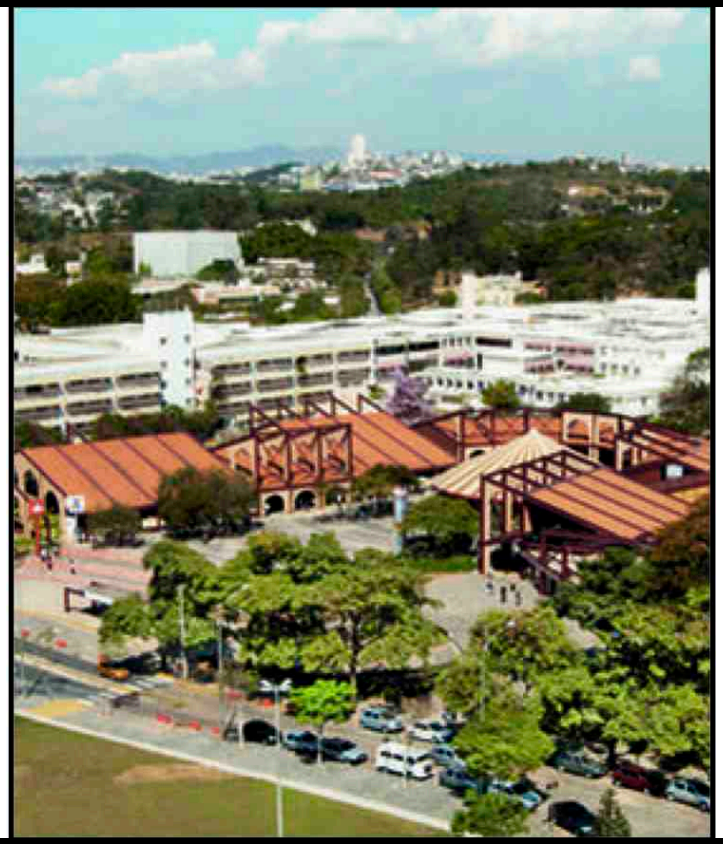
Compilers Laboratory

LABORATORIO DE COMPILADORES

Department of Computer Science – Universidade Federal de Minas Gerais – Brazil



Restrictification of Function Arguments

- Victor Campos
- Péricles Alves
- Henrique Nazaré
- Fernando Pereira

UFMG ICEx DCC

fernando@dcc.ufmg.br

Goal

The goal of this work is to disambiguate pointers used as arguments of functions

fernando@dcc.ufmg.br

Goal

The goal of this work is to disambiguate pointers used as arguments of functions

```
void acc_sum_chal(int src[], int *s, int *acc) {
  int i;
  for (i = 0; i < *s; i++) {
    *acc += src[i];
  }
}
```

Under which assumptions **src**, **s** and **acc** do not alias each other?

fernando@dcc.ufmg.br

Goal

The goal of this work is to disambiguate pointers used as arguments of functions

Restrictification lets us enable more compiler optimizations

fernando@dcc.ufmg.br

Enabling More Optimizations

```
void acc_sum_chal(int src[restrict], int *restrict s, int *restrict acc) {
  int i;
  for (i = 0; i < *s; i++) {
    *acc += src[i];
  }
}
```

[-O0]:
Time spent = 0.306336

[-O1]:
Time spent = 0.294689

[-O2]:
Time spent = 0.328521

[-O3]:
Time spent = 0.328576

[Restrictification]:
Time spent = 0.122120

2.6x faster
than LLVM –O3

fernando@dcc.ufmg.br

Another Example

```
void prefix_sum_chal(int *restrict v, int *restrict w, int N) {
  int i, j;
  for (i = 0; i < N; ++i) {
    v[i] = 0;
    for (j = 0; j < N; ++j) {
      v[i] += w[j];
    }
  }
}
```

```
$> ./prefix_sum.exe 10000 1

[-O0]:
Time spent = 0.360765

[-O1]:
Time spent = 0.111973

[-O2]:
Time spent = 0.121257

[-O3]:
Time spent = 0.118897

[Restrictification]:
Time spent = 0.024659
```

18x faster than LLVM –O0
and **5x** faster than LLVM –03

fernando@dcc.ufmg.br

Example: Vectorization

```
void acc_sum_chal(int src[restrict], int *restrict s, int *restrict acc) {
  int i;
  for (i = 0; i < *s; i++) {
    *acc += src[i];
  }
}
```

```
vector.body.prol:
%index.prol = phi i64 [ %index.next.prol, %vector.body.prol ], [ 0,
                                                                %2,
                                                                , [
                                                                %xtraiter,
%7 = getelementptr inbounds i32, i32* %src, i64 %index.prol
%8 = bitcast i32* %7 to <4 x i32>*
%wide.load.prol = load <4 x i32>, <4 x i32>* %8, align 4
%9 = getelementptr i32, i32* %7, i64 4
%10 = bitcast i32* %9 to <4 x i32>*
%wide.load11.prol = load <4 x i32>, <4 x i32>* %10, align 4
%11 = add nsw <4 x i32> %vec.phi.prol, %wide.load.prol
%12 = add nsw <4 x i32> %vec.phi10.prol, %wide.load11.prol
%index.next.prol = add i64 %index.prol, 8
%prol.iter.sub = add i64 %prol.iter, -1
%prol.iter.cmp = icmp eq i64 %prol.iter.sub, 0
br i1 %prol.iter.cmp, label %vector.body.preheader.split.loopexit, label
... %vector.body.prol, !llvm.loop !2
```

fernando@dcc.ufmg.br

In Previous Work...§

- We showed how to disambiguate pointers at the entry of loops

```
double **kmeans(int is_perform_atomic, /* in: */
            double **objects,     /* in: [numObjs][numCoords] */
            int numCoords,        /* no. coordinates */
            int numObjs,          /* no. objects */
            int numClusters,      /* no. clusters */
            double threshold,     /* % objects change membership */
            int *membership)      /* out: [numObjs] */
{
  int i, j, k, index, loop = 0, rc;
  int *newClusterSize;  /* [numClusters]: no. objects assigned in each
                   new cluster */
  double **clusters;    /* out: [numClusters][numCoords] */
  double **newClusters; /* [numClusters][numCoords] */
  double timing;
  int *local_newClusterSize;
  double *local_newClusters;

  /* [numClusters] clusters of [numCoords] double coordinates each */
  clusters = create_array_2d_f(numClusters, numCoords);

  /* Pick first numClusters elements of objects[] as initial cluster centers */
  for (i = 0; i < numClusters; i++)
    for (j = 0; j < numCoords; j++)
      clusters[i][j] = objects[i][j];

  /* Initialize membership, no object belongs to any cluster yet */
  for (i = 0; i < numObjs; i++)
    membership[i] = -1;

  do {
    delta = 0.0;

    for (i = 0; i < numClusters; i++) {

      newClusterSize[i] += local_newClusterSize[i];
      local_newClusterSize[i] = 0.0;
      for (k = 0; k < numCoords; k++) {
        newClusters[i][k] += local_newClusters[i * numCoords + k];
        local_newClusters[i * numCoords + k] = 0.0;
      }
    }
  }
```

- Using two techniques
  - Appending size information to data structures
  - Backward array size inference

§: *Runtime Pointer Disambiguation* – OOPSLA 2016 – Alves *et al*.

fernando@dcc.ufmg.br

In this Work:

- Larger scope: change the function with info from the whole program

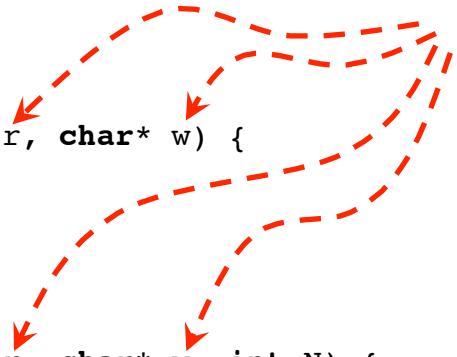
```
double **kmeans(int is_perform_atomic, /* in: */
            double **objects,     /* in: [numObjs][numCoords] */
            int numCoords,        /* no. coordinates */
            int numObjs,          /* no. objects */
            int numClusters,      /* no. clusters */
            double threshold,     /* % objects change membership */
            int *membership)      /* out: [numObjs] */
{
  int i, j, k, index, loop = 0, rc;
  int *newClusterSize;  /* [numClusters]: no. objects assigned in each
                   new cluster */
  double **clusters;    /* out: [numClusters][numCoords] */
  double **newClusters; /* [numClusters][numCoords] */
  double timing;
  int *local_newClusterSize;
  double *local_newClusters;

  /* [numClusters] clusters of [numCoords] double coordinates each */
  clusters = create_array_2d_f(numClusters, numCoords);

  /* Pick first numClusters elements of objects[] as initial cluster centers */
  for (i = 0; i < numClusters; i++)
    for (j = 0; j < numCoords; j++)
      clusters[i][j] = objects[i][j];

  /* Initialize membership, no object belongs to any cluster yet */
  for (i = 0; i < numObjs; i++)
    membership[i] = -1;

do {
    delta = 0.0;

  for (i = 0; i < numClusters; i++) {

      newClusterSize[i] += local_newClusterSize[i];
      local_newClusterSize[i] = 0.0;
      for (k = 0; k < numCoords; k++) {
        newClusters[i][k] += local_newClusters[i * numCoords + k];
        local_newClusters[i * numCoords + k] = 0.0;
      }
    }
}
```

- New techniques
  - Appending size information to data structures
  - Backward array size inference
  - Forward array size inference
  - Static alias analysis with dynamic linkage

fernando@dcc.ufmg.br

The Golden Question

```
1  char* get_new_array(int size, char init) {
2    char* A = malloc(size);
3    for (int i = 0; i < size - 1; i++) {
4      A[i] = init;
5    }
6    A[size - 1] = 0;
7    return A;
8  }
9
10 void sumTillZero(char* r, char* w) {
11   while (*r) {
12     *w += *(r++);
13   }
14 }
15
16 void sumTillSize(char* r, char* w, int N) {
17   for (int i = 0; i < N; i++) {
18     *w += r[i];
19   }
20 }
21
22 int main(int argc, char** argv) {
23   char A1[argc], A2 = 0;
24   char* A3 = get_new_array(argc, 0);
25   char* A4 = get_new_array(1, 0);
26   char* A5 = get_new_array(1 + argc, 1);
27   sumTillZero(A1, &A2);
28   sumTillZero(A3, A4);
29   sumTillSize(A3, A4, argc);
30   sumTillSize(A5, A5 + argc, argc);
31 }
```



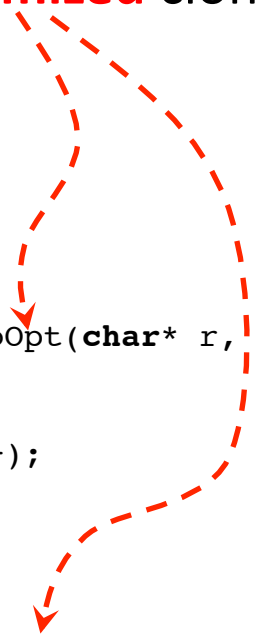
Question: under which assumptions can we show that the **arguments** of these functions: sumTillZero and sumTillSize, do not point to overlapping memory regions?

fernando@dcc.ufmg.br

The Motivation

```
1  char* get_new_array(int size, char init) {
2    char* A = malloc(size);
3    for (int i = 0; i < size - 1; i++) {
4      A[i] = init;
5    }
6    A[size - 1] = 0;
7    return A;
8  }
9
10 void sumTillZero(char* r, char* w) {
11   while (*r) {
12     *w += *(r++);
13   }
14 }
15
16 void sumTillSize(char* r, char* w, int N) {
17   for (int i = 0; i < N; i++) {
18     *w += r[i];
19   }
20 }
21
22 int main(int argc, char** argv) {
23   char A1[argc], A2 = 0;
24   char* A3 = get_new_array(argc, 0);
25   char* A4 = get_new_array(1, 0);
26   char* A5 = get_new_array(1 + argc, 1);
27   sumTillZero(A1, &A2);
28   sumTillZero(A3, A4);
29   sumTillSize(A3, A4, argc);
30   sumTillSize(A5, A5 + argc, argc);
31 }
```

Why? whenever we prove the absence of aliasing, we can invoke an **optimized** clone of the function.



```
1  void sumTillZeroOpt(char* r, char* w) {
2    int tmp = *w;
3    while (*r) {
4      tmp += *(r++);
5    }
6    *w = tmp;
7  }
8
9  void sumTillSizeOpt(char* r,char* w,int N) {
10   int tmp = *w;
11   for (int i = 0; i < N; i++) {
12     tmp += r[i];
13   }
14   *w = tmp;
15 }
```

fernando@dcc.ufmg.br

The Solution

```
1  int main(int argc, char** argv) {
2    char A1[argc], A2 = 0;
3    char* A3 = get_new_array(argc, 0);
4    char* A4 = get_new_array(1, 0);
5    char* A5 = get_new_array(1 + argc, 1);
6                                          Section 3.1
7    // PSA:ok, FRI:ok, BRI:no
8    sumTillZeroOpt(A1, &A2);
9                                          Section 3.2.1
10   // PSA:no, FRI:ok, BRI:no
11   if (A3 + argc ≤ A4 || A4 + argc ≤ A3)
12     sumTillZeroOpt(A3, A4);
13   else
14     sumTillZero(A3, A4);
15                                         Section 3.2.2
16   // PSA:no, FRI:ok, BRI:ok
17   sumTillSizeBri(A3, A4, argc);
18
19   // PSA:no, FRI:no, BRI:ok
20   sumTillSizeBri(A5, A5 + argc, argc);
21 }
```

How? We have invented three different techniques to disambiguate pointers:

- Purely static approach (PSA)
- Forward range inference (FRI)
- Backward range inference (BRI)

fernando@dcc.ufmg.br

The Need for Versioning

```
1  void sumTillZeroOpt(char* r, char* w) {
2    int tmp = *w;
3    while (*r) {
4      tmp += *(r++);
5    }
6    *w = tmp;
7  }
8
9  void sumTillZero(char* r, char* w) {
10   while (*r) {
11     *w += *(r++);
12   }
13 }
14
15 int main(int argc, char** argv) {
16   char A1[argc], A2 = 0;
17   sumTillZeroOpt(A1, &A2);
18   sumTillZero(A1, A1);
19 }
```

- There are calling sites that indeed contain aliasing
- The original function can still be called from outside

The Three Techniques

fernando@dcc.ufmg.br

Purely Static Approach

```
1  int main(int argc, char** argv) {
2    char A1[argc], A2 = 0;
3    char* A3 = get_new_array(argc, 0);
4    char* A4 = get_new_array(1, 0);
5    char* A5 = get_new_array(1 + argc, 1);
6
                                          Section 3.1
7    // PSA:ok, FRI:ok, BRI:no
8    sumTillZeroOpt(A1, &A2);
9
                                          Section 3.2.1
10   // PSA:no, FRI:ok, BRI:no
11   if (A3 + argc ≤ A4 || A4 + argc ≤ A3)
12     sumTillZeroOpt(A3, A4);
13   else
14     sumTillZero(A3, A4);
15
                                          Section 3.2.2
16   // PSA:no, FRI:ok, BRI:ok
17   sumTillSizeBri(A3, A4, argc);
18
19   // PSA:no, FRI:no, BRI:ok
20   sumTillSizeBri(A5, A5 + argc, argc);
21 }
```

PSA: sometimes simple pointer analysis is able to disambiguate memory locations statically

- Purely static approach (PSA)
- Forward range inference (FRI)
- Backward range inference (BRI)

```
1 void sumTillZeroOpt(char* r, char* w) {
2   int tmp = *w;
3   while (*r) {
4     tmp += *(r++);
5   }
6   *w = tmp;
7 }
```

fernando@dcc.ufmg.br

Purely Static Approach: Dynamic Linkage

```
1  int main(int argc, char** argv) {
2    char A1[argc], A2 = 0;
3    char* A3 = get_new_array(argc, 0);
4    char* A4 = get_new_array(1, 0);
5    char* A5 = get_new_array(1 + argc, 1);
6
7    // PSA:ok, FRI:ok, BRI:no                    Section 3.1
8    sumTillZeroOpt(A1, &A2);
9
10   // PSA:no, FRI:ok, BRI:no                    Section 3.2.1
11   if (A3 + argc ≤ A4 || A4 + argc ≤ A3)
12     sumTillZeroOpt(A3, A4);
13   else
14     sumTillZero(A3, A4);
15
16   // PSA:no, FRI:ok, BRI:ok                    Section 3.2.2
17   sumTillSizeBri(A3, A4, argc);
18
19   // PSA:no, FRI:no, BRI:ok
20   sumTillSizeBri(A5, A5 + argc, argc);
21 }
```

- Non-aliasing is proved statically.
- What if we do not have the body of sumTillZero at compilation time?

fernando@dcc.ufmg.br

Purely Static Approach: Dynamic Linkage

```
1  int main(int argc, char** argv) {
2    char A1[argc], A2 = 0;
3    char* A3 = get_new_array(argc, 0);
4    char* A4 = get_new_array(1, 0);
5    char* A5 = get_new_array(1 + argc, 1);
6                                          Section 3.1
7    // PSA:ok, FRI:ok, BRI:no
8    sumTillZeroOpt(A1, &A2);
9                                          Section 3.2.1
10   // PSA:no, FRI:ok, BRI:no
11   if (A3 + argc ≤ A4 || A4 + argc ≤ A3)
12     sumTillZeroOpt(A3, A4);
13   else
14     sumTillZero(A3, A4);
15                                         Section 3.2.2
16   // PSA:no, FRI:ok, BRI:ok
17   sumTillSizeBri(A3, A4, argc);
18
19   // PSA:no, FRI:no, BRI:ok
20   sumTillSizeBri(A5, A5 + argc, argc
21 }
```

- Non-aliasing is proved statically.
- What if we do not have the body of sumTillZero at compilation time?
- This approach lets us go a bit further than inlining.

```
1  int main(int argc, char** argv) {
2    char A1[argc], A2 = 0;
3    char* A3 = get_new_array(argc, 0);
4    char* A4 = get_new_array(1, 0);
5    char* A5 = get_new_array(1 + argc, 1);
6    sumTillZero(A1, &A2);
7    void* opt = dlsym(RTLD_NEXT, "sumTillZeroOpt_03FD");
8    if (opt) {
9      void (*sumTillZeroOpt)(int*, int*) = opt;
10     sumTillZeroOpt(A1, &A2);
11   } else {
12     sumTillZeroOpt(A1, &A2);
13   }
14 }
```

fernando@dcc.ufmg.br

Forward Range Inference

```
1  int main(int argc, char** argv) {
2    char A1[argc], A2 = 0;
3    char* A3 = get_new_array(argc, 0);
4    char* A4 = get_new_array(1, 0);
5    char* A5 = get_new_array(1 + argc, 1);
6                                          Section 3.1
7    // PSA:ok, FRI:ok, BRI:no
8    sumTillZeroOpt(A1, &A2);
9                                          Section 3.2.1
10   // PSA:no, FRI:ok, BRI:no
11   if (A3 + argc ≤ A4 || A4 + argc ≤ A3)
12     sumTillZeroOpt(A3, A4);
13   else
14     sumTillZero(A3, A4);
15                                         Section 3.2.2
16   // PSA:no, FRI:ok, BRI:ok
17   sumTillSizeBri(A3, A4, argc);
18
19   // PSA:no, FRI:no, BRI:ok
20   sumTillSizeBri(A5, A5 + argc, argc);
21 }
```

FRI: find and propagate the ranges of pointers throughout the program's flow graph.

- Purely static approach (PSA)
- Forward range inference (FRI)
- Backward range inference (BRI)

fernando@dcc.ufmg.br

Forward Range Inference

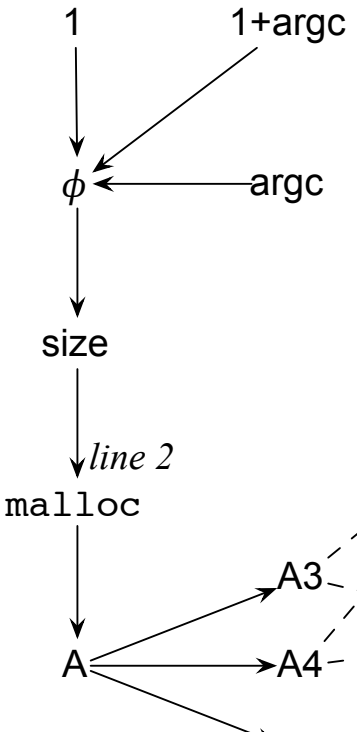
```
char* get_new_array(int size, char init) {
  char* A = malloc(size);
  for (int i = 0; i < size - 1; i++) {
    A[i] = init;
  }
  A[size - 1] = 0;
  return A;
}

int main(int argc, char** argv) {
  char A1[argc], A2 = 0;
  char* A3 = get_new_array(argc, 0);
  char* A4 = get_new_array(1, 0);
  char* A5 = get_new_array(1 + argc, 1);
  sumTillZero(A1, &A2);
  sumTillZero(A3, A4);
  sumTillSize(A3, A4, argc);
  sumTillSize(A5, A5 + argc, argc);
}
```

FRI: we propagate array size information forwardly through the program by means of an inter-procedural flow analysis.

fernando@dcc.ufmg.br

Forward Range Inference

```
22 int main(int argc, char** argv) {
23   char A1[argc], A2 = 0;
24   char* A3 = get_new_array(argc, 0);
25   char* A4 = get_new_array(1, 0);
26   char* A5 = get_new_array(1 + argc, 1);
27   sumTillZero(A1, &A2);
28   sumTillZero(A3, A4);
29   sumTillSize(A3, A4, argc);
30   sumTillSize(A5, A5 + argc, argc);
31 }
```



line 28

```
if (A3 + argc + 1 ≤ A4 || A4 + argc + 1 ≤ A3)
   sumTillZeroOpt(A3, A4);
else
   sumTillZero(A3, A4);
```

line 29

```
if (A3 + argc + 1 ≤ A4 || A4 + argc + 1 ≤ A3)
   sumTillSizeOpt(A3, A4, argc);
else
   sumTillSize(A3, A4, argc);
```

line 30

~~if (A5 + argc + 1 ≤ A5 || A5 + argc + 1 ≤ A5)~~
~~sumTillSizeOpt(A5, A5 + argc, argc);~~

```
else
   sumTillSize(A5, A5 + argc, argc);
```

This check is trivially false, and we do not try to call the optimized function in this case.

fernando@dcc.ufmg.br

Backward Range Inference

```
1  int main(int argc, char** argv) {
2    char A1[argc], A2 = 0;
3    char* A3 = get_new_array(argc, 0);
4    char* A4 = get_new_array(1, 0);
5    char* A5 = get_new_array(1 + argc, 1);
6                                          Section 3.1
7    // PSA:ok, FRI:ok, BRI:no
8    sumTillZeroOpt(A1, &A2);
9                                          Section 3.2.1
10   // PSA:no, FRI:ok, BRI:no
11   if (A3 + argc ≤ A4 || A4 + argc ≤ A3)
12     sumTillZeroOpt(A3, A4);
13   else
14     sumTillZero(A3, A4);
15                                         Section 3.2.2
16   // PSA:no, FRI:ok, BRI:ok
17   sumTillSizeBri(A3, A4, argc);
18
19   // PSA:no, FRI:no, BRI:ok
20   sumTillSizeBri(A5, A5 + argc, argc);
21 }
```

BRI: analyze the code of a function to find out the bounds of arrays that avoid aliasing.

- Purely static approach (PSA)
- Forward range inference (FRI)
- Backward range inference (BRI)

```
1 void sumTillSizeBri(char* r,char* w,int N) {
2   if (r + N ≤ w || w + 1 ≤ r) {
3     // See Figure 2, lines 9-15
4     sumTillSizeOpt(r, w, N);
5   } else {
6     // See Figure 1, lines 16-20
7     sumTillSize(r, w, N);
8   }
9 }
```

fernando@dcc.ufmg.br

Backward Range Inference

```
1  void countCh(int** A, char* B, char* C
2               int M, int N, int X, int Y) {
3    int bA = max(N*M*4,(X*N+Y)*4);
4    int bB = M;
5    int bC = N;
6    if ((A + bA ≤ bB || B + bB ≤ bA)
7    && (A + bA ≤ bC || C + bC ≤ bA)
8    && (B + bB ≤ bC || C + bC ≤ bB)) {
9      // call optimized version of countCh
10   } else {
11     // original code:
12     for (int i = 0; i < M; i++) {
13       for (int j = 0; j < N; j++) {
14         A[i*N+j] += B[i] > C[j] ? 1 : 0;
15       }
16     }
17     return A[X*N+Y];
18   }
19 }
```

- What is the minimum size of each array, so that their indices will never overlap?
  - Run symbolic range analysis on the memory indices.
  - Size of array is convex hull of every variable used to index it.

fernando@dcc.ufmg.br

Comparison

Technique	WPI	Call Site	Fun. Body	Dynamic
PSA	Yes	Yes	No	No
FRI	Yes	Yes	No	Yes
BRI	No	No	Yes	Yes

- **PSA**: Purely Static Approach; **FRI**: Forward Range Inference; **BRI**: Backward Range Inference.
- **WPI**: Whole Program Information affects precision.
- **Call Site**: inserts computation at call sites.
- **Fun. Body**: inserts computation at the target's function body.
- **Dynamic**: may call either original or optimized function, based on runtime values.

Evaluation

- We have implemented restrictification in LLVM 3.7
- **Small Benchmarks**:
  - BitBench
  - MiBench
  - Shootout-C++
  - Stanford
- **Large Benchmarks**: the OpenCV library.
- **Environment**: Intel Xeon 2.0GHz, with 16GB of RAM running Linux Ubuntu.

fernando@dcc.ufmg.br

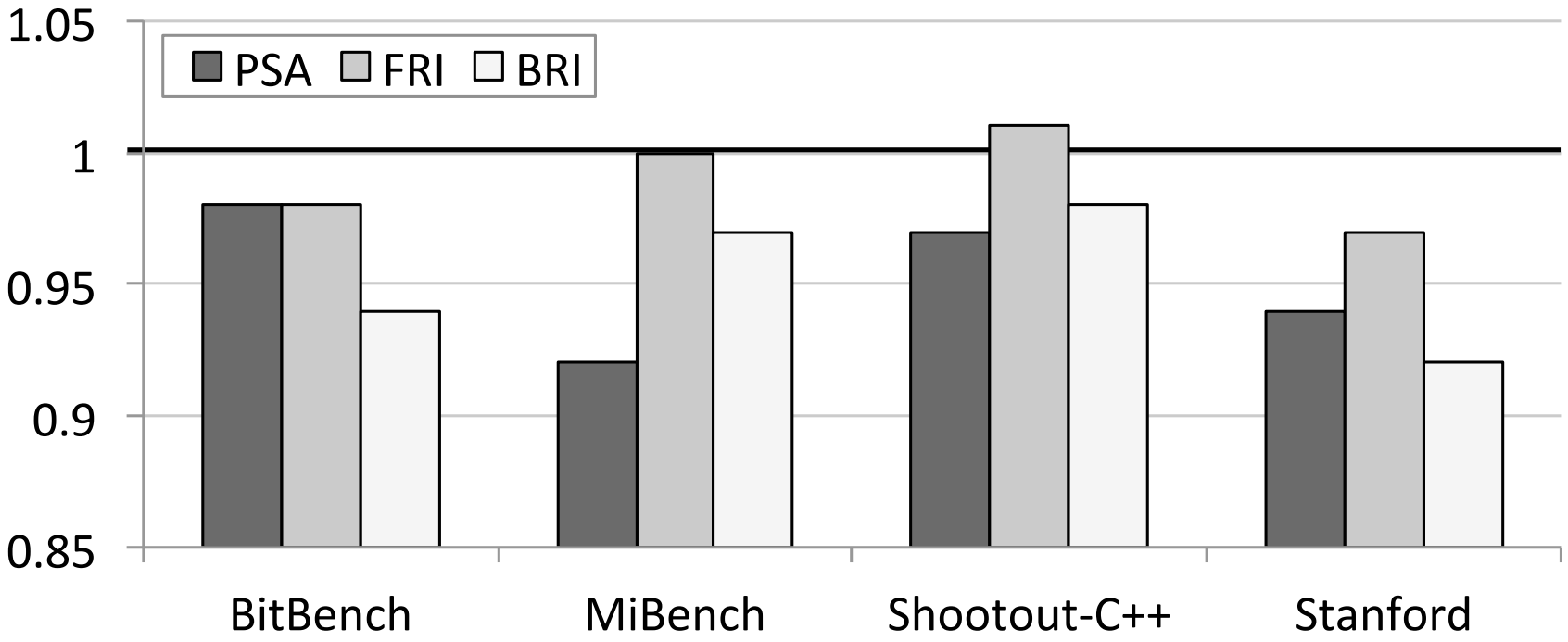
Reach

Benchmark	PSA	FRI	BRI
BitBench	37%	6%	17%
MiBench	15%	1%	72%
Shootout-C++	26%	4%	44%
Stanford	60%	10%	63%

- Reach of the three different disambiguation methods.
- **PSA**: Number of call sites optimized by the Purely Static Approach.
- **FRI**: Number of call sites instrumented by Forward Range Inference.
- **BRI**: Number of functions guarded by Backward Range Inference.

fernando@dcc.ufmg.br

Speedup on Small Benchmarks

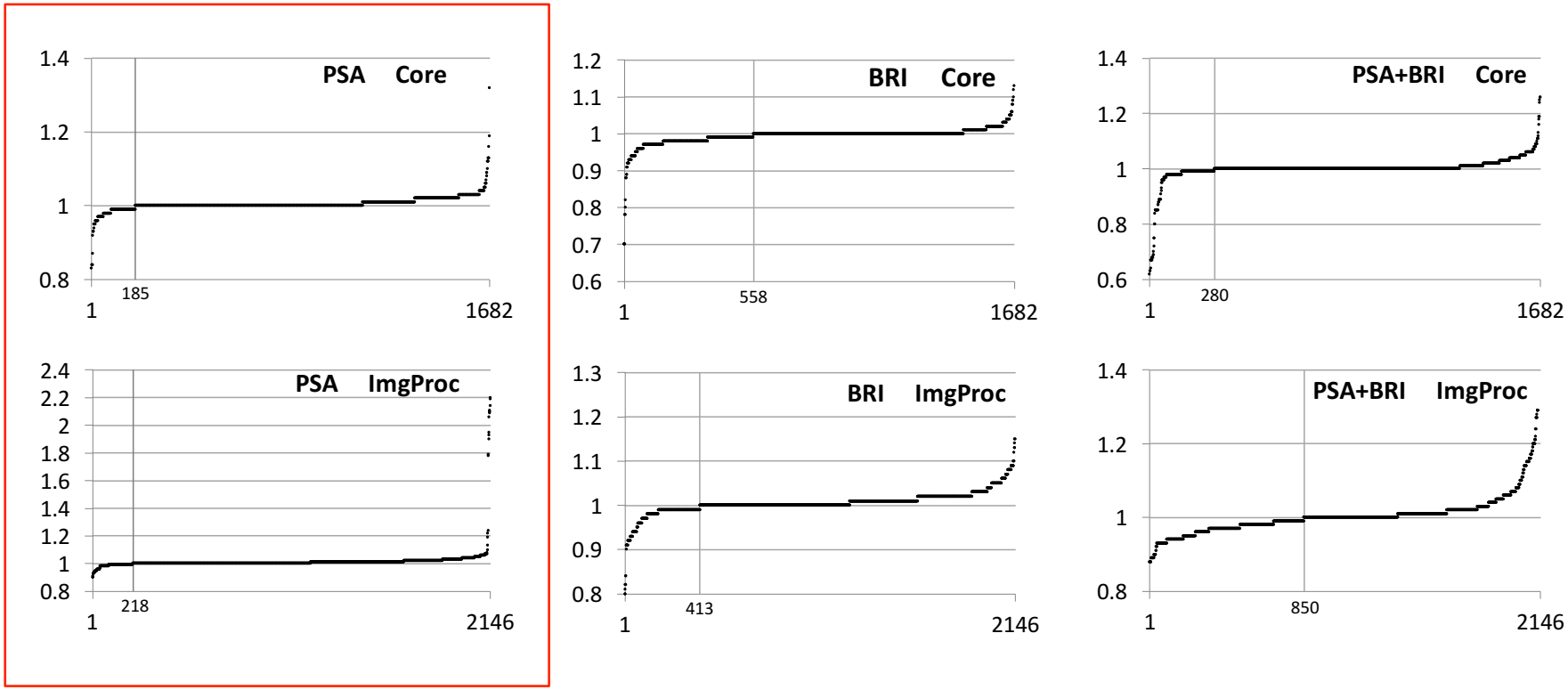


- **PSA**: Purely Static Approach; **FRI**: Forward Range Inference; **BRI**: Backward Range Inference.
- Bars represent normalized runtime. The lower the bar, the better.
- Baseline: LLVM 3.7 –O3

fernando@dcc.ufmg.br

PSA on OpenCV

Each dot is a benchmark. Each suite contains around 2000 benchmarks.

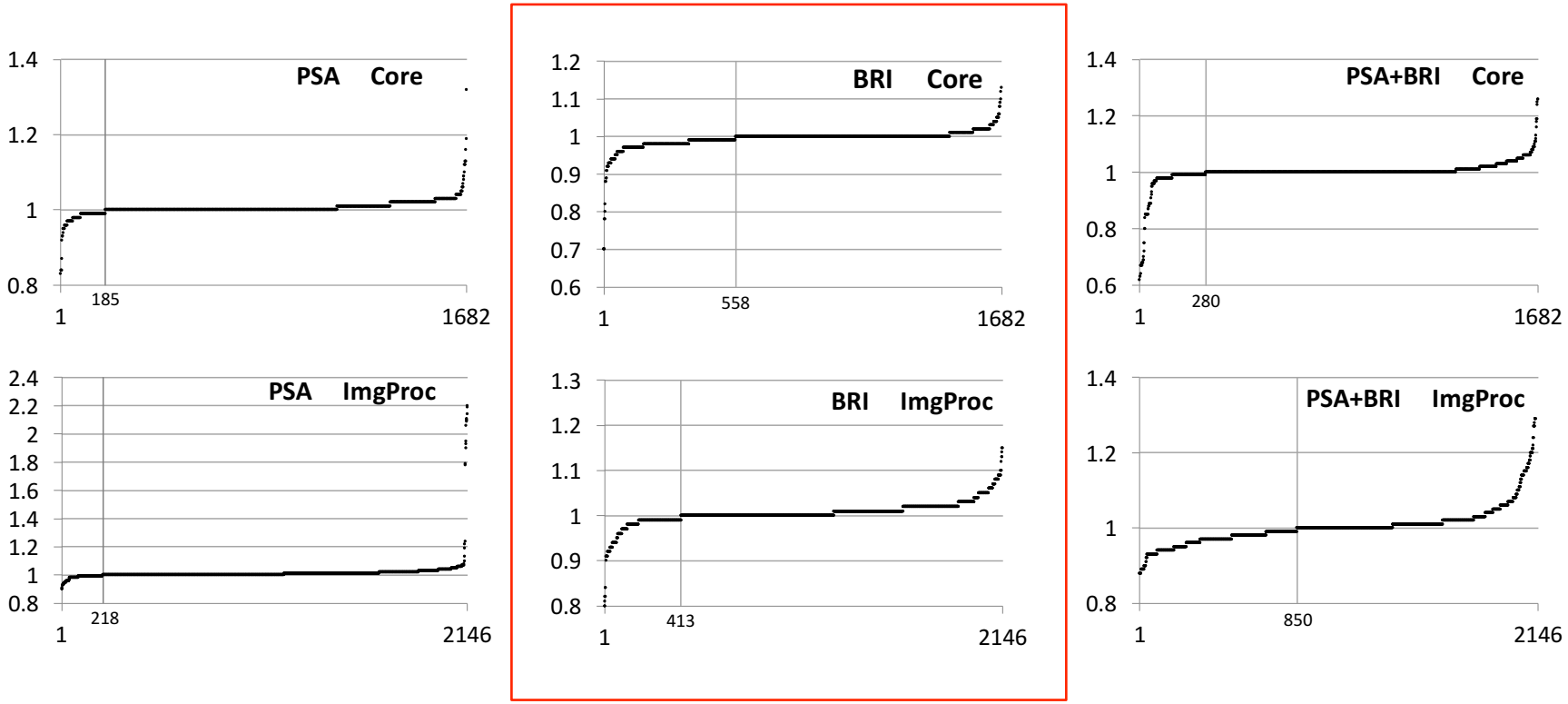


PSA: In the **core** suite, 11% of the benchmarks had improvements in runtime, with individual speedups of as much as 1.2x. In the imgproc, 10% of the benchmarks showed speedups, with individual numbers as high as 10%.

fernando@dcc.ufmg.br

BRI on OpenCV

Each dot is a benchmark. Each suite contains around 2000 benchmarks.

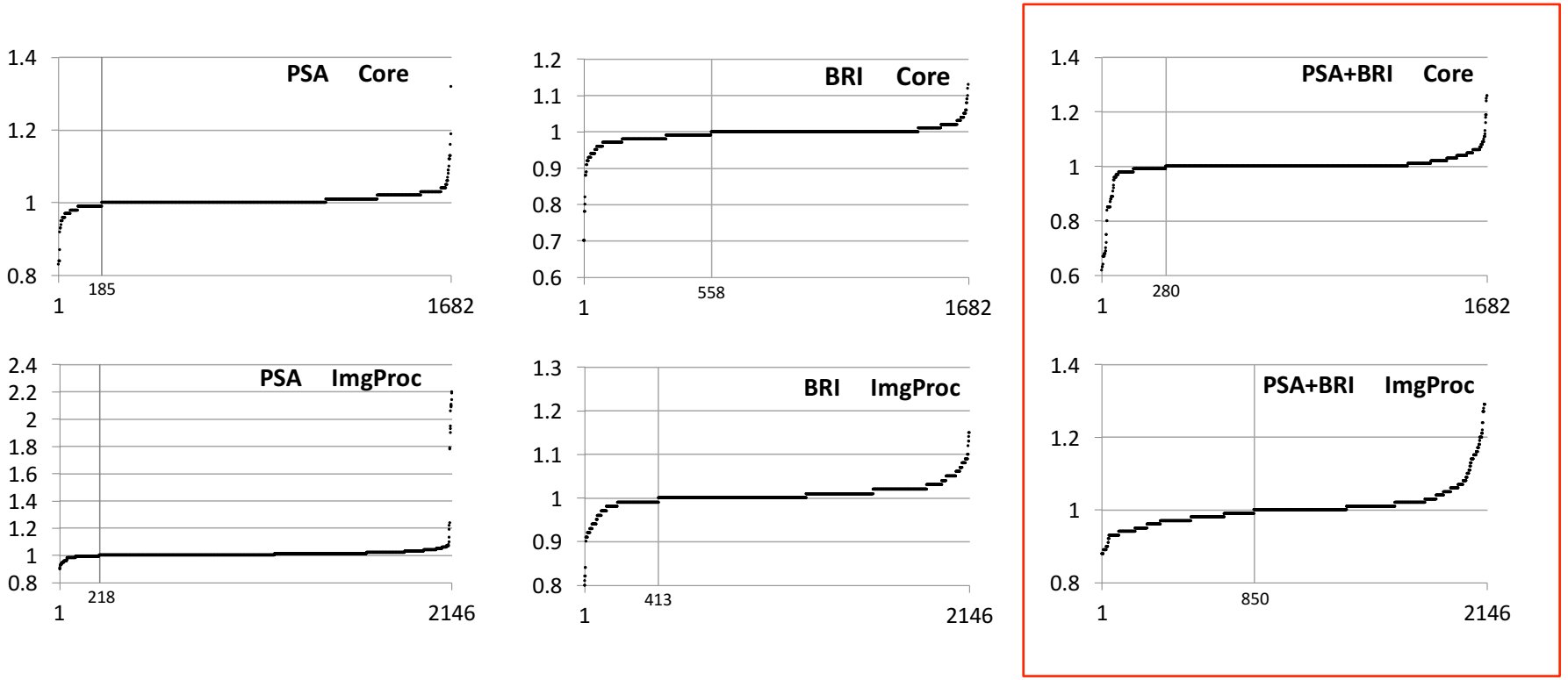


BRI: we notice improvements over PSA: in the **core** suite, the BRI produced speedups as high as 42%; in the second, 19% of the benchmarks experienced speedups, with the highest decrease in runtime of 20%.

fernando@dcc.ufmg.br

Combining the Approaches

Each dot is a benchmark. Each suite contains around 2000 benchmarks.



Combined approach: PSA goes over programs in a first moment, and BRI is then used to catch cases that PSA misses. On the **core** suite, 16% of the benchmarks had gains, of which the best gain gave us 61% of speedup.

fernando@dcc.ufmg.br

Increase in Parallelism

- Total of loops: 28,607
- Parallel loops:
  - Without restrictification: 1,454
  - With restrictification: +332
    - Increase of 23%

We have coded a simple parallel loop analysis, and we have applied it on the loops available in the OpenCV test cases.

fernando@dcc.ufmg.br

Code Size Expansion

- Code may increase due to cloning.
- Backward range inference brings the largest increase.
  - Around 25% increase in OpenCV
  - Greatest expansion: 74% in MiBench
  - Least expansion: 1% in BitBench

fernando@dcc.ufmg.br

Conclusions

- Paper: Restrictification of Function Arguments

```
void acc_sum_chal(int src[restrict], int *restrict s, int *restrict acc) {
  int i;
  for (i = 0; i < *s; i++) {
    *acc += src[i];
  }
}
```

[-O3]:
Time spent = 0.328576

[Restrictifiction]:
Time spent = 0.122120

- We disambiguate pointers used as arguments of functions
- This disambiguation gives the compiler more freedom to optimize code

- Victor Campos - victorsc@dcc.ufmg.br
- Péricles Alves - periclesroalves@gmail.com
- Henrique Nazaré - henrique.nazare.santos@gmail.com
- Fernando Pereira - fernando@dcc.ufmg.br (that's me!)

Video tutorial: hCps://youtu.be/1blYSsDV5Qc