Loop-Oriented Array- and Field-Sensitive Pointer Analysis for Automatic SIMD Vectorization

Yulei Sui, Xiaokang Fan, Hao Zhou and Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
2052 Sydney Australia

Jun 13, 2016
Contributions

- A new loop-oriented array- and field-sensitive inter-procedural pointer analysis using access-based location sets built in terms of a lazy memory modeling.

- The technique improves the effectiveness of both SLP and Loop-Level Vectorization by vectorizing more basic blocks and reducing runtime checks.

- Improves the performance of LLVM’s SLP (best speedup of 2.95%) and Loop vectorizer (best speedup of 7.18%).
Outline

- Background and Motivation
- Our approach: LPA
- Evaluation
Pointer Analysis

- Statically approximate runtime values of a pointer.
- Serves as the foundation for compiler optimisations and software bug detection.
- Generally answers the questions, such as does two pointer expressions (e.g., *a and *b) may access the same memory.

Automatic SIMD Vectorization

- Superword-Level Parallelism (SLP) vectorization packs isomorphic scalar instructions in the same basic block into a vector instruction.
- Loop-Level Vectorization (LLV) combines multiple consecutive iterations of a loop into a single iteration of a vector instruction.

Aim of this work:
- Study and develop interprocedural pointer analysis to generate more vectorized code (SLP) and reduce dynamic dependence checks (LLV).
Pointer Alias Analysis and SIMD Vectorization

Pointer Analysis

- Statically approximate runtime values of a pointer.
- Serves as the foundation for compiler optimisations and software bug detection.
- Generally answers the questions, such as does two pointer expressions (e.g., *a and *b) may access the same memory.
Pointer Alias Analysis and SIMD Vectorization

Pointer Analysis

- Statically approximate runtime values of a pointer.
- Serves as the foundation for compiler optimisations and software bug detection.
- Generally answers the questions, such as does two pointer expressions (e.g., *a and *b) may access the same memory.

Automatic SIMD Vectorization

- *Superword-Level Parallelism (SLP)* vectorization packs isomorphic scalar instructions in the same basic block into a vector instruction.
- *Loop-Level Vectorization (LLV)* combines multiple consecutive iterations of a loop into a single iteration of a vector instruction.
Pointer Alias Analysis and SIMD Vectorization

Pointer Analysis

- Statically approximate runtime values of a pointer.
- Serves as the foundation for compiler optimisations and software bug detection.
- Generally answers the questions, such as does two pointer expressions (e.g., *a and *b) may access the same memory.

Automatic SIMD Vectorization

- Superword-Level Parallelism (SLP) vectorization packs isomorphic scalar instructions in the same basic block into a vector instruction.
- Loop-Level Vectorization (LLV) combines multiple consecutive iterations of a loop into a single iteration of a vector instruction.

Aim of this work:

- Study and develop interprocedural pointer analysis to generate more vectorized code (SLP) and reduce dynamic dependence checks (LLV).
Pointer Alias Analysis and SLP Vectorization

void foo(float* A, float* B){
    A[0] = B[0];
}

vectorization

void foo(float* A, float* B){
    A[0:3] = B[0:3];
}

SLP vectorization: pack isomorphic non-alias memory accesses
void foo(float* A, float* B){
    A[0] = B[0];
}

void foo(float* A, float* B){
    A[0:3] = B[0:3];
}

SLP vectorization: pack isomorphic non-alias memory accesses

Imprecise alias information (e.g., A[i] and B[i] are aliases)
miss the vectorization opportunity!
void foo(float* A, float* B){
    for(int i = 0; i < N; i++)
}

vectorization
void foo(float* A, float* B){
    if( (&A[N-1] >= &B[0]) && (&B[N-1]) >= &A[0])
        for(int i = 0; i < N; i++)
    else
        for(int i = 0; i < N; i+=4)
}

Loop vectorization: Dynamic checks due to imprecise aliases
void foo(float* A, float* B) {
    for(int i = 0; i < N; i++)
}

void foo(float* A, float* B) {
    // vectorization
    if ( (&A[N-1] >= &B[0]) && (&B[N-1]) >= &A[0] )
        for(int i = 0; i < N; i++)
    else
        for(int i = 0; i < N; i+=4)
}

Loop vectorization: Dynamic checks due to imprecise aliases

Imprecise alias information (e.g., A[i] alias B[i])
increases the runtime overhead!
Motivation

Impact of LLVM’s Basic Alias Analysis on the effectiveness of SLP in LLVM

SLP: number of vectorizable and non-vectorizable basic blocks
Motivation

Impact of LLVM’s Basic Alias Analysis on the effectiveness of SLP in LLVM

SLP: number of vectorizable and non-vectorizable basic blocks

On average, up to 30.04% of basic blocks are vectorizable if more precise alias analysis is used in the above benchmarks!
Motivation

Impact of LLVM’s Basic Alias Analysis on the effectiveness of LLV in LLVM

LLV: percentage of runtime checks for disjoint and overlapping memory

On average, up to 96.35% of dynamic alias checks which return disjoint regions can be removed in the above benchmarks!
Precision of Pointer Alias Analysis

- Analysis dimensions (Most previous works):
  - flow-sensitivity
  - context-sensitivity
  - path-sensitivity
- Abstract memory modeling (This work)
  - Partition the infinite-size concrete addresses (stack/global/heap) into a finite number of abstract objects.
Abstract Memory modeling for Pointer Analysis

Abstract memory modeling is to partition the infinite-size concrete addresses (stack/global/heap) into a finite number of abstract objects.

```c
struct ST{
    int f1;
    int f2;
    int f3;
}

struct ST st;
int* p = &st.f1;
int* q = &st.f2;
```

Field-Insensitive Modeling:

Sensitive modeling: costly and overkill for precision

Insensitive modeling: coarse-grained (commonly used in pointer analysis)
Abstract Memory modeling for Pointer Analysis

Abstract memory modeling is to partition the infinite-size concrete addresses (stack/global/heap) into a finite number of abstract objects.

```c
struct ST{
    int f1;
    int f2;
    int f3;
}
struct ST st;
int* p = &st.f1;
int* q = &st.f2;
```

Field-Insensitive Modeling:

```
Alias(*p,*q) = false
```

Field-Sensitive Modeling:

```
Alias(*p,*q) = true
```

Insensitive modeling: coarse-grained (commonly used in pointer analysis)

Sensitive modeling: costly and overkill for precision.
Abstract Memory modeling for Pointer Analysis

Abstract memory modeling is to partition the infinite-size concrete addresses (stack/global/heap) into a finite number of abstract objects.

```c
struct ST{
    int f1;
    int f2;
    int f3;
}

struct ST st;
int* p = &st.f1;
int* q = &st.f2;
```

Field-Insensitive Modeling:

```
Alias(*p,*q) = true
```

Field-Sensitive Modeling:

```
Alias(*p,*q) = false
```

Insensitive modeling: coarse-grained (commonly used in pointer analysis)
Sensitive modeling: costly and overkill for precision:
Abstract Memory modeling for Pointer Analysis

Abstract memory modeling is to partition the infinite-size concrete addresses (stack/global/heap) into a finite number of abstract objects.

```
struct ST{
    int f1;
    int f2;
    int f3;
}
```

```
struct ST st;
int* p = &st.f1;
int* q = &st.f2;
```

Field-Sensitive Modeling:

- Alias(*p,*q) = false

Field-Insensitive Modeling:

- Alias(*p,*q) = true

Sensitive modeling: costly and overkill for precision.

Insensitive modeling: coarse-grained (commonly used in pointer analysis).

```
int* p = &a[0];
int a[3];
a[0] a[1] a[2]
int* q = &a[1];
```

Array-Sensitive Modeling:

- Alias(*p,*q) = false

Array-Insensitive Modeling:

- Alias(*p,*q) = true
Abstract Memory modeling for Pointer Analysis

Abstract memory modeling is to partition the infinite-size concrete addresses (stack/global/heap) into a finite number of abstract objects.

```
struct ST{
    int f1;
    int f2;
    int f3;
}
```

```
struct ST st;
int* p = &st.f1;
int* q = &st.f2;
```

Field-Sensitive Modeling:
Field-Insensitive Modeling:

```
Alias(*p,*q) = false
```
```
Alias(*p,*q) = true
```

```
int a[3];
```

Array-Insensitive Modeling:

```
Alias(*p,*q) = true
```

```
int* p = &a[0];
int* q = &a[1];
```

Array-Sensitive Modeling:

```
Alias(*p,*q) = false
```

Insensitive modeling: coarse-grained (commonly used in pointer analysis)
Sensitive modeling: costly and overkill for precision
Challenges

- How to find the right balance between efficiency and precision to model abstract objects?
- How to model an array access when its index is variant including nested aggregates (e.g., array of struct, struct of array)?
- How to integrate byte-precise abstract modeling into an inter-procedural pointer analysis to improve vectorization?
LPA: Loop-oriented Pointer Analysis

- Loop-oriented array- and field-sensitive inter-procedural pointer analysis using access-based location sets built in terms of lazy memory modeling.
  - Statically evaluate the symbolic range of pointers according to loop information.
  - Generate location sets lazily during points-to resolution.
LPA: Loop-oriented Pointer Analysis

- **Loop-oriented** array- and field-sensitive inter-procedural pointer analysis using **access-based location sets** built in terms of lazy memory modeling.
  - *Statically evaluate the symbolic range of pointers according to loop information.*
  - *Generate location sets lazily during points-to resolution.*
- **Separates** memory modeling as an independent concern from the rest of the pointer analysis.
  - *Facilitating the development of pointer analyses with desired efficiency and precision tradeoffs by reusing existing pointer resolution algorithms.*
LPA: Loop-oriented Pointer Analysis

- Loop-oriented array- and field-sensitive inter-procedural pointer analysis using access-based location sets built in terms of lazy memory modeling.
  - *Statically evaluate the symbolic range of pointers according to loop information.*
  - *Generate location sets lazily during points-to resolution.*
- Separates memory modeling as an independent concern from the rest of the pointer analysis.
  - *Facilitating the development of pointer analyses with desired efficiency and precision tradeoffs by reusing existing pointer resolution algorithms.*
- Generate efficient vector code and improves the performance of both SLP and loop vectorizer (best speedup over 7%).
Access-based location set

A location set $\sigma$ represents memory locations in terms of numeric offsets from the beginning of an abstract memory block\(^1\).

Our array-sensitive modeling, e.g., $arr[i]$ inside a loop:

- Interval range $i \in [lb, ub]$
- Access step $X \in \mathbb{N}^+$ (e.g., $X = 1$ if $arr$ is accessed consecutively inside the loop)

---

\(^1\)R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In PLDI ’95 (Field-Sensitive array-insensitive modeling based on location set)
Access-based location set

A location set $\sigma$ represents memory locations in terms of numeric offsets from the beginning of an abstract memory block$^1$.

Our array-sensitive modeling, e.g., $arr[i]$ inside a loop:

- **Interval range** $i \in [lb, ub]$
- **Access step** $X \in \mathbb{N}^+$ (e.g., $X = 1$ if $arr$ is accessed consecutively inside the loop)
- **Access trip** is a pair $(t, s)$ consists of
  - A trip count $t = \frac{(ub - lb)}{(X - 1)}$
  - A stride $s = es \times X$ where $es$ is the size of an array element.

---

$^1$R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In PLDI ’95 (Field-Sensitive array-insensitive modeling based on location set)
Access-based location set

An access-based location set derived from an object $a$ is:

$$\sigma = \langle \text{off}, \[(t_1, s_1), \ldots, (t_m, s_m)\]\rangle_a$$

where $\text{off} \in \mathbb{N}$ is an offset from the beginning of object $a$, and $T = \[(t_1, s_1), \ldots, (t_m, s_m)\]$ is an access-trip stack containing a sequence of (trip count, stride) pairs for handling a nested struct of arrays.
Access-based Location Set (Examples)

(a) An array with **consecutive** accesses

```
float a[16]; float *p = &a[0];
for(i=0;i<8;i++)
    p[i] = p[i+8];
```

(b) An array with **non-consecutive** accesses

```
struct {
    float f1[2]; float f2[2];
} a[4], *p, *q;
float *r;
p = &a[0];
for(i=0;i<4;i++)
    q = &p[i];
r = q->f1;
x = q->f2;
for(j=0;j<2;j++)
r[j] = x[j];
```
Access-based Location Set (Examples)

```c
float a[16]; float *p = &a[0];
for(i=0;i<16;i+=4){
    p[i] = i; i∈[0, 12], X=4, es=4
    p[i+1] = i + 1; i+1∈[1, 13], X=4, es=4
    p[i+2] = i + 2; i+2∈[2, 14], X=4, es=4
    p[i+3] = i + 3; i+3∈[3, 15], X=4, es=4
}
```

(b) An array with non-consecutive accesses

```c
struct {
    float f1[2]; float f2[2];
} a[4], *p, *q;
float *r;
p = &a[0];
for(i=0;i<4;i++){
    q = &p[i]; i∈[0, 3], X=1, es=16
    r = q->f1;  
    x = q->f2;
    for(j=0;j<2;j++)
        r[j] = x[j]; j∈[0, 1], X=1, es=4
}
```
Access-based Location Set (Examples)

(a) An array with

```c
float a[16]; float *p = &a[0];
for(i=0;i<8;i++)
    p[i] = p[i+8];
```

(b) An array with non-consecutive accesses

```c
struct {
    float f1[2]; float f2[2];
} a[4], *p, *q;
float *r;
p = &a[0];
for(i=0;i<4;i++)
    q = &p[i];
r = q->f1;
x = q->f2;
for(j=0;j<2;j++)
r[j] = x[j];
```

c) Nested arrays and structs with consecutive accesses

```
0 16 32 48 64
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
```
Andersen’s Pointer Analysis based on Access-based Location Set

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[S-ALLOC]</strong></td>
<td>( p = &amp;a ) ( \sigma = \langle 0, [] \rangle_a ) ( {\sigma} \subseteq pt(p) )</td>
</tr>
<tr>
<td><strong>[S-COPY]</strong></td>
<td>( p = q ) ( pt(q) \subseteq pt(p) )</td>
</tr>
<tr>
<td><strong>[S-LOAD]</strong></td>
<td>( q = e_p ) ( \sigma \in pt(q) ) ( \sigma' = GetLS(\sigma, e_q) ) ( pt(\sigma') \subseteq pt(q) )</td>
</tr>
<tr>
<td><strong>[S-STORE]</strong></td>
<td>( e_p = q ) ( \sigma \in pt(p) ) ( \sigma' = GetLS(\sigma, e_p) ) ( pt(q) \subseteq pt(\sigma') )</td>
</tr>
</tbody>
</table>

**GetLS**(
\( \langle off, T \rangle_a \),
\( \langle off + off_f, T \rangle_a \),
\( \langle off + C \times es, T \rangle_a \),
\( \langle off + lb \times es, T . push(\ \frac{ub' - lb'}{X} + 1, X \times es)\rangle_a \)

Points-to target in the points-to set of a pointer is not an abstract object but rather a location set derived from it.

If \( e_p \) is \( *p \)
else if \( e_p \) is \( p \rightarrow f \), where \( off_f \) is the offset of field \( f \) in array object \( a \)
else if \( e_p \) is \( p[i] \), where \( i \) is constant \( C \)
else if \( e_p \) is \( p[i] \), where \( i \in [lb, ub] \) with step \( X \),
\( [lb', ub'] = [lb, ub] \cap [0, m - 1] \) and \( m \) is size of array object \( a \)
Disambiguation of location sets

\[ \text{alias}(e_p, e_q) = \begin{cases} 
  \text{true} & \text{if } \exists \sigma'_p \in \text{pt}(p) \land \sigma'_q \in \text{pt}(q) : (\sigma_p, sz_p) \bowtie (\sigma_q, sz_q), \\
  \text{false} & \text{otherwise}
\end{cases} \]

where \( \sigma_p = \text{GetLS}(\sigma'_p) \land \sigma_q = \text{GetLS}(\sigma'_q) \) \hfill (1)

\[ (\sigma_p, sz_p) \bowtie (\sigma_q, sz_q) = \begin{cases} 
  \text{true} & \text{if } \text{obj}(\sigma_p) = \text{obj}(\sigma_q) \text{ and } \\
  \exists l_p \in \text{LS}(\sigma_p) \land l_q \in \text{LS}(\sigma_q): (l_p < l_q + sz_q) \land (l_q < l_p + sz_p) \\
  \text{false} & \text{otherwise}
\end{cases} \]

\hfill (2)
Disambiguation of location sets

(a) Disjoint location sets

float a[16]; float *p = &a[0];
for(i=0;i<16;i+=2){
    p[i] = ...;
    p[i+1] = ...;
    (0, [(8, 8)]_a, 4) \not\sqsubset (0, [(8, 8)]_a, 4)
}

(b) Overlapping location sets

float a[16]; float *p = &a[0];
for(i=0;i<16;i++)
p[i] = ...;
for(j=0;j<16;j+=2)
    (0, [(16, 4)]_a, 4) \not\sqsubset (0, [(8, 8)]_a, 4)
p[j] = ...;

0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64

p[i]  p[i+1]  overlapping
Field Unification

```
struct {float f1[8]; float f2[8];} a, *p;
p = &a;
float *q = p->f1, *r = p->f2;
for(int i=0;i<8;i=i+2){
    q[i]  = ...;
    q[i+1] = ...;
}
for(int j=0;j<8;j=j+2){
    r[j]   = ...;
    r[j+1] = ...;
}
```

(a) Code

(b) Default location sets

(c) Location sets with max offset limit: F = 32
Experiments

Compilation Process

Our experiments are conducted on

- An Intel Core i7-4770 CPU (3.40GHz) with an AVX2 SIMD extension, which supports 256 bit floating point and integer SIMD operations.
- 64-bit Ubuntu (14.0.4) with 32 GB memory.
## Experiments
### Program Characteristics

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>#Stmt</th>
<th>#Ptrs</th>
<th>#Objs</th>
<th>#CallSite</th>
</tr>
</thead>
<tbody>
<tr>
<td>173.applu</td>
<td>3.9</td>
<td>3361</td>
<td>20951</td>
<td>159</td>
<td>346</td>
</tr>
<tr>
<td>176.gcc</td>
<td>226.5</td>
<td>215312</td>
<td>545962</td>
<td>16860</td>
<td>22595</td>
</tr>
<tr>
<td>177.mesa</td>
<td>61.3</td>
<td>99154</td>
<td>242317</td>
<td>9831</td>
<td>3641</td>
</tr>
<tr>
<td>183.equake</td>
<td>1.5</td>
<td>2082</td>
<td>6688</td>
<td>236</td>
<td>235</td>
</tr>
<tr>
<td>188.ammp</td>
<td>13.4</td>
<td>14665</td>
<td>56992</td>
<td>2216</td>
<td>1225</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>60.1</td>
<td>119914</td>
<td>276301</td>
<td>6497</td>
<td>18713</td>
</tr>
<tr>
<td>197.parser</td>
<td>11.3</td>
<td>13668</td>
<td>36864</td>
<td>1177</td>
<td>1776</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4.6</td>
<td>1556</td>
<td>10650</td>
<td>436</td>
<td>380</td>
</tr>
<tr>
<td>300.twolf</td>
<td>20.4</td>
<td>23354</td>
<td>75507</td>
<td>1845</td>
<td>2059</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>168.1</td>
<td>130640</td>
<td>296288</td>
<td>3398</td>
<td>15399</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>8.2</td>
<td>7493</td>
<td>28965</td>
<td>669</td>
<td>439</td>
</tr>
<tr>
<td>433.milc</td>
<td>15</td>
<td>11219</td>
<td>30373</td>
<td>1871</td>
<td>1661</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>108.5</td>
<td>84966</td>
<td>224967</td>
<td>12302</td>
<td>8690</td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>103.8</td>
<td>62106</td>
<td>188284</td>
<td>2980</td>
<td>8006</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>3.8</td>
<td>12228</td>
<td>38850</td>
<td>513</td>
<td>2003</td>
</tr>
<tr>
<td>454.calculix</td>
<td>166.7</td>
<td>135182</td>
<td>532836</td>
<td>18814</td>
<td>23520</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td>11.5</td>
<td>25681</td>
<td>107656</td>
<td>3136</td>
<td>6566</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>51.5</td>
<td>55548</td>
<td>184660</td>
<td>3747</td>
<td>3553</td>
</tr>
<tr>
<td>465.tonto</td>
<td>143.1</td>
<td>418494</td>
<td>932795</td>
<td>28704</td>
<td>58756</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>25</td>
<td>20918</td>
<td>60347</td>
<td>1917</td>
<td>2775</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1208.2</strong></td>
<td><strong>1457541</strong></td>
<td><strong>3898253</strong></td>
<td><strong>117308</strong></td>
<td><strong>182338</strong></td>
</tr>
</tbody>
</table>
LPA’s analysis times ranging from 94.4 secs to 240.8 secs
On average, LPA’s analysis time occupies 42% over the total compilation time.
# Experiments

## SLP Vectorization Static Statistics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>basicaa</th>
<th>SCEVAA</th>
<th>LPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>173.applu</td>
<td>20</td>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>176.gcc</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>177.mesa</td>
<td>24</td>
<td>23</td>
<td>64</td>
</tr>
<tr>
<td>183.equake</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>188.ammp</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>46</td>
<td>23</td>
<td>53</td>
</tr>
<tr>
<td>433.milc</td>
<td>21</td>
<td>13</td>
<td>69</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>53</td>
<td>35</td>
<td>57</td>
</tr>
<tr>
<td>454.calculix</td>
<td>161</td>
<td>92</td>
<td>166</td>
</tr>
<tr>
<td>465.tonto</td>
<td>19</td>
<td>21</td>
<td>32</td>
</tr>
<tr>
<td>482.sphinx</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>351</strong></td>
<td><strong>217</strong></td>
<td><strong>482</strong></td>
</tr>
</tbody>
</table>

Number of basic blocks vectorized by SLP under the three alias analyses (larger is better).
## Experiments

### Loop Vectorization Static Statistics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BASICAA</th>
<th>SCEVAA</th>
<th>LPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>176.gcc</td>
<td>4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>177.mesa</td>
<td>121</td>
<td>137</td>
<td>88</td>
</tr>
<tr>
<td>197.parser</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>300.twolf</td>
<td>11</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>23</td>
<td>21</td>
<td>13</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>6</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>71</td>
<td>112</td>
<td>2</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>21</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>454.calculix</td>
<td>83</td>
<td>90</td>
<td>57</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td>65</td>
<td>79</td>
<td>16</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>30</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>465.tonto</td>
<td>110</td>
<td>118</td>
<td>38</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>551</strong></td>
<td><strong>652</strong></td>
<td><strong>238</strong></td>
</tr>
</tbody>
</table>

Number of static alias checks inserted by LLV under the three alias analyses (smaller is better).
Experiments

SLP: whole-program performance speedups

The whole-program speedups achieved by SLP under LPA normalized with respect to LLVM’s alias analyses
Experiments
Loop Vectorization: whole-program performance speedups

The whole-program speedups achieved by LLV under LPA normalized with respect to LLVM’s alias analyses
Conclusion

- A new loop-oriented array- and field-sensitive inter-procedural pointer analysis using access-based location sets built in terms of lazy memory model.

- The technique improves the effectiveness of both SLP and Loop-Level Vectorization by vectorizing more basic blocks and reducing dynamic checks.

- Improves the performance of LLVM’s SLP (best speedup of 2.95%) and Loop vectorizer (best speedup over 7%)
Thanks!

Q & A
Limitations

- Range analysis
  - SCEV in LLVM
  - Indirect array access e.g., a[*p]
  - Irregular loops, e.g., iterating arrays inside a loop with variant bounds
- More precise analysis methods, e.g., context-, heap-sensitivity
Andersen’s Analysis based on Field-Insensitive Modeling

\[ p = \&a \subseteq pt(p) \]
\[ p = q \subseteq pt(q) \subseteq pt(p) \]
\[ \ast p = q \quad o \in pt(p) \quad pt(q) \subseteq pt(o) \]
\[ p = \ast q \quad o \in pt(q) \quad pt(o) \subseteq pt(p) \]

Every allocation site is treated as a single memory object. Array and field accesses like \( p[i] = .. \) and \( p \rightarrow f = ... \) are treated as copies.