Workshop on Essential Abstractions in GCC

Introduction to Data Flow Analysis

GCC Resource Center
(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

1 July 2012
Outline

• Motivation

• Live Variables Analysis

• Available Expressions Analysis

• Pointer Analysis
Part 2

Motivation
Dead Code Elimination

- No uses for variables a_3, b_4, c_5, and n_6
- Assignments to these variables can be deleted

How can we conclude this systematically?
Liveness Analysis of Variables

Find out at each program point \( p \), the variables that are used beyond \( p \)

\[
\begin{align*}
\text{B2} & : a_3 = 1; b_4 = 2 \quad c_5 = 3; n_6 = 6 \\
\text{B4} & : a_1 = \phi(1, a_7) \\
\text{B3} & : a_7 = a_1 + 1 \\
\text{B5} & : \begin{cases} \text{T} & \text{if } a_1 \leq 6 \\
\text{F} & \text{if } a_1 \leq 11 \\
\end{cases}
\end{align*}
\]

Which variables are used beyond this point?

\[
\begin{align*}
\text{B6} & : D.1200_8 = a_1 + 2 \\
& \quad a_9 = D.1200_8 + 3
\end{align*}
\]

Which variables are used beyond this point?

What about \( a_2 \)?

\[
\begin{align*}
\text{B7} & : a_2 = \phi(a_1, a_9) \\
& \quad \text{return } a_2
\end{align*}
\]

\[
\begin{align*}
\emptyset & \quad \emptyset
\end{align*}
\]
Liveness Analysis of Variables: Iteration 2

Find out at each program point \( p \), the variables that are used beyond \( p \)

\[
\begin{align*}
B2: & \quad a_{3} = 1; b_{4} = 2 \\
& \quad c_{5} = 3; n_{6} = 6
\end{align*}
\]

\[
\begin{align*}
B4: & \quad a_{1} = \phi (1, a_{7}) \\
& \quad \text{if } a_{1} \leq 6
\end{align*}
\]

\[
\begin{align*}
B5: & \quad \text{if } a_{1} \leq 11
\end{align*}
\]

\[
\begin{align*}
B6: & \quad D.1200_{8} = a_{1} + 2 \\
& \quad a_{9} = D.1200_{8} + 3
\end{align*}
\]

\[
\begin{align*}
B7: & \quad a_{2} = \phi (a_{1}, a_{9}) \\
& \quad \text{return } a_{2}
\end{align*}
\]

\[
\begin{align*}
\{a_{7}, a_{9}\} \\
\{a_{7}, a_{9}\} \\
\{a_{1}, a_{9}\} \\
\emptyset \text{(Conservative assumption)} \\
\{a_{1}, a_{9}\} \\
\{a_{1}, a_{9}\} \\
\emptyset
\end{align*}
\]

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Using Liveness Analysis for Dead Code Elimination

- Values of $a_{3}$, $a_{4}$, $c_{5}$, and $n_{6}$ are guaranteed not to be used.
- Why are the values of $a_{7}$ and $a_{9}$ meaningful at the exit of B2?
- We have assumed a $\phi$ function to be an ordinary expression in which operands are computed along every path reaching the computation.

$\phi(1, a_{7})$

if $a_{1} \leq 6$

F

$B_{3}$

$a_{7} = a_{1} + 1$

$B_{5}$

if $a_{1} \leq 11$

$B_{6}$

$D.1200.8 = a_{1} + 2$

$a_{9} = D.1200.8 + 3$

$B_{7}$

$a_{2} = \phi(a_{1}, a_{9})$

return $a_{2}$

$\emptyset$

$\{a_{7}, a_{9}\}$

$\{a_{7}, a_{9}\}$
Part 3

Live Variables Analysis
Defining Live Variables Analysis

A variable \( v \) is live at a program point \( p \), if some path from \( p \) to program exit contains an r-value occurrence of \( v \) which is not preceded by an l-value occurrence of \( v \).

- \( v \) is live at \( p \)
- \( v \) is not live at \( p \)
- \( v \) is live at \( p \)

Path based specification

\[
\begin{align*}
\text{Start} & \quad v = a \ast b \\
\text{End} & \quad a = v + 2
\end{align*}
\]
Defining Data Flow Analysis for Live Variables Analysis

Basic Blocks $\equiv$ Single statements or Maximal groups of sequentially executed statements

$Gen_k, \ Kill_k$

$Gen_i, \ Kill_i$

$Gen_j, \ Kill_j$

Local Data Flow Properties

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Local Data Flow Properties for Live Variables Analysis

- **Gen**<sub>n</sub> = \{ v | variable v is used in basic block n and is not preceded by a definition of v \}
- **Kill**<sub>n</sub> = \{ v | basic block n contains a definition of v \}

**r-value occurrence**
Value is only read, e.g. x, y, z in
x.sum = y.data + z.data

**l-value occurrence**
Value is modified e.g. y in
y = x.lptr

** Essential Abstractions in GCC **
** GCC Resource Center, IIT Bombay **
Local Data Flow Properties for Live Variables Analysis

- $Gen_n$: Use not preceded by definition
  
  Upwards exposed use

- $Kill_n$: Definition anywhere in a block
  
  Stop the effect from being propagated across a block
Defining Data Flow Analysis for Live Variables Analysis

Global Data Flow Properties

\[ \text{In}_k = \text{Gen}_k \cup (\text{Out}_k - \text{Kill}_k) \]

\[ \text{Out}_k = \text{In}_i \cup \text{In}_j \]

Edge based specifications

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Data Flow Equations For Live Variables Analysis

\[ \text{In}_n = \ (\text{Out}_n - \text{Kill}_n) \cup \text{Gen}_n \]

\[ \text{Out}_n = \begin{cases} 
BL & \text{if } n \text{ is End block} \\
\cup_{s \in \text{succ}(n)} \text{In}_s & \text{otherwise}
\end{cases} \]

\[ \text{In}_n \text{ and } \text{Out}_n \text{ are sets of variables.} \]
Performing Live Variables Analysis

\[
\{a_7, a_9\} \quad a_3 = 1; \ b_4 = 2 \\
\{a_7, a_9\} \quad c_5 = 3; \ n_6 = 6
\]

\[
\{a_7, a_9\} \quad a_1 = \phi (1, a_7) \\
\{a_7, a_9\} \quad \text{if } a_1 \leq 6
\]

\[
\{a_1, a_9\} \quad a_7 = a_1 + 1 \\
\{a_7, a_9\} \quad \text{if } a_1 \leq 11
\]

\[
\{a_1, a_9\} \quad D.1200_8 = a_1 + 2 \\
\{a_1, a_9\} \quad a_9 = D.1200_8 + 3
\]

\[
\{a_1, a_9\} \quad a_2 = \phi (a_1, a_9) \\
\{a_1, a_9\} \quad \text{return } a_2
\]

---

<table>
<thead>
<tr>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>B4</td>
<td>{a_7}</td>
</tr>
<tr>
<td>B3</td>
<td>{a_1}</td>
</tr>
<tr>
<td>B5</td>
<td>{a_1}</td>
</tr>
<tr>
<td>B6</td>
<td>{a_1}</td>
</tr>
<tr>
<td>B7</td>
<td>{a_1, a_9}</td>
</tr>
</tbody>
</table>
Strongly Live Variables Analysis

A variable $v$ is strongly live if it is used in

- in statement other than assignment statement, or  
  (this case is same as simple liveness analysis)
- in defining other strongly live variables in an assignment statement  
  (this case is different from simple liveness analysis)
Understanding Strong Liveness

Simple Liveness

\[ \{ x \} \]

\[ y = x \]

\[ \{ x \} \]

\[ \text{print}(x) \]

\[ \emptyset \]

Same

Strong Liveness

\[ \{ x \} \]

\[ y = x \]

\[ \{ x \} \]

\[ \text{print}(x) \]

\[ \emptyset \]

Same

Simple Liveness

\[ \{ x \} \]

\[ y = x \]

\[ \{ y \} \]

\[ \text{print}(y) \]

\[ \emptyset \]

\[ \{ y \} \]

\[ \emptyset \]

Same

Strong Liveness

\[ \{ z, x \} \]

\[ y = x \]

\[ \{ z \} \]

\[ \text{print}(z) \]

\[ \emptyset \]

\[ \{ z \} \]

\[ \emptyset \]

Different
Comparision of Simple and Strong Liveness for our Example

**Simple Liveness**

- \{a_7, a_9\}
- \{a_7, a_9\}
- \{a_7, a_9\}
- \{a_1, a_9\}
- if \(a_1 \leq 6\)
  - \(a_3 = 1; b_4 = 2\)
  - \(c_5 = 3; n_6 = 6\)
  - \(a_7 = a_1 + 1\)
  - \(D.1200_8 = a_1 + 2\)
  - \(a_9 = D.1200_8 + 3\)
  - print "Hello"

**Strong Liveness**

- \{a_7\}
- \{a_7\}
- \{a_7\}
- \{a_1\}
- if \(a_1 \leq 6\)
  - \(a_3 = 1; b_4 = 2\)
  - \(c_5 = 3; n_6 = 6\)
  - \(a_7 = a_1 + 1\)
  - \(D.1200_8 = a_1 + 2\)
  - \(a_9 = D.1200_8 + 3\)
  - print "Hello"
Using Data Flow Information of Live Variables Analysis

- Used for register allocation.
  If variable $x$ is live in a basic block $b$, it is a potential candidate for register allocation.

- Used for dead code elimination.
  If variable $x$ is not live after an assignment $x = \ldots$, then the assignment is redundant and can be deleted as dead code.
Part 4

Available Expressions Analysis
Defining Available Expressions Analysis

An expression $e$ is available at a program point $p$, if 
\textit{every} path from program entry to $p$ contains an evaluation of $e$
which is not followed by a definition of any operand of $e$.
Local Data Flow Properties for Available Expressions Analysis

Gen\(_n\) = \{ e \mid \text{expression } e \text{ is evaluated in basic block } n \text{ and this evaluation is not } \text{followed} \text{ by a definition of any operand of } e \}\)

Kill\(_n\) = \{ e \mid \text{basic block } n \text{ contains a definition of an operand of } e \}\)

<table>
<thead>
<tr>
<th>Entity</th>
<th>Manipulation</th>
<th>Exposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen(_n)</td>
<td>Expression</td>
<td>Use</td>
</tr>
<tr>
<td>Kill(_n)</td>
<td>Expression</td>
<td>Modification</td>
</tr>
</tbody>
</table>
Data Flow Equations For Available Expressions Analysis

\[ \begin{align*}
In_n & = \begin{cases} 
B & \text{n is Start block} \\
\bigcap_{p \in \text{pred}(n)} Out_p & \text{otherwise}
\end{cases} \\
Out_n & = Gen_n \cup (In_n - Kill_n)
\end{align*} \]

Alternatively,

\[ Out_n = f_n(In_n), \quad \text{where} \]

\[ f_n(X) = Gen_n \cup (X - Kill_n) \]

\[ In_n \text{ and } Out_n \text{ are sets of expressions.} \]
Using Data Flow Information of Available Expressions Analysis

- Common subexpression elimination
  - If an expression is available at the entry of a block $b$ and a computation of the expression exists in $b$ such that it is not preceded by a definition of any of its operands

  Then the expression is redundant

- Redundant expression must be upwards exposed

- Expressions in $Gen_n$ are downwards exposed
Part 5

Introduction to Pointer Analysis
## Code Optimization In Presence of Pointers

### Program

1. \( q = p; \)
2. \( \text{while ( \ldots ) \{ do \{ \)  
3. \( \quad q = q \rightarrow \text{next}; \)
4. \( \} \) \text{while ( \ldots ) } \)
5. \( p \rightarrow \text{data} = r1; \)
6. \( \text{print (} q \rightarrow \text{data}); \)
7. \( p \rightarrow \text{data} = r2; \)

### Memory graph at statement 5

- Is \( p \rightarrow \text{data} \) live at the exit of line 5? Can we delete line 5?
  - No, if \( p \) and \( q \) can be possibly aliased  
    (while loop or do-while loop with a circular list)
  - Yes, if \( p \) and \( q \) are definitely not aliased  
    (do-while loop without a circular list)
Code Optimization In Presence of Pointers

Original Program  Constant Propagation without aliasing  Constant Propagation with aliasing

\[ a = 5 \]
\[ x = &a \]
\[ b = *x \]

\[ a = 5 \]
\[ x = &a \]
\[ b = *x \]

\[ a = 5 \]
\[ x = &a \]
\[ b = 5 \]
The World of Pointer Analysis

Alias Analysis
- Alias analysis of reference parameters, fields of unions, array indices
- Alias analysis of data pointers

Pointer Analysis
- Points-to analysis of data and function pointers

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Alias Information Vs. Points-To Information

1. \( a \):
   - \( x = \&a \) denoted \( x \rightarrow a \)

2. \( b = x \):
   - “\( x \) and \( b \) are Aliases” denoted \( x \bowtie b \)

- What about transitivity?
  - Points-To: No.
  - Alias: Depends.
Introduction

Two important dimensions for precise pointer analysis are

- Flow Sensitivity
- Context Sensitivity
A flow-sensitive analysis computes the data flow information at each program point according to the control-flow of a program.

At the exit of node $n_4$

Flow insensitive information:
\{a\rightarrow b, a\rightarrow c, a\rightarrow d\}

Flow sensitive information:
\{a\rightarrow d\}
Context Sensitivity in Interprocedural Analysis
Issues with Pointer Analysis

• For precise pointer information, we require flow and context sensitive pointer analysis

• Flow and context sensitive pointer analysis computes a large size of information
Example of Points-to Analysis

```
x = &y  
y = &z  
z = &u
```

```
\{x \rightarrow y, y \rightarrow z, z \rightarrow u\}
```

```
\{x \rightarrow y, y \rightarrow z, z \rightarrow u\}
```

```
\{x \rightarrow y, y \rightarrow z, z \rightarrow u\}
```

```
\{x \rightarrow y, y \rightarrow z, z \rightarrow u\}
```

```
\{x \rightarrow y, y \rightarrow z, z \rightarrow z, u \rightarrow z\}
```

```
\{x \rightarrow y, y \rightarrow z, z \rightarrow u, z \rightarrow z, u \rightarrow z\}
```

```
\{x \rightarrow y, y \rightarrow z, z \rightarrow u, z \rightarrow z, u \rightarrow z\}
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Is All This Information Useful?

\[
\begin{aligned}
n_1 & \quad x &= \& y \\
& \quad y &= \& z \\
& \quad z &= \& u \\
& \quad \emptyset & \quad \{x \rightarrow y, y \rightarrow z, z \rightarrow u\} \\
\end{aligned}
\]

\[
\begin{aligned}
n_2 & \quad *z &= y \\
& \quad \{x \rightarrow y, y \rightarrow z, z \rightarrow u\} \\
\end{aligned}
\]

\[
\begin{aligned}
n_3 & \quad z &= y \\
& \quad \{x \rightarrow y, y \rightarrow z, z \rightarrow u\} \\
\end{aligned}
\]

\[
\begin{aligned}
n_4 & \quad \text{use } u \\
& \quad \text{use } x \\
\end{aligned}
\]

\[
\begin{aligned}
& \quad \{x \rightarrow y, y \rightarrow z, z \rightarrow u, z \rightarrow z, u \rightarrow z\} \\
\end{aligned}
\]
Improving pointer analysis

For a fast flow and context sensitive pointer analysis, we can reduce the number of computations done at a program point. This can be done in following ways:

- Computing pointer information for only those variables that are being used at some later program point.
- Propagating only the new data flow values obtained in current iteration to the next iteration.
Liveness Based Pointer analysis (L-FCPA)

- A flow and context sensitive pointer analysis
- Pointer information is not computed unless a variable becomes live.

  - Strong liveness is used for computing liveness information.
    If basic block contains statement like $x = y$, then $y$ is said to be live, if $x$ is live at the exit of basic block.

- Pointer information is propagated only in live range of the pointer
First Round of Liveness Analysis and Points-to Analysis

\[ x = &y \]
\[ y = &z \]
\[ z = &u \]

\[ \{ u \} \quad \{ u \rightarrow ? \} \]
\[ n_1 \]
\[ \{ u, x, z \} \quad \{ u \rightarrow ?, x \rightarrow y, z \rightarrow u \} \]

\[ \{ z \rightarrow u \} \quad \{ z \} \]
\[ \{ u \rightarrow ? \} \quad \{ u, x \} \]
\[ n_2 \]
\[ \{ u, x \} \quad \{ u \rightarrow ?, x \rightarrow y \} \]
\[ \{ u \rightarrow ? \}, \{ u \rightarrow ?, x \rightarrow y \} \]

\[ *z = y \]
\[ \{ u, x \} \quad \{ u \rightarrow ?, x \rightarrow y \} \]
\[ n_3 \]
\[ \{ u, x \} \quad \{ u \rightarrow ?, x \rightarrow y \} \]
\[ \{ u, x \} \quad \{ u \rightarrow ?, x \rightarrow y \} \]

\[ \{ u, x \} \quad \{ u \rightarrow ?, x \rightarrow y \} \]
\[ n_4 \]
\[ \{ u, x \} \quad \{ u \rightarrow ?, x \rightarrow y \} \]
Second Round of Liveness Analysis and Points-to Analysis

\[ x = \& y \]
\[ y = \& z \]
\[ z = \& u \]

\[ z \rightarrow u \]
\[ \{ z \rightarrow u \} \cup \{ y \rightarrow z, x \rightarrow y \} \]
\[ \{ z, x \{ z \} \} \]

\[ \{ u \} \{ u \rightarrow ? \} \]
\[ \{ u \rightarrow ?, x \rightarrow y \} \]
\[ \{ u, x \} \]

\[ \{ u \rightarrow ?, x \rightarrow y \} \]
\[ \{ u, x \} \]
\[ \{ u \rightarrow ?, x \rightarrow y \} \]
\[ \{ u \rightarrow ?, x \rightarrow y \} \]

\[ u \rightarrow z, x \rightarrow y \]
\[ \{ u \rightarrow z, x \rightarrow y \} \]
\[ \{ u, x \} \]

\[ \{ u \rightarrow ?, x \rightarrow y \} \cup \{ u \rightarrow z \} \]

\[ \{ u \rightarrow ?, x \rightarrow y \} \]
\[ \{ u \rightarrow z \} \]

\[ \{ u \rightarrow ?, x \rightarrow y \} \cup \{ u \rightarrow z \} \]

\[ \{ u \rightarrow ?, x \rightarrow y \} \]
\[ \{ u \rightarrow z \} \]

\[ \{ u \rightarrow ?, x \rightarrow y \} \cup \{ u \rightarrow z \} \]

\[ \{ u \rightarrow ?, x \rightarrow y \} \]
\[ \{ u \rightarrow z \} \]
Observation

• L-FCPA has 2 fixed point computations:
  ▶ Strong Liveness analysis
  ▶ Points-to analysis

• Liveness and Points-to passes are interdependent.

• Both the computations are done alternatively until final value converges.
Conclusions: New Insights in Pointer Analysis

- Usable pointer information is very small and sparse
- Earlier approaches reported inefficiency and non-scalability because they computed far more information than the actual usable information
- Triumph of *The Genius of AND over the Tyranny of OR*
- Future work
  - Redesign data structures by hiding them behind APIs
    - Current version uses linked lists and linear search
  - Incremental version
  - Using precise pointer information in other passes in GCC
Precise Context Information is Small and Sparse

Our contributions: Value based termination, liveness

<table>
<thead>
<tr>
<th>Program</th>
<th>Total no. of functions</th>
<th>0 call strings</th>
<th>1-4 call strings</th>
<th>5-8 call strings</th>
<th>9+ call strings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>L-FCPA</td>
<td>FCPA</td>
<td>L-FCPA</td>
<td>FCPA</td>
</tr>
<tr>
<td>lbm</td>
<td>22</td>
<td>16 (72.7%)</td>
<td>3 (13.6%)</td>
<td>6 (27.3%)</td>
<td>19 (86.4%)</td>
</tr>
<tr>
<td>mcf</td>
<td>25</td>
<td>16 (64.0%)</td>
<td>3 (12.0%)</td>
<td>9 (36.0%)</td>
<td>22 (88.0%)</td>
</tr>
<tr>
<td>bzip2</td>
<td>100</td>
<td>88 (88.0%)</td>
<td>38 (38.0%)</td>
<td>12 (12.0%)</td>
<td>62 (62.0%)</td>
</tr>
<tr>
<td>libquantum</td>
<td>118</td>
<td>100 (84.7%)</td>
<td>56 (47.5%)</td>
<td>17 (14.4%)</td>
<td>62 (52.5%)</td>
</tr>
<tr>
<td>sjeng</td>
<td>151</td>
<td>96 (63.6%)</td>
<td>37 (24.5%)</td>
<td>43 (28.5%)</td>
<td>45 (29.8%)</td>
</tr>
<tr>
<td>hmmmer</td>
<td>584</td>
<td>548 (93.8%)</td>
<td>320 (56.5%)</td>
<td>32 (5.5%)</td>
<td>175 (30.0%)</td>
</tr>
<tr>
<td>parser</td>
<td>372</td>
<td>246 (66.1%)</td>
<td>76 (20.4%)</td>
<td>118 (31.7%)</td>
<td>135 (36.3%)</td>
</tr>
<tr>
<td>h264ref</td>
<td>624</td>
<td>351 (56.2%)</td>
<td>?</td>
<td>240 (38.5%)</td>
<td>?</td>
</tr>
</tbody>
</table>

9+ call strings in L-FCPA: Tot 4, Min 10, Max 52, Mean 32.5, Median 29, Mode 10

9+ call strings in L-FCPA: Tot 14, Min 9, Max 56, Mean 27.9, Median 24, Mode 9
Precise Usable Pointer Information is Small and Sparse

Our contribution: liveness

<table>
<thead>
<tr>
<th>Program</th>
<th>Total no. of BBs</th>
<th>No. and percentage of basic blocks (BBs) for points-to (pt) pair counts</th>
<th>0 pt pairs</th>
<th>1-4 pt pairs</th>
<th>5-8 pt pairs</th>
<th>9+ pt pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>L-FCPA</td>
<td>FCPA</td>
<td>L-FCPA</td>
<td>FCPA</td>
<td>L-FCPA</td>
</tr>
<tr>
<td>lbm</td>
<td>252</td>
<td>229 (90.9%)</td>
<td>61 (24.2%)</td>
<td>23 (9.1%)</td>
<td>82 (32.5%)</td>
<td>0</td>
</tr>
<tr>
<td>mcf</td>
<td>472</td>
<td>356 (75.4%)</td>
<td>160 (33.9%)</td>
<td>116 (24.6%)</td>
<td>2 (0.4%)</td>
<td>0</td>
</tr>
<tr>
<td>libquantum</td>
<td>1642</td>
<td>1520 (92.6%)</td>
<td>793 (48.3%)</td>
<td>119 (7.2%)</td>
<td>796 (48.5%)</td>
<td>3</td>
</tr>
<tr>
<td>bzip2</td>
<td>2746</td>
<td>2624 (95.6%)</td>
<td>1085 (39.5%)</td>
<td>118 (4.3%)</td>
<td>12 (0.4%)</td>
<td>3</td>
</tr>
</tbody>
</table>

9+ pt pairs in L-FCPA: Tot 1, Min 12, Max 12, Mean 12.0, Median 12, Mode 12

| sjeng      | 6000             | 4571 (76.2%) | 3239 (54.0%) | 1208 (20.1%) | 12 (0.2%) | 221 (3.7%) | 41 (0.7%) | 0 | 2708 (45.1%) |
| hmer       | 14418            | 13483 (93.5%) | 8357 (58.0%) | 896 (6.2%) | 21 (0.1%) | 24 (0.2%) | 91 (0.6%) | 15 | 5949 (41.3%) |

9+ pt pairs in L-FCPA: Tot 6, Min 10, Max 16, Mean 13.3, Median 13, Mode 10

| parser     | 6875             | 4823 (70.2%) | 1821 (26.5%) | 1591 (23.1%) | 25 (0.4%) | 252 (3.7%) | 154 (2.2%) | 209 | 4875 (70.9%) |

9+ pt pairs in L-FCPA: Tot 13, Min 9, Max 53, Mean 27.9, Median 18, Mode 9

| h264ref    | 21315            | 13729 (64.4%) | ? | 4760 (22.3%) | ? | 2035 (9.5%) | ? | 791 (3.7%) | ? |

9+ pt pairs in L-FCPA: Tot 44, Min 9, Max 98, Mean 36.3, Median 31, Mode 9