Automatic speculative POlyhedraL Loop Optimizer

Aravind S R

University of of Strasbourg

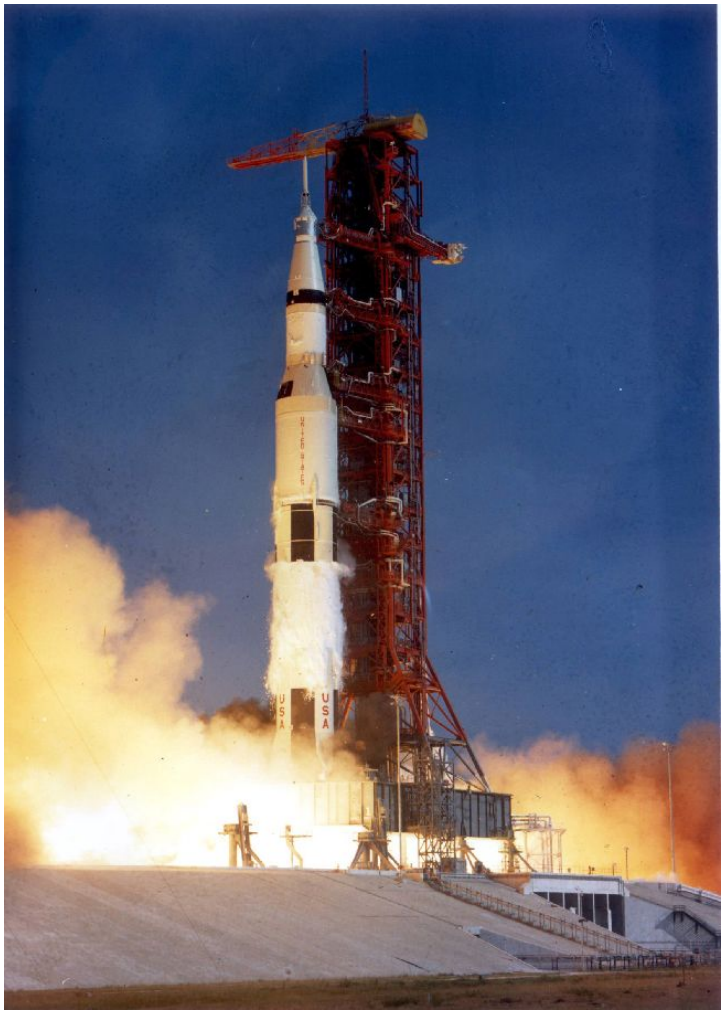
aravind.sukumaran-rajam@inria.fr

December 6, 2013

Motivation

- Parallelism is going to be the key for performance in future
- Writing efficient parallel code is not trivial
- Static polyhedral tools are very limited
  (do not handle pointer, while loops, etc..)

Apollo



Apollo

Apollo stands for the Automatic speculative POLyhedral Loop Optimizer

Features

- Dynamic
- Polytope model
- Speculative
- Handle all loop types
- Support all optimizations in current compilers

Apollo uses the experience from VMAD

Background {Thread level speculation}

- Used for codes having dynamic dependence behavior
- Execute thread(s) without waiting for dependence resolution
- Rollback on dependence violation

Background {Thread level speculation}

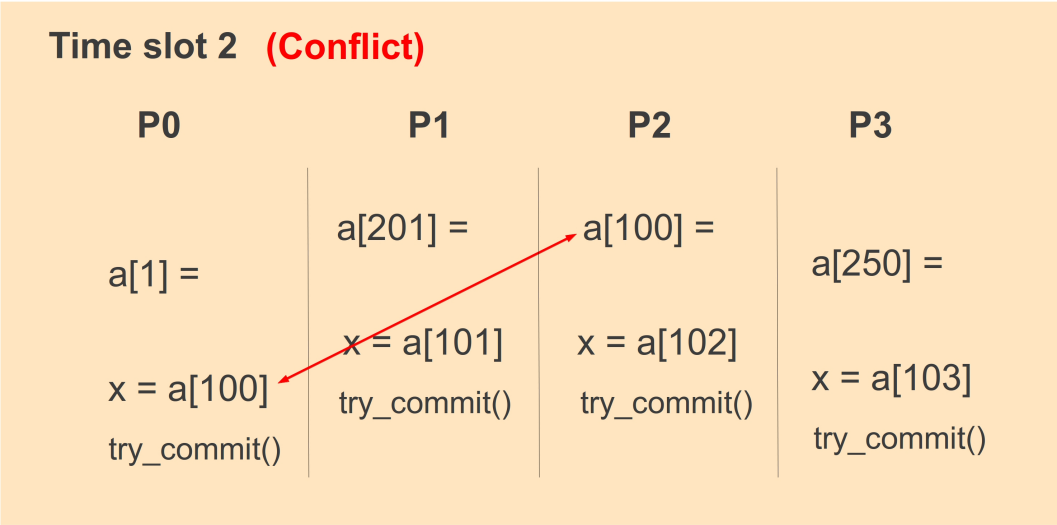
```
while (cond)
{
    a[i] = ...;
    ..
    ..
    ..
    x = a[j];
}
```

Time slot 1

P0	P1	P2	P3
a[1] =	a[2] =	a[3] =	a[4] =
x = a[100]	x = a[101]	x = a[102]	x = a[103]
try_commit()	try_commit()	try_commit()	try_commit()

Sequential execution order

a[1] = ...
x = a[100]
a[201] = ...
x = a[101]
a[100] = ...
x = a[102]
a[250] = ...
x = a[103]



Background

Polytope model: Dependence characterization

- Iteration domain constraints: Ensures the statement instance is part of the domain
- Access constraints: Ensures that the statement instances accesses the same memory location
- Order constraints: Ensures the lexicographical ordering of the statements

Background {Dependence representation}

Original user code

```
for(i = 1; i < 10; i++){
  a[i] = a[i - 2] + 1;
}
```

Dependence constraints

```
i  >= 1
i  <= 9
i' >= 1
i' <= 9
i' =  i + 2
i' >= i + 1
```

Background {Polytope model}

Dependence constraints

```
i  >= 1
i  <= 9
i' >= 1
i' <= 9
i' =  i + 2
i' >= i + 1
```

Dependence representation (pluto)

```
 i   i'  1
---------------
 1   0  -1  >= 0
-1   0   9  >= 0
 0   1  -1  >= 0
 0  -1   9  >= 0
 1  -1   2  == 0
-1   1  -1  >= 0
```

Apollo

Apollo consists of two core components

- Static module
- Runtime module

Apollo

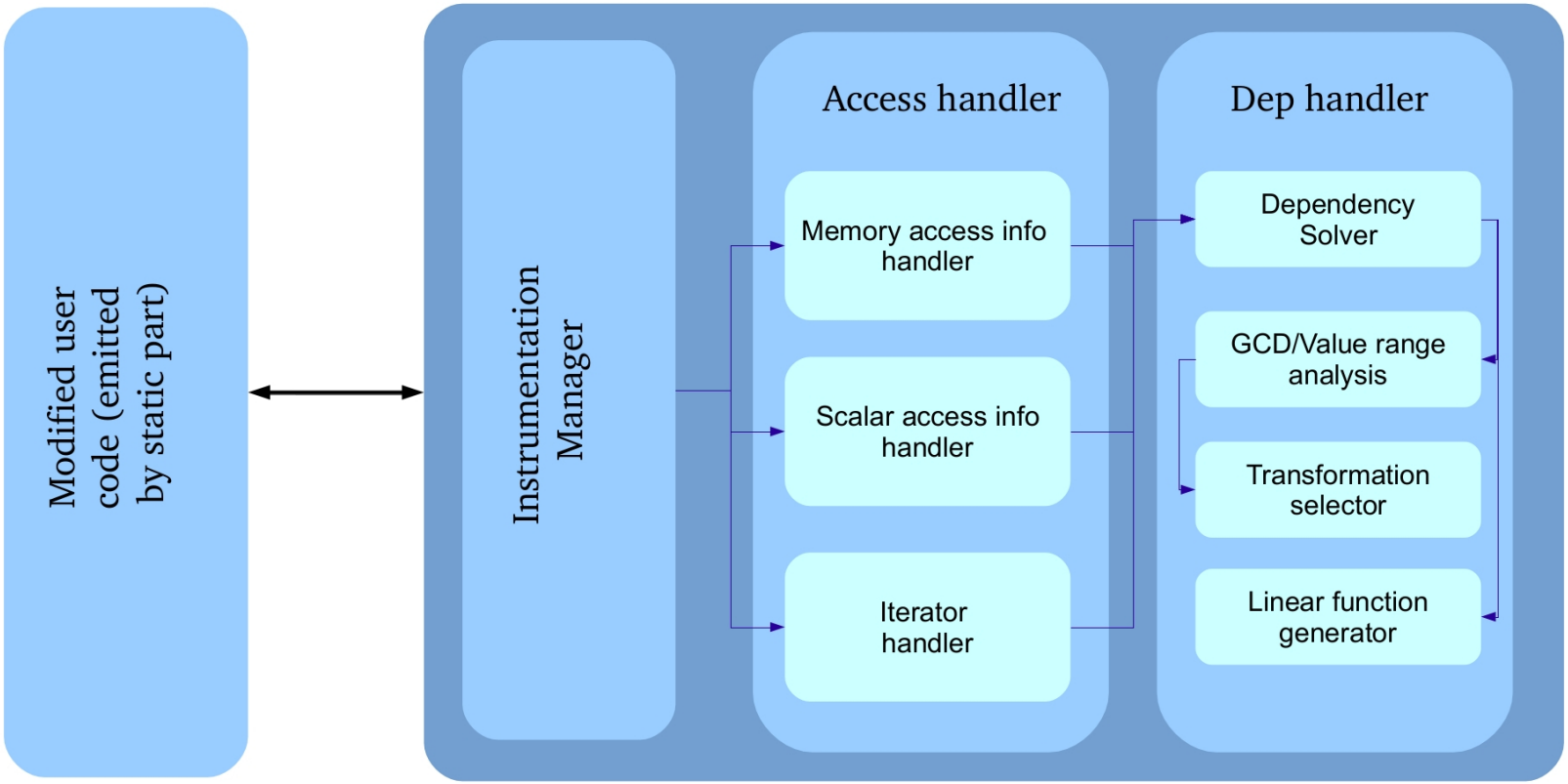


Figure: Apollo global view

Apollo {Static Module}

Static Module

- A set of dedicated llvm passes
- Modifies the control flow of user code
- Adds calls to runtime for information exchange
- Creates code skeletons

Apollo

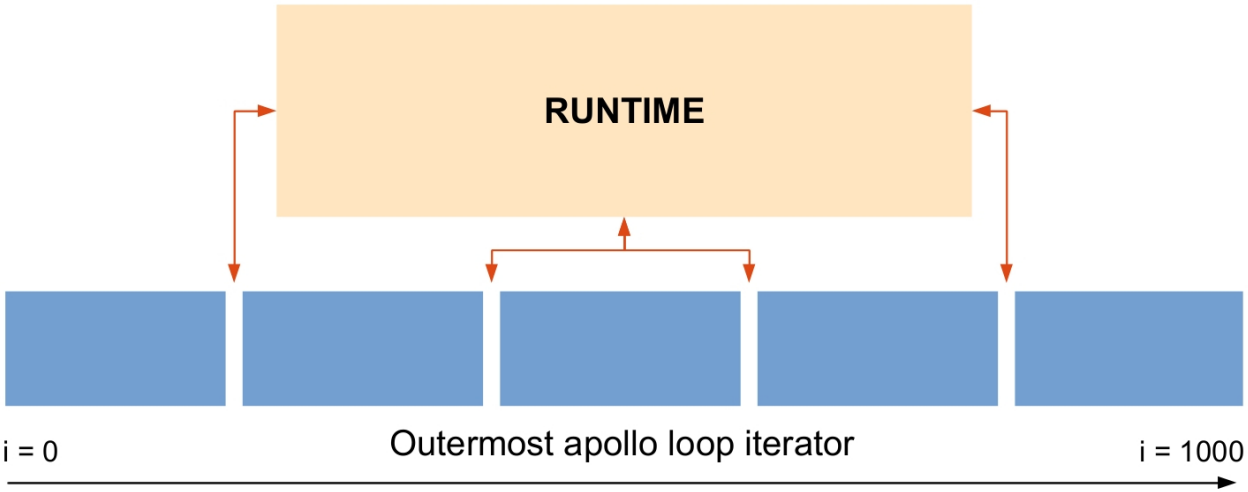


Figure: chunking mechanism

Apollo {Static Module}

Example

- The examples shown is for understanding the concepts
- The examples are shown in C style
- For convenience some information is omitted

Apollo {virtual iterator}

Original user code

```
for(i = 1; i < 1000; i++){
    ptr = start;
    while(ptr){
      a[j] = a[j - 2] *
          ptr->data;
      ptr = ptr->next;
   }
   j = ...
}
```

inserting VI

```
vi = 0; \\VI for i loop
for(i = 1; i < 1000; i++){
  vj = 0; \\VI for j loop
  while(ptr){
    a[j] = a[j - 2] *
        ptr->data;
    vj++;
   }
   j = ...
 vi++;
 }
```

Example {registering access info to runtime}

Original

```
for(i = 1; i < 1000; i++){
    ptr = start;
    while(ptr){
      a[j] = a[j - 2] * ptr->data;
      ptr = ptr->next;
    }
    j = ...
}
```

Example {registering access info to runtime}

Instrumentation skeleton

```
for( vi = chunk_lb; vi < chunk_ub; ++vi){
    ptr = start;
    .
    .
    while(ptr){
      apollo_reg_access(st1);
      a[j] = a[j - 2] * ptr->data; //st1
      apollo_reg_access(st2);
      ptr = ptr->next; //st2
      .
      .
    }
    j = ...
}
```

Partial instrumentation

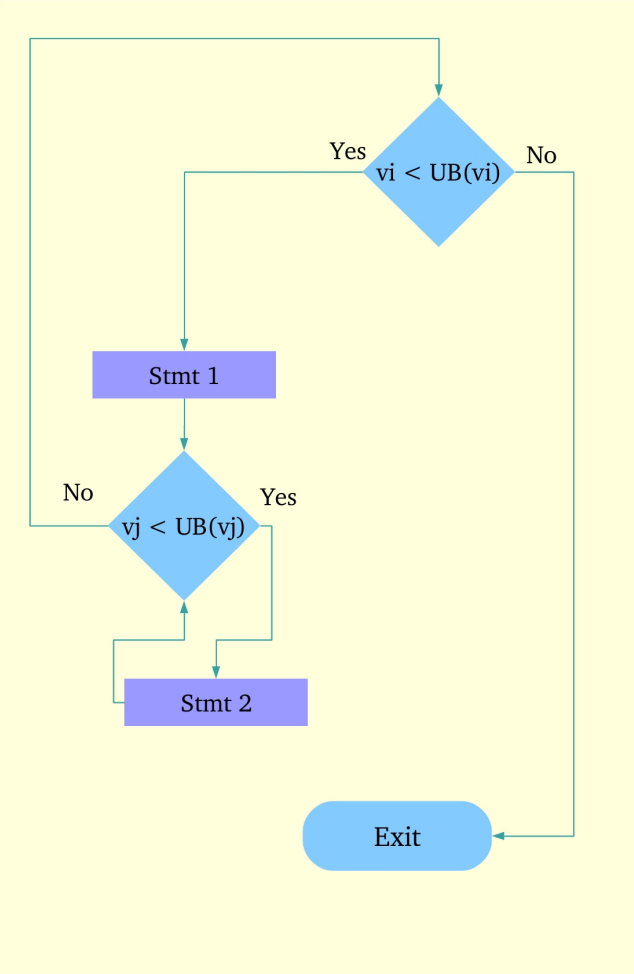


Figure: Original user code

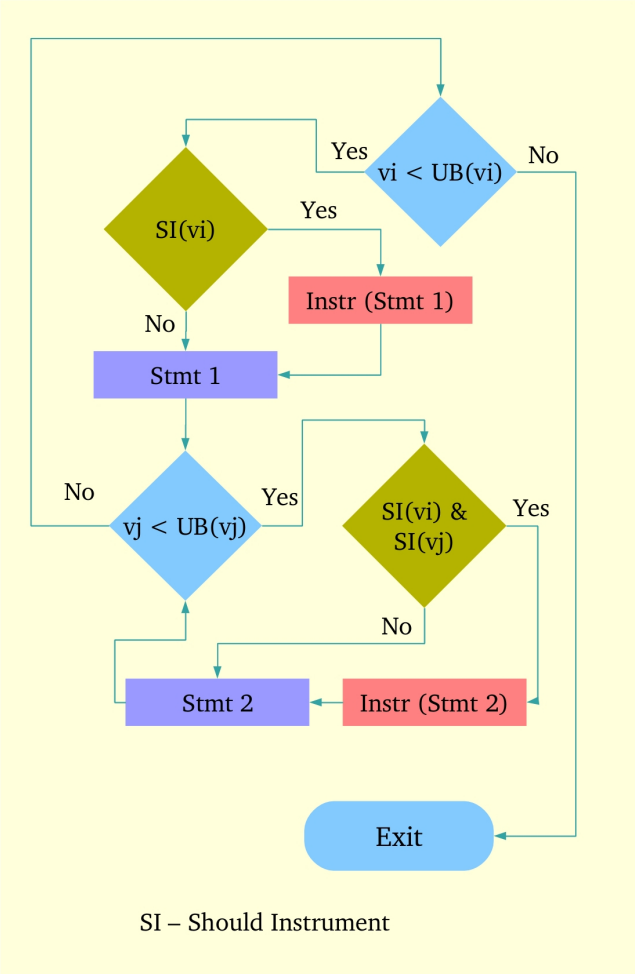


Figure: Modified user code

Apollo {Runtime Module}

Runtime Module

- Collects the required information
- Computes distance vectors
- Interpolates the access function for each memory access
- Interpolates the iterator bounds function
- Controls the behavior of code skeletons

Apollo {Runtime Module}

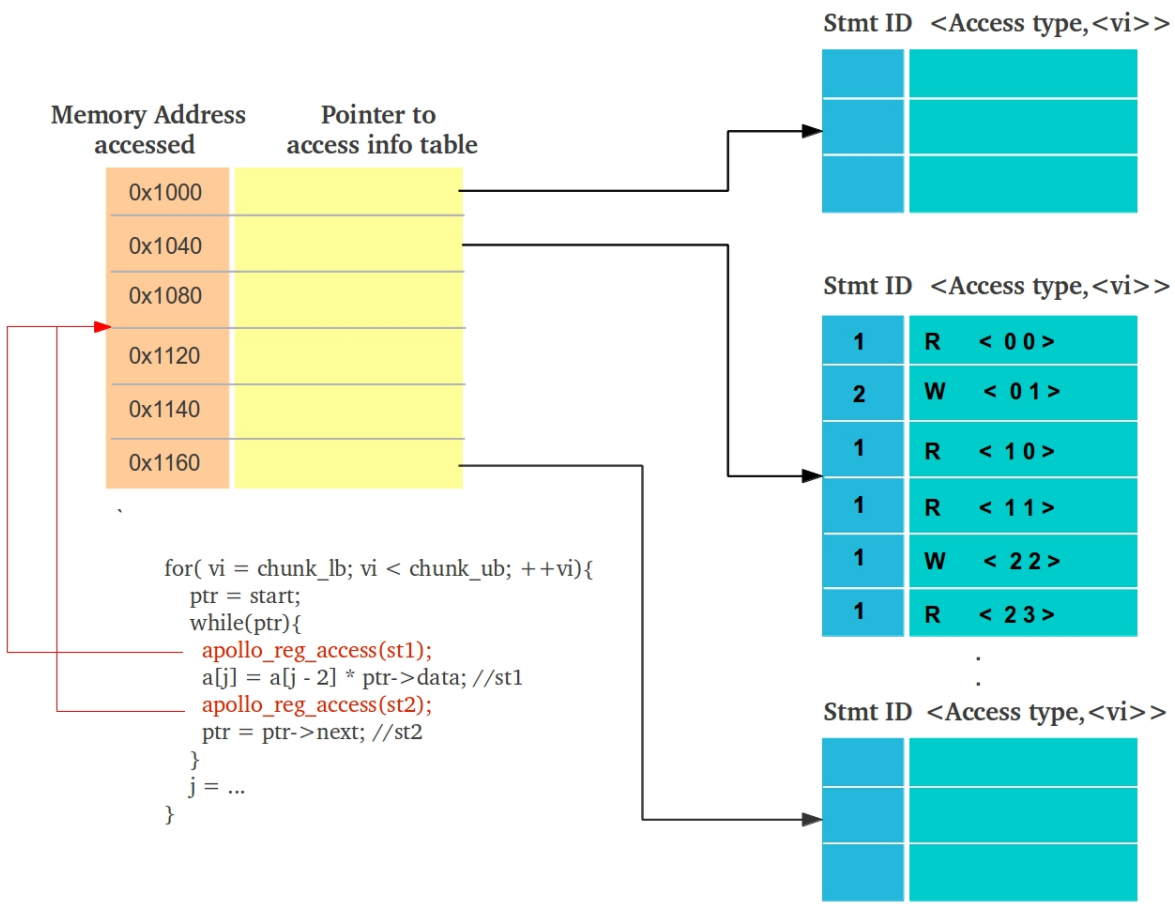


Figure: Computing distance vectors

Apollo {Runtime Module}

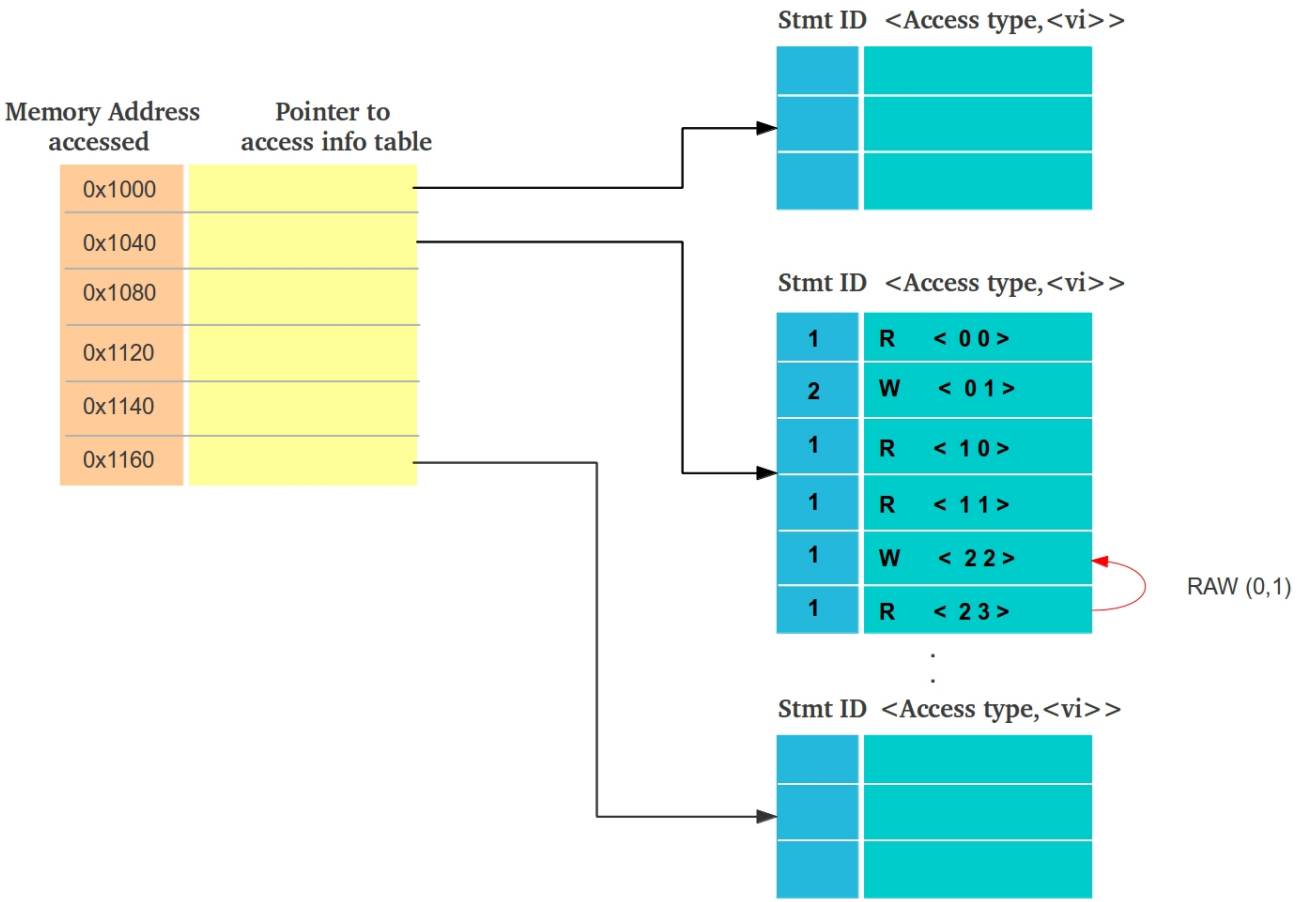


Figure: Computing distance vectors

Apollo {Runtime Module}

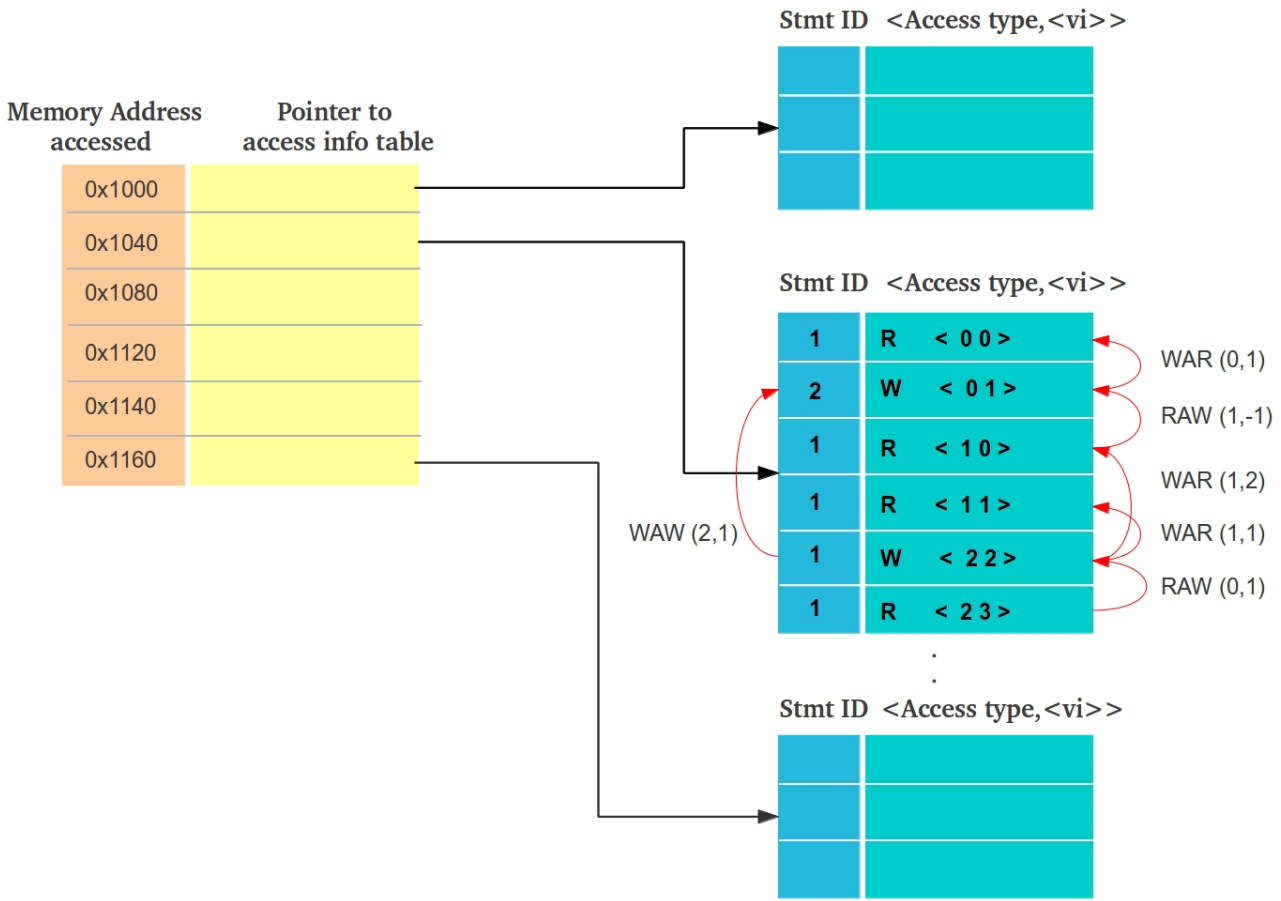


Figure: Computing distance vectors

Apollo {Runtime Module}

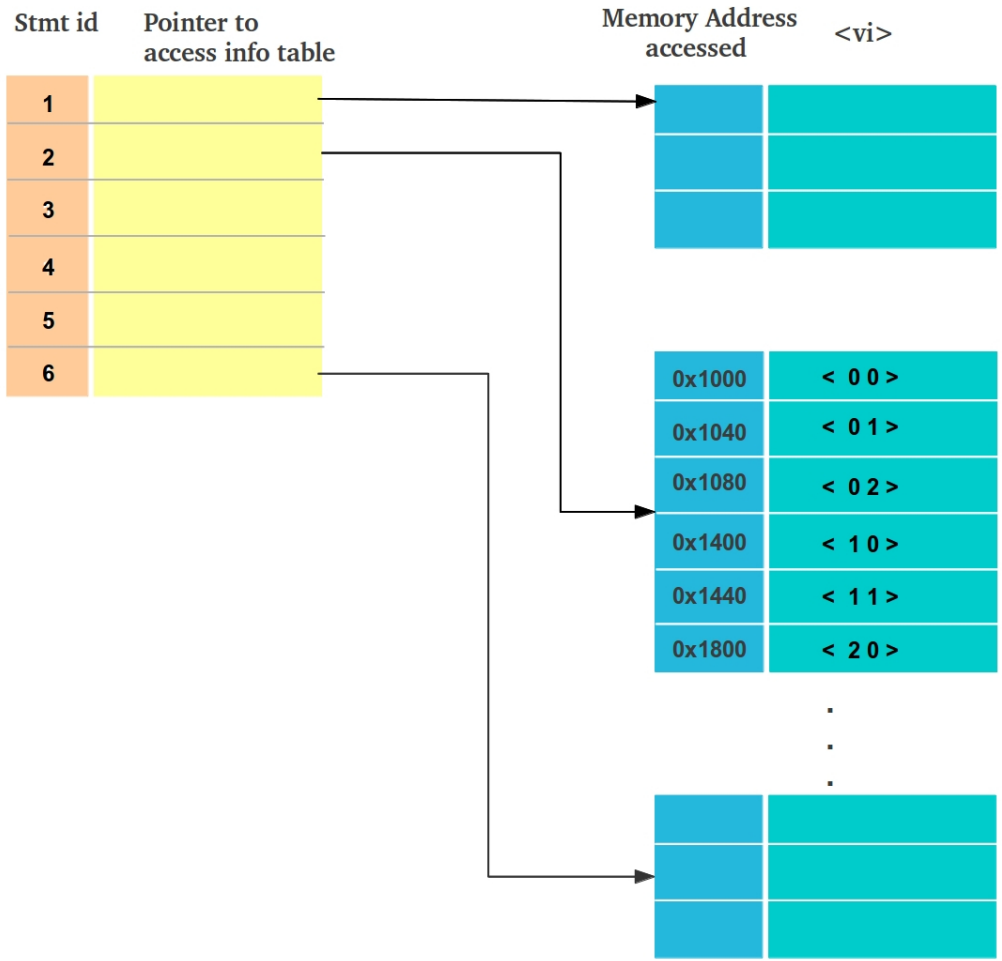


Figure: Computing linear access functions for statement

Apollo {Runtime Module}

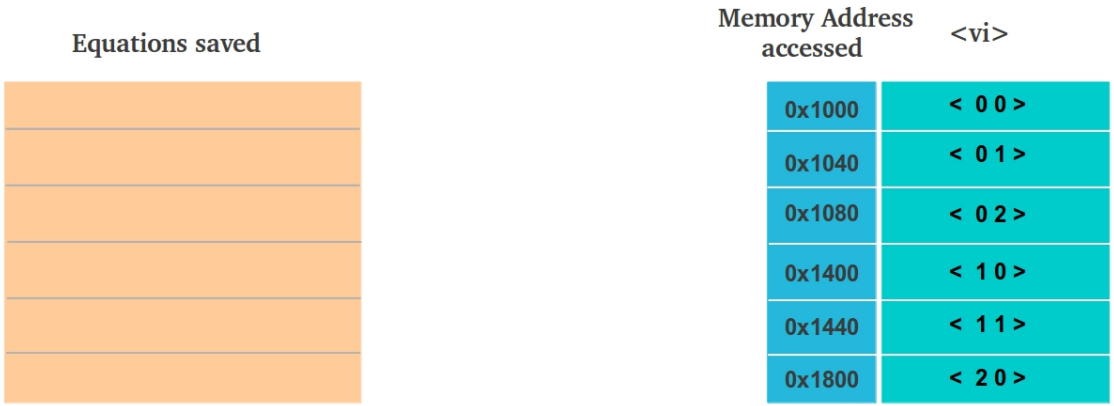


Figure: Computing linear access functions for statement

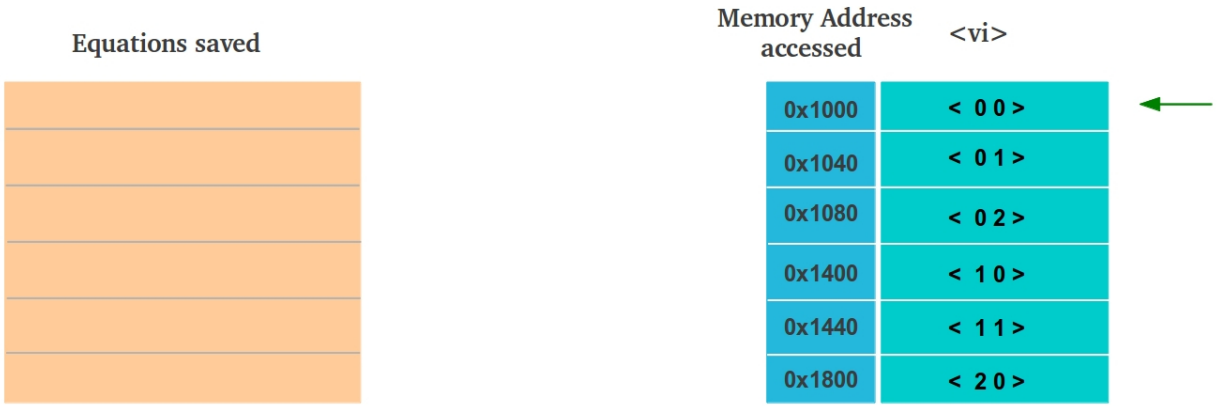


Figure: Computing linear access functions for statement

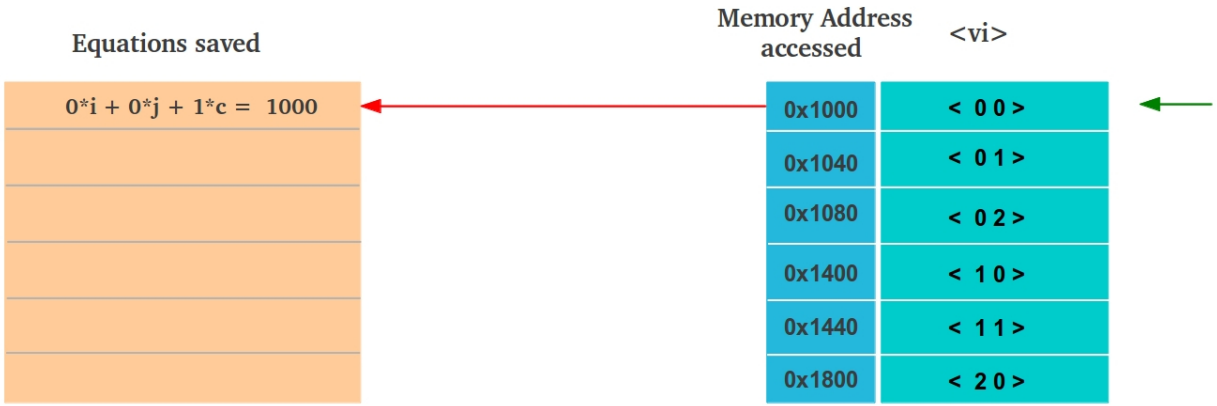


Figure: Computing linear access functions for statement

Apollo {Runtime Module}

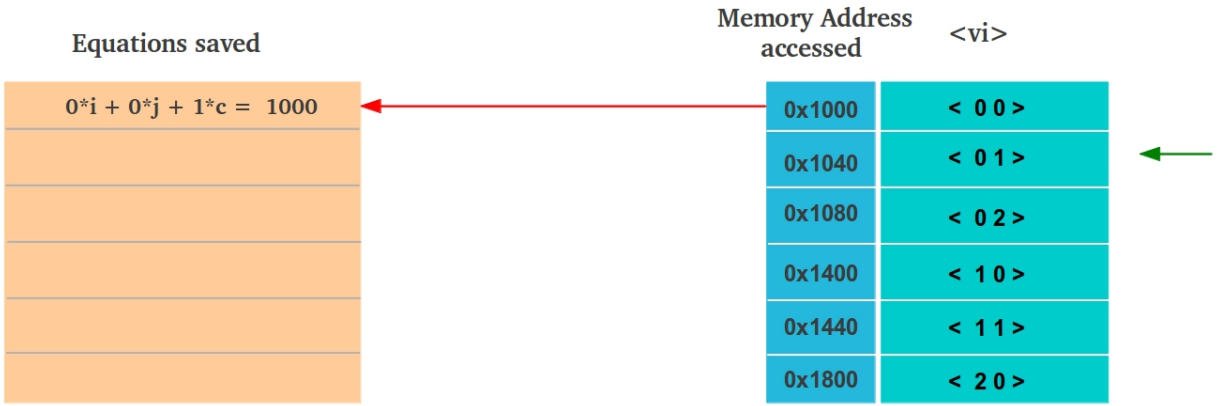


Figure: Computing linear access functions for statement

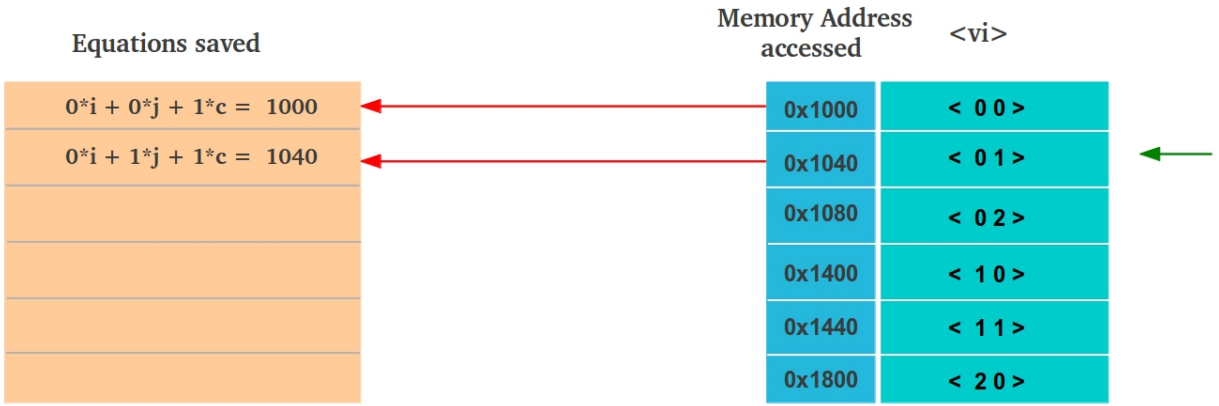


Figure: Computing linear access functions for statement

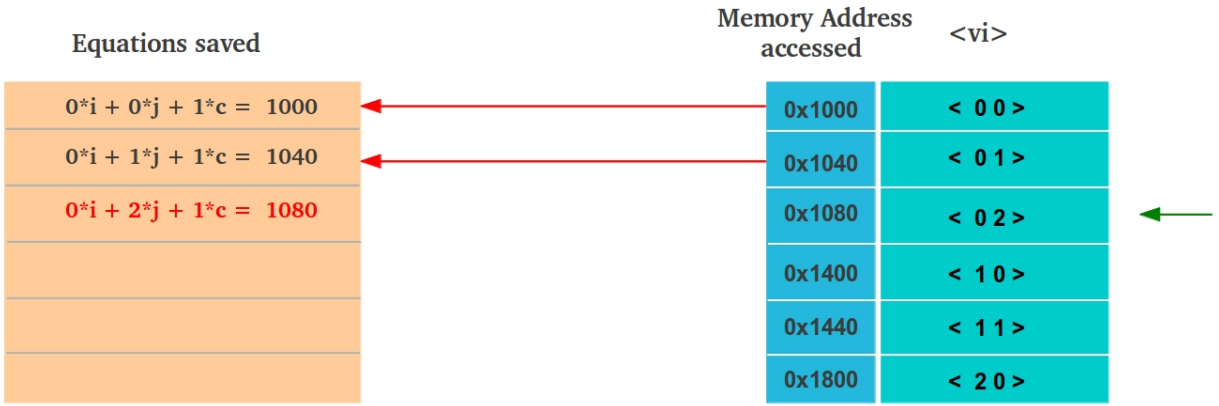


Figure: Computing linear access functions for statement

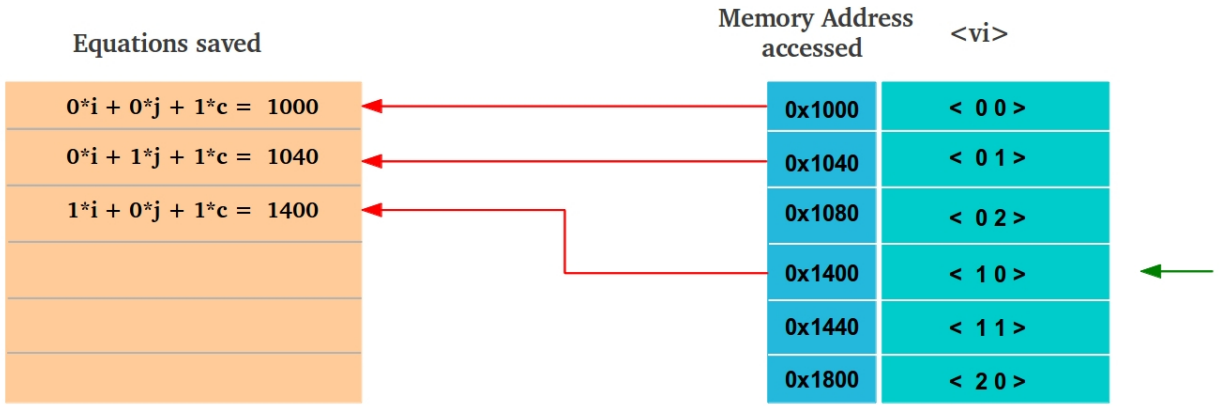


Figure: Computing linear access functions for statement

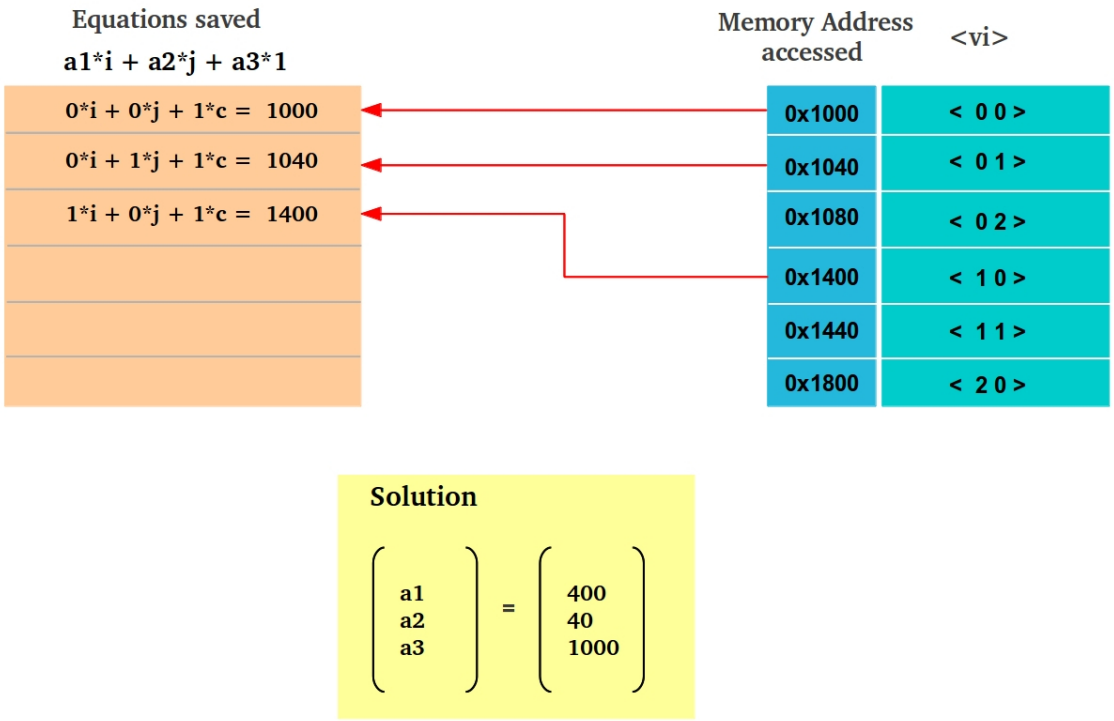


Figure: Computing linear access functions for statement

Apollo {Runtime Module}

Computing transformation

- Runs value range analysis
- Runs GCD
- Computes the transformation matrix
- Selects a code skeleton

Apollo {Runtime Module}

Computing transformation

- Vmad uses a limited set of statically determined transformations
- Apollo uses modified Pluto
- The transformations to be applied are selected at runtime
- Dependence info directly fed to Pluto

Verification Module

- Uses the linear access functions generated
- Each iteration verifies the next sequential iteration
- If verification fails a rollback is triggered
- Only used in patterns which do not follow sequential execution

Original user code

```
for(i = 1; i < 10; i++){
  while(ptr){
    ptr = ptr->next;
    .
    .
    .
  }
}
```

inserting VI

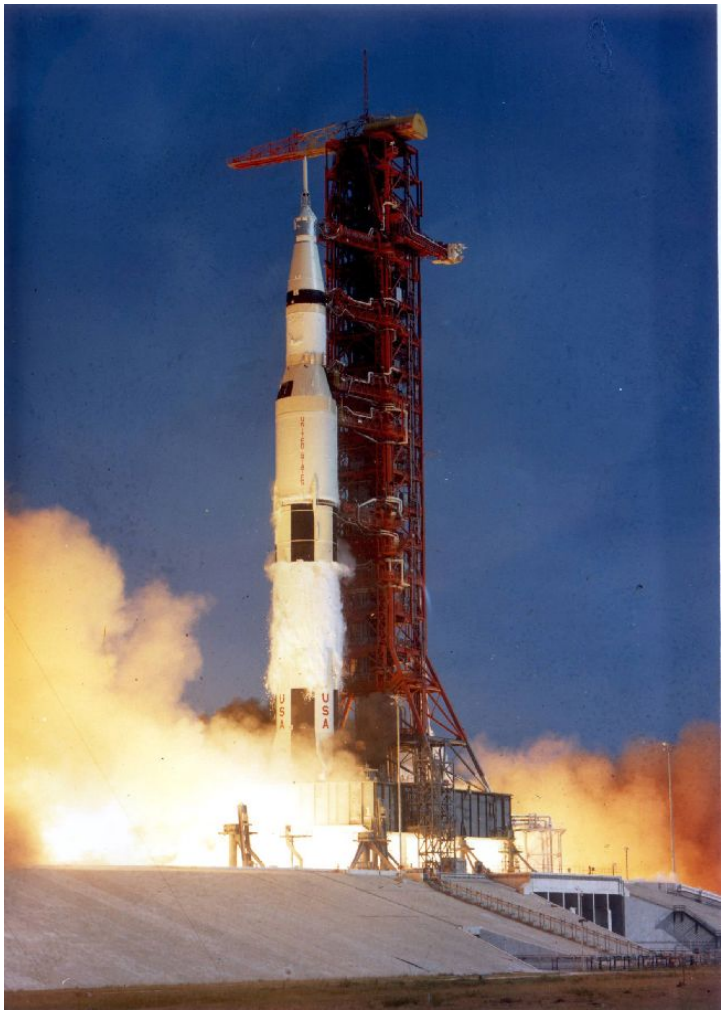
```
a * vi + b * vj + c;
for(vi = 0; vi < 9; vi++){
  vj = 0;
  while(vj < 100){
   ptr = a*vi + b*vj + c;
   ptr = ptr->next;
   vi_next = getnext_VI(vi);
   vj_next = getnext_VI(vj);
   if(ptr != a * vi_next +
      b * vj_next + c)
    rollback();
  }
}
```

Apollo

Why Apollo

- VMAD was a prototype
- Ability to handle more loop classes
- Advanced transformation selection
- Combination of static and dynamic analysis
- Runtime module is more efficient and parallel by design
- Faster resolution of distance vectors and access functions
- No assembly code, No dlsym
- Precise control of target loop
- Highly modularized & extensible
- Open platform

Apollo



Apollo

Apollo {Results}

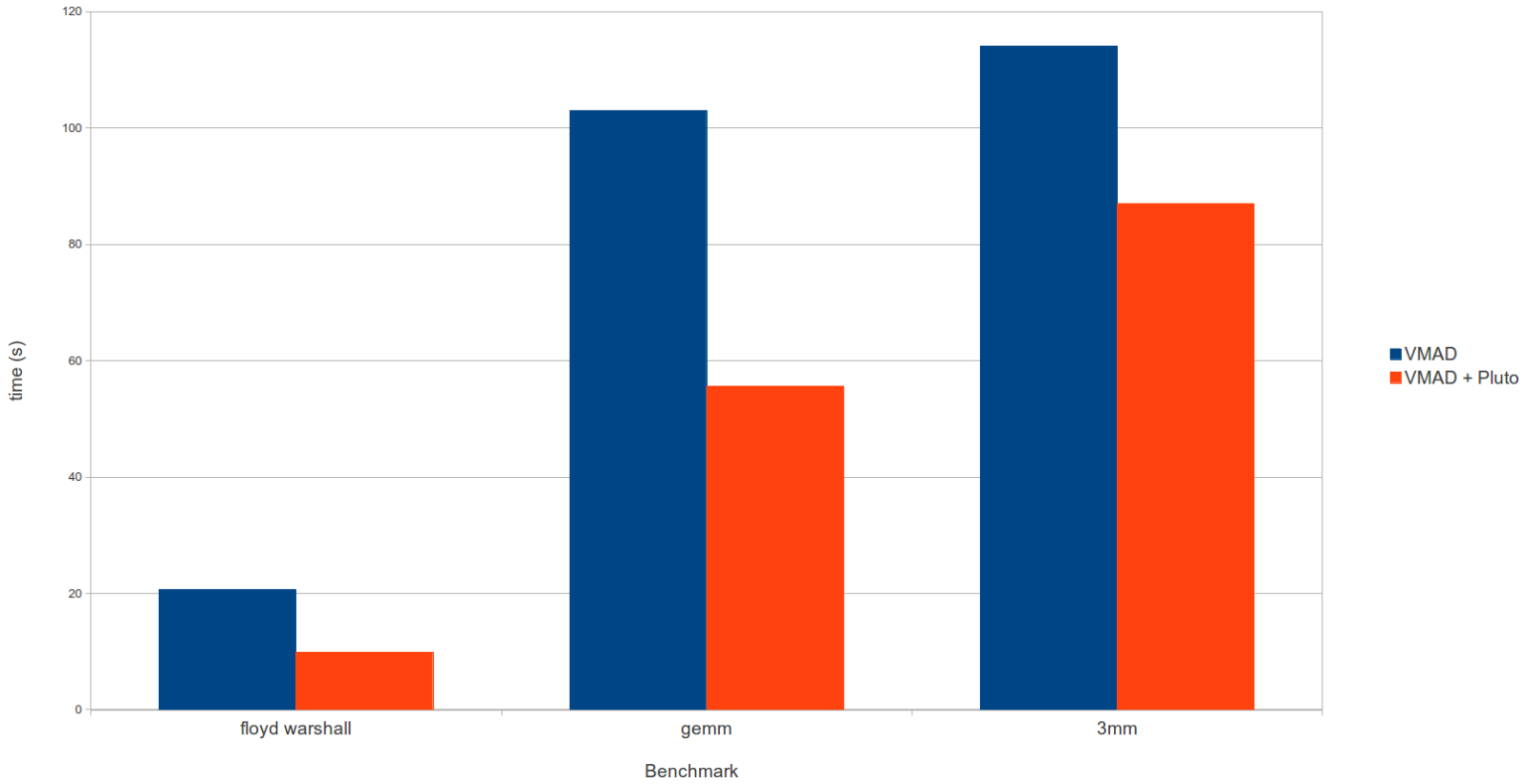


Figure: Speed up of using pluto to determine transformation

Example

User code

```
#pragma apollo_scop
{
    for (i = 0; i < n; i++) {
      ptr = head;
      while (ptr!=NULL) {
          a[i] += ptr->value; \\st1
          ptr = ptr->next; \\st2
      }
    }
}
```

Example

Apollo output

```
iterator bound for vj loop :   < 0 100 > (0*vi + 100)
distance vector for st1    :   < 0 1 >
access function for st1    :   < 4 0 0x7fffb196c364 >
                               (4*vi + 0*vj + 0x7fffb196c364)
scalar function for st2    :   < 0 16 0x2531010 >
                               (0*vi + 16*vj + 0x2531010)
```

Apollo {Results}

overhead

In general the overhead of apollo instrumentation varies from 5% to 25% on instrumentation chuck

Apollo {Non-affine}

Non-affine access

- The dynamic approach of apollo opens the door for more code classes
- Certain classes of Non-affine access can be handled
- Class of codes having monotonic memory access behavior
- Dependence polytope which is a subset of previous phase
- Verification mechanism needs to be found as there is no linear function to validate against

Apollo {Non-affine}

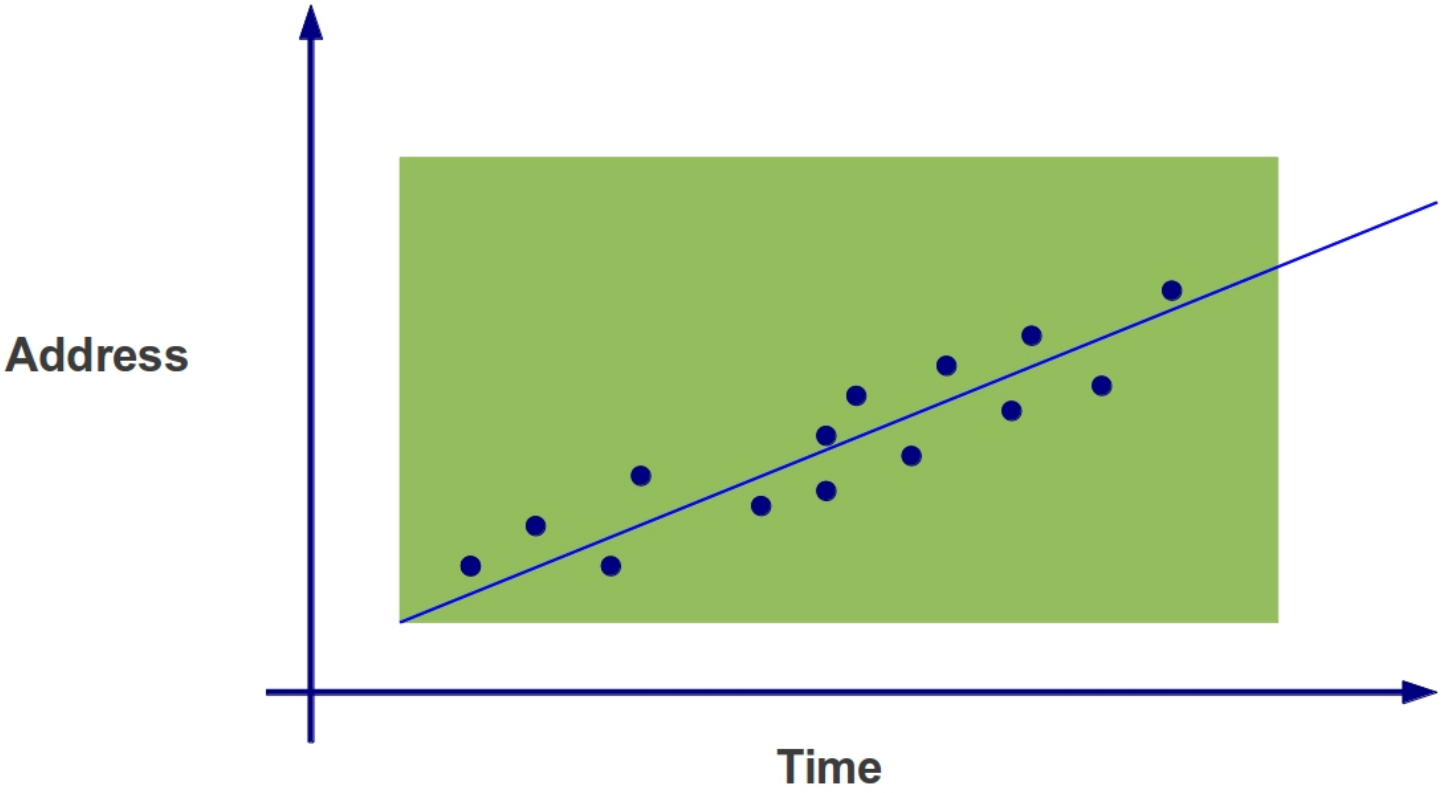


Figure: Handling non affine access

Apollo {Non-affine}

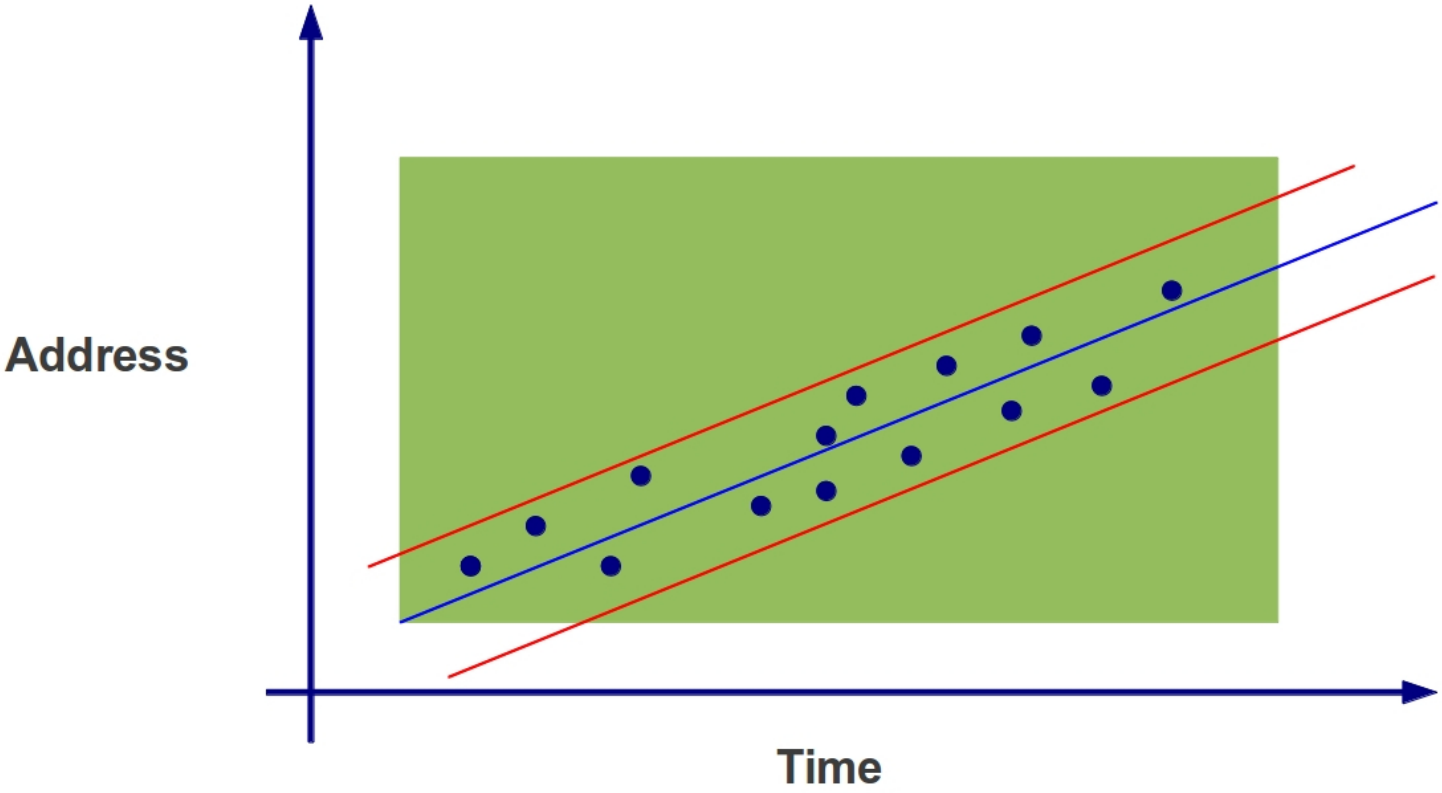


Figure: Handling non affine access

Questions?