Parallelism for Free:

Efficient and Optimal Bitvector Analyses for Parallel Programs

Jens Knoop, Bernhard Steffen, and Jürgen Vollmer

Presented by Aaron Kimball, October 20, 2005, COM S 711

Motivation

- Finite lattice analysis is the basis of lots of compiler optimizations
- Programs with parallel components need optimization too
- Few analyses available for such parallel structures

Existing techniques / research

- McDowell ’89: Developed heuristics to ignore some interleavings
- Grunwald and Srinivasan ’93: Mandate that interleaved components have no data dependencies (any interleavings are ok)
- Long and Clarke ’91: Analyzed Ada programs with parallel tasks, but no shared variables

Goals of the paper

- Provide analysis for programs with sections of parallel or interleaved execution
- Optimally cover “interference”
- Efficiency on par with sequential algorithms
- Be easy to implement

Bitvector algorithms

- Use bitvectors as the only analysis mechanism
- Lattice with only elements {tt, ff}
- Still powerful! Bitvectors can characterize liveness, “very busy”-ness, reaching definitions, def-use, availability, code motion, cse, dce, strength reduction, etc

How they meet their goals (1)

(1) Optimally cover “interference”

- Claim: PMOP (Parallel Meet Over Paths) = $PMFP_{bv}$ (Parallel Maximal Fixed Point for bitvectors)
- The key observation: “Though the various executions of parallel components are semantically different, they need not be considered during bitvector analysis.”

How they meet their goals (2)

(2) Be as efficient as
sequential counterparts

- These are standard bitvector algorithms; no special techniques or global state space are actually used. Therefore, the efficiency is the same as sequential techniques.

How they meet their goals (3)

(3) Be easy to implement

- The ease of implementation comes from the fact that the bitvector analysis is the same as on any sequential program, after a single fixed-point “preprocessing” step.

Sequential flow graphs

- Sequential flow graphs: G = (N, E, s, e)

N – set of nodes (statements)

E – set of directed edges (n, m) $\in$ E implies nondeterministic flow from node n to m.

s, e are the “start” and “end” nodes (always a “skip”)

All nodes n $\in$ N are on at least one path from s to e.

Parallel flow graphs

- Parallel flow graph G* = (N*, E*, s*, e*)

Identical to normal flow graphs, except for treatment of `par{`s_1`}{`s_2`}` structure:

`par` structure begins with *ParBegin* node, and ends with *ParEnd* (both 'skip' nodes).

ParBegin has edges to the two separate flow graphs for each of the two sides of the parallel structure. These separate subgraphs then each flow toward the same *ParEnd* structure.

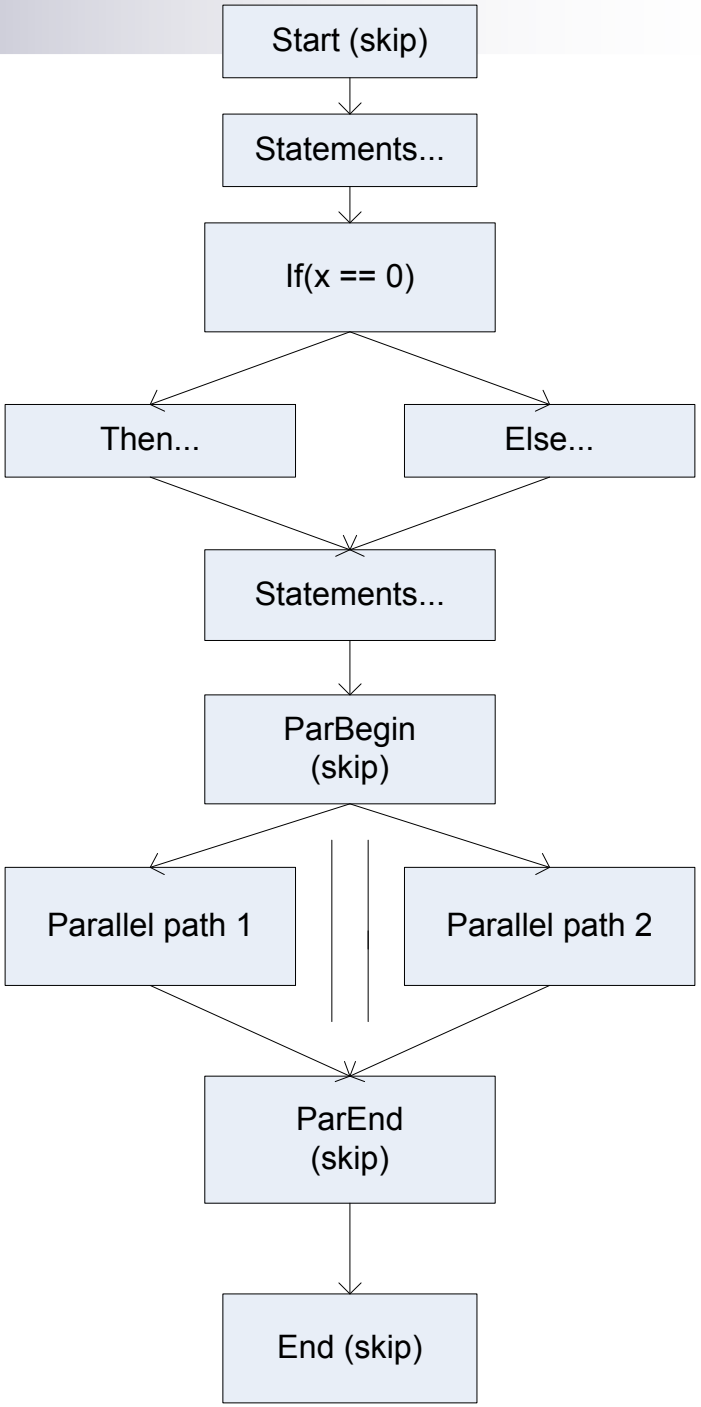
`par` structures

`par{`s_1`}{`s_2`}` denotes two sets of statements to execute simultaneously, with any interleaving between the elements of s_1 and s_2 acceptable

- No synchronization inside a `par` structure
- No jumps into or out of a `par` structure
- Both sides must reach ParEnd for program flow to continue past the `par` structure (implicit synchronization)

A parallel flow graph example

A flow graph of statements (basic blocks); Parallel blocks are denoted with the || symbol between them, and share common ParBegin and ParEnd elements.



Definitions

$pred_{G^*}(n) =_{df}$ all nodes { m | (m, n) $\in$ E*}

$succ_{G^*}(n) =_{df}$ all nodes { m | (n ,m) $\in$ E*}

$G_P(G^*)$ = all subgraphs of G* representing `par` statements

$G_C(G')$ for G' $\in$ $G_P(G^*)$ are all component graphs of a particular subgraph

$G_C(G^*)$ is shorthand for the set of all such component graphs (single-entry/exit regions of G*)

Definitions, continued

- rank(G) $=_{df}$ The nesting level of a particular `par` statement, with 0 as inner-most, then 1, 2, etc.
- pfg(n): maps a node n to the smallest enclosing flow graph in $G_P(G^*)$, or to G* if no such flow graph exists.
- cfg(n): maps a node n to the smallest enclosing flow graph in $G_C(G^*)$, or to G* if no such flow graph exists.

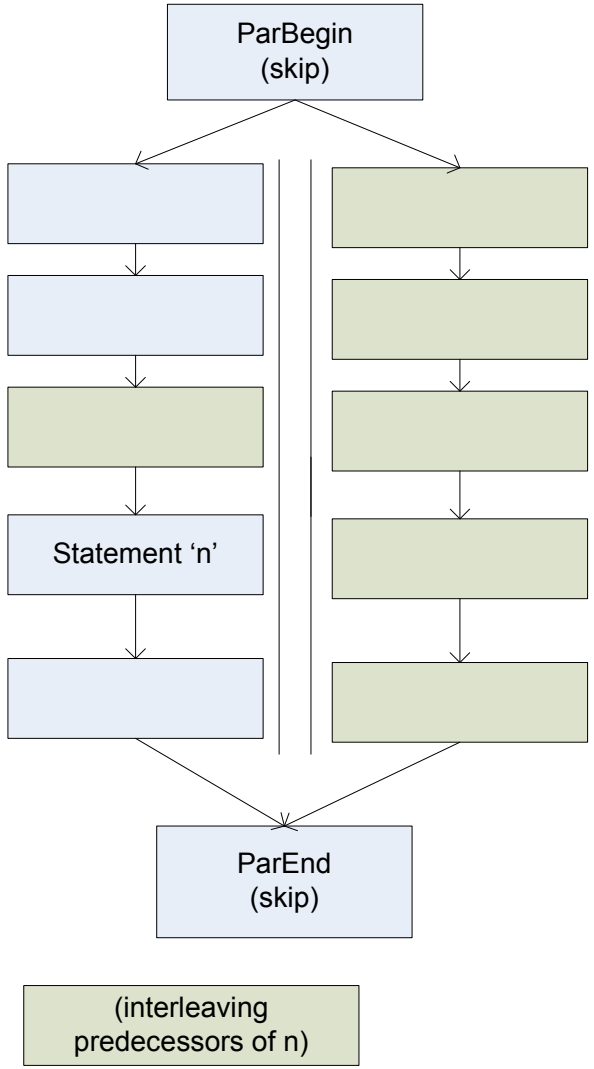
Interleaving predecessors

- For a sequential flow graph, $pred_G(n)$ gives the predecessors of a node n

- For possibly interleaved program segments, $ItlvgPred_{G^*}(n)$ gives all possible predecessor nodes; essentially, $pred_{G^*}(n)$ U { all nodes in all complementary subgraphs of the `par` statement }

Interleaving predecessors example

All nodes in the component subgraph complementary to cfg(n) are interleaving predecessors of n, as well as all static predecessors of n.



Data flow analysis

- Use Kam/Ullman definitions for data flow analyses:
- [[n]] : N* → (C → C), where C is a lattice

- Bitvectors: [[n]] : N* → (B → B), where B is the lattice ({ff, tt}, ⊠, ❍}

Bitvectors

- Meet operator is AND (OR in reverse lattice {tt, ff})
- ff ❍ tt
- F_B: monotonic functions (B → B)
  - Elements are {$Const_{tt}$, $Const_{ff}$, Id_B}
  - F_B itself is a complete lattice
  - All functions of F_B are distributive

Restrictions on bitvectors

- Functions on $B^k \rightarrow B$ are not distributive for $k \geq 2$
- OK because assignment motion, partial dce, liveness, availability, etc can all be computed with only functions in F_B.

Naïve solution to bitvector analysis

- Any given set of two parallel subgraphs can be decomposed into a single sequential subgraph by using if statements to cover all possible interleaving program flows…
- But this solution introduces exponential number of statements; effectively incomputable.

Solution: Main Lemma

LEMMA 3.3.3. (MAIN LEMMA). *Let* $f_i : \mathcal{F}_\mathcal{B} \to \mathcal{F}_\mathcal{B}$, $1 \leq i \leq q$, $q \in \mathbb{N}$, *be functions from* $\mathcal{F}_\mathcal{B}$ *to* $\mathcal{F}_\mathcal{B}$. *Then we have:*

$$\exists\, k \in \{1, \ldots, q\}.\ f_q \circ \ldots \circ f_2 \circ f_1 = f_k \ \wedge\ \forall\, j \in \{k+1, \ldots, q\}.\ f_j = Id_\mathcal{B}.$$

Given a sequence of *q* transformations on the lattice, there exists some transformation f_k that has the "final" and therefore only effect on the lattice, and al subsequent "transformations" are Id_B.

Furthermore: for $m \in ltlvgPred_{G^*}(n)$, ⌒ a parallel path to leading to n with m as the last step before n.

Significance: Only one statement among the interleaving predecessors of a given statement can actually affect the value of the lattice property as seen by that statement!

Interference

- Only possible interference between parallel flow graphs is *destruction*.
- NonDestructible(n): no interleaving predecessors of *n* are $Const_{ff}$.
- The MFP solution essentially resolves to: is a particular property *NonDestructable* over a `par` flow graph?

The Algorithm:

(1) Terminate, if G does not contain any parallel statement. Otherwise, select successively all maximal flow graphs G' occurring in a graph of $\mathcal{G}_{\mathcal{P}}(G)$ that do not contain any parallel statement, and determine the effect $[\![G']\!]$ of this (purely sequential) graph according to the equational system of Definition 2.2.3.1.

(2) Compute the effect $[\![\bar{G}]\!]^*$ of the innermost parallel statements $\bar{G}$ of G by

$$[\![\bar{G}]\!]^* = \begin{cases} Const_{ff} & \text{if } \exists\, G' \in \mathcal{G}_{\mathcal{C}}(\bar{G}).\ [\![end(G')]\!] = Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall\, G' \in \mathcal{G}_{\mathcal{C}}(\bar{G}).\ [\![end(G')]\!] = Id_{\mathcal{B}} \\ Const_{tt} & \text{otherwise.} \end{cases}$$

(3) Transform G by replacing all innermost parallel statements $\bar{G} = (\bar{N}, \bar{E}, \bar{\mathbf{s}}, \bar{\mathbf{e}})$ by $(\{\bar{\mathbf{s}}, \bar{\mathbf{e}}\}, \{(\bar{\mathbf{s}}, \bar{\mathbf{e}})\}, \bar{\mathbf{s}}, \bar{\mathbf{e}})$, and replace the local semantics of $\bar{\mathbf{s}}$ and $\bar{\mathbf{e}}$ by $Id_{\mathcal{B}} \sqcap \bigsqcap\{ [\![n]\!] \mid n \in \bar{N}\}$ and $[\![\bar{G}]\!]^*$, respectively. Continue with step 1.

The $PMFP_{bv}$ solution

$$PMFP_{BV\,(G^*,[\![\ ]\!])} : N^* \to \mathcal{F}_{\mathcal{B}} \ \text{ defined by}$$
$$\forall\, n \in N^*\ \forall\, b \in \mathcal{B}.\ PMFP_{BV\,(G^*,[\![\ ]\!])}(n)(b) = [\![\, n \,]\!](b).$$

$$[\![\, n \,]\!] = \begin{cases} Id_{\mathcal{B}} & \text{if } n = \mathbf{s}^* \\ [\![\, pfg(n) \,]\!]^* \circ [\![\, start(pfg(n)) \,]\!] \sqcap Const_{NonDestructible(n)} & \text{if } n \in N^*_X \\ \bigsqcap\{\, [\![\, m \,]\!] \circ [\![\, m \,]\!] \mid m \in pred_{G^*}(n)\} \sqcap Const_{NonDestructible(n)} & \text{otherwise.} \end{cases}$$

where N_X^* is the set of all ParEnd statements in the graph G*

The parallel bitvector coincidence theorem

- The parallel bitvector coincidence theorem shows that PMOP = $PMFP_{bv}$ for *B*.
- Within a side of a parallel structure, we simply use $pred_{G^*}(n)$ to calculate the value of the lattice at that point.
- But we also consider the other parallel component by taking the Meet of that value with the function $Const_{NonDestructible(n)}$. This captures the value computed across all interleaving predecessors – handling all possible interleavings at once.

PMOP = $PMFP_{bv}$

LEMMA 3.3.4. *The PMOP-solution of a parallel flow graph* $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ *that only consists of purely sequential parallel components* $G_1, \ldots, G_k$ *is given by:*

$$PMOP_{(G,[\![\ ]\!])}(end(G)) = \begin{cases} Const_{ff} & \text{if } \exists\, 1 \leq i \leq k.\ [\![end(G_i)]\!] = Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall\, 1 \leq i \leq k.\ [\![end(G_i)]\!] = Id_{\mathcal{B}} \\ Const_{tt} & \text{otherwise.} \end{cases}$$

Because the functions are distributive, and by the parallel coincidence theorem, the maximal fixed point solution is equivalent to the meet over paths solution.

Performance, implementation & results

- While they have a section for “implementation”, they don’t actually mention many specifics about implementing this algorithm.
- The paper computes code motion for cse, and partial dead code elimination for an example routine.
- Timing, memory usage? “The parallel version often runs faster than the sequential version.”

Discussion Topics / Questions

- Did the authors meet their stated goals?
- Could it run “fast enough”?
- What does this paper give us over the previous work cited?
- Possible extensions? Application to real languages? Are `par` blocks a reasonable abstraction?
- Can `par` blocks be extended to handle other control flow mechanisms?