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
- Evalution

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.

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];
   A[1] = B[1];
   A[2] = B[2];
   A[3] = B[3];
}
```

vectorization $\Longrightarrow$

```
void foo(float* A, float* B){

   A[0:3] = B[0:3];

}
```

SLP vectorization: pack isomorphic non-alias memory accesses

Pointer Alias Analysis and SLP Vectorization

```
void foo(float* A, float* B){
    A[0] = B[0];
    A[1] = B[1];
    A[2] = B[2];
    A[3] = B[3];
}
```

vectorization ⟹

```
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!

Pointer Alias Analysis and LLV Vectorization

```
void foo(float* A, float* B){
    for(int i = 0; i < N; i++)
        A[i] = B[i] + K;
}
```

vectorization $\Longrightarrow$

```
void foo(float* A, float* B){
  if( (&A[N-1] >= &B[0]) && (&B[N-1]) >= &A[0]))
      for(int i = 0; i < N; i++)
          A[i] = B[i] + K;
  else
      for(int i = 0; i < N; i+=4)
          A[i:i+3] = B[i:i+3] + K;
}
```

Loop vectorization: Dynamic checks due to imprecise aliases

Pointer Alias Analysis and LLV Vectorization

```
void foo(float* A, float* B){
    for(int i = 0; i < N; i++)
        A[i] = B[i] + K;
}
```

vectorization ⟹

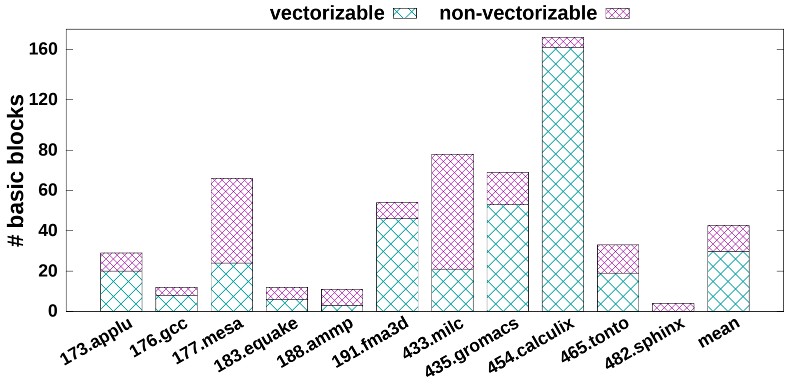
```
void foo(float* A, float* B){
  if( (&A[N-1] >= &B[0]) && (&B[N-1]) >= &A[0]))
      for(int i = 0; i < N; i++)
          A[i] = B[i] + K;
  else
      for(int i = 0; i < N; i+=4)
          A[i:i+3] = B[i:i+3] + K;
}
```

Loop vectorization: Dynamic checks due to imprecise aliases

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

Motivation

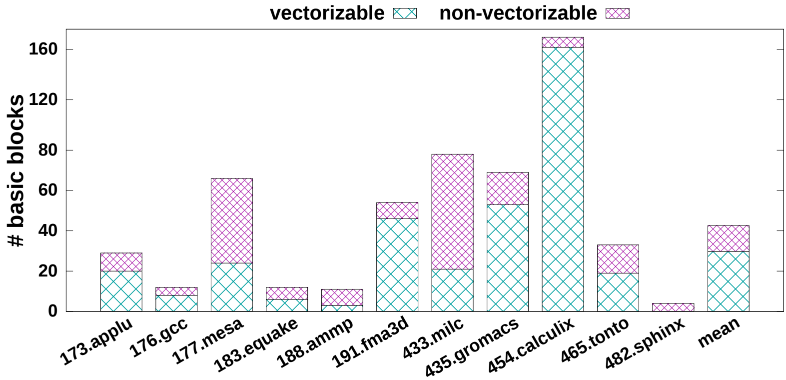
Impact of LLVM's Basic Alias Analysis on the effectiveness of SLP in LLVM



SLP: number of vectorizable and non-vectorizable basic blocks

Motivation

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

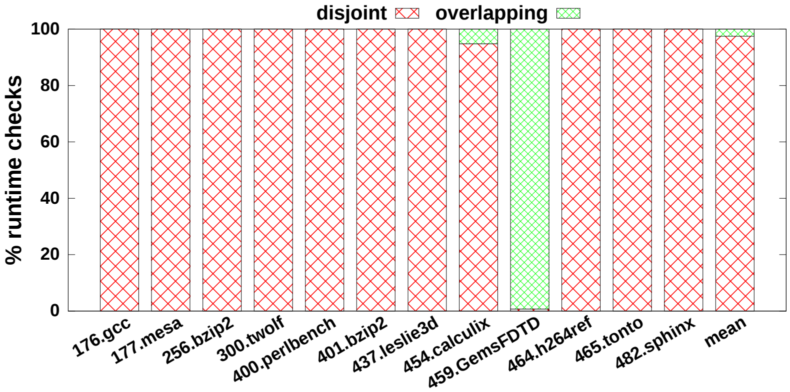


SLP: number of vectorizable and non-vectorizable basic blocks

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.

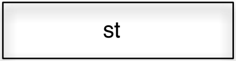
Abstract Memory modeling for Pointer Analysis

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

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

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

Field-Insensitive Modeling:



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-Insensitive Modeling:

Field-Sensitive Modeling:

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-Insensitive Modeling:

st

Alias(*p,*q) = true

Field-Sensitive Modeling:

st.f1	st.f2	st.f3

Alias(*p,*q) = false

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.

Array-Insensitive Modeling:

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

a

Alias(*p,*q) = true

Array-Sensitive Modeling:

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

a[0]	a[1]	a[2]

Alias(*p,*q) = false

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.

Array-Insensitive Modeling:

int a[3];

a

Alias(*p,*q) = true

Array-Sensitive Modeling:

int* p = &a[0];

int* q = &a[1];

a[0]	a[1]	a[2]

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 σ 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 σ 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 $t = \dfrac{(ub - lb)}{(X - 1)}$
  - A stride $s = es * X$ 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:

$$\sigma = \langle off, [\![(t_1, s_1), ..., (t_m, s_m)]\!]\rangle_a$$

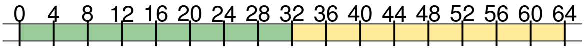
where $off \in \mathbb{N}$ is an offset from the beginning of object *a*, and $T = [\![(t_1, s_1), ..., (t_m, s_m)]\!]$ is an access-trip stack containing a sequence of (trip count, stride) pairs for handling a nested struct of arrays.

Access-based Location Set (Examples)

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

$\texttt{i} \in [0, 7]$, $X=1$, $es=4$

$\texttt{i+8} \in [8, 15]$, $X=1$, $es=4$



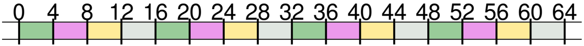
p[i]: $\langle 0, [\![(8,4)]\!]\rangle_a$ p[i+8]: $\langle 32, [\![(8,4)]\!]\rangle_a$

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

Access-based Location Set (Examples)

```
float a[16]; float *p = &a[0];
for(i=0;i<16;i+=4){
  p[i]   = i ;
  p[i+1] = i + 1;
  p[i+2] = i + 2;
  p[i+3] = i + 3;
}
```

$\mathtt{i}\in[0,12]$, X=4, es=4
$\mathtt{i+1}\in[1,13]$, X=4, es=4
$\mathtt{i+2}\in[2,14]$, X=4, es=4
$\mathtt{i+3}\in[3,15]$, X=4, es=4

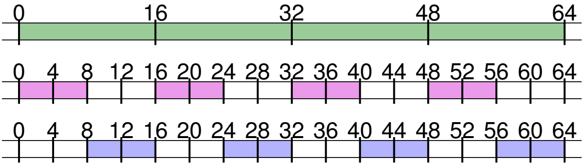


p[i]: $\langle 0, [\![(4,16)]\!]\rangle_a$ p[i+1]: $\langle 4, [\![(4,16)]\!]\rangle_a$
p[i+2]: $\langle 8, [\![(4,16)]\!]\rangle_a$ p[i+3]: $\langle 12, [\![(4,16)]\!]\rangle_a$

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

Access-based Location Set (Examples)

```
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
}
```



p[i]: $\langle 0, [\![(4,16)]\!]\rangle_a$ r[j]: $\langle 0, [\![(4,16),(2,4)]\!]\rangle_a$ x[j]: $\langle 8, [\![(4,16),(2,4)]\!]\rangle_a$

(c) **Nested arrays and structs** with consecutive accesses

Andersen's Pointer Analysis based on Access-based Location Set

$$[\texttt{S-ALLOC}]\ \frac{p = \&a \quad \sigma = \langle 0, [\,] \rangle_a}{\{\sigma\} \subseteq \mathrm{pt(p)}} \qquad [\texttt{S-LOAD}]\ \frac{q = e_p \quad \sigma \in \mathrm{pt(q)} \quad \sigma' = GetLS(\sigma, e_q)}{\mathrm{pt}(\sigma') \subseteq \mathrm{pt(q)}}$$

$$[\texttt{S-COPY}]\ \frac{p = q}{\mathrm{pt(q)} \subseteq \mathrm{pt(p)}} \qquad [\texttt{S-STORE}]\ \frac{e_p = q \quad \sigma \in \mathrm{pt(p)} \quad \sigma' = GetLS(\sigma, e_p)}{\mathrm{pt(q)} \subseteq \mathrm{pt}(\sigma')}$$

$$GetLS(\langle off, T\rangle_a, e_p) = \begin{cases} \langle off, T\rangle_a & \text{if } e_p \text{ is } *p \\ \langle off + off_f, T\rangle_a & \text{else if } e_p \text{ is } p \to f\text{, where } off_f \text{ is the offset of field } f \text{ in array object } a \\ \langle off + C * es, T\rangle_a & \text{else if } e_p \text{ is } p[i]\text{, where } i \text{ is constant C} \\ \langle off + lb * es, T.\mathtt{push}(\frac{ub'-lb'}{X} + 1, X * es)\rangle_a & \text{else if } e_p \text{ is } p[i]\text{, where } i \in [lb, ub] \text{ with step } X\text{, } [lb', ub'] = [lb, ub] \sqcap [0, m-1] \text{ and } m \text{ is size of array object } a \end{cases}$$

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

Disambiguation of location sets

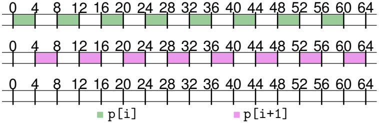
$$alias(e_p, e_q) = \begin{cases} true & \text{if } \exists\, \sigma'_p \in pt(p) \wedge \sigma'_q \in pt(q) : (\sigma_p, sz_p) \bowtie (\sigma_q, sz_q), \\ & \quad \text{where } \sigma_p = GetLS(\sigma'_p) \wedge \sigma_q = GetLS(\sigma'_q) \\ false & \text{otherwise} \end{cases} \tag{1}$$

$$(\sigma_p, sz_p) \bowtie (\sigma_q, sz_q) = \begin{cases} true & \text{if } obj(\sigma_p) = obj(\sigma_q) \text{ and} \\ & \exists\, l_p \in LS(\sigma_p) \wedge l_q \in LS(\sigma_q) \text{: } (l_p < l_q + sz_q) \wedge (l_q < l_p + sz_p) \\ false & \text{otherwise} \end{cases} \tag{2}$$

Disambiguation of location sets

```
float a[16]; float *p = &a[0];
for(i=0;i<16;i+=2){
  p[i]   = ...;
  p[i+1] = ...;
}
```

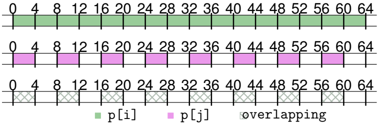
$$(\langle 0, [\![(8,8)]\!]\rangle_a, 4) \not\bowtie (\langle 4, [\![(8,8)]\!]\rangle_a, 4)$$



(a) Disjoint location sets

```
float a[16]; float *p = &a[0];
for(i=0;i<16;i++)
  p[i] = ...;
for(j=0;j<16;j+=2)
  p[j] = ...;
```

$$(\langle 0, [\![(16,4)]\!]\rangle_a, 4) \bowtie (\langle 0, [\![(8,8)]\!]\rangle_a, 4)$$

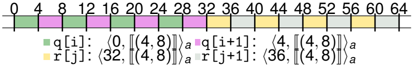


(b) Overlapping location sets

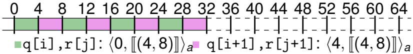
Field Unification

```
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



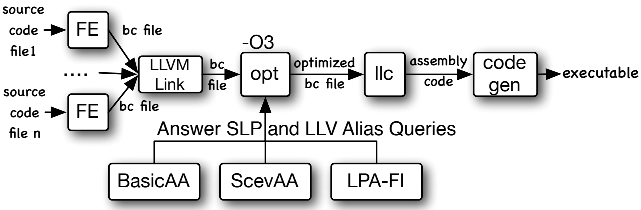
(b) **Default** location sets



(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.

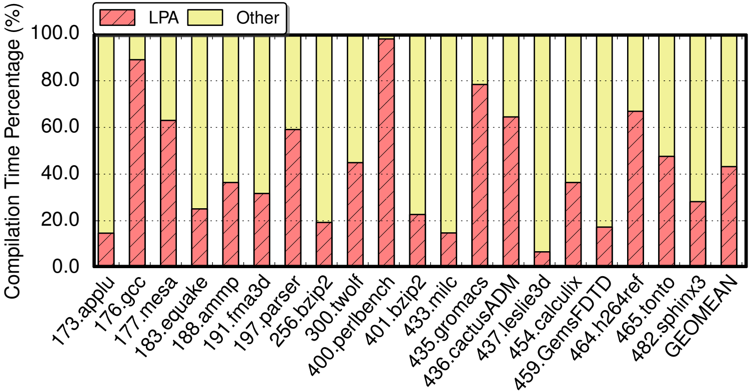
Experiments

Program Characteristics

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

Analysis Time

Percentage of analysis time over total compilation time



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

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

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

Experiments

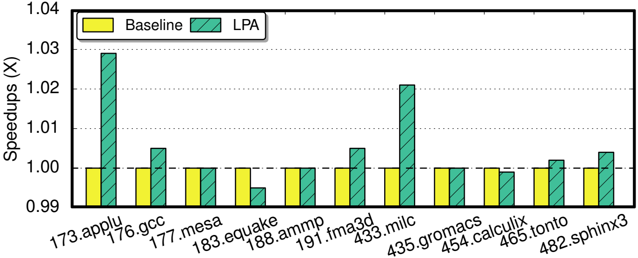
Loop Vectorization Static Statistics

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

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

Experiments

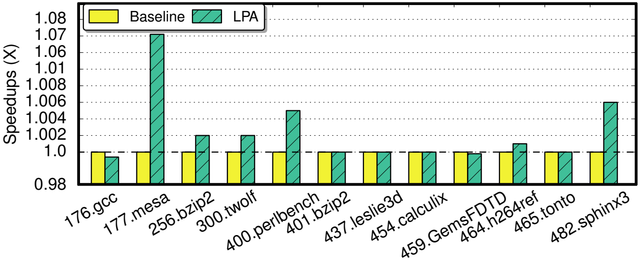
SLP: whole-program performance speedups



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

$$\frac{p = \&a}{\{a\} \subseteq pt(p)} \qquad \frac{p = q}{pt(q) \subseteq pt(p)}$$

$$\frac{*p = q \quad o \in pt(p)}{pt(q) \subseteq pt(o)} \qquad \frac{p = *q \quad o \in pt(q)}{pt(o) \subseteq pt(p)}$$

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.