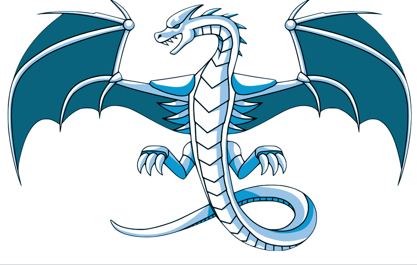
INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD



MemorySSA-Based Reaching Definitions for IR2Vec Flow-Aware Embeddings

Nishant Sachdeva[1], S. VenkataKeerthy[2], and Ramakrishna Upadrasta[2]

1. IIIT Hyderabad · 2. Indian Institute of Technology Hyderabad

BACKGROUND

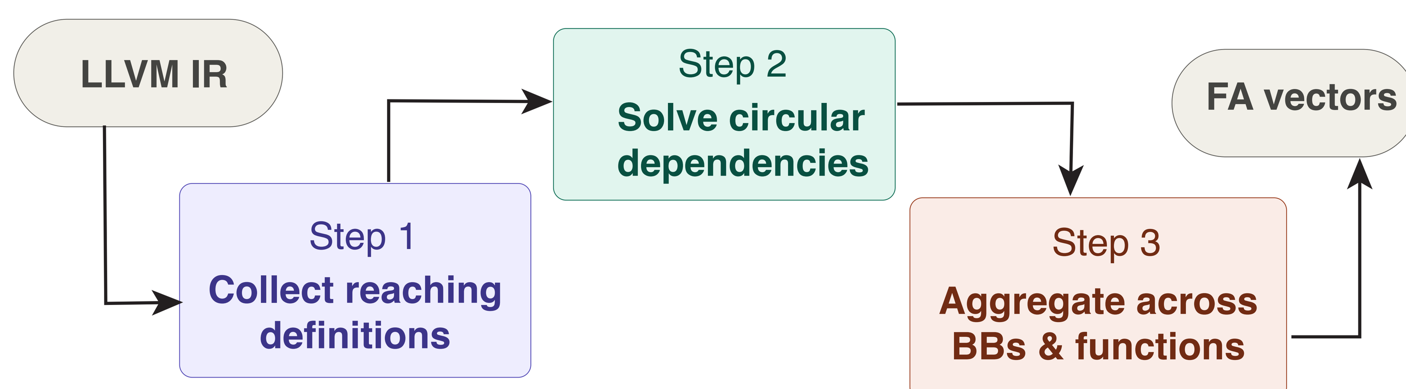
IR2Vec is a scalable, language- and machine-independent program embedding framework built on LLVM IR. It models IR entities as a knowledge graph and learns distributed seed embedings via TransE. IR2Vec has demonstrated its effectiveness on different ML-driven optimizations like phase ordering, loop distribution, and register allocation.

POSET-RL Jain et. al., (2022)	Loop Distribution Jain et. al., (2022)	RL4ReaL VenkataKeerthy et. al., (2022)

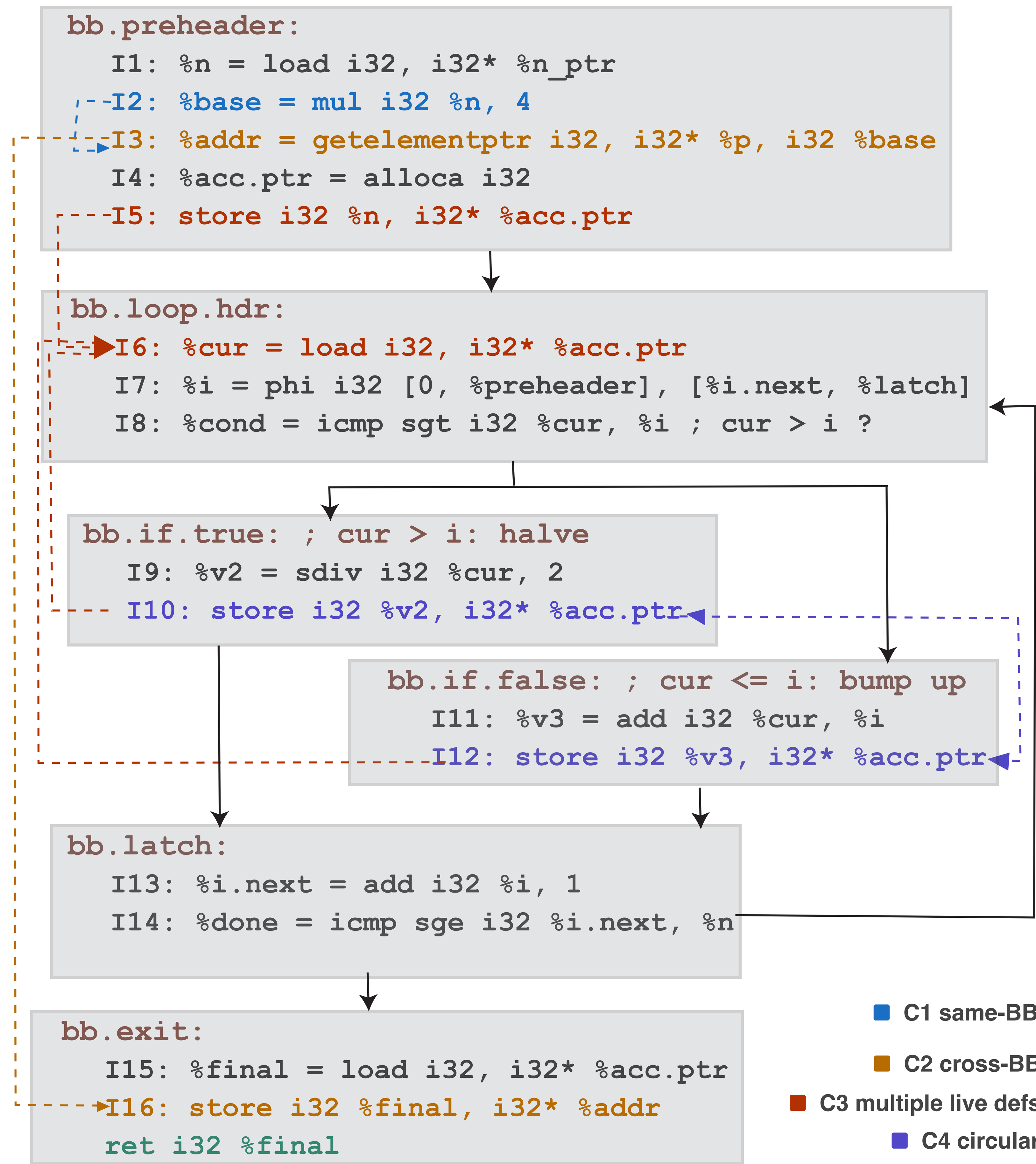
IR2Vec embeddings integrated into MLGO inlining yield up to 4.2% additional code size reduction over -Os.

Ref- https://discourse.llvm.org/t/rfc-enhancing-mlgo-inlining-with-ir2vec-embeddings/86250

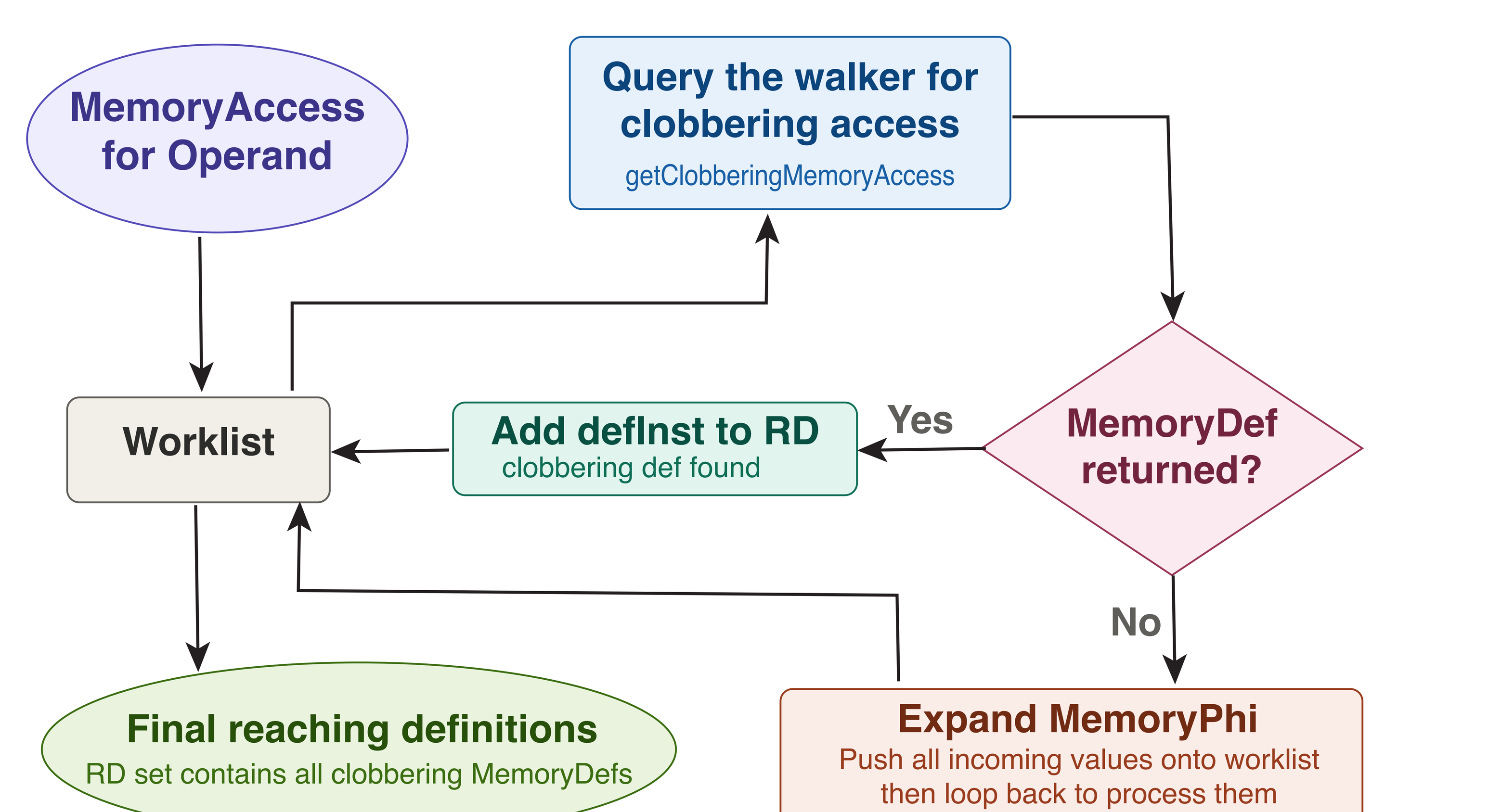
FLOW-AWARE EMBEDDINGS



REACHING DEFINITIONS



MEMORYSSA REACHING DEFINITIONS



CASE STUDY1: POINTER CHAIN

A pointer is initialised, loaded, and used - the original algorithm misclassifies the intermediate load asa reaching defination.

```
; Initialize pointer
store ptr %2, ptr %4, align 8
%8 = mul nsw i32 %7, 2,
; Load pointer value into temp
%9 = load ptr, ptr %4, align 8
; Store through pointer ← query
; target
store i32 %8, ptr %9, align 4
```

Original - incorrect dependency

```
store i32 %8, ptr %9 → {
  %8 = mul nsw i32 %7, 2,
  %9 = load ptr, ptr %4 ← WRONG
}
```

MemorySSA - correct

```
store i32 %8, ptr %9 → {
  %8 = mul nsw i32 %7, 2,
  store ptr %2, ptr %4 ← correct
}
```

MemorySSA skips the load (MemoryUse) and finds the actual store that established the pointer.

CASE STUDY2: LOOP WITH CONDITIONAL UPDATES

A pointer is conditionally reassigned inside a loop. The original algorithm identifies a mismatched store as a false reaching defination for the pointer loads.

```
;Conditional ptr reassignment in loop
%10 = load ptr, ptr %4 ; load query
store ptr %3, ptr %4 ; conditional update
%15 = load ptr, ptr %4 ; load query
store i32 %17, ptr %15 ; unrelated
int store
```

Original - wrong type included

```
%10 = load ptr, ptr %4 → {
  store ptr %2, ptr %4,
  store i32 %17, ptr %15 ← WRONG
}
```

MemorySSA - correct

```
%10 = load ptr, ptr %4 → {
  store ptr %3, ptr %4, ← correct
  store ptr %2, ptr %4, ← correct
}
```

MemorySSA correctly excludes the integer store.

CASE STUDY3: GEP AND ARRAY ELEMENT ACCESS

An array element is accessed via a GEP instruction. The original algorithm stops at the GEP rather than tracing through the actual store that wrote the value.

```
%ptr2 = getelementptr
          inbounds [2 x [3 x i32]],
ptr %arr, i64 0, i64 0, i64 1

store i32 %val1, ptr %ptr2, align 4
%result = load i32, ptr %ptr2,
 align 4 ; query
```

Original - GEP as def

```
load i32, ptr %ptr2 → {
  %ptr2 = getelementptr ← WRONG
}
```

MemorySSA - correct

```
load i32, ptr %ptr2 → {
  store i32 %val1, ptr %ptr2 ← correct
}
```

GEP is not a MemoryDef. MemorySSA correctly returns the store that wrote the value.

CONCLUSION

- **Why MemorySSA fits?**
  The Walker API — `getClobberingMemoryAccess(MA, Loc)` — answers: "which MemoryDef actually clobbers this location?" using LLVM's alias analysis, mapping directly onto IR2Vec's need to find all relevant memory defs that reach the current instruction.
- **Tunable precision.**
  MemoryLocation buffer size is user-configurable for field-level vs whole-object aliasing.
- **Downstream impact.**
  More accurate defs produce higher-fidelity FA embedding and using memorySSA improves stability and robustness and reduces maintenance cost.

ROAD AHEAD

- Extending the MemorySSA API : when a MemoryPhi is encountered, `getClobberingMemoryAccess()` returns a single dominant def from all the clobbering defs, or the phi node itself. This necessitates manual traversal of the predecessors to recover all clobbering definitions. We propose adding a new API that directly exposes the full clobbering def set, with a patch planned for upstream submission to LLVM.
- Usage metrics will be collected to identify downstream tasks where enriched reaching definitions yield the most benefit, and evaluated against existing IR2Vec-based downstream tasks to assess whether more precise data-flow encodings translate to measurable performance gains.

REFERENCES

LLVM - IR2VEC - https://llvm.org/docs/CommandGuide/llvm-ir2vec.html

MemorySSA - https://llvm.org/docs/MemorySSA.html

VenkataKeerthy, S., Aggarwal, R., Jain, S., Desarkar, M. S., Upadrasta, R., & Srikant, Y. N. (2020). IR2VEC. ACM Transactions on Architecture and Code Optimization, 17(4), 1–27. https://doi.org/10.1145/3418463

ARTIFACTS