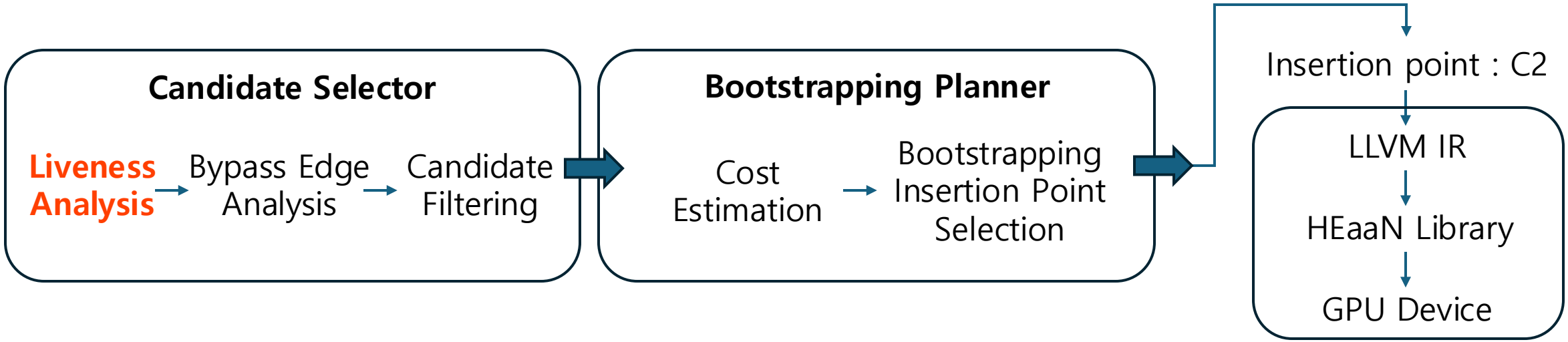


DACAPO - Candidate Selector

❖ 1) 생존성 분석(Liveness Analysis)

- 각 프로그램 지점에서 **live-out** 암호문 개수를 분석

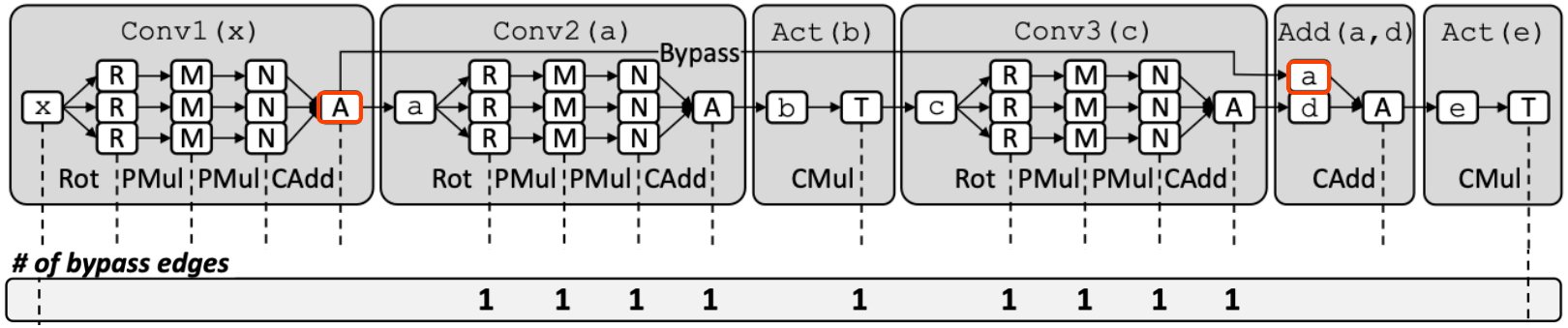


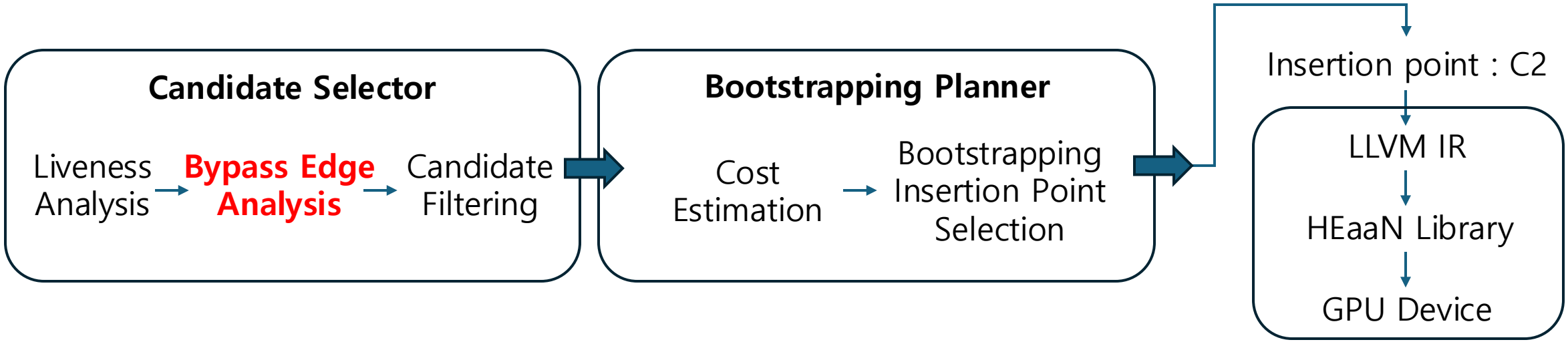


DACAPO - Candidate Selector

❖ 2) 바이패스 엣지 분석(Bypass Edge Analysis)

- 오랫동안 사용되지 않다가 나중에 다시 사용되는 암호문 탐지

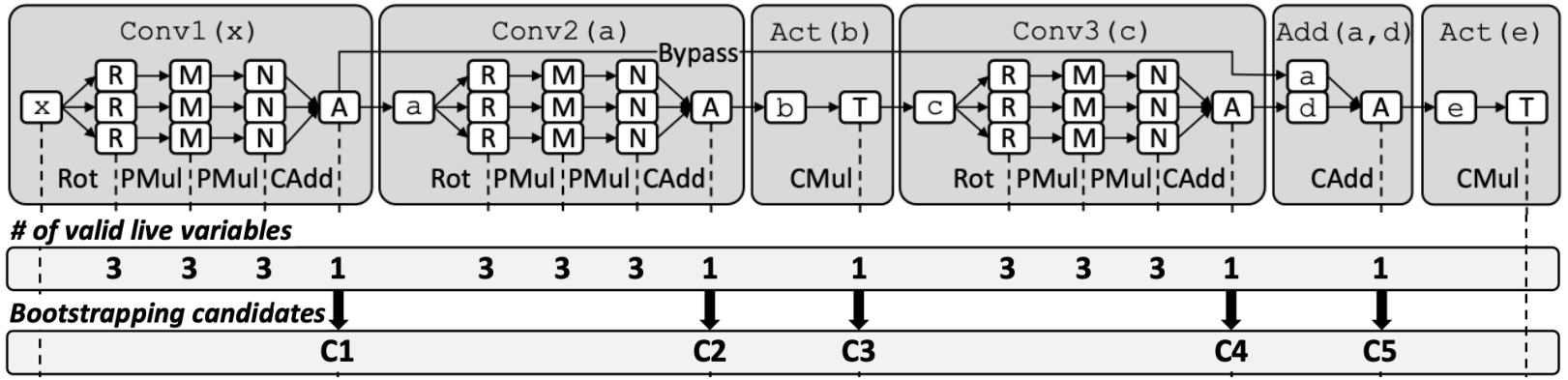


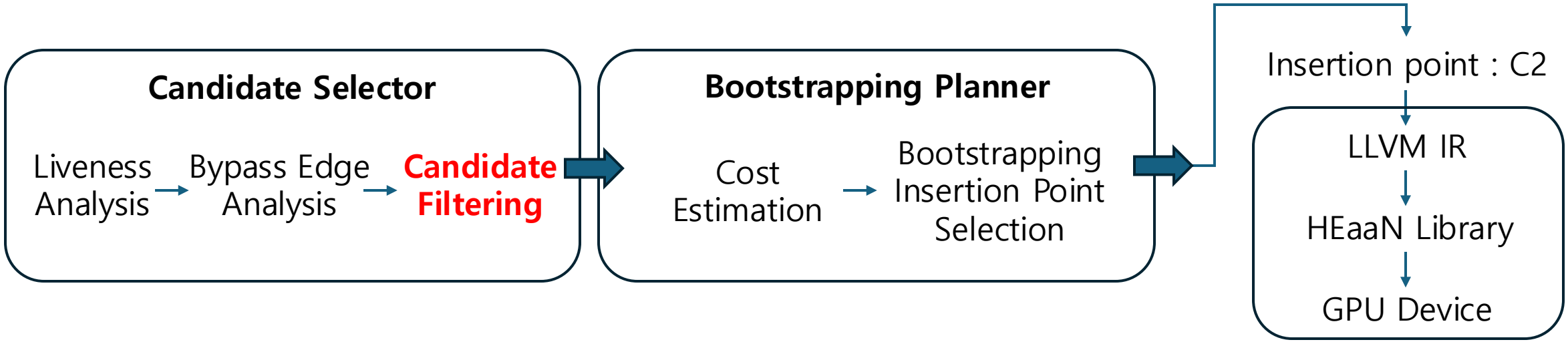


DACAPO - Candidate Selector

❖ 3) 후보 필터링(Candidate Filtering)

- **최종적**으로 부트스트래핑 **후보 지점을 결정** (Bypass된 암호문은 제외하고, 필요한 live-out 암호문만 고려)

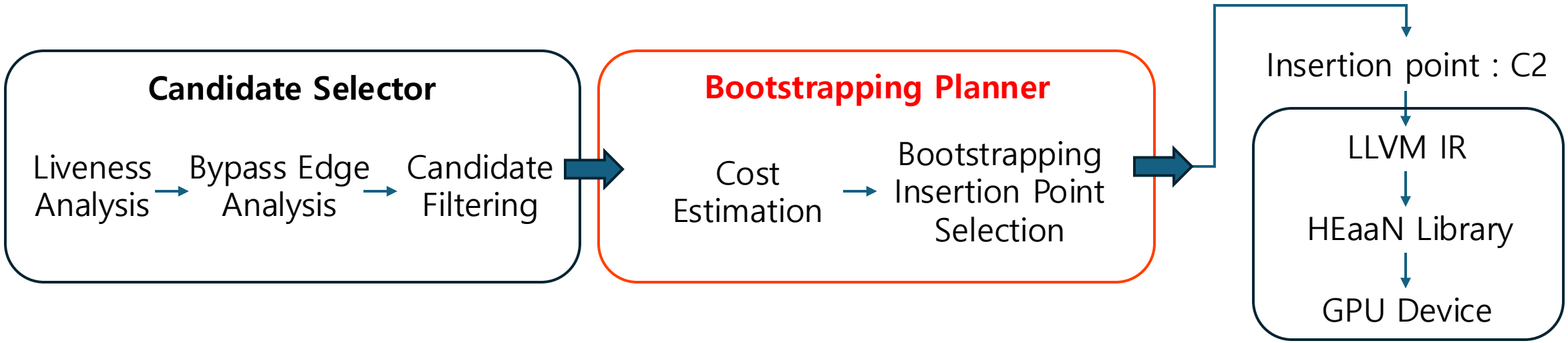




DACAPO - Bootstrapping Planner

❖ Bootstrapping Planner

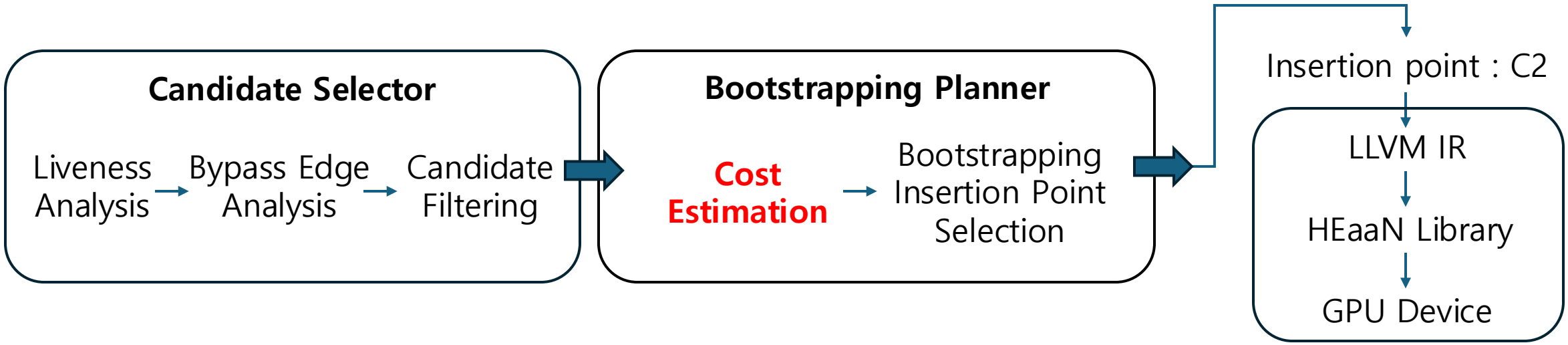
- 목적: RNS-CKKS 연산의 **비용을 계산**하여, 전체 **latency를 최소화**하는 부트스트래핑 배치 계획을 수립
  - 두 단계로 구성
    1) **각 후보 지점의 비용 계산**
    2) **최적의 부트스트래핑 삽입 지점 선택**



DACAPO - Bootstrapping Planner

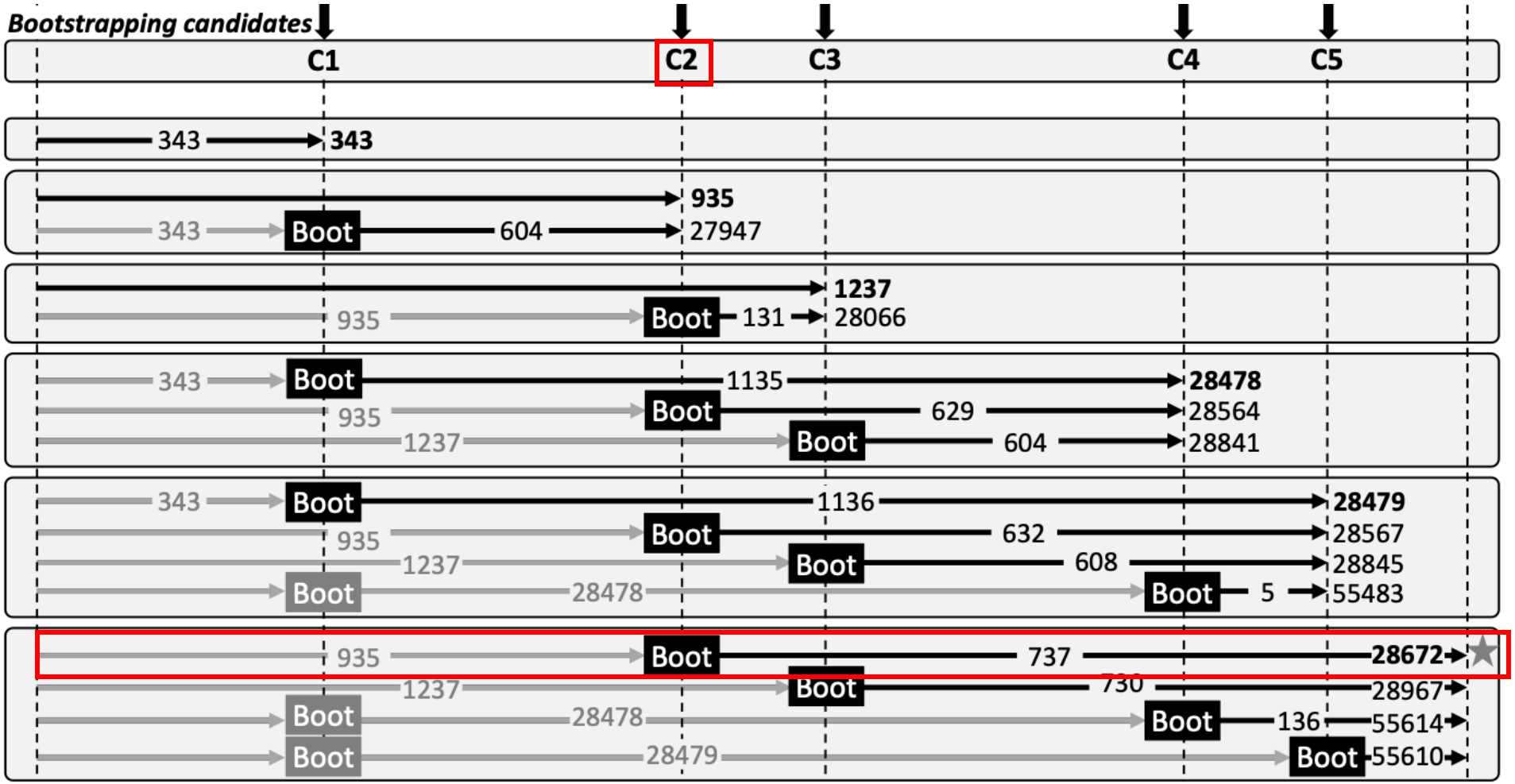
❖ 1) 비용 추정 (Cost Estimation)

- **비용 추정 결과**를 분석하여 각 후보 지점에서 RNS-CKKS 연산의 **예상 latency 계산**



DACAPO - Bootstrapping Planner

❖ 1) 비용 추정 (Cost Estimation)

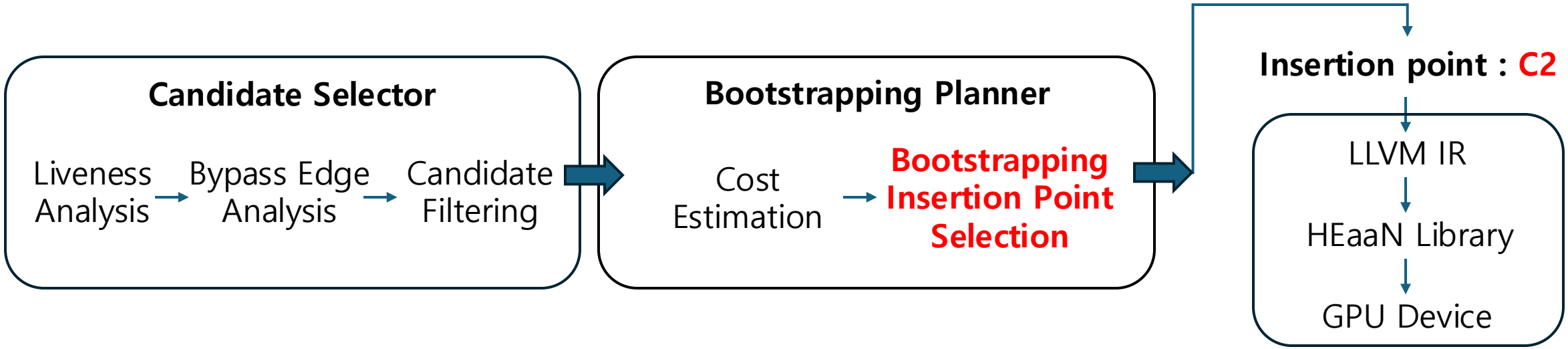


DACAPO - Bootstrapping Planner

❖ 2) 부트스트래핑 삽입 지점 선택 (Bootstrapping Insertion Point Selection)

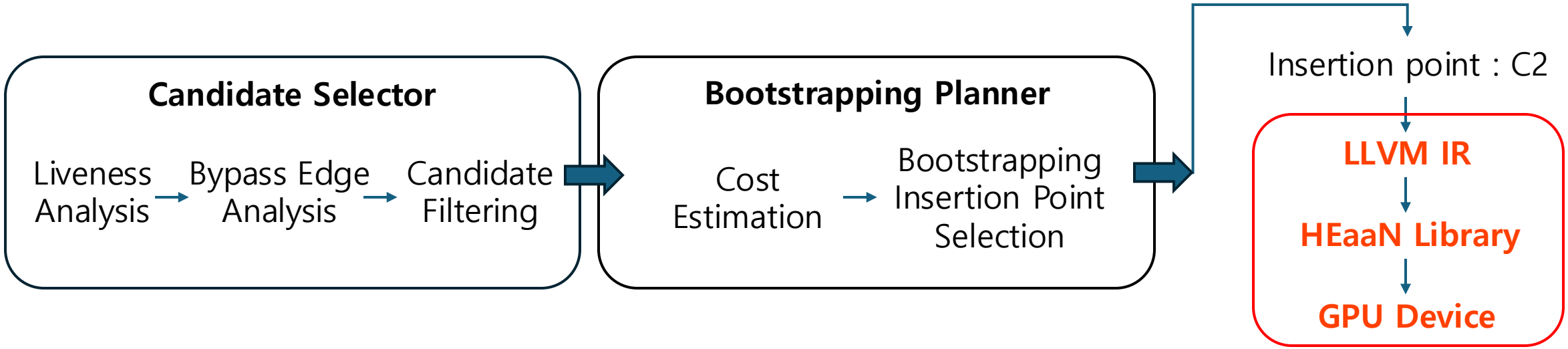
- 계산된 비용 데이터를 바탕으로 **최소 비용**으로 부트스트래핑을 배치할 수 있는 **최적의 지점**을 선택
- 부트스트래핑 지점이 결정되면 최종적으로 부트스트래핑 배치 계획 완성

  ⇒ **전체 연산 비용을 최소화**하면서 **최적의 부트스트래핑 배치** 가능



❖ 이후 과정

- **LLVM IR:** 부트스트래핑을 포함한 스케일 관리 연산을 적용하여 GPU 가속 HEaaN 라이브러리를 호출하는 LLVM IR 코드 생성
- **HEaaN Library**: 생성된 LLVM IR 코드를 HEaaN 라이브러리와 통합하여 암호화된 데이터에 대한 연산 수행
- **GPU Device**: HEaaN 라이브러리를 통해 GPU 디바이스에서 병렬로 RNS-CKKS 연산 실행



DACAPO: 성능 평가 및 결과 분석

Q6. ReLU 왜 하필 {15, 15, 27}차 인지
-> 차수는 상황에 따라 변형 가능한지

Q7. 각 활성화 함수 방법마다 왜 저런 풀링 계층 사용했는지

Q. SiLU ReLU 왜 두개 썼을지

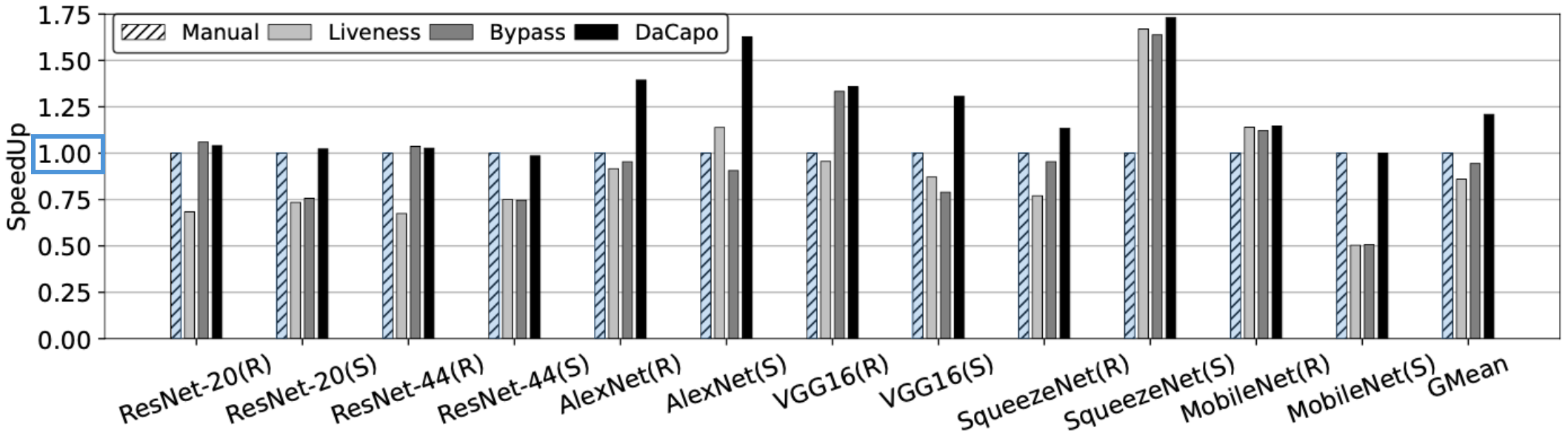
❖ 실험 환경 (Experimental Setup)

- 하드웨어 환경:
  - CPU: Intel Core i7-12700
  - GPU: NVIDIA GeForce RTX 3090 (24GB 메모리)
- 벤치마크 모델: ResNet-20/44, AlexNet, VGG16, SqueezeNet, MobileNet
- 입력데이터: CIFAR-10 데이터셋, 1,000개 이미지 테스트
- 활성화 함수:
  - SiLU: 96차 단일 다항식 근사 – $depth\ 7$
    - Max Pooling 이용
  - ReLU: Minimax 조합 다항식 근사 {15, 15, 27} - $depth\ 13$
    - Average Pooling 이용

DACAPO: 성능 평가 및 결과 분석

❖ 성능 평가 (Performance Evaluation)

- **부트스트래핑 배치 방식에 따른 속도 향상 비교**
  - **수동 배치 방식을 기준점(1.00)**으로 두고 **SpeerUp 정도** 비교
  - 각 딥러닝 모델에서 사용된 활성화 함수는 *ReLU(R)* 및 *SiLU(S)*로 표시됨.

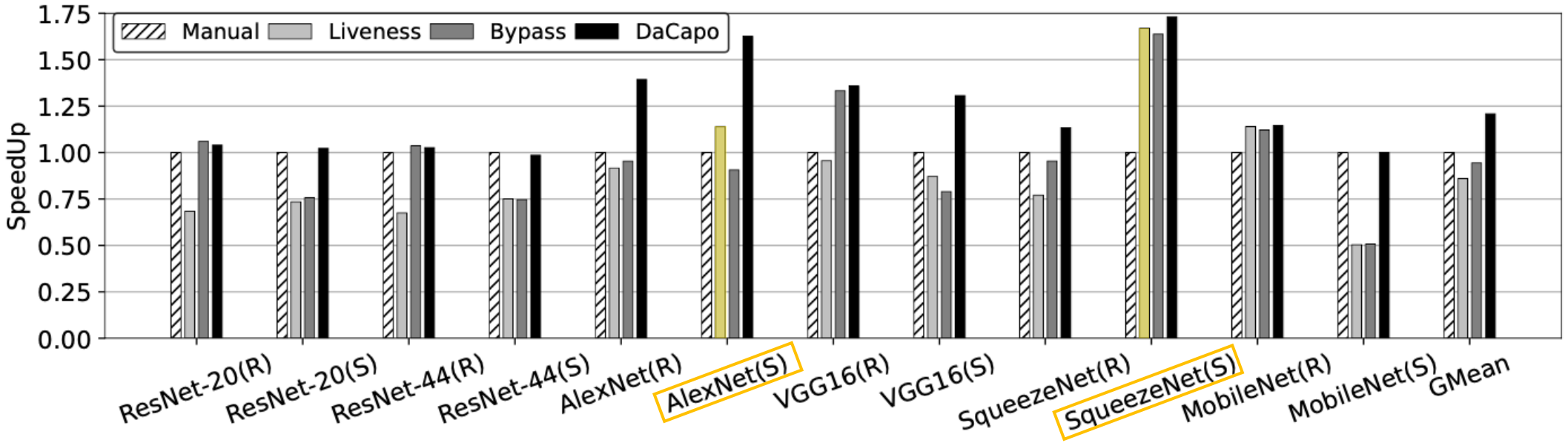


[그림 10] 부트스트래핑 배치 방식에 따른 속도 향상 비교

DACAPO: 성능 평가 및 결과 분석

❖ 성능 평가 (Performance Evaluation)

- **Liveness 방식**
  - AlexNet(S), SqueezeNet(S)에서 성능이 더 향상됨
    - 이유: Liveness 분석을 통해 사용자가 고려하기 어려운 내부 연산에서도 자동으로 부트스트래핑을 수행했기 때문
      - ex) 활성화 함수 내부에서 부트스트래핑 수행 가능



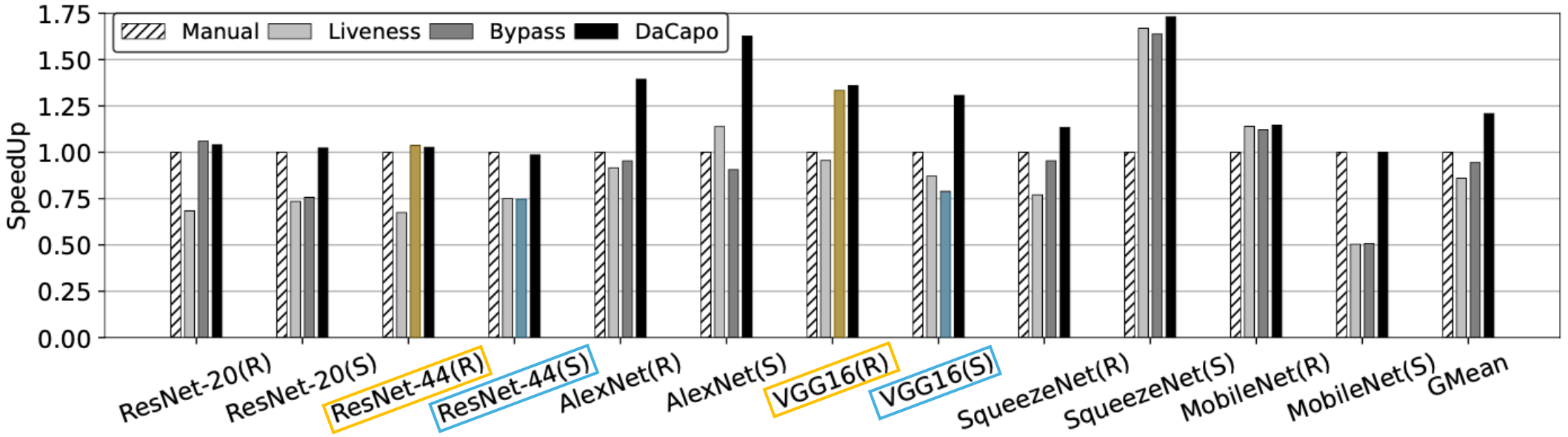
[그림 10] 부트스트래핑 배치 방식에 따른 속도 향상 비교

DACAPO: 성능 평가 및 결과 분석

Q7. SqueezeNet 등 처럼 성능이 나아진 부분 더 있는데 이부분은 논문에서 왜 언급을 안했을지

❖ 성능 평가 (Performance Evaluation)

- **Bypass 방식**
  - ResNet-20/44(R), VGG16(R)에서 성능 개선
    - 이유: 불필요한 부트스트래핑을 줄여 최적의 위치에서 수행했기 때문
    - 그러나 VGG16(S), AlexNet(S) 등에서는 지연 시간이 증가 → 바이패스 엣지를 제외하면서 연산 지연이 증가한 것으로 추정

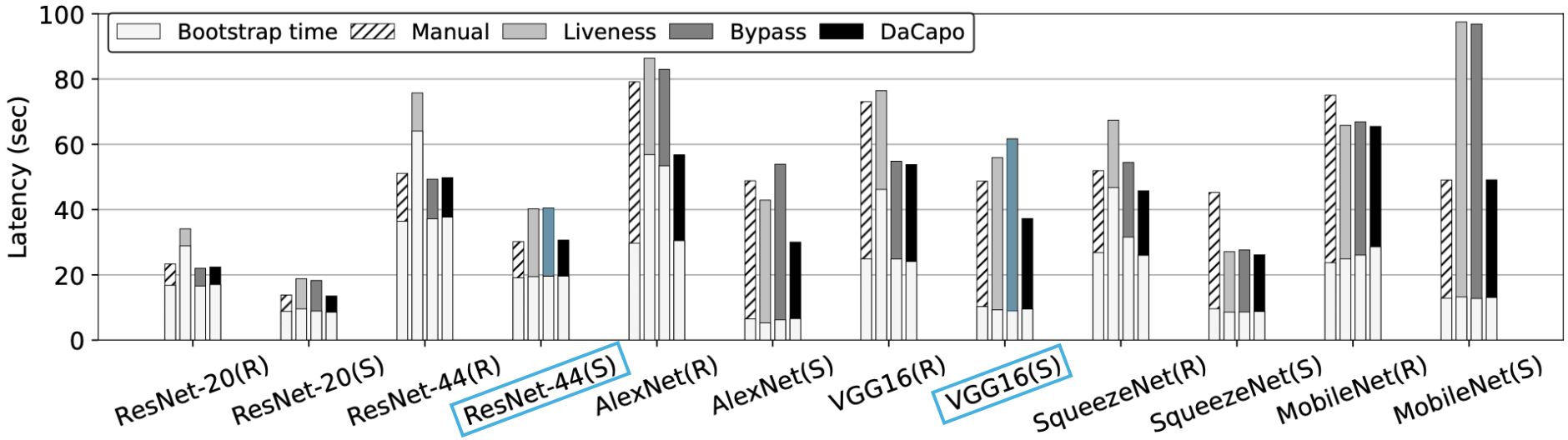


[그림 10] 부트스트래핑 배치 방식에 따른 속도 향상 비교

DACAPO: 성능 평가 및 결과 분석

❖ 성능 평가 (Performance Evaluation)

- **latency 비교**
  - **Bypass** 방식 → **ReLU 기반 모델에서 효과적**이지만, **일부 SiLU 모델에서는 성능 저하** 가능
  - **DACAPO** 방식 → 전체적으로 가장 우수한 성능을 보이며, 수동 배치보다 빠른 연산 속도 달성

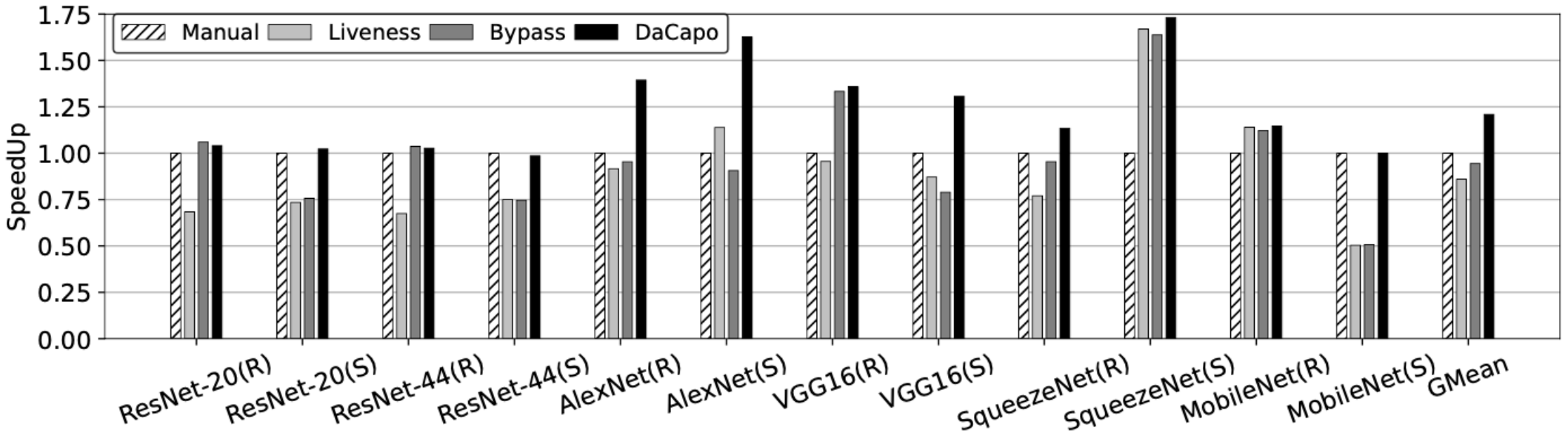


[그림 11] 부트스트래핑 배치 방식에 따른 latency 비교

DACAPO: 성능 평가 및 결과 분석

❖ 성능 평가 (Performance Evaluation)

- **DACAPO 방식**
  - 모든 모델에서 수동 배치 대비 뛰어난 성능
    - 평균적으로 **1.21**배 속도 향상



[그림 11] 부트스트래핑 배치 방식에 따른 속도 향상 비교

DACAPO: 성능 평가 및 결과 분석

❖ 부트스트래핑 횟수 비교 (Bootstrapping Counts)

- 대부분의 모델에서 Manual과 DACAPO의 **부트스트래핑 횟수는 거의 비슷**함
- **DACAPO: 부트스트래핑 자체의 횟수는 거의 동일하지만, 부트스트래핑을 수행하는 위치를 최적화하여 연산 속도를 향상**
  - **연산 비용이 낮은 위치**에서 부트스트래핑을 수행하여 전체 지연 시간을 줄이는 전략을 사용

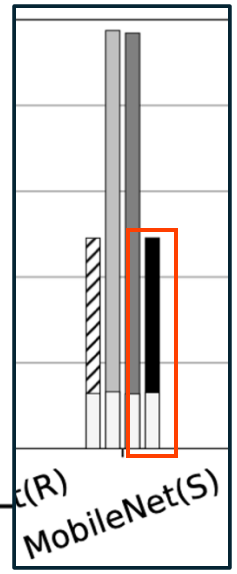
[표 1] 부트스트래핑 횟수 비교: Manual vs. DACAPO

Model	ResNet-20		ResNet-44		AlexNet		VGG16		SqueezeNet		MobileNet	
	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU
Manual	37	19	85	43	66	12	54	20	58	18	53	27
DACAPO	37	19	85	43	67	12	53	20	57	19	61	27

DACAPO: 성능 평가 및 결과 분석

❖ 부트스트래핑 횟수 비교 (Bootstrapping Counts)

- 대부분의 모델에서 Manual과 DACAPO의 **부트스트래핑 횟수는 거의 비슷**함
- **DACAPO: 부트스트래핑 자체의 횟수는 거의 동일하지만, 부트스트래핑을 수행하는 위치를 최적화하여 연산 속도를 향상**
  - **연산 비용이 낮은 위치**에서 부트스트래핑을 수행하여 전체 지연 시간을 줄이는 전략을 사용



⇒ **단순히 부트스트래핑 횟수를 줄이는 것이 아닌, 연산 지연 시간까지 고려해야 최적의 성능을 확보할 수 있음**

[표 1] 부트스트래핑 횟수 비교: Manual vs. DACAPO

Model	ResNet-20		ResNet-44		AlexNet		VGG16		SqueezeNet		MobileNet	
	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU
Manual	37	19	85	43	66	12	54	20	58	18	53	27
DACAPO	37	19	85	43	67	12	53	20	57	19	61	27

DACAPO: 성능 평가 및 결과 분석

Q8. 앞 [표 1]에서 부트스트래핑 횟수는 ReLU, SiLU 별 차이가 없었는데 왜 후보 지점 수는 ReLU가 더 많은지 (후보 중 어떤곳에서 실행하길래 ReLU, SiLU 횟수가 거의 똑같은지)

❖ 컴파일 시간 (Compile Time)

- 대부분 벤치마크에서 부트스트래핑 관리 시간 **5분 미만** → 실질적으로 **컴파일 시간이 허용 가능한 수준**
- **부트스트래핑 관리 시간**이 전체 컴파일 시간의 **상당 부분을 차지** (**약 52%~90%**)
  - 후보 개수가 많아질수록 컴파일 시간이 급증 → **후보 필터링을 보다 정교하게 조정할 필요**가 있음
- **ReLU**를 사용하는 모델에서 후보 지점이 더 많음
  - {15, 15, 27}차 다항식을 조합한 ReLU 근사 이용

[표 2] 컴파일 시간 및 부트스트래핑 관리 시간 비교

Model	ResNet-20		ResNet-44		AlexNet		VGG16		SqueezeNet		MobileNet	
	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU
# Ops	8784	9144	19412	20204	53533	50535	45798	44766	21404	19264	44919	45411
# Candidates	138	100	306	220	662	95	166	71	251	52	191	136
Compile Time (s)	33.0	31.0	116.5	109.2	1163.8	454.0	332.7	290.2	131.2	84.8	302.3	302.1
Bootstrap Mgmt. Time (s)	15.8	14.4	79.4	72.8	1042.3	336.5	230.1	188.1	89.1	44.1	222.8	218.0

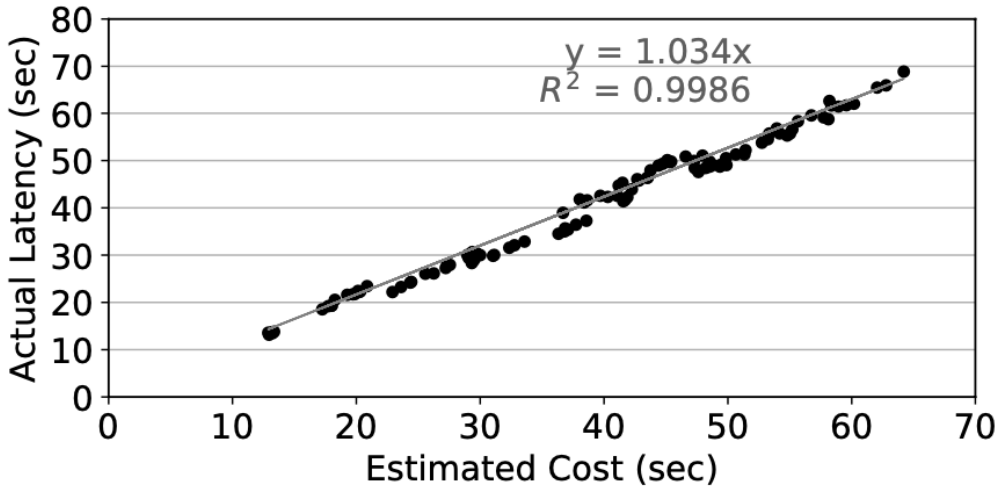
각 모델에서 수행된 전체 연산(operation) 수
부트스트래핑 후보 지점 개수
전체 컴파일 시간(초)
부트스트래핑 관리 시간(초)

DACAPO: 성능 평가 및 결과 분석

❖ 예측 정확도 평가(Performance Estimation)

- 부트스트래핑 배치 계획을 최적화하기 위해 예측된 값과 실제 실행 값 비교
  - $y = 1.034x$ → 예측 값이 실제 값과 3.4% 오차 발생
  - $R^2 = 0.9986$ → 예측 모델의 정확도가 매우 높음

⇒ 부트스트래핑 배치 최적화에 유용한 정확한 성능 추정 가능



[그림 12] DACAPO의 예측 값(X축)과 실제 실행 값(Y축) 비교

결론

❖ DACAPO

- 최초의 자동 부트스트래핑 관리 컴파일러
- **주요 기능:**
  - 생존성 분석(Liveness Analysis): 부트스트래핑 후보 제한
  - 바이패스 엣지(Bypass Edges) 고려: 불필요한 부트스트래핑 최소화
  - 최소 비용 부트스트래핑 계획: 최적 위치 자동 결정
- **성능 평가**
  - **평균 속도 향상: 1.21배 → 딥러닝 모델에서 수동 구현 FHE 프로그램 대비 우수한 성능**

⇒ DACAPO는 자동 부트스트래핑 관리로 FHE 프로그램 성능 최적화

⇒ 불필요한 부트스트래핑 최소화 및 전체 실행 시간 감소

End