IBM Research, Tokyo Research Laboratory

Fast Predicate-Aware Dataflow Analysis

Kazuaki Ishizaki (kiszk@acm.org)
Akira Koseki
Hideaki Komatsu
Toshio Nakatani
IBM Research, Tokyo Research Laboratory

Background and goal of this research

- Background
  - Predicated intermediate code is efficient for optimizing the programs for the underlying EPIC architecture.
  - Previously-known predicate-aware dataflow analysis algorithms are too slow to be used for dynamic compilers.
- Goal
  - Develop a fast algorithm for predicate-aware dataflow analysis.

Detailed algorithm is written in our paper on web site.

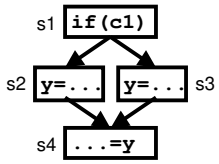
Predicate-aware dataflow analysis

- Example: Live analysis for variable y in a hyperblock *i*.
  - Definitions of y kill the value at S1.
  - Predicate-aware dataflow analysis must tell that y is not live at S1 since it knows the relationship of p0, p1, and p2.

Original program

```
if (c1) {
  y = ...;
} else {
  y = ...;
}
... = y;
```

Control flow graph

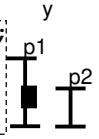


Predicated code

hyperblock *i*

```
S1: (p0) cmp.unc  p1,p2 = (c1);
S2: (p1) y = ...
S3: (p2) y = ...;
S4: (p0) ... = y;
```

Live range y



A predicate p0 is always '**true**'.

The cmp.unc sets a predicate p1 as '**true**' and a predicate p2 as '**false**' complementarily if condition variable c1 and guarding predicate p0 are '**true**'.

A table to represent which statements are executed depending on C1

		S1	S2	S3	S4
C1	T	Y	Y	N	Y
	F	Y	N	Y	Y

Previous algorithm [JohnsonMICRO96]

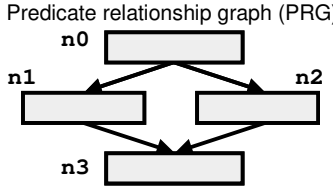
- It consists of the following steps:
  - Create a *partition graph* to represent the relationship of the predicates of the target instructions.
  - Calculate liveness of each variable in hyperblock *i*, by performing many queries to the partition graph.
- Each query needs a traversal of the partition graph.
  - A graph traversal is expensive, in comparison to the bitwise operation used in conventional dataflow analysis.
  - Performance has been improved by caching the results of the query.

Our algorithm

- Avoid any expensive graph traversal by
  - Pre-computing *reachable* and *dominance sets* for each group of contiguous instructions under the same predicate.
  - Calculating liveness of each variable in hyperblock *i* only using set operations on the pre-computed sets.

```
S1: (p0) cmp.unc  p1,p2 = (c1);    // n0
S2: (p1) y = ...                   // n1
S3: (p2) y = ...;                  // n2
S4: (p0) ... = y;                  // n3
```

Predicate relationship graph (PRG)



LIVE(y) can be represented by bitvectors.

UR(n3) = {n0, n1, n2, n3}

PD(n0) = {n0}, PD(n1) = {n1}, PD(n2) = {n2}

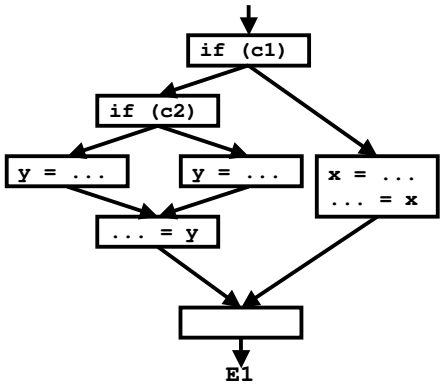
LIVE(y) represents a set of nodes where y is available

LIVE(y) at S4 = UR(n3) = {n0, n1, n2, n3}
LIVE(y) at S3 = {n0, n1, n2, n3} – PD(n2) = {n0, n1, n3}
LIVE(y) at S2 = {n0, n1, n3} - PD(n1) = {n0, n3}
LIVE(y) at S1 = {n0, n3} - PD(n0) = {n3} ⇒ y is not live at S1 since n0 is not included.

Sample program and control flow graph

- Sequential code

```
if (c1) {
  if (c2) {
    y = ...;
  } else {
    y = ...;
  }
  ... = y;
} else {
  x = ...;
  ... = x;
}
goto E1;
```

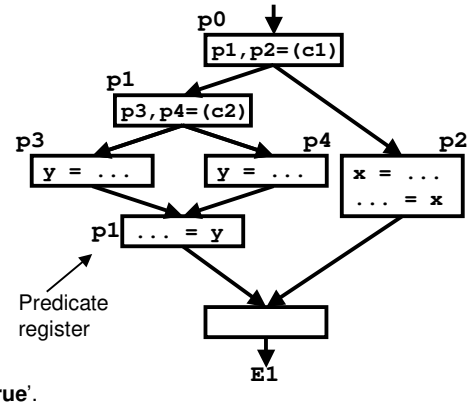


Sample program and control flow graph

- Predicated code

hyperblock *i*

```
(p0) cmp.unc  p1,p2 = (c1);
(p1) cmp.unc  p3,p4 = (c2);
(p2) x = ...
(p3) y = ...
(p4) y = ...;
(p2) ... = x
(p1) ... = y;
(p0) goto E1;
```

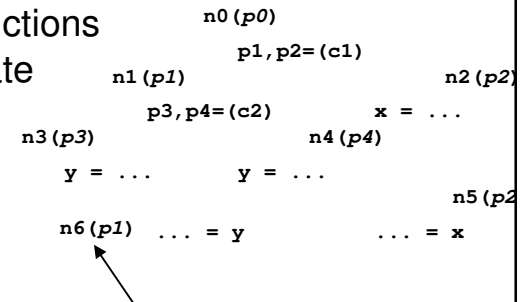


p0 is always '**true**'.

The first cmp.unc instruction sets a predicate p1 as '**true**' and a predicate p2 as '**false**' complementarily if condition variable c1 and guarding predicate p0 are '**true**'.

Define a Predicate Relationship Graph (PRG)

- Node: contiguous instructions under the same predicate register.
- Arc: Execution dependence between two nodes.




```
(P0) cmp.unc  p1,p2 = (c1);// n0
(P1) cmp.unc  p3,p4 = (c2);// n1
(p2) x = ...               // n2
(p3) y = ...               // n3
(p4) y = ...;              // n4
(p2) ... = x               // n5
(p1) ... = y;              // n6
(p0) goto E1;              // n7
```

Node number (Predicate register)

n7(p0)

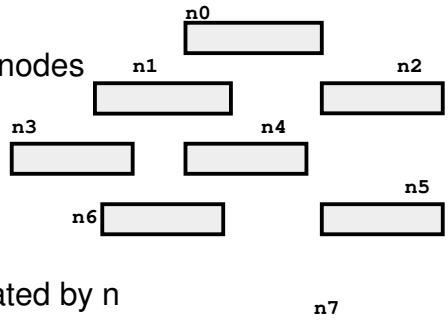
Virtually defined, but not actually used

Compute UR(n) and PD(n, e)

- UR(n)
  - A set of upward reachable nodes
  - Each node is executed if n is executed.

    UR(n6) = {n0,n1,n3,n4,n6}
- PD(n, e)
  - A set of nodes post-dominated by n
  - Every path from each node to e includes n.

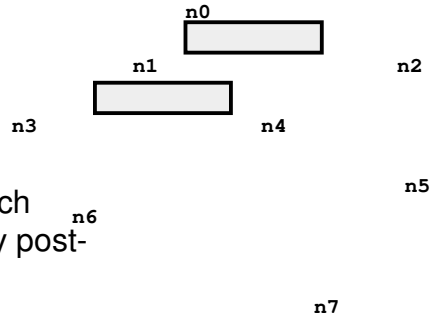
    PD(n3, n7) = {n3}
    PD(n5, n7) = {n2, n5}
    PD(n1, n7) = {n1}



Compute PDF(n)

- PDF(n)
  - The dominance frontier for the reverse graph
  - Node n post-dominates an immediate successor of each node, but n does not strictly post-dominate the node.

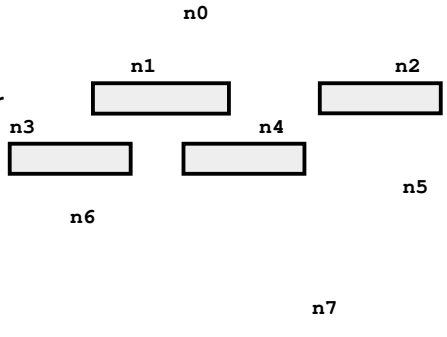
    PDF(n6) = {n0}
    PDF(n3) = {n0, n1}



Compute GIPD(n)

- GIPD(n)
  - The generalized immediate dominator set [Gupta92] for the reverse graph
  - Node n immediately dominates each node.

  GIPD(n0) = {n1, n2}
  GIPD(n1) = {n3, n4}



Local analysis of variable y using precomputed sets

- Initialize LIVE(y)
  - LIVE(y) = Ø at the end of hyperblock *i*
- Traverse instructions backward from the bottom
  - Compute LIVE(y) based on the references and definition of y
- Compute GEN(i) for y based on LIVE(y) at the top of hyperblock *i*

```
(P0) cmp.unc  p1,p2 = (c1);// n0
(P1) cmp.unc  p3,p4 = (c2);// n1
(p2) x = ...               // n2
(p3) y = ...               // n3
(p4) y = ...;              // n4
(p2) ... = x               // n5
(p1) ... = y;              // n6
(p0) goto E1;              // n7
```

GEN(i) is a set of live variables at the top of *i* within *i*

In this presentation, GEN(i) stands for GEN(i, n7).

Compute GEN(i) for variable y (y is live ?)

- GEN(i) for definition
  - LIVE(y) at n0
    = UR(n6)-PD(n4)-PD(n3)-PD(n1)-PD(n0)
    = {n0,n1,n3,n4,n6}-{n4}-{n3}-{n1}-{n0}
    = {n6}

y is not live.

```
(P0) cmp.unc  p1,p2 = (c1);// n0
(P1) cmp.unc  p3,p4 = (c2);// n1
.
(p3) ...                   // n3
(p4) y=...;                // n4
.
.
(p1) ...y;                 // n6
.
```

Precomputed sets:

UR(n6)={n0,n1,n3,n4,n6}
PD(n0)={n0}, PD(n1)={n1}, PD(n3)={n3}, PD(n4)={n4}
PDF(n3)={n0,n1}, PDF(n4)={n0,n1}, PDF(n6)={n0}
GPID(n0)={n1,n2}, GPID(n1)={n3,n4}

In this presentation,
PD(n) stands for PD(n, n7).

Global analysis

- Solve the dataflow equations using the results of local analysis
  - We use the following flow equations for a hyperblock.

Changes for a hyperblock

$$IN(i) = \bigcup_{e \in ExitPath(i)} ((OUT(i,e) - KILL(i,e)) \cup GEN(i,e))$$

$$OUT(i,e) = \bigcup_{j \in Succ(i,e)} IN(j)$$

ExitPath(i): A set of exit paths from hyperblock *i* to other hyperblocks.
Succ(i, e): A set of successors of hyperblock *i* at exit node *e*.

Evaluation of our algorithm

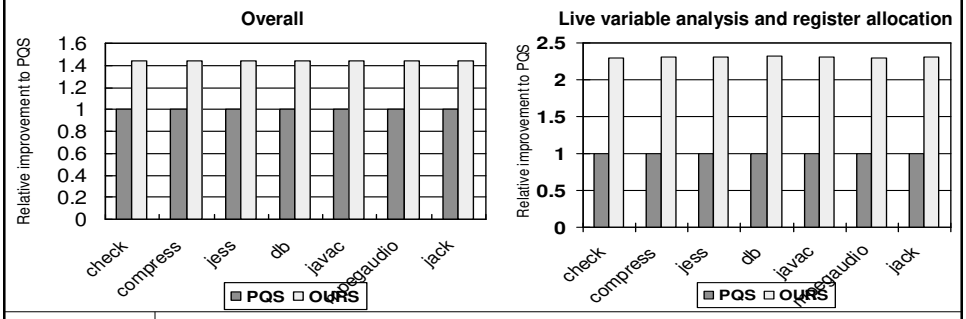
- Measure the compilation speed of live variable analysis and register allocation that uses predicate-aware dataflow analysis.
  - OURS: our algorithm
  - PQS: Johnson's algorithm [JohnsonMICRO96]

    Port this PQS code from Trimaran 2.0 implemented by Johnson.
- Environments
  - IBM Developers Kit, Java Technology Edition, 1.4.0
  - 4-Way 900MHz Itanium2 with 1GB memory, Windows 2003 Server Datacenter edition
- Benchmark program
  - 7 programs in SPECjvm98

Experimental results

- OURS improves the compilation time over PQS:
  - by 1.4 times for overall
  - by 2.3 times for live variable analysis and register allocation

Improvement of compilation time by OURS over PQS (higher bars are better)



Conclusion

- We developed a faster algorithm for predicate-aware dataflow analysis.
  - Pre-compute UR(n), PD(n, e), PDF(n), and GIPD(n).
  - Compute GEN(i, e) and KILL(i, e) without any query with graph traversal.
- Live variable analysis and register allocation using our algorithm are 2.3 times faster than that using PQS.
- Overall compilation time using our algorithm is 1.4 times faster than that using PQS.

Thank you

Any questions ?