

Design Document: Adding Register Data Flow to LLVM

Krzysztof Parzyszek

Qualcomm Innovation Center, Inc. is a member of Code Aurora Forum,
hosted by The Linux Foundation

Introduction

- Machine instruction-level optimization sequence:
 - Aggressive optimizations on SSA form.
 - Register allocation sequence (exiting SSA).
 - Post-RA transformations on non-SSA representation.
- Few optimizations happen after register allocation:
 - Many transformations that happen after RA deal with finalizing the code: pseudo-instruction expansion, prolog/epilog insertion, packetization (on Hexagon), etc.
 - Many deal with the control flow or execution flow: branch folding, if conversion, tail duplication, post-RA scheduling, etc.
 - Not many are focused on data flow: machine copy propagation.
- Post-RA data-flow opportunities:
 - Register allocation process can insert register copies (phi node elimination), or replicate instructions (rematerialization).
 - Much less concern for register pressure, (no new spills, but potential interference with post-RA scheduler).

Motivation

- Code quality concerns: redundant register assignments, unnecessary register copies, etc.
 - Optimizing these cases may require cross-block analysis.
- Some data flow optimizations may create further data flow optimization opportunities:
 - Copy propagation may create dead code.
 - Code motion can make the “update” dead in load/store-with-update (e.g. pre- or post-increment).
- Cross-block data flow analysis may be expensive: no general framework to use.
- The goal: provide a framework that will minimize the complexity of data flow optimizations after register allocation.
 - Use a form similar to SSA.

The approach

- The framework itself is transparent:
 - There are no required changes to the program representation.
 - The analysis framework will not modify the program.
- Create a graph that represents the flow of data between registers:
 - Structure of the graph mimics SSA: graph is created for an entire machine function.
 - Graph contains nodes for register definitions and uses, edges connecting related nodes.
 - Additional nodes help represent the structure of the function: blocks and statements, as well as the information required by SSA, such as phi nodes.

Design considerations

- Data flow graph will contain large number of nodes:
 - Several thousand nodes is not uncommon.
 - Large functions can have several hundred thousand nodes.
 - Memory consumption is a concern.
- Reduce memory usage by avoiding pointers:
 - Pointers are usually 64-bit long.
 - Use 32-bit integers to identify nodes, translate between the id's and addresses at run-time.
- Represent all nodes through PODs:
 - Simplifies memory allocation scheme required for efficient translation between addresses and id's.
- Represent all information as nodes:
 - Besides defs and uses, basic blocks, statements, phi nodes are also nodes.
 - All data is allocated using the same mechanism.

Nodes and their structure

- All nodes are derived from NodeBase:
 - The only structure that defines data members: union of many layouts.
 - Derived classes can only provide function definitions to operate on the appropriate members of the union.
- There are two types of nodes: CodeNode and RefNode.
- CodeNode is a container that holds other nodes:
 - Basic block, phi node, statement, as well as the function itself are all represented through CodeNodes.
 - Nodes within a container are connected by a circular linked list.
- RefNode is a node that represents a definition or a use of some entity from the program.
 - Vast majority of RefNodes are defs or uses of registers.
 - Phi nodes contain RefNodes that refer to basic blocks.

SSA property

- Every node has a single reaching def.
 - Given def may not be the only source of data at the point of use:
 - Imprecise and shadow nodes deal with this (described later)
 - Both are a deviation from the pure SSA.
- Join points are expressed through phi nodes.
 - There are no phi nodes in the actual code: they only exist in the graph.

CodeNodes

- Subclasses:
 - FuncNode – corresponds to the machine function.
 - BlockNode – corresponds to a machine basic block.
 - InstrNode – corresponds to an instruction, one of:
 - PhiNode
 - StmtNode – actual machine instruction.
- Each CodeNode contains:
 - Pointer to the corresponding element of the program: MachineFunction, MachineBasicBlock, MachineInstr, MachineOperand.

RefNodes

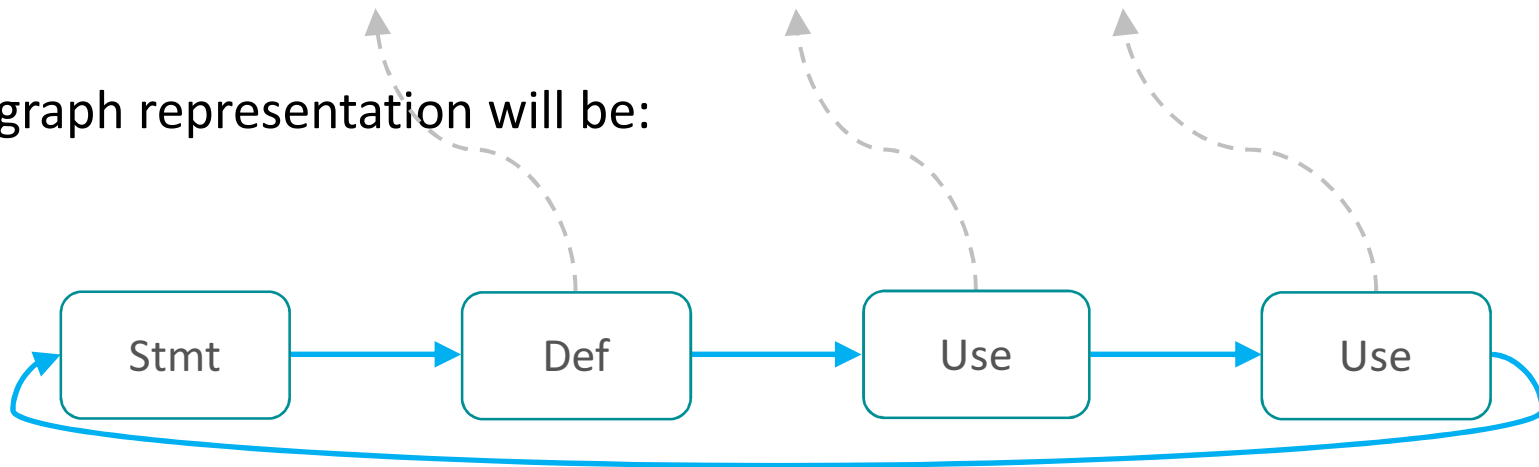
- Each RefNode contains:
 - Id of the reaching def node: it can be 0 if there is no actual reaching definition.
 - Pointer to the MachineOperand to which the node corresponds, except defs and uses in phi nodes.
- Subclasses:
 - DefNode contains:
 - Id of the first reached def
 - Id of the first reached use
 - Id of the sibling def: next def with the same (non-zero) reaching def.
 - UseNode contains:
 - Id of the sibling use: next use with the same (non-zero) reaching def.
- Sibling chains:
 - Separate chains for defs and for uses reached by the same def.
 - Terminated by a node id of 0.

Examples

Consider

$R_0\langle\text{def}\rangle = \text{add } R_1\langle\text{use}\rangle, R_2\langle\text{use}\rangle$

The graph representation will be:

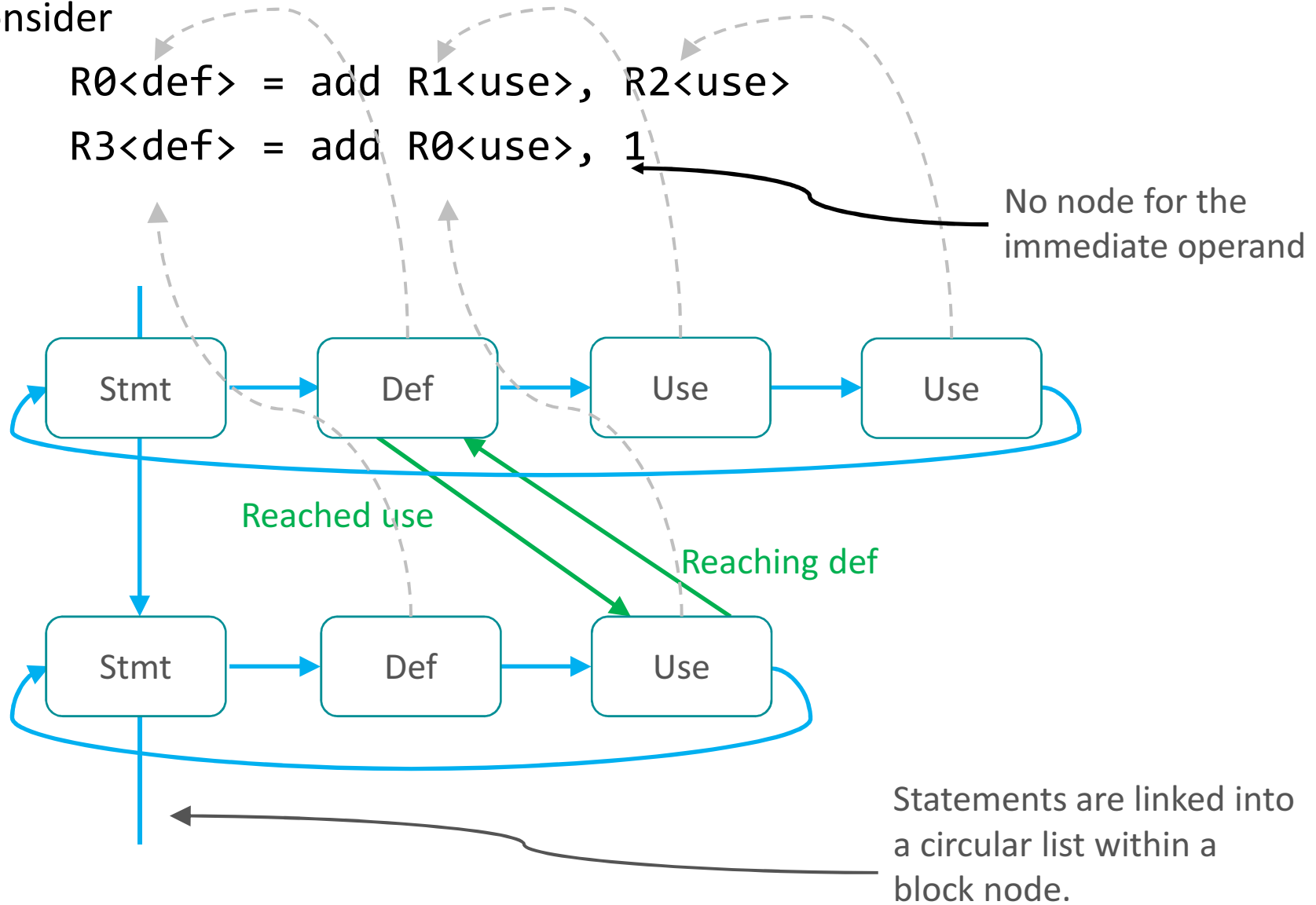


Examples

Consider

$R0\langle\text{def}\rangle = \text{add } R1\langle\text{use}\rangle, R2\langle\text{use}\rangle$

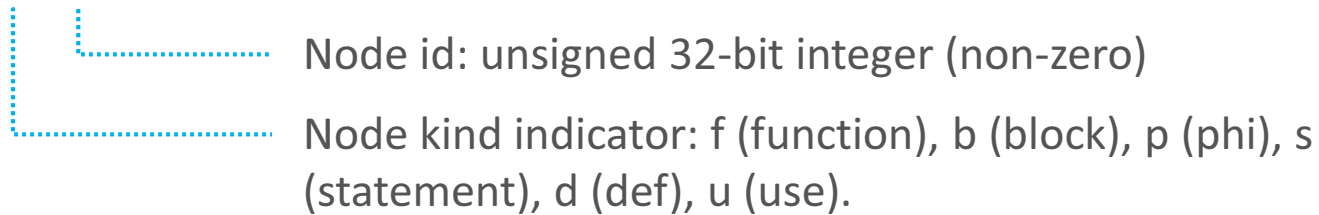
$R3\langle\text{def}\rangle = \text{add } R0\langle\text{use}\rangle, 1$



Graph dumps

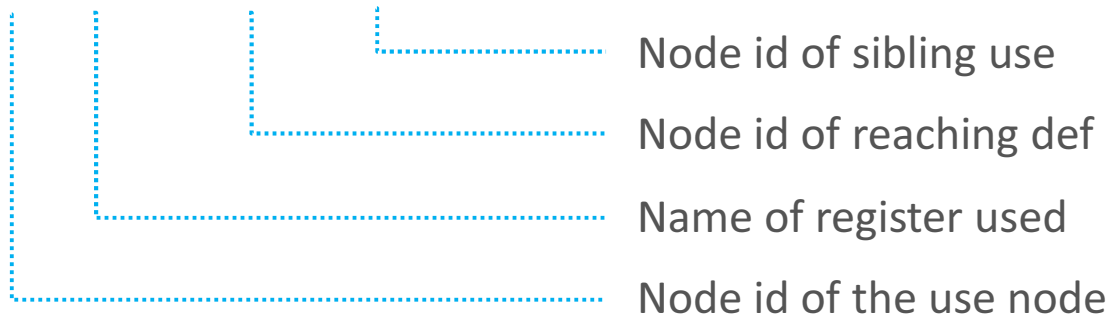
- Node id format:

p321



- UseNode format:

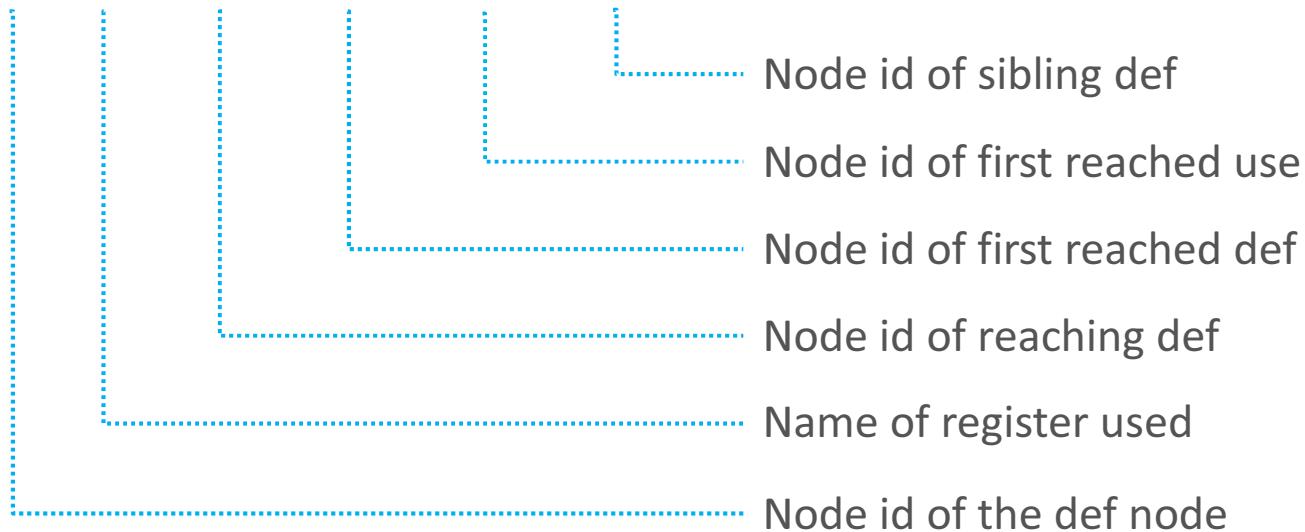
u123<R0>(d59):u87



Graph dumps —continued

- DefNode format:

d141<R2>(d17,d243,u155):d117



- Any node id that is 0 will be omitted:
 - Minimal def node: `d123<R0>(, ,):`
 - Minimal use node: `u123<R0>():`

Shadow ref nodes

Consider the following case

```
R3    = #0           ; load 0 into R3
R2    = add R4, #1   ; add 1 to R4, store result in R2
R5:4 = asl R3:2, #2 ; shift pair R3:R2 left by 2 bits
```

- What is the reaching def of R3:2 in the last instruction?
- Slightly different scenario from the case for imprecise nodes:
 - The defs of R3 and R2 are not related: R2 and R3 do not overlap.
 - There is only room in a node for one reaching def.
 - If R2 is the reaching def, then the def of R3 may appear dead (and the other way around): the def-use relationship will be missing from the graph.
 - Create an extra “shadow” use node to hold the additional reaching def:
 - Defs can also be “shadows”.
 - There can be as many shadows as there are needed.

Shadow ref nodes —continued

- The “shadowed” node is also considered a “shadow”:
 - The two uses for R3:2 (one for R2 and one for R3) will both be marked as “shadows”.
 - In the original design, the “first” node was not a shadow, only the “extra” ones.
 - In data-flow analysis it may be necessary to know if a node has a shadow: marking all related nodes as shadows makes the check faster.

Shadow ref nodes: example

Consider the following case

```
R3    = #0           ; load 0 into R3
R2    = add R4, #1   ; add 1 to R4, store result in R2
R5:4  = asl R3:2, #2 ; shift pair R3:R2 left by 2 bits
```

- Graph representation:
 - Shadow nodes marked with “.

```
d2<R3>(…,…, u7):
d4<R2>(…,…, u8):, u5<R4>(…):
d6<R5:4>(…,…,…) :, u7”<R3:2>(d2):, u8”<R3:2>(d4):
```


Phi nodes

- Phi nodes do not exist in the actual code.
 - Defs and uses in phi nodes cannot point to actual operands.
 - The register shown in dumps comes from MachineOperand.
 - Store the actual register in the node.
- Phi nodes may form cycles.
 - Some cycles in the graph may be unnecessary.
 - Phi nodes are created speculatively.
 - Trivially unused ones are removed in the build process.