Pointer analysis for semi-automatic code parallelizers

Vyavahare, K.S.

Award date:
2014

5T746

Master Thesis *of*
Ketan Sudhakar Vyavahare (0827911)

on

Pointer Analysis for Semi-Automatic Code Parallelizers

May 2014 - Oct 2014

Wednesday 22^{nd} October, 2014

Supervisors Océ : Jaccon Bastiaansen (jaccon.bastiaansen@oce.com)

Supervisor TU/e: Dr. Sander Stuijk (s.stuijk@tue.nl)

Publication Date: Tuesday 28^{th} October, 2014

Contents

Abstract

Code parallelizers are employed these days to reduce the efforts needed in manually parallelizing sequential code. But they are ineffective when it comes to handling programming constructs like pointers. Code parallelizers like Par4all have a limited support for pointers while approaches like the ASET + BONES cannot handle pointers at all. In this thesis we have developed a pointer analysis infrastructure to enable pointer support for the ASET + BONES approach. Our pointer analysis infrastructure is based upon LLVM's compiler infrastructure and relies on an indigenously developed flow insensitive, context sensitive analysis pass called *ex-ptrinfo*. The pass is designed to perform static analysis of source code and extract required pointer analysis information regarding pointer constructs that have been employed for performing memory accesses in PAINt and the Data Path algorithms[1]. Our results show that the developed pointer analysis infrastructure can correctly identify such pointer constructs and extracts the required pointer analysis information which can be used for parallelizing the PAINt and the Data Path algorithms with the ASET + BONES approach. At the moment the ASET + BONES approach is under development and has several limitations. In future we intend to overcome these limitations and apply the ASET + BONES approach for parallelizing real world code.

[1]PAINt and the Data Path algorithms are proprietary algorithms developed at Océ

1 Introduction

The following report is a brief overview of the graduation thesis project jointly carried out at the Eindhoven University of Technology and Océ Technologies.

Océ N.V. is a Netherlands-based company that develops, manufactures and sells printing and copying hardware and its related software. Unlike table top printers these are huge machines and are targeted for commercial printing. An Océ printer realizes its functionality by implementing several algorithms. We will talk about two such algorithms called **PAINt** (see Sec. 1.1) and the **Data Path** (see Sec. 1.2)

Figure 1: Arizona flatbed printer

1.1 PAINt Algorithm

PAINt is a compute intensive data processing algorithm. The algorithm is used for detecting blockages in the nozzles of a print head. If a blocked nozzle is encountered it can identify the nature of the blockage. Such information can be employed for taking corrective action for maintaining and improving print quality. PAINt also increases the life expectancy of a print head making them more profitable.

1.2 Data Path Algorithms

To print any image we have to transform the input image into a form called *printing image*. A *printing image* determines which nozzle of the print head should jet the ink at a given instance in time while printing. The data path is a collection of data intensive image processing algorithms used for deriving the *printing image* from the input image.

Currently Océ employs PC based platforms for running PAINt and the Data Path algorithms. These algorithms have been implemented sequentially for the ease of development, portability and maintainability of code. Although the current implementation meets all existing deadlines, the use of PC based platforms is disadvantageous because of the following reasons.

1. PC based platforms are expensive. So it is not desirable to use them with commercial products.
2. PC based platforms are not guaranteed for long term availability. So maintaining product compatibility becomes an issue.

To overcome these limitations Océ proposes the use of embedded platforms as an alternative to PC based platforms. Such platforms are cheaper in comparison to PC based platforms and are guaranteed for long term availability [1]. But using embedded platforms has its own limitations. They are slower in comparison to PC based platforms and so cannot meet the existing deadlines for PAINt

and the Data Path algorithms.

Océ is working with an **i.MX6Q** based platform. i.MX6Q is an embedded heterogeneous SoC consisting of multi-core CPU and GPUs. It can be programmed parallely and can be exploited for speedup of PAINt and the Data Path algorithms to meet existing deadlines. Océ understands the complexities of parallel programming [20] and wants to explore the feasibility of a semi-automatic code parallelizing tool. Such a tool can reduce the effort otherwise needed in manually parallelizing sequential code. Literature [12], [17], [9], [25] shows that many attempts have been made in the past to develop automatic or semi-automatic code parallelizing tools.

Accordingly a feasibility study [2] was conducted for selecting a semi-automatic code parallelizer suitable for the PAINt and the Data Path algorithms. To choose the right code parallelizing tool for our purpose, we had derived requirements (see Sec. 1.3) from the hardware (i.MX6Q), the software (PAINt and the Data Path algorithms) and the work practices adopted at Océ. These requirements have been listed below.

1.3 Requirements for our code parallelizing tool

1. Implementations that are created semi-automatic as Océ wants to semi-automatically parallelize sequential code. Océ sells small number of printing units each year. Also their algorithms get modified from time to time. As a result it is economically inviable for the company to invest huge man hours in the manual parallelization of sequential code.

2. Implementations which support multi-core, GPU and heterogeneous platforms as we would be generating code for i.MX6Q which is a heterogeneous platform.

3. Semi automatic parallelizers that can generate code in human readable form thereby allowing for custom hand optimizations.

4. Preferably open source implementations as they can be legally modified as per ones requirement.

5. Implementations capable of handling pointers as PAINt and the Data Path algorithms employ pointers.

Based on these requirements we had evaluated (see Tab. 1) implementations like **Par4all**, **ASET + BONES** approach (see Chap. 2) and **Pareon**. We had considered Pareon as it was one of the few commercially sold code parallelization tools available in the market.

Feature	Tool		
	Par4all	ASET + BONES	Pareon
Automation 1	Fully automatic	Semi automatic	Manual
Platform Support 2	Multi-core, GPU and heterogeneous platforms	Multi core, GPU and heterogeneous platforms	Multi core platforms
Code Readability 3	Generates human readable code	Generates human readable code	Output is human readable
Distribution 4	Open Source	Open Source	Licensed (closed source)
Pointer Capability 5	Par4all has a limited support for pointers	ASET + BONES cannot handle pointers	Can handle pointers

Table 1: A comparison of Par4all, ASET + BONES Approach and Pareon

[2] The feasibility study report was submitted earlier for the preparation phase of the thesis

Pareon is a manual code parallelization tool. It lacks support for GPUs and heterogeneous platforms. It is a licensed tool and it's implementation is closed source. Because of all these limitations using Pareon in our case was not a feasible option.

Par4all and the ASET + BONES approach equally fulfill most of our requirements. Both are semi-automatic and open-source implementations. Both are capable of generating human readable code for multi-core, GPU and heterogeneous platforms. But Par4all has a very limited support for pointers where as the ASET + BONES approach cannot handle pointers at all.

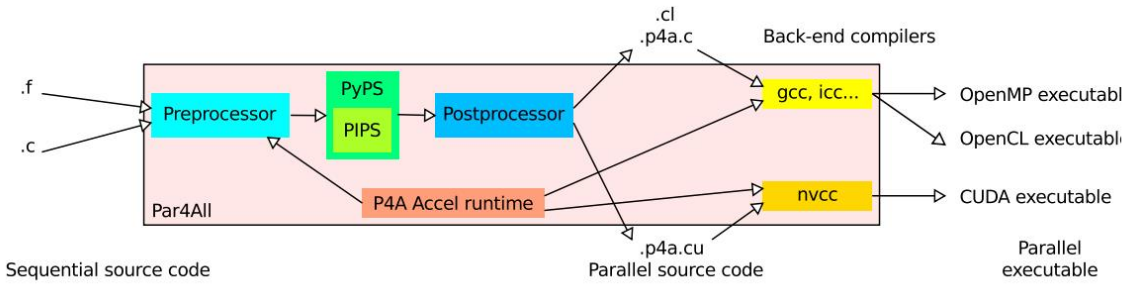


Figure 2: Internal blocks of Par4all[10]

Par4all is a complex implementation composed of multiple blocks (see Fig. 2) (preprocessor, pass manger, optimizer and source to source compiler). Performing any modifications to the tool would require considerable amount of time and effort. So it was unsuitable for quick development. On the other hand the ASET + BONES approach (see Fig. 6) is a much simpler implementation. It is intuitive and can be easily modified in a short amount of time. So it was selected for our work

It was observed that tools like Par4all and the ASET + BONES approach offer limited or no support for pointers due to lack of alias analysis [3] capability. Alias analysis enables us to identify hidden dependencies in a code, which can then be resolved for achieving parallelism.

In this thesis we have developed pointer analysis infrastructure to enable the ASET to handle pointer constructs used for performing memory accesses in PAINt and the Data Path algorithms. Our results show that the developed infrastructure is capable of handling the pointer constructs used for performing memory accesses in PAINt and the Data Path algorithms. We integrated the developed infrastructure into ASET and tested the ASET + BONES approach for parallelizing these algorithm.

The report is organized as follows, Chap. 2 gives the details of the ASET + BONES approach and the reasons behind its inability of handling pointer constructs. Chap. 3 describes the problem statement for our work. Chap. 4 describes the solution proposed as part of the thesis. Chap. 5 discusses the results obtained by applying the proposed solution. Chap. 6 describes the limitations of our proposed solution. Chap. 7 discusses the conclusions that can be drawn from our work and then finally we conclude by Chap. 8 and suggest scope for future work as part of our implementation.

[3] Alias analysis is the technique used to identify if a memory location can be accessed in more than one way at a given instance.

2 The ASET + BONES Approach

The following section describes the ASET + BONES [25] approach in more detail. We study the architecture and the work flow for the approach. We then discuss the reasons behind the lack of pointer handling capabilities in this approach.

ASET + BONES approach is a semi-automatic code parallelization approach developed at Eindhoven University of Technology. It is useful for semi-automatically parallelizing sequential programs written in C language. Before we dive into the details of the approach we would like to explain some basic terminology.

2.1 Terminology

2.1.1 Blocks

We have to delimit code sections with pragmas to indicate potentially parallelizable code fragments. Such delimited code sections are called *blocks*. ASET uses delimiters like *#pragma scop* and *#pragma endscop* to identify a *block*. The example in Fig. 3 is an illustration of a *block*.

```
1 #include<stdio.h>
2 int main()
3 {
4     int i;
5     int A[20];
6     int B[20];
7 #pragma scop
8     for(i=0;i<10;i++)
9     {
10        B[i] = A[i];
11    }
12 #pragma endscop
13    return 0;
14 }
```

Figure 3: for loop is an example of a *block*

2.1.2 Skeletons

Skeletons are highly optimized code fragments used for interpreting specific algorithms in the source code. *Skeletons* are typically hand coded and are maintained in the form of library implementations for individual target-language pairs. For example the version of BONES used as part of our work offers *skeletons* for the following combinations *gpu-cuda*, *cpu-c*, *gpu-opencl-amd*, *cpu-opencl-intel*, *cpu-opencl-amd* and *cpu-openmp*.

2.1.3 Species

Species are annotations applied to a *block* to represent the memory access patterns performed in it. *Species* are based on the **grammar** and the **vocabulary** proposed in *algorithm classification theory* [21]. The theory identifies five access patterns called *element*, *chunk*, *neighbourhood*, *shared* and *full*. Fig. 4 illustrates an annotated block. Fig. 5 shows the *specie* derived by ASET in Fig. 4. Par(10) implies we have 10 parallel iterations possible. The annotation means we can perform 10 parallel element wise iterations between array A and array B.

```
1 #include<stdio.h>
2 int main()
3 {
4     int i;
5     int A[20];
6     int B[20];
7 #pragma scop
8     #pragma species kernel par(10) A[0:9]|element -> B[0:9]|element
9     for(i=0;i<10;i++)
10    {
11        B[i] = A[i];
12    }
13    #pragma species endkernel
14 #pragma endscop
15    return 0;
16 }
```

Figure 4: Example of an annotated block

```
#pragma species kernel par(10) A[0:9]|element -> B[0:9]|element
```

Figure 5: Illustration of a specie

2.2 Architecture

Fig. 6 describes the architecture and the work flow of the ASET + BONES approach. The approach employs two tools viz ASET and BONES. ASET is a code annotation tool and BONES is a source to source compiler.

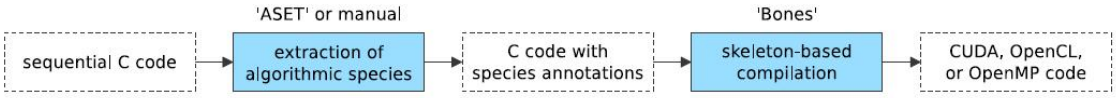


Figure 6: ASET + BONES work flow[25].

Source code to be parallelized is first delimited using the pragmas *#pragma scop* and *#pragma endscop* to identify *blocks* (see Sec. 2.1.1). ASET (code annotation tool) processes such delimited code to produce annotations for identified *blocks* called *species* (see Sec. 2.1.3). BONES (source to source compiler) then interprets such *species* annotated code in term of *skeletons* (see Sec. 2.1.2) to compile parallelized human readable code in CUDA, OpenCL or OpenMP.

2.3 ASET

In this section we give a brief overview of the code annotation tool ASET. ASET employs the *theory of algorithmic species* [14] to derive *species* for the identified *blocks*. Fig. 7 describes the work flow employed in ASET. The details of the work-flow have been documented in [14].

Source code to be annotated (see Fig. 3) is first delimited using the pragmas *#pragma scop* and *#pragma endscop* to identify *blocks*. ASET relies on *pet (polyhedral extraction tool)* [28] to generate (see step a in Fig. 7) per statement polyhedral representation (see Fig. 8a) for the identified *block*.

ASET processes (see step b in Fig. 7) the statement information (see lines 9-24 in Fig. 8a) from the polyhedral representation to build an abstract syntax tree of loops and the statements inside the loop bodies as shown in Fig. 8b. ASET employs the domain information (see line 11) to identify loop bounds. It creates nodes based on the type of operation encountered in the body information

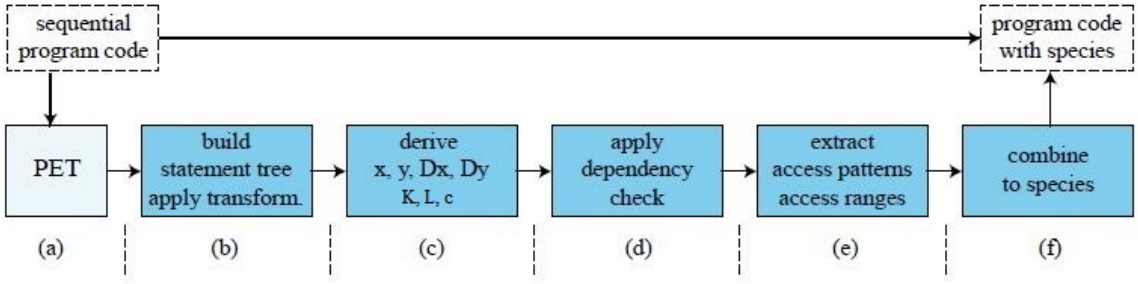
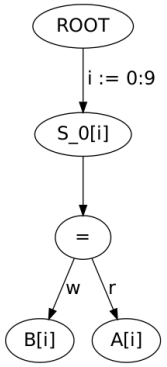


Figure 7: ASET work flow

```
1 context: '{  :  }'
2 arrays:
3 - context: '{  :  }'
4   extent: '{ A[i0] : i0 >= 0 and i0 <= 19 }'
5   element_type: int
6 - context: '{  :  }'
7   extent: '{ B[i0] : i0 >= 0 and i0 <= 19 }'
8   element_type: int
9 statements:
10 - line: 10
11   domain: '{ S_0[i] : i >= 0 and i <= 9 }'
12   schedule: '{ S_0[i] -> [0, i, 0] }'
13   body:
14     type: binary
15     operation: =
16     arguments:
17     - type: access
18       relation: '{ S_0[i] -> B[i] }'
19       read: 0
20       write: 1
21     - type: access
22       relation: '{ S_0[i] -> A[i] }'
23       read: 1
24       write: 0
```

(a) Polyhedral representation



(b) Statement tree built from polyhedral representation in Fig. 8a.

Figure 8: Illustration of polyhedral representation and statement tree for block in Fig. 3

(see lines 13-24). An accesses operation implies a read (see line 21-24) or a write (see line 17-20) operation to a memory location. A binary operation implies an arithmetic (see line 14) operation on the read or written memory locations.

The domain information (see line 11) D_S $\{i \mid 0 \leq i \leq 9\}$ can be represented in homogeneous coordinates as

$$D_S = \mathbf{D_S} \cdot \overrightarrow{t_S} \leq 0$$
$$= \begin{bmatrix} 1 & 0 \\ -1 & 9 \end{bmatrix} \cdot \begin{pmatrix} i \\ 0 \end{pmatrix} \leq 0$$

where $\overrightarrow{t_S}$ is called the iteration vector. Based on the domain information the array access functions for array A and array B can be represented as

$$f_A(\overrightarrow{t_S}) = \mathbf{F_{S,A}} \cdot \overrightarrow{t_S}$$
$$= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ 1 \end{pmatrix} = (i)$$

$$f_B(\overrightarrow{t_S}) = \mathbf{F_{S,B}} \; . \; \overrightarrow{t_S}$$

$$= \begin{bmatrix} 1 \\ 0 \end{bmatrix} . \begin{pmatrix} i \\ 1 \end{pmatrix} = (i)$$

The array access function generated by *pet* represents the access behavior of an array in isolation. It does not consider the other array accesses in the *block*. So it is unsuitable for deriving *species* as the access patters for two distinct arrays cannot be correlated

To overcome this problem ASET identifies *structure* loops and *base* loops in a *block*. A *structure* loop can be formally defined as a loop containing either: (1) a *read* access that is dependent on the loop iterator while all *write* accesses are not, or (2), a *write* access that is dependent on the loop iterator while all *read* accesses are not. All other loops which contain at least an array access are considered *base* loops.

ASET employs the algorithm (see Fig. 10a) described in [24] to identify *structure* and *base* loops in a *block*. ASET creates the modified vector $\overrightarrow{t'_S}$ by eliminating the constants in $\overrightarrow{t_S}$

$$\overrightarrow{t'_S} = \begin{pmatrix} i \end{pmatrix}$$

Accordingly the access functions for array A and array B are updated as $\mathbf{F'_{S,A}}$ and $\mathbf{F'_{S,B}}$.

$$\mathbf{F'_{S,A}} = \begin{bmatrix} 1 \end{bmatrix}$$

$$\mathbf{F'_{S,B}} = \begin{bmatrix} 1 \end{bmatrix}$$

ASET then constructs vectors $\overrightarrow{R}_\mathbf{S}$ and $\overrightarrow{W}_\mathbf{S}$ by summing all values per column (projection) of all $\mathbf{F'_{S,}}_a$ matrices where a is a *read* or a *write* access.

$$\overrightarrow{R}_\mathbf{S} = \begin{bmatrix} 1 \end{bmatrix}$$

$$\overrightarrow{W}_\mathbf{S} = \begin{bmatrix} 1 \end{bmatrix}$$

Based on the values of $\overrightarrow{R}_\mathbf{S}$ and $\overrightarrow{W}_\mathbf{S}$ ASET identifies the loop in Fig. 3 with iterator i as a *base* loop. The corresponding iteration vector and domain for the base loop are represented as $\overrightarrow{x}$ and $D_{x,S}$

$$D_{x,S} = \mathbf{D_{x,S}} \; . \; \overrightarrow{x} \leq 0$$

$$= \begin{bmatrix} 1 & 0 \\ -1 & 9 \end{bmatrix} . \begin{pmatrix} i \\ 1 \end{pmatrix} \leq 0$$

As no structure loops were determined the iteration vector and domain for the structure loop are zero.

$$\overrightarrow{y} = \begin{pmatrix} 0 \end{pmatrix}$$

$$D_{y,S} = \begin{bmatrix} 0 \end{bmatrix}$$

[24] introduces two new matrices K and L. K gives the relation between the array indices and the base loop iterator $\overrightarrow{x}$, while L gives the relation between the array indices and the structure loop iterator $\overrightarrow{y}$. Additionally, they introduce $\overrightarrow{c}$ to set a constant offset. It was identified that the value for $\mathbf{K_a} = [1]$, $\mathbf{L_a} = [0]$ and $\overrightarrow{c} = [0]$. Together K, L and $\overrightarrow{c}$ are used for deriving the new access function (I_a) for array access a, described as

$$I_a = \mathbf{K_a}.\overrightarrow{x} + \mathbf{L_a}.\overrightarrow{y} + \overrightarrow{c}$$

Till this point ASET has identified $\overrightarrow{x}$, $\overrightarrow{y}$, $D_{x,S}$, $D_{y,S}$, $\mathbf{K_a}$, $\mathbf{L_a}$ and $\overrightarrow{c}$ (see step c in Fig. 7).

Accesses to the same array inside a loop is checked for dependencies. GCD-test [18] and the Banerjee-test [18] are applied to compare addresses of *reads* and *writes*. The GCD-test searches for integer solutions, but does not consider loop bounds. On the contrary, the Banerjee-test does consider loops bounds and searches for non-integer solutions as well. Within ASET, both tests are combined for checking dependencies (see step d in Fig. 7). If dependencies are found then no *species* are produced for the corresponding *block*.

For the current example in Fig. 3 the array accesses are independent. ASET employs the algorithm (see Fig. 10b) described in [24] to identify *species* from the values of matrices $\mathbf{K_a}$ and $\mathbf{L_a}$ (see step d and e in Fig. 7). Based on the values of $\mathbf{K_a}$ and $\mathbf{L_a}$ the algorithm derives a *specie* with access pattern of *element* for the example in Fig. 3

2.4 BONES

In this section we give a brief overview of the source to source compiler BONES. Fig. 9 describes the working of BONES. The details of the compiler have been documented in [23].

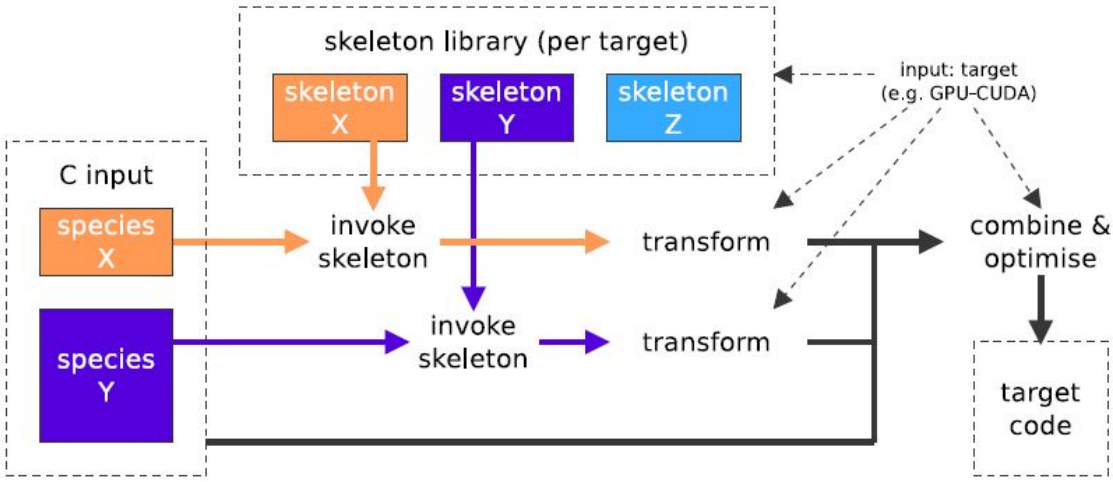


Figure 9: Working of BONES.

Bones [23] is based on the *skeleton based compilation theory* [22]. Skeleton based compilation is a source to source translation technique which relies on the interpretation of the source program in terms of optimized *skeleton* code. It transforms *species* annotated code to OpenMP, CUDA or OpenCL code with the help of a *skeleton* library.

ALGORITHM : Algorithm to deriving base loops and structure loops for a loop body.

Input: Polyhedral description of all statements $\mathcal{S}$ in the loop body
Output: Base loop vector $\vec{x}$ and structure loop vector $\vec{y}$
$\vec{x} = \emptyset; \vec{y} = \emptyset$
for *each statement* S *in* $\mathcal{S}$ **do**
  $\vec{t'}_S \leftarrow \vec{t}_S$ without constants
  $n = length(\vec{t'}_S)$
  for *all accesses* a *in* S **do**
    $\mathbf{F'}_{S,a} \leftarrow$ the first n columns of $\mathbf{F}_{S,a}$
  end
  $\vec{R}_S \leftarrow$ projection of $\mathbf{F'}_{S,a}$ into a row vector, for which a are read accesses
  $\vec{W}_S \leftarrow$ projection of $\mathbf{F'}_{S,a}$ into a row vector, for which a are write accesses
  if $\vec{R}_S = \vec{0}$ *or* $\vec{W}_S = \vec{0}$ **then**
    continue;
  end
  for i *in* $[0 : n-1]$ **do**
    if $(R_{S,i} \neq 0$ *and* $W_{S,i} = 0)$ *or* $(R_{S,i} = 0$ *and* $W_{S,i} \neq 0)$ **then**
      $\vec{y} \leftarrow \vec{y} \bigcup \vec{t}_{S,i}$
    end
  end
end
$\vec{x} \leftarrow \vec{t}_S \bigcap \vec{y}$
Result: $\vec{x}, \vec{y}$

(a) Algorithm for deriving *structure* and *base* loops in a *block*.

ALGORITHM : Algorithm to derive the access patterns for an array.

Input: Access description matrices ($\mathbf{K}_a$,$\mathbf{L}_a$) and domain descriptions ($\mathcal{D}_{x,S}$, $\mathcal{D}_{y,S}$) for array access a in statement S
Output: Access pattern $\mathcal{P}_a$ for array access a
if $(\mathbf{L}_a = 0)$ **then**
  if $(\mathbf{K}_a = 0)$ **then**
    $\mathcal{P}_a \leftarrow$ "*shared*"
  else
    $\mathcal{P}_a \leftarrow$ "*element*"
  end
else
  if $(\mathbf{K}_a = 0)$ **then**
    $\mathcal{P}_a \leftarrow$ "*full*"
  else if *(partial overlap)* **then**
    $\mathcal{P}_a \leftarrow$ "*neighbourhood*"
  else
    $\mathcal{P}_a \leftarrow$ "*chunk*"
  end
end
Result: $\mathcal{P}_a$

(b) Algorithm for deriving *species* from the values of matrices $\mathbf{K}_a$ and $\mathbf{L}_a$.

Figure 10: Algorithms employed for deriving *species*

2.5 Limitations of ASET

In the previous sections we have talked about the ASET + BONES approach. One of the limitation of the approach is its lack of support for pointers. In the following section we analyze this limitation and investigate the reason behind it.

2.5.1 Lack of pointer handling capability in ASET

For the ASET + BONES approach to work it is a strict requirement that the memory accesses are performed using arrays. The approach fails if memory accesses are performed using pointers.

If we consider the examples in Fig. 11 the code snippet in Fig. 11a involves memory accesses on line 10. These memory accesses are performed using arrays A[20] and B[20]. As memory accesses are performed using arrays ASET can successfully produce annotations (line 8) as shown in Fig. 11b.

On the other hand the code snippet shown in Fig. 11c involves memory accesses on line 13. These memory accesses are performed using pointers A and B. ASET is incapable of handling memory accesses using pointer constructs. As a result it crashes and fails to produce any annotations.

```
1 #include<stdio.h>
2 int main()
3 {
4     int i;
5     int A[20];
6     int B[20];
7 #pragma scop
8     for(i=0;i<10;i++)
9     {
10        B[i] = A[i];
11    }
12 #pragma endscop
13    return 0;
14 }
```

(a) Arrays for memory access

```
1 #include<stdio.h>
2 int main()
3 {
4     int i;
5     int A[20];
6     int B[20];
7 #pragma scop
8     #pragma species kernel par(10) A[0:9]|element -> B[0:9]|element
9     for(i=0;i<10;i++)
10    {
11        B[i] = A[i];
12    }
13    #pragma species endkernel
14 #pragma endscop
15    return 0;
16 }
```

(b) Example of an annotated *block*

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 int main()
4 {
5     int i;
6     int *A;
7     int *B;
8     A=(int*)malloc(20*sizeof(int));
9     B=(int*)malloc(20*sizeof(int));
10 #pragma scop
11    for(i=0;i<10;i++)
12    {
13        B[i] = A[i];
14    }
15 #pragma endscop
16    free(A);
17    free(B);
18    return 0;
19 }
```

(c) Pointers for memory access - No annotations are produced by ASET

Figure 11: Arrays vs. pointers for memory access

It was found that *pet (polyhedral extraction tool)* used for extracting *polyhedral representation* (see step a in Fig. 7) had certain limitations (see Sec. 2.6). These limitations prevented ASET from handling pointer constructs when used for performing memory accesses.

2.6 Limitations in *pet (polyhedral extraction tool)*

In this section we talk about the limitations of *pet* which prevent ASET from handling pointer constructs when used for performing memory accesses.

Before we discuss the limitations we would like to introduce some terminology.

1. **Range**: For a contiguous memory allocation the term *range* implies the number of bytes that have been allocated for the chunk of memory.
2. **Extent**: For a contiguous memory allocation of a given type the term *extent* implies a ratio of range of the memory allocation to the size of the type of the allocation.

If I[10] is an array of 10 integers and assuming that integers are 4 bytes wide we would say, *range* for I is 40 while the *extent* is 40/4 which is 10. Also for I the lower bound of the extent would be 0 while the upper bound of the extent would be 10.

Similarly if I is an integer type of pointer and holds the address of a dynamically allocated memory chunk (*I=(int*)malloc(10*sizeof(int))*) we would say, *range* for I is 40 while the *extent* is 40/4 which is 10. Also for I the lower bound of the extent would be 0 while the upper bound of the extent would be 10.

2.6.1 Limitation 1: cannot determine the upper bound of the extent for pointers

When arrays are used for performing memory accesses *pet* can correctly determines the value of the upper bound of the extent. But when pointers are used for performing memory accesses the value of the upper bound of the extent is unknown. ASET requires the value of the upper bound of the extent for deriving *species*.

```
1 #include<stdio.h>
2 int main()
3 {
4     int i;
5     int A[20];
6     int B[20];
7 #pragma scop
8     for(i=0;i<10;i++)
9     {
10        B[i] = A[i];
11    }
12 #pragma endscop
13    return 0;
14 }
```

(a) Arrays for memory access

```
1 context: '{  :  }'
2 arrays:
3 - context: '{  :  }'
4   extent: '{ A[i0] : i0 >= 0 and i0 <= 19 }'
5   element_type: int
6 - context: '{  :  }'
7   extent: '{ B[i0] : i0 >= 0 and i0 <= 19 }'
8   element_type: int
9 statements:
10 - line: 10
11   domain: '{ S_0[i] : i >= 0 and i <= 9 }'
12   schedule: '{ S_0[i] -> [0, i, 0] }'
13   body:
14     type: binary
15     operation: =
16     arguments:
17     - type: access
18       relation: '{ S_0[i] -> B[i] }'
19       read: 0
20       write: 1
21     - type: access
22       relation: '{ S_0[i] -> A[i] }'
23       read: 1
24       write: 0
```

(b) Polyhedral representation for arrays

Figure 12: Polyhedral representation for array based memory accesses

Fig. 12 shows the polyhedral representation for array based memory accesses. The code snippet in Fig. 12a involves memory accesses at line 10 using arrays A[20] and B[20]. *Pet* can identify the value of the lower bound of the extent as 0 and can also correctly identify the value of the upper bound of the extent as 20 for arrays A[20] and B[20]. In Fig. 12b we can see that *pet* generates suitable entries at line 4 as i0 >= 0 and i0 <= 19 for array A[20] and at line 7 as i0 >= 0 and i0 <= 19 for array B[20] based on the values of their lower and upper bound of the extent respectively.

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;
    int *A;
    int *B;
    A=(int*)malloc(20*sizeof(int));
    B=(int*)malloc(20*sizeof(int));
#pragma scop
    for(i=0;i<10;i++)
    {
        B[i] = A[i];
    }
#pragma endscop
    free(A);
    free(B);
    return 0;
}
```

(a) Pointers for memory access

```
context: '{  :  }'
arrays:
- context: '{  :  }'
  extent: '{ A[i0] : i0 >= 0 }'
  element_type: int
- context: '{  :  }'
  extent: '{ B[i0] : i0 >= 0 }'
  element_type: int
statements:
- line: 13
  domain: '{ S_0[i] : i >= 0 and i <= 9 }'
  schedule: '{ S_0[i] -> [0, i, 0] }'
  body:
    type: binary
    operation: =
    arguments:
    - type: access
      relation: '{ S_0[i] -> B[i] }'
      read: 0
      write: 1
    - type: access
      relation: '{ S_0[i] -> A[i] }'
      read: 1
      write: 0
```

(b) Polyhedral representation for pointers

Figure 13: Polyhedral representation for pointer based memory accesses

Fig. 13 shows the polyhedral representation for pointer based memory accesses. The code snippet in Fig. 13a involves memory accesses at line 13 using pointers A and B. *Pet* can only identify the value of the lower bound of the extent as 0 but cannot identify the value of the upper bound of the extent for pointers A and B. In Fig. 13b we can see that *pet* generates entries at line 4 as i0 >= 0 for pointer A and at line 7 as i0 >= 0 for pointer B based on the values of their lower bound of the extent. But no entries are produced for the values corresponding to their upper bound of the extent. As a result the entries for *extent* information at line 4 and at line 7 in polyhedral representation are incomplete when used for deriving *species*. Because of this limitation ASET fails to derive *species* when pointers are used for performing memory accesses.

2.6.2 Limitation 2: *Pet* cannot identify pointer aliases

ASET checks for dependencies in a *block* (see step d in Fig. 7) while deriving *species*. *Species* are produced if the iterations in a *block* are found to be independent. The dependency check can yield false positives if aliasing pointers are used for performing memory accesses.

The example in Fig. 14a employs arrays for performing memory accesses. ASET can correctly annotate the *block* as there is no dependency on line 10. The annotated code is shown in Fig. 14b. If ASET could handle pointers the example in Fig. 14c would yield a false positive. In the absence of alias analysis capability ASET would have failed to identify the alias created on line 10 which lies outside the delimitations of the *block*. As a result the iteration on line 14 would be tested as independent and would get annotated. Because of this limitation ASET cannot perform correct dependency analysis and hence fails to derive *species*.

2.6.3 Limitation 3: *Pet-0.1* has support for limited code constructs

It was observed that the version of *pet* (version 0.1) originally used with ASET allowed only a small set of code constructs to be used in a *block*. As a result any unsupported code construct would prevent the identification of a potential *block*. For example constructs like shift operators were not supported by *pet-0.1*. We have overcome this limitation by substituting *pet-0.1* by *pet-0.5*. *Pet-0.5* is capable of handling most of the code constructs of C language.

```c
#include<stdio.h>
int main()
{
    int i;
    int A[20];
    int B[20];
#pragma scop
    for(i=0;i<10;i++)
    {
        B[i] = A[i];
    }
#pragma endscop
    return 0;
}
```

(a) Arrays for memory access

```c
#include<stdio.h>
int main()
{
    int i;
    int A[20];
    int B[20];
#pragma scop
    #pragma species kernel par(10) A[0:9]|element -> B[0:9]|element
    for(i=0;i<10;i++)
    {
        B[i] = A[i];
    }
    #pragma species endkernel
#pragma endscop
    return 0;
}
```

(b) Example of an annotated *block*

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;
    int *A;
    int *B;
    A=(int*)malloc(20*sizeof(int));
    B=(int*)malloc(20*sizeof(int));
    B=A;
#pragma scop
    for(i=0;i<10;i++)
    {
        B[i] = A[10-i];
    }
#pragma endscop
    free(A);
    free(B);
    return 0;
}
```

(c) Dependency in pointers

Figure 14: Significance of alias analysis

3 Problem statement

In the earlier chapter we have discussed the reasons behind ASET's limitation of not handling pointers. The limitation arises because of the specific deficiencies (see Sec. 2.6.1, 2.6.2) of *pet (polyhedral extraction tool)*. These deficiencies prevent ASET from handling pointer constructs.

In this thesis we develop a pointer analysis infrastructure for ASET. The infrastructure enables ASET to handle pointer constructs used for performing memory accesses in the PAINt and Data algorithms. To realize our pointer analysis infrastructure we must achieve the following goals.

3.1 Proposed goals

1. We should be able to identify the pointer constructs that can be employed for performing memory access. Such kind of pointers are typically encountered in a *block*.
2. We should be able to identify the value of the upper bounds of the extents for such pointer constructs as it is needed for deriving *species*.
3. We should be able to identify the pointer aliases for such constructs. By identifying pointer aliases ASET can correctly perform dependency analysis essential for deriving *species*.

4 Proposed solution

In this thesis we have developed a custom pointer analysis infrastructure to enable pointer support for the ASET + BONES approach. We handle pointer constructs used for performing memory accesses in a *block*.

4.1 Pointer analysis

In the following section we give a brief introduction to pointer analysis and its related terminology.

4.1.1 Background about pointer analysis

Pointer analysis is a technique used to determine if two pointers refer to the same memory location at any program execution point. Pointer analysis has been studied in literature for a long period of time. We found [15] very helpful in understanding the basics of pointer analysis and designing our pointer analysis infrastructure. We would like to recall some terminology related to the subject of pointer analysis.

1. **Flow sensitive analysis**: A pointer analysis scheme is said to be flow sensitive if it takes into consideration the control flow information of the program. For such schemes an analysis solution is produced for each program execution point. As a result flow sensitive analysis schemes are more accurate but less efficient.

2. **Flow insensitive analysis**: A pointer analysis scheme is said to be flow insensitive if the control flow information of the program is not taken into consideration for analysis. For such schemes only one analysis solution is produced for the entire program. As a result flow insensitive analysis schemes are more efficient but are conservative and less accurate.

3. **Context sensitive analysis**: A pointer analysis scheme is said to be context sensitive if it produces analysis solutions by taking into consideration the context of execution at a given program point. For such schemes arguments are correctly handled for inter-procedural calls and as return values.

4. **Context insensitive analysis**: A pointer analysis scheme is said to be context insensitive if it produces analysis solutions for the current execution context (current procedure) only. Such schemes are not designed to handle inter-procedural calls or return statements.

Most pointer analysis schemes produce analysis solutions of the following forms.

1. **Must Alias**: If two pointers refer to the same memory location at some program execution point then the alias analysis yields a solution of Must Alias.

2. **Partially Alias**: If two pointers refer to a common chunk of memory allocation but are at a continuous offset then the alias analysis yields a solution of Partially Alias.

3. **May Alias**: If two pointers might refer to the same memory location for some program execution point then the alias analysis yields a solution of May Alias.

4. **Do Not Alias**: If two pointers do nor refer to the same memory location for any program point then the alias analysis yields Do Not Alias.

4.1.2 Observations about pointer analysis

1. It was identified by [19, 26] that computing aliases in presence of general pointers is an undecidable problem.

2. As a result implementations like [16] apply approximations to perform alias analysis in polynomial time. They only treat a subset of constructs like pointers, reference formals and recursive functions.

3. A popular approach suggested in [15] is to design custom pointer analysis infrastructures as per the clients need.

We are interested in creating pointer support for ASET. To derive *species* for pointer based code ASET requires the pointers to be non aliasing. Also it needs additional information like value of the upper bound of the extent for these pointer variables. Based on these observations we have designed a custom flow insensitive, context sensitive pointer analysis infrastructure. It fulfills both the requirements of ASET. It not only perform alias identification but also extracts required pointer information essential for deriving *species*.

There are several other flow insensitive, context sensitive pointer analyses discussed in literature like [11], [27]. They are useful for performing alias analysis only. They do not generate information like upper bound of the extent for pointer variables which is required by ASET. As a result using them for our work is not suitable. We would like to mention that if implementations for [11], [27] are available they can be extended for deriving information like upper bound of the extent and can be adopted for our work.

4.2 Modified work flow for the ASET + BONES approach

We modify the ASET + BONES approach as shown in Fig. 15. We introduce an additional stage called *pointer analysis* for realizing our pointer analysis infrastructure.

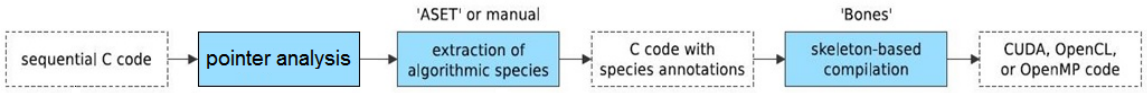


Figure 15: Modified ASET + BONES approach

Source code to be annotated is first delimited using the pragmas *#pragma scop* and *#pragma endscop* to identify *blocks*. Such delimited code (see Fig. 16a) is then subjected to the *pointer analysis* stage. The *pointer analysis* stage performs static analysis of the source code and derives required information (see Fig. 16b) regrading pointer constructs used for performing memory accesses. This extracted data is then made available to ASET. ASET has been suitably modified to process this additional data and generate *species* for pointer based code. Fig. 17 illustrates the *species* produced by ASET for the *block* in Fig16a. We will elaborated the details of the pointer analysis infrastructure in later sections.

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 int main()
4 {
5     int i;
6     int *A;
7     int *B;
8     A=(int*)malloc(20*sizeof(int));
9     B=(int*)malloc(20*sizeof(int));
10 #pragma scop
11     for(i=0;i<10;i++)
12     {
13         B[i] = A[i];
14     }
15 #pragma endscop
16     free(A);
17     free(B);
18     return 0;
19 }
```

(a) Pointer based code

```
- Patch: True
- File: pointers.c
  Function: main
  Line: 13
  Variable: A
  Scope: Local
  Extent: 20
- File: pointers.c
  Function: main
  Line: 13
  Variable: B
  Scope: Local
  Extent: 20
```

(b) Pointer Analysis information

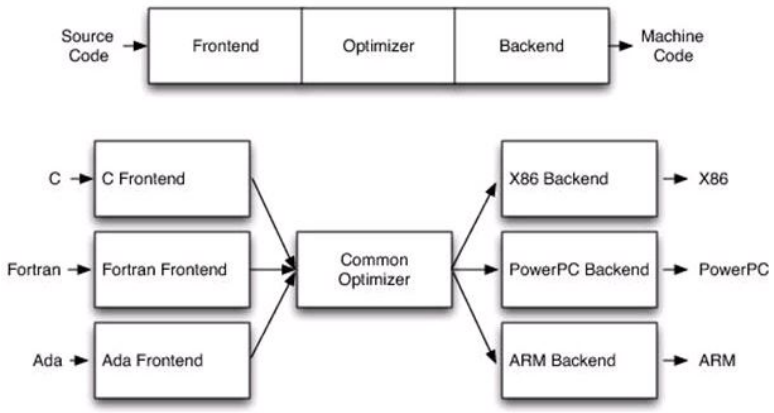
Figure 16: Illustration of pointer analysis information

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;
    int *A;
    int *B;
    A=(int*)malloc(20*sizeof(int));
    B=(int*)malloc(20*sizeof(int));
#pragma scop
    #pragma species kernel par(10) A[0:9]|element -> B[0:9]|element
    for(i=0;i<10;i++)
    {
        B[i] = A[i];
    }
    #pragma species endkernel
#pragma endscop
    free(A);
    free(B);
    return 0;
}
```

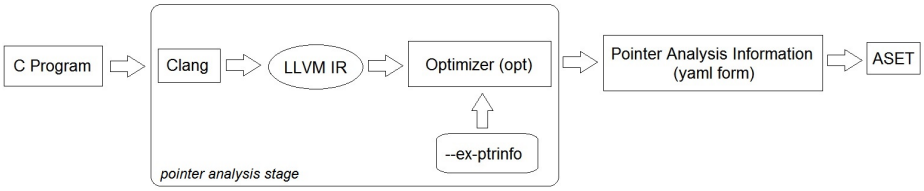
Figure 17: Species produced by ASET for *block* in Fig. 16a

4.3 Architecture of the *pointer analysis stage*

The *pointer analysis* stage employs LLVM [2] infrastructure for performing static analysis of source code. We have preferred LLVM for its intuitiveness, modularity and extensive documentation. Fig. 18a illustrates the individual components of LLVM infrastructure. Fig. 18b illustrates the architecture of our pointer analysis stage.



(a) LLVM Components



(b) Architecture of the pointer analysis stage

Figure 18: LLVM components and the architecture of *pointer analysis* stage

As we are working with C language we use LLVM's front-end for C language called *Clang*. *Clang* transforms source code to a form called LLVM IR (intermediate representation). LLVM IR has a SSA [4] form. Such LLVM IR code is then processed by LLVM's optimizer called *opt*. *Opt* comes

[4]SSA: Stands for single state assignment. It is a popular form of representation employed in compilers. SSA variables can be assigned only once and must be defined before they are used.

pre-built with many optimization passes which can be used for optimizing and analyzing LLVM IR code. LLVM allows writing custom optimization and analysis passes for opt.

To perform static analysis we have designed a flow insensitive, context sensitive, pointer analysis pass called *ex-ptrinfo* (extract pointer information). The pass can be run with LLVM's optimizer *opt* to statically analyze source code and extract required pointer information like upper bound of the extent and pointer aliases. The pointer analysis information produced by the *pointer analysis* stage is encoded into yaml [5] format to make it compatible with ASET.

4.4 Algorithms employed in ex-ptrinfo pass

In this section we will discuss the algorithms employed in the ex-ptrinfo pass. Fig. 19 illustrates the hierarchy adopted in LLVM.

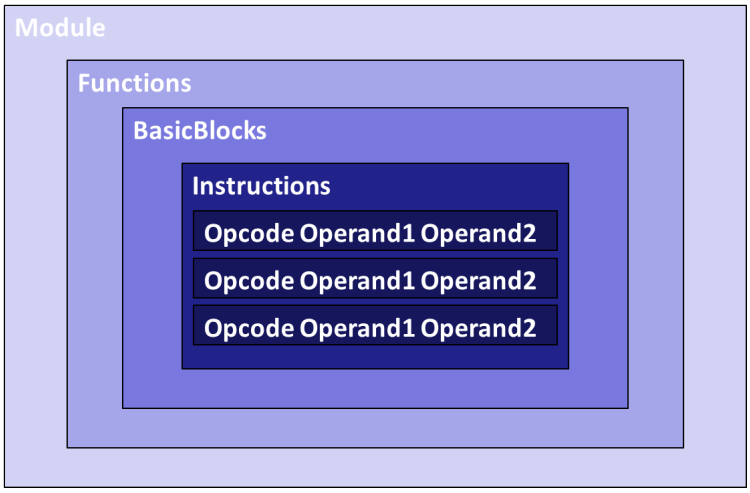


Figure 19: Hierarchy in LLVM IR

Every working program is identified by a module. A module may be composed of several function. Each function can be composed of several basic blocks. Each basic block can be composed of several instructions. Every instruction is a three address form of instruction involving an opcode and two operands.

We are interested in handling pointer constructs used for performing memory operations in PAINt and the Data Path algorithms. We illustrate such pointer constructs in Tab. 2.

Pointer based memory assignment	Pointer based memory access
ptr=ptr+offset	ptr[offset]
ptr = malloc()	structmember.ptr[offset]
ptr = & Array[ ]	structptr -> ptr[offset]
structmember.ptr = malloc()	-
structmember.ptr = & Array[ ]	-
structptr -> ptr = malloc()	-
structptr -> ptr = & Array[ ]	-

Table 2: Pointer constructs treated by the *pointer analysis* stage

[5] Yaml: Yaml is a data serialization format

Such form of memory operations fall under the category of *getelementpointer* (GEP) [3] instructions in LLVM. GEP instructions are used for address calculation of aggregate datastructures (arrays and structures) in LLVM. Because of this reason we identify GEP instructions in our algorithm for ex-ptrinfo pass (see Algo 1 step 8).

We generate LLVM IR code with debug information. It is very useful for identifying the details of a pointer variable like the line number of the variable in source code, name of the parent function and name of the source file. LLVM stores this information in two type of instructions called *llvm.debug.declare* (dbgDec) instructions [4] and *llvm.debug.value* (dbgVal) instructions [5]. We use these values for deriving the pointer information in all our algorithms.

Algorithm 1 is the principle algorithm and forms the body of the ex-ptrinfo pass. It is responsible for identifying the GEP instructions in the LLVM IR of the source code. It iterates over all the instructions in a module looking for GEP instructions. Once a GEP instruction is found it extracts memory access value (accessVal) and the type of memory access (accessType) performed. Depending on the values of accessVal and accessType it employs specific algorithm (algorithm 2-7) to identify the nature of the memory accesses performed. Once the memory access is recognized the information of the corresponding variable is recorded. Every algorithm (algorithm 2-7) employs algorithm 10 to identify *extent* value for the corresponding variable. When all the instructions in a module have been checked algorithm 8 is employed for performing alias identification. Finally algorithm 9 is employed for generating pointer analysis information in yaml format.

Algorithm 1: Algorithm for the *ex-ptrinfo* pass

Output: Produce a yaml document with pointer analysis information

```
1  ex-ptrinfo()
2  for every module in the file do
3      moduleValue = getModule();
4      for every function in the module do
5          functionValue = getFunction();
6          for every basic block in a function do
7              for every instruction in a basic block do
8                  if instruction is a GEP instruction then
9                      accessValue = GEP -> getPointerOperand();
10                     accessType = GEP -> getPointerOperandType();
11                     if accessType == arrayType then
12                         error = checkArray(moduleValue, functionValue, accessValue);
13                         if error is equal to zero then
14                             continue;
15                     else
16                         if accessType == structType then
17                             structOffset = GEP -> getOffset();
18                             error = checkStruct(moduleValue, functionValue, accessVal,
                               structOffset);
19                             if error is equal to zero then
20                                 continue;
21                         error = checkAutoScope(moduleValue, functionValue, accessVal);
22                         if error is equal to zero then
23                             continue;
24                         error = checkGlobalScope(moduleValue, functionValue, accessVal);
25                         if error is equal to zero then
26                             continue;
27                         error = checkStructArgument(moduleValue, functionValue, accessVal);
28                         if error is equal to zero then
29                             continue;
30                         error = checkArgument(moduleValue, functionValue, accessVal);
31                         if error is equal to zero then
32                             continue;
33     checkAlias();
34     yamlPrint();
```

In algorithm 1 we iterate through all the instructions in the module to look for GEP instructions. If a GEP instruction is found we extract the memory access value (accessValue) and the type of the value (accessType). We treat array based values by function checkArray(). We treat structure based values by function checkStruct(). Other values are checked based on their scope. Local values are checked by function checAutoScope() while global values are checked by function checkGlobalScope(). If the access values are arguments we check if they are structure based arguments by checkStructArgument() or regular arguments by checkArgument().

Once all the instructions in the module are processed we perform alias analysis of the identified values with the function checkAlias(). Finally the analysis results are exported as a yaml document with the function yamlPrint().

```
struct Data
{
    std::string name;
    const Value* value;
    uint64_t extent;
    std::string function;
    std::string file;
    AliasSeq partialAlias;
    AliasSeq mayAlias;
    AliasSeq mustAlias;
};
```

(a) struct Data

```
struct Arg
{
    Value *actualArgument;
    Function *callingFunction;
};
```

(b) struct Arg

```
typedef std::vector<std::string> AliasSeq;
typedef std::vector<Data> DataSequence;
typedef std::vector<Value*> StoreSeq;
```

(c) typedefs

Figure 20: structures and typedefs in ex-ptrinfo pass

Algorithm 2: Algorithm for a checkArray function

Output: Return zero if algorithm runs successfully

```
1  checkArray(moduleValue, sentFunctionValue, sentAccessValue)
2  for every basic block in the sentfunction do
3      for every instruction in a basic block do
4          if instruction is a dbgDec instruction then
5              arrayValue = dbgDec -> getAddress();
6              if sentAccessValue is equal to arrayValue then
7                  entry.name = dbgDec -> getName();
8                  entry.value = arrayValue;
9                  entry.extent = getExtent(moduleValue, sentFunctionValue, arrayValue);
10                 entry.function = dbgDec -> getFunction();
11                 entry.file = dbgDec -> getFile();
12                 yamlSequence.push_back(entry);
```

In algorithm 2 we check for all the dbgDec instructions in the sentFunction and check if their value matches to the sentAccessValue. If a match is found we extract the information from that dbgDec instruction and upload the entry to yamlSequence.

6 7

[6]entry is a instance of struct Data

[7]yamlSequence is an instance of vector DataSequence

Algorithm 3: Algorithm for a checkStruct function

Output: Return zero if algorithm runs successfully

```
1  checkStruct(moduleValue, sentFunctionValue, sentAccessValue, sentStructOffset)
2  for every basic block in the sentfunction do
3      for every instruction in a basic block do
4          if instruction is a dbgDec instruction then
5              structValue = dbgDec -> getAddress();
6              storeValList = getStore(moduleValue, structVal);
7              for every storeValue in the storeValueList do
8                  storeValueOffset = storeValue -> getOffset();
9                  if storeValueOffset is equal to sentStructOffset then
10                     entry.name = dbgDec -> getName();
11                     entry.value = storeValue;
12                     entry.extent = getExtent(moduleValue, sentFunctionValue, storeValue);
13                     entry.function = dbgDec -> getFunction();
14                     entry.file = dbgDec -> getFile();
15                     yamlSequence.push_back(entry);
```

In algorithm 3 we check for all the dbgDec instructions in the sentFunction. We extract the structValue from these instructions and find a corresponding storeValue for it. From this storeValue we extract the structure offset called storeValueOffset which represents the position of a member in a structure. If the storeValueOffset matches to the value of sentStructOffset we extract the information from that dbgDec instruction and upload the entry to yamlSequence. sentStructOffset represents the position of the structure member that was used for performing memory access.

[8]

[8]storeValList is instance of vector StoreSeq

Algorithm 4: Algorithm for a checkAutoScope function

Output: Return zero if algorithm runs successfully

```
1  checkAutoScope(moduleValue, sentFunctionValue, sentAccessValue)
2  for every basic block in the sentfunction do
3      for every instruction in a basic block do
4          if instruction is a dbgVal instruction then
5              autoValue = dbgVal -> getValue();
6              if sentAccessValue is equal to autoValue then
7                  entry.name = dbgVal -> getName();
8                  entry.value = autoValue;
9                  entry.extent = getExtent(moduleValue, sentFunctionValue, autoValue);
10                 entry.function = dbgVal -> getFunction();
11                 entry.file = dbgVal -> getFile();
12                 yamlSequence.push_back(entry);
```

In algorithm 4 we check for all the dbgVal instructions in the sentFunction and check if their value matches to the sentAccessValue. If a match is found we extract the information from that dbgVal instruction and upload the entry to yamlSequence.

Algorithm 5: Algorithm for a checkGlobalScope function

```
   Output: Return zero if algorithm runs successfully
 1 checkGlobalScope(moduleValue, sentFunctionValue, sentAccessValue)
 2 globalList = getGlobalList();
 3 for every globalVariable in the globalList do
 4     globalValue = globalVariable -> getValue();
 5     if sentAccessValue is equal to globalValue then
 6         entry.name = globalVariable -> getName();
 7         entry.value = globalValue;
 8         entry.extent = getExtent(moduleValue, sentFunctionValue, globalValue);
 9         entry.function = ' ';
10         entry.file = globalVariable -> getFile();
11         yamlSequence.push_back(entry);
```

In algorithm 5 we try to identify global variables. First we get access to the global variable list maintained for all the global variables in a module. For every variable in this global list we extract its global value and try to find a match to the sentAccessValue. If a match is found we extract the information from the corresponding globalVariable and upload the entry to yamlSequence.

Algorithm 6: Algorithm for a checkStructArgument function

```
   Output: Return zero if algorithm runs successfully
 1 checkStructArgument(moduleValue, sentFunctionValue, sentAccessValue)
 2 Initialize argument.callingFunction;
 3 Initialize argument.actualArgument;
 4 structOffset = sentAccessValue -> getOffset();
 5 while argument.callingFunction != NULL do
 6 |   argument = getArgument(moduleValue, argument.callingFunction,
   |   argument.actualArgument);
 7 for every basic block in the argument.callingFunction do
 8 |   for every instruction in a basic block do
 9 |   |   if instruction is a dbgDec instruction then
10 |   |   |   tempValue = dbgDec -> getAddress();
11 |   |   |   storeValList = getStore(moduleValue, tempVal);
12 |   |   |   for every storeValue in a storeValueList do
13 |   |   |   |   storeValueOffset = storeValue -> getOffset();
14 for every basic block in the sentfunction do
15 |   for every instruction in a basic block do
16 |   |   if instruction is a dbgVal instruction then
17 |   |   |   structValue = dbgVal -> getValue();
18 |   |   |   if sentAccessValue is equal to structValue and storeValueOffset is equal to
   |   |   |   structOffset then
19 |   |   |   |   entry.name = dbgVal -> getName();
20 |   |   |   |   entry.value = sentAccessValue;
21 |   |   |   |   entry.extent = getExtent(moduleValue, argument.calling function, storeVal);
22 |   |   |   |   entry.function = dbgVal -> getFunction();
23 |   |   |   |   entry.file = dbgVal -> getFile();
24 |   |   |   |   yamlSequence.push_back(entry);
```

We rely on getArgument() function to implement algorithm 6. getArgument() checks if the sentAccessValue matches to any of the formal arguments in the sentFunction. If a match is found it returns the corresponding calling function and the corresponding actual arguments.

In algorithm 6 we check if the sentAccessValue is an argument of structure type. For the sentAccessValue we get the corresponding actual argument and the calling function. We identify the offset for the actual argument called storeValueOffset. Also we derive the offset from sentAccessValue called structOffset

We check for all the dbgVal instructions in the sentFunction. From these dbgVal instructions we extract structValues. If the structValue matches to sentAccessValue and if the structOffset matches storeValueOffset it implies we have found the appropriate member position in the structure of the calling function. The additional check is important as the argument can be a structure pointer, a structure member as a pointer or a structure member it self. If a match is found then we extract the information from that dbgVal instruction and upload the entry to yamlSequence.

9

[9] argument is a instance of struct Arg

Algorithm 7: Algorithm for a checkArgument function

Output: Return zero if algorithm runs successfully

```
1  checkArgument(moduleValue, sentFunctionValue, sentAccessValue)
2  argument = getArgument(moduleValue, sentFunctionValue, sentAccessValue);
3  for every basic block in the sentfunction do
4      for every instruction in a basic block do
5          if instruction is a dbgVal instruction then
6              argumentValue = dbgVal -> getValue();
7              if sentAccessValue is equal to argumentValue then
8                  entry.name = dbgVal -> getName();
9                  entry.value = sentAccessValue;
10                 entry.extent = getExtent(moduleValue, argument.callingFunction,
                   argument.actualArgument);
11                 entry.function = dbgVal -> getFunction();
12                 entry.file = dbgVal -> getFile();
13                 yamlSequence.push_back(entry);
```

We rely on getArgument() function to implement algorithm 7. getArgument() checks if the sentAccessValue matches to any of the formal arguments in the sentFunction. If a match is found it returns the corresponding calling function and the corresponding actual arguments. In algorithm 7 we check for all the dbgVal instructions in the sentFunction and check if their value matches to the value if the actual argument produced by getArgument(). If a match is found we extract the information from that dbgVal instruction and upload the entry to yamlSequence.

Algorithm 8: Algorithm for a checkAlias function

```
1  checkAlias()
2  while count <yamlSequence.size() do
3      if yamlSequence[count].function is equal to yamlSequence[count+1].function and
       yamlSequence[count].name is not equal to yamlSequence[count+1].name then
4          query LLVM's basicaa pass to check if the two pointers alias or not;
5          result = basicaa(yamlSequence[count].value, yamlSequence[count+1].value);
6          switch result do
7              case 0
8                  Do not Alias;
9              case 1
10                 May Alias;
11                 entry.mayAlias.push_back(yamlSequence[count+1].name);
12             case 2
13                 Partially Alias;
14                 entry.partiallyAlias.push_back(yamlSequence[count+1].name);
15             case 3
16                 Must Alias;
17                 entry.mustAlias.push_back(yamlSequence[count+1].name);
```

In algorithm 8 we check if two entires in the yamlSequence alias or not. We rely on LLVM pass *-basicaa* [6] to perform alias analysis for us. For a given entry in yaml sequence if an alias is found the corresponding AliasSeq vector gets appended with the name of the aliasing pointer. We check for aliases if the two entries in the yamlSequence belong to the same function and do not have the same name. In case of global variable we check for aliases based on the file names rather than the function names.

10

[10] partiallyAlias, mayAlias, mustAlias are instances of Vector AliasSeq

Algorithm 9: Algorithm for a yamlPrint function

```
1 yamlPrint()
2 while count <yamlSequence.size() do
3 |  export yamlSequence[count].entry as a yaml entry to interface.txt;
```

The algorithm 9 employs APIs from *yamlcpp* (a yaml emitter for c++) for generating yaml documents. For every entry in the yamlSequence a yaml description is produced. This yaml data is then exported as a yaml document called *interface.txt* to ASET.

11

[11] interface.txt is the yaml file containing pointer analysis information. This file is read by ASET for handling pointer constructs

Algorithm 10: Algorithm for a getExtent function

Output: Return the extent value for the sentAccessValue

```
1  getExtent(moduleValue, sentFunctionValue, sentAccessValue)
2  size = sentAccessValue -> getSize();
3  if sentAccessValue is a local variable or sentAccessValue is a global variable then
4      if Check for malloc is true then
5          derive range from malloc();
6          extent=range/size;
7          return extent;
8      if Check for malloc + offset is true then
9          derive range from malloc();
10         derive offset from sentAccessValue;
11         extent=((range/size)-offset);
12         return extent;
13     if Check for array is true then
14         derive range from array[];
15         extent=range/size;
16         return extent;
17     if Check for array + offset is true then
18         derive range from array[];
19         derive offset from sentAccessValue;
20         extent=((range/size)-offset);
21         return extent;
22 if sentAccessValue is a structure variable then
23     structOffset = sentAccessValue -> getOffset();
24     storeValList = getStore(moduleValue, structVal);
25     for every storeValue in the storeValueList do
26         storeValueOffset = storeValue -> getOffset();
27         if structOffset is equal to storeValueOffset then
28             extent = getExtent(moduleValue, sentFunctionValue, storeValue);
29             return extent;
30 if sentAccessValue is an argument variable then
31     argument = getArgument(moduleValue, sentFunctionValue, sentAccessValue);
32     extent = getExtent(moduleValue, argument.callingFunction, argument.actualArgument);
33     return extent;
```

Algorithm 10 is used for extracting the value of the upper bound of the extent for pointer constructs used for performing memory accesses. Depending on the type of the access a suitable method is applied and the value for extent is derived.

4.5 Working

The following examples illustrate the input and the output of the *pointer analysis* stage. Fig. 21a represents the delimited source code which acts as the input. Fig. 21b illustrates the pointer analysis information generated by the *pointer analysis* stage at the output.

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;
    int *A;
    int *B;
    A=(int*)malloc(20*sizeof(int));
    B=(int*)malloc(20*sizeof(int));
#pragma scop
    for(i=0;i<10;i++)
    {
        B[i] = A[i];
    }
#pragma endscop
    free(A);
    free(B);
    return 0;
}
```

(a) Pointer based code

```
- Patch: True
- File: pointers.c
  Function: main
  Line: 13
  Variable: A
  Scope: Local
  Extent: 20
- File: pointers.c
  Function: main
  Line: 13
  Variable: B
  Scope: Local
  Extent: 20
```

(b) Pointer Analysis information

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;
    int *A;
    int *B;
    A=(int*)malloc(20*sizeof(int));
    B=(int*)malloc(20*sizeof(int));
#pragma scop
    #pragma species kernel par(10) A[0:9]|element -> B[0:9]|element
    for(i=0;i<10;i++)
    {
        B[i] = A[i];
    }
    #pragma species endkernel
#pragma endscop
    free(A);
    free(B);
    return 0;
}
```

(c) Species produced by ASET for *block* in Fig. 21a

Figure 21: Pointer analysis information and its usage for deriving *species* for non aliasing pointers

It is evident that the two pointers do not alias and hence no alias information is generated (see Algo 1, 4, 8, 9, 10). As a result the *block* in Fig. 21a can be annotated. ASET has been modified to process the additional information in Fig. 21b to derive *species* as shown in Fig. 21c. Such annotated code in Fig. 21c can be used with BONES for compiling OpenCL, CUDA, or OpenMP code.

The examples in Fig. 22 illustrate the case of aliasing pointers without dependency. Fig. 22a represents the delimited source code with aliasing pointers. Fig. 22b illustrates the pointer analysis information generated by the *pointer analysis* stage. It can be observed that the *pointer analysis* stage could correctly identify Must Aliases at line 14 (see Algo 1, 4, 8, 9, 10). In case of Must Aliases ASET has been modified to perform dependency analysis for the memory accesses performed (line 14). ASET found the iterations to be independent and so annotations were produced as shown in Fig. 22c

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;
    int *A;
    int *B;
    A=(int*)malloc(20*sizeof(int));
    B=(int*)malloc(20*sizeof(int));
    B=A;
#pragma scop
    for(i=0;i<4;i++)
    {
        B[i] = A[10-i];
    }
#pragma endscop
    free(A);
    free(B);
    return 0;
}
```

(a) Pointer based code

```
- Patch: True
- File: pointers.c
  Function: main
  Line: 14
  Variable: A
  Scope: Local
  Extent: 20
  MustAliases:
    - B
- File: pointers.c
  Function: main
  Line: 14
  Variable: B
  Scope: Local
  Extent: 20
  MustAliases:
    - A
```

(b) Pointer Analysis information

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;
    int *A;
    int *B;
    A=(int*)malloc(20*sizeof(int));
    B=(int*)malloc(20*sizeof(int));
    B=A;
#pragma scop
    #pragma species kernel par(4) A[7:10]|element -> B[0:3]|element
    for(i=0;i<4;i++)
    {
        B[i] = A[10-i];
    }
    #pragma species endkernel
#pragma endscop
    free(A);
    free(B);
    return 0;
}
```

(c) Species produced by ASET for *block* in Fig. 22a

Figure 22: Pointer analysis information and its usage for deriving *species* for aliasing pointers with no dependency

The examples in Fig. 23 illustrate the case of aliasing pointers with dependency. Fig. 23a represents the delimited source code with aliasing pointers. Fig. 23b illustrates the pointer analysis information generated by the *pointer analysis* stage. It can be observed that the *pointer analysis* stage could correctly identify Must Aliases at line 14 (see Algo 1, 4, 8, 9, 10).

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 int main()
4 {
5     int i;
6     int *A;
7     int *B;
8     A=(int*)malloc(20*sizeof(int));
9     B=(int*)malloc(20*sizeof(int));
10    B=A;
11 #pragma scop
12    for(i=0;i<10;i++)
13    {
14        B[i] = A[10-i];
15    }
16 #pragma endscop
17    free(A);
18    free(B);
19    return 0;
20 }
```

(a) Aliasing pointers

```
- Patch: True
- File: pointers.c
  Function: main
  Line: 14
  Variable: A
  Scope: Local
  Extent: 20
  MustAliases:
    - B
- File: pointers.c
  Function: main
  Line: 14
  Variable: B
  Scope: Local
  Extent: 20
  MustAliases:
    - A
```

(b) Pointer Analysis information

Figure 23: Illustration of pointer analysis information for aliasing pointers with dependent accesses

In case of Must Aliases (see Algo 8 line 15) ASET performs dependency analysis for the memory accesses performed (line 14). ASET found the iterations to be dependent (see loop bound in Fig. 23a at line 12) and so no annotations were produced in this case.

Same treatment is applied to Partial Aliases (see Algo 8 line 12). ASET perform dependency analysis for partially aliasing pointers and derives *species* if the pointers are found to be independent. But in case of May Aliases (see Algo 8 line 9) we reject the *block* as there is an uncertainty that the pointers may alias.

4.6 Treatment for May Aliases

We rely on LLVMS's alias analysis infrastructure [7] to identify pointer aliases in source code. Specifically we rely on passes *-basicaa* [6] and *-globalsmodref* [8] to perform alias identification. *-basicaa* is a intra-procedural alias analysis pass. It requires that one of the aliasing candidates be declared within the local scope of execution. The pass is incapable of performing inter-procedural alias analysis. So if the aliasing candidates are declared beyond the current scope of execution they are marked as May Aliases (see Algo 8 line 9). In such circumstances we inline the code in a bottom-up approach. As a result *main* function can be treated as a composite executional unit to identify aliases over the entire program.

In the examples in Fig. 24 we illustrate the process of resolving May Aliases. We make a call from main() in Fig. 24a to foo() in Fig. 24c. In absence of inlining we get the interface data as shown in Fig. 24e. . We identify that pointer X and pointer Y May Alias (see Algo 1, 4, 8, 9, 10). When the functions are inlined we get interface data as shown in Fig. 24f. We derive that pointer X and pointer Y Do Not Alias(see Algo 1, 4, 8, 9, 10). So the array operations on line 8 in Fig. 24c can be annotated. Annotations are produced as shown in Fig. 24d

```c
#include<myheader.h>
int main()
{
    int *A;
    int *B;
    A=(int*)malloc(20*sizeof(int));
    B=(int*)malloc(20*sizeof(int));
    foo(A,B);
    free(A);
    free(B);
    return 0;
}
```

(a) caller.c

```c
#include<stdio.h>
#include<stdlib.h>
void foo(int *X, int *Y);
```

(b) myheader.h

```c
#include<myheader.h>
void foo(int *X, int *Y)
{
    int i;
#pragma scop
    for(i=0;i<4;i++)
    {
        X[i] = Y[i];
    }
#pragma endscop
}
```

(c) callee.c

```c
#include<myheader.h>
void foo(int *X, int *Y)
{
    int i;
#pragma scop
    #pragma species kernel par(4) Y[0:3]|element -> X[0:3]|element
    for(i=0;i<4;i++)
    {
        X[i] = Y[i];
    }
    #pragma species endkernel
#pragma endscop
}
```

(d) annotations produced after resolving May Aliases in callee.c

```
- Patch: True
- File: callee.c
  Function: foo
  Line: 8
  Variable: Y
  Scope: Argument
  Extent: 20
  MayAliases:
    - X
- File: callee.c
  Function: foo
  Line: 8
  Variable: X
  Scope: Argument
  Extent: 20
  MayAliases:
    - Y
```

(e) Interface data before inline

```
- Patch: True
- File: caller.c
  Function: main
  Line: 8
  Variable: B
  Scope: Local
  Extent: 20
- File: callee.c
  Function: foo
  Line: 8
  Variable: Y
  Scope: Local
  Extent: 20
- File: caller.c
  Function: main
  Line: 8
  Variable: A
  Scope: Local
  Extent: 20
- File: callee.c
  Function: foo
  Line: 8
  Variable: X
  Scope: Local
  Extent: 20
```

(f) Interface data after inline

Figure 24: Resolving May Aliases

5 Experiments

The following section describes the experiments we have performed to test our pointer analysis infrastructure. We test the original version of the ASET + BONES approach (based on *pet-0.1*) against the modified ASET + BONES approach (based on *pet-0.5*) for handling pointer constructs. The original version of the ASET + BONES (based on *pet-0.1*) is incapable of handling pointer constructs. So in all the conducted tests the approach crashes and fails to produce any annotations.

5.1 Results

The following tests have been performed with the modified ASET + BONES approach (based on *pet-0.5*).

5.1.1 Test to evaluate the correctness of *species* produced with ASET

The following test evaluates the capability of our pointer analysis infrastructure in correctly enabling the ASET in deriving *species* for pointer based constructs.

Figs. 25a, 25b, 26a,27a, 28a and 29a are the source files and header files used for the test.

```c
#include"my_header.h"
void setup(int *A, int *U, int *W)
{
    int i,j;
    for(i=0;i<20;i++)
    {
        A[i] = i;
    }
    for(j=0;j<4;j++)
    {
        U[j]=1;
        W[j]=2;
    }
    element(A);
    chunk_full(A,U);
    neighbourhood(A);
    shared(U,W);
}

int main()
{
    int *A;
    int *U;
    int *W;
    A=(int *)malloc(20*sizeof(int));
    U=(int *)malloc(4*sizeof(int));
    W=(int *)malloc(4*sizeof(int));
    setup(A,U,W);
    return 0;
}
```

(a) main.c

```c
#include <stdio.h>
#include <stdlib.h>

void setup(int *A, int *U, int *W);
void element(int *A);
void chunk_full(int *A, int *U);
void neighbourhood(int *A);
void shared(int *U, int *W);
```

(b) my_header.h

Figure 25: main.c and my_header.h

Fig. 26a represents element function which involves memory accesses of *element* type at line 10. Fig. 26b represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 26a. Fig. 26c represents the annotated code generated by ASET by processing the information in Fig. 26b.

Pointer based memory assignment	Pointer based memory access
ptr = malloc()	ptr[offset]

Table 3: Pointer constructs treated in the example

```c
#include"my_header.h"
void element(int *A)
{
    int i,j;
    int *B;
    B=(int *)malloc(16*sizeof(int));
#pragma scop
    for(i=0;i<16;i++)
    {
        B[i] = (2*A[i]);
    }
#pragma endscop
    free(B);
}
```

(a) element.c

```
- Patch: True
- File: element.c
  Function: element
  Line: 10
  Variable: A
  Scope: Local
  Extent: 20
- File: element.c
  Function: element
  Line: 10
  Variable: B
  Scope: Local
  Extent: 16
```

(b) Pointer analysis data

```c
#include"my_header.h"
void element(int *A)
{
    int i,j;
    int *B;
    B=(int *)malloc(16*sizeof(int));
#pragma scop
    #pragma species kernel par(16) A[0:15]|element -> B[0:15]|element
    for(i=0;i<16;i++)
    {
        B[i] = (2*A[i]);
    }
    #pragma species endkernel
#pragma endscop
    free(B);
}
```

(c) *element specie*

Figure 26: Deriving the *element specie*

It can be observed that the pointer analysis information in Fig. 26b correctly represents the details of the memory accesses performed in Fig. 26a. It can be verified that pointer A and pointer B both have local scope. Both are enclosed within the element function and are located at line 10. Also both the pointers are present in the source file element.c. The pointer analysis information can correctly represent the extent value for pointer A as 20 and for pointer B as 16. As the pointer accesses on line 10 do not alias they are independent and can be annotated as shown in Fig. 26c

Fig. 27a represents chunk_full function which involves memory accesses of *chunk*, *full* and *element* type at line 13. Fig. 27b represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 27a. Fig. 27c represents the annotated code generated by ASET by processing the pointer analysis information in Fig. 27b.

Pointer based memory assignment	Pointer based memory access
ptr = malloc()	ptr[offset]

Table 4: Pointer constructs treated in the example

```c
#include"my_header.h"
void chunk_full(int *A, int *U)
{
    int i,j;
    int *X;
    X=(int *)malloc(4*sizeof(int));
#pragma scop
    for(i=0;i<4;i++)
    {
        X[i]=0;
        for(j=0;j<4;j++)
        {
            X[i] += A[j+i*4] * U[j] ;
        }
    }
#pragma endscop
    free(X);
}
```

(a) chunk_full.c

```
- Patch: True
- File: chunk_full.c
  Function: chunk_full
  Line: 10
  Variable: X
  Scope: Local
  Extent: 4
- File: chunk_full.c
  Function: chunk_full
  Line: 13
  Variable: A
  Scope: Local
  Extent: 20
  MayAliases:
    - U
- File: chunk_full.c
  Function: chunk_full
  Line: 13
  Variable: U
  Scope: Local
  Extent: 4
  MayAliases:
    - A
- File: chunk_full.c
  Function: chunk_full
  Line: 13
  Variable: X
  Scope: Local
  Extent: 4
```

(b) Pointer analysis data

```c
#include"my_header.h"
void chunk_full(int *A, int *U)
{
    int i,j;
    int *X;
    X=(int *)malloc(4*sizeof(int));
#pragma scop
    #pragma species kernel par(4) A[0:15]|chunk(0:3) ^ U[0:3]|full -> X[0:3]|element
    for(i=0;i<4;i++)
    {
        X[i]=0;
        for(j=0;j<4;j++)
        {
            X[i] += A[j+i*4] * U[j] ;
        }
    }
    #pragma species endkernel
#pragma endscop
    free(X);
}
```

(c) *chunk specie* and *full specie*

Figure 27: Deriving the *chunk specie* and *full specie*

It can be observed that the pointer analysis information in Fig. 27b correctly represents the details of the memory accesses performed in Fig. 27a. The extent value for pointer A was identified as 20, for pointer U as 4 and for pointer X as 4. As A and U are both arguments our pointer analysis infrastructure classifies them as May Aliases. We perform read operations on pointers A and U and do not write to their addresses. So the accesses are independent and can be safely annotated. Annotations are produced as shown in Fig. 27c

Fig. 28a represents neighbourhood function which involves memory accesses of *neighbourhood* and *element* type at line 10. Fig. 28b represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 28a. Fig. 28c represents the annotated code generated by ASET by processing the pointer analysis information in Fig. 28b.

Pointer based memory assignment	Pointer based memory access
ptr = malloc()	ptr[offset]

Table 5: Pointer constructs treated in the example

```
1 #include"my_header.h"
2 void neighbourhood(int *A)
3 {
4     int i,j;
5     int *B;
6     B=(int *)malloc(16*sizeof(int));
7 #pragma scop
8     for(i=1;i<16;i++)
9     {
10        B[i] = A[i-1]+A[i]+A[i+1];
11    }
12 #pragma endscop
13    free(B);
14 }
```

(a) neighbourhood.c

```
1 - Patch: True
2 - File: neighbourhood.c
3   Function: neighbourhood
4   Line: 10
5   Variable: A
6   Scope: Local
7   Extent: 20
8 - File: neighbourhood.c
9   Function: neighbourhood
10  Line: 10
11  Variable: A
12  Scope: Local
13  Extent: 20
14 - File: neighbourhood.c
15  Function: neighbourhood
16  Line: 10
17  Variable: A
18  Scope: Local
19  Extent: 20
20 - File: neighbourhood.c
21  Function: neighbourhood
22  Line: 10
23  Variable: B
24  Scope: Local
25  Extent: 16
```

(b) Pointer analysis data

```
1 #include"my_header.h"
2 void neighbourhood(int *A)
3 {
4     int i,j;
5     int *B;
6     B=(int *)malloc(16*sizeof(int));
7 #pragma scop
8     #pragma species kernel par(15) A[0:16]|neighborhood(-1:1) -> B[1:15]|element
9     for(i=1;i<16;i++)
10    {
11        B[i] = A[i-1]+A[i]+A[i+1];
12    }
13    #pragma species endkernel
14 #pragma endscop
15    free(B);
16 }
```

(c) *neighbourhood specie*

Figure 28: Deriving the *neighbourhood specie*

It can be observed that the pointer analysis information in Fig. 28b correctly represents the details of the memory accesses performed in Fig. 28a. The extent value for pointer A was identified as 20 and for pointer B as 16. Although A is an argument we do not classify it as May Alias (see step 3 in Algo 8) because we do not check a pointer with itself for alias analysis. It would get checked in dependency analysis stage (see step d Fig. 7) in ASET. All the accesses on line 11 were found to be independent. So annotations were produced as shown in Fig. 28c

Fig. 29a represents shared function which involves memory accesses of *shared* and *element* type at line 10. Fig. 29b represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 29a. Fig. 29c represents the annotated code generated by ASET by processing the pointer analysis information in Fig. 29b.

Pointer based memory assignment	Pointer based memory access
ptr = malloc()	ptr[offset]

Table 6: Pointer constructs treated in the example

```c
#include"my_header.h"
void shared(int *U, int *W)
{
    int i;
    int *X;
    X=(int *)malloc(1*sizeof(int));
#pragma scop
    for(i=0;i<4;i++)
    {
        X[0] += U[i] + 3 * W[i];
    }
#pragma endscop
    free(X);
}
```

(a) shared.c

```
- Patch: True
- File: shared.c
  Function: shared
  Line: 10
  Variable: U
  Scope: Local
  Extent: 4
  MayAliases:
    - W
- File: shared.c
  Function: shared
  Line: 10
  Variable: W
  Scope: Local
  Extent: 4
  MayAliases:
    - U
- File: shared.c
  Function: shared
  Line: 10
  Variable: X
  Scope: Local
  Extent: 1
```

(b) Pointer analysis data

```c
#include"my_header.h"
void shared(int *U, int *W)
{
    int i;
    int *X;
    X=(int *)malloc(1*sizeof(int));
#pragma scop
    #pragma species kernel par(4) U[0:3]|element ^ W[0:3]|element -> X[0:0]|shared
    for(i=0;i<4;i++)
    {
        X[0] += U[i] + 3 * W[i];
    }
    #pragma species endkernel
#pragma endscop
    free(X);
}
```

(c) *shared specie*

Figure 29: Deriving the *shared specie*

It can be observed that the pointer analysis information in Fig. 29b correctly represents the details of the memory accesses performed in Fig. 29a. The extent value for pointer U and W was identified as 4, and for pointer X as 1. As A and U are both arguments our pointer analysis infrastructure classifies them as May Aliases. We only perform read operations on pointers U and W and do not write to their addresses. So they are independent and can be safely annotated as shown in Fig. 29c.

The results shown in Figs. 26, 27, 28, 29 demonstrate that the pointer analysis infrastructure when integrated into ASET can successfully generate *species* based on all five types of the access patters.

5.1.2 Test to determine the ability to handle system defined pointer constructs

The following test evaluates the capability of our pointer analysis infrastructure to handle the system defined pointer constructs that have been employed for performing memory accesses in PAINt and the Data Path algorithms.

Figs. 30a, 30b, 30c, 30d are the source files and header files used for the test.

```c
#include"myheader.h"
int K[128];
int main()
{
    int i,*A,*L,no1=128,no2 = 256;
    A=(int*)malloc(no1*sizeof(int));
    L=&K[7];
#pragma scop
    for(i=0;i<4;i++)
    {
        L[i] = K[i+2];
    }
#pragma endscop
    mycode(A,no2);
    free(A);
    return 0;
}
```

(a) main.c

```c
#include<stdio.h>
#include<stdlib.h>
void mycode(int *, int);
void yourcode(int *, int *);
```

(b) myheader.h

```c
#include"myheader.h"
void mycode(int *X, int No)
{
    int i,*B,*F;
    B=(int*)malloc(No*sizeof(int));
    F=X+5;
#pragma scop
    for(i=0;i<4;i++)
    {
        B[i] = F[i]*2;
    }
#pragma endscop
    yourcode(B,F);
    free(B);
}
```

(c) myc.c

```c
#include"myheader.h"
void yourcode(int *U, int *V)
{
    int i,*L,Z[256];
    L=&Z[5];
#pragma scop
    for(i=0;i<4;i++)
    {
        Z[i] = U[i]*4+8*V[i]+L[i];
    }
#pragma endscop
}
```

(d) yrc.c

Figure 30: source files main.c myheader.h myc.c yrc.c

Fig. 30a represents the main.c file which involves memory accesses at line 11. Fig. 31a represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 30a. Fig. 31b represents the annotated code generated by ASET by processing the pointer analysis information in Fig. 31a.

Pointer based memory assignment	Pointer based memory access
ptr = malloc()	ptr[offset]
ptr = &Array[]	-

Table 7: Pointer constructs treated in the example

```
- Patch: True
- File: main.c
  Function: ""
  Line: 11
  Variable: K
  Scope: Global
  Extent: 128
  PartialAliases:
    - L
- File: main.c
  Function: main
  Line: 11
  Variable: L
  Scope: Local
  Extent: 121
```

(a) Pointer analysis data

```
1  #include"myheader.h"
2  int K[128];
3  int main()
4  {
5      int i,*A,*L,no1=128,no2 = 256;
6      A=(int*)malloc(no1*sizeof(int));
7      L=&K[7];
8  #pragma scop
9      #pragma species kernel par(4) K[2:5]|element -> L[0:3]|element
10     for(i=0;i<4;i++)
11     {
12         L[i] = K[i+2];
13     }
14     #pragma species endkernel
15 #pragma endscop
16     mycode(A,no2);
17     free(A);
18     return 0;
19 }
```

(b) annotated main.c

Figure 31: main.c

It can be observed that the pointer analysis information in Fig. 31a correctly represents the details of the memory accesses performed in Fig. 30a. We could correctly identify the scope of array K as global. Also the extent value for array K was identified as 128 (after adjusting the offset by 7), and for pointer L as 121 after adjusting the offset. It was correctly identified that K and L Partially Alias. As the accesses on line 11 were found to be independent annotations were produced by ASET as shown in Fig. 31b.

Fig. 30c represents the myc.c file which involves memory accesses at line 10. Fig. 32a represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 30c. Fig. 32b represents the annotated code generated by ASET by processing the pointer analysis information in Fig. 32a.

Pointer based memory assignment	Pointer based memory access
ptr = malloc()	ptr[offset]
ptr = ptr + offset	-

Table 8: Pointer constructs treated in the example

```
- Patch: True
- File: myc.c
  Function: mycode
  Line: 6
  Variable: X
  Scope: Argument
  Extent: 128
  PartialAliases:
    - F
- File: myc.c
  Function: mycode
  Line: 10
  Variable: F
  Scope: Local
  Extent: 123
  PartialAliases:
    - X
- File: myc.c
  Function: mycode
  Line: 10
  Variable: B
  Scope: Local
  Extent: 256
```

(a) Pointer analysis data

```
1 #include"myheader.h"
2 void mycode(int *X, int No)
3 {
4     int i,*B,*F;
5     B=(int*)malloc(No*sizeof(int));
6     F=X+5;
7 #pragma scop
8     #pragma species kernel par(4) F[0:3]|element -> B[0:3]|element
9     for(i=0;i<4;i++)
10    {
11        B[i] = F[i]*2;
12    }
13    #pragma species endkernel
14 #pragma endscop
15    yourcode(B,F);
16    free(B);
17 }
```

(b) annotated myc.c

Figure 32: myc.c

It can be observed that the pointer analysis information in Fig. 32a correctly represents the details of the memory accesses performed in Fig. 30c. The extent value for pointers X was identified as 128, for pointer F as 123 (after adjusting the offset by 5) and for pointers B as 256. As the accesses on line 10 were found to be independent annotations were produced by ASET as shown in Fig. 31b.

Fig. 30d represents the yrc.c file which involves memory accesses at line 9. Fig. 33a represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 30d. Fig. 33b represents the annotated code generated by ASET by processing the pointer analysis information in Fig. 33a.

Pointer based memory assignment	Pointer based memory access
ptr = &Array[]	ptr[offset]

Table 9: Pointer constructs treated in the example

```
- Patch: True
- File: yrc.c
  Function: yourcode
  Line: 5
  Variable: Z
  Scope: Local
  Extent: 256
  PartialAliases:
    - L
- File: yrc.c
  Function: yourcode
  Line: 9
  Variable: U
  Scope: Argument
  Extent: 256
  MayAliases:
    - V
- File: yrc.c
  Function: yourcode
  Line: 9
  Variable: V
  Scope: Argument
  Extent: 123
  MayAliases:
    - U
- File: yrc.c
  Function: yourcode
  Line: 9
  Variable: L
  Scope: Local
  Extent: 251
  PartialAliases:
    - Z
- File: yrc.c
  Function: yourcode
  Line: 9
  Variable: Z
  Scope: Local
  Extent: 256
  PartialAliases:
    - L
```

(a) Pointer analysis data

```
1 #include"myheader.h"
2 void yourcode(int *U, int *V)
3 {
4     int i,*L,Z[256];
5     L=&Z[5];
6 #pragma scop
7     #pragma species kernel par(4) U[0:3]|element ^ L[0:3]|element ^ V[0:3]|element -> Z[0:3]|element
8     for(i=0;i<4;i++)
9     {
10        Z[i] = U[i]*4+8*V[i]+L[i];
11    }
12    #pragma species endkernel
13 #pragma endscop
14 }
```

(b) annotated yrc.c

Figure 33: yrc.c

It can be observed that the pointer analysis information in Fig. 33a correctly represents the details of the memory accesses performed in Fig. 30d. The extent value for pointers U, V, L and array Z was correctly identified. As pointers U and V are arguments they were identified as May Aliases. Pointer L and array Z were found as Partial Aliases. We have perform read operations on May Aliases U and V, so they were independent. Also the rest of the accesses on line 9 were found to be independent. So annotations were produced by ASET as shown in Fig. 31b.

The following set of results shown in Figs. 31, 32, 33, demonstrates that our pointer analysis infrastructure can handle the system defined pointer constructs that have been employed for performing memory accesses in PAINt and the Data Path algorithms.

5.1.3 Test to determine the ability to handle structure based pointer constructs

The following test evaluates the capability of our pointer analysis infrastructure to handle structure based pointer constructs that have been employed for performing memory accesses in PAINt and the Data Path algorithms.

Figs. 34a, 34b, 34c, 34d are the source files and header files used for the test.

```c
#include"myheader.h"
int main()
{
    int i;
    int no1=128, no2 = 256,K[512];
    struct mystruct A,*G;
    G=&A;
    A.P = (int*)malloc(no1 * sizeof(int));
    A.Q = &K[6];
    G -> R = (int*)malloc(no2 * sizeof(int));
    G -> S = &K[10];
#pragma scop
    for(i=0;i<4;i++)
    {
        A.P[i] = A.Q[i];
    }
#pragma endscop
    theircode(G,no2);
    free(A.P);
    free(A.R);
    return 0;
}
```

(a) struct.c

```c
#include<stdio.h>
#include<stdlib.h>
struct mystruct
{
    int *P;
    int *Q;
    int *R;
    int *S;
};
void theircode(struct mystruct *, int);
void yourcode(int *, int *);
```

(b) myheader.h

```c
#include"myheader.h"
void theircode(struct mystruct *X, int No)
{
    int i, *B;
    B=(int*)malloc(No*sizeof(int));
#pragma scop
    for(i=0;i<4;i++)
    {
        B[i] = X->R[i] + X->S[i];
    }
#pragma endscop
    yourcode(B,X->P);
    free(B);
}
```

(c) trc.c.c

```c
#include"myheader.h"
void yourcode(int *U, int *V)
{
    int i,*L;
    int Z[256];
    L=&Z[5];
#pragma scop
    for(i=0;i<4;i++)
    {
        Z[i] = U[i]*4+8*V[i]+L[i];
    }
#pragma endscop
}
```

(d) yrc.c

Figure 34: source files struct.c myeader.h trc.c yrc.c

Fig. 34a represents the struct.c file which involves memory accesses at line 15. Fig. 35a represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 34a. Fig. 35b represents the annotated code generated by ASET by processing the pointer analysis information in Fig. 35a.

Pointer based memory assignment	Pointer based memory access
structmember.ptr = malloc()	structmember.ptr[offset]
structmember.ptr = & Array[ ]	-
structptr -> ptr = malloc()	-
structptr -> ptr = & Array[ ]	-

Table 10: Pointer constructs treated in the example

```
1 - Patch: True
2 - File: struct.c
3   Function: main
4   Line: 9
5   Variable: K
6   Scope: Local
7   Extent: 512
8   PartialAliases:
9     - Q
10 - File: struct.c
11   Function: main
12   Line: 11
13   Variable: K
14   Scope: Local
15   Extent: 512
16   PartialAliases:
17     - Q
18 - File: struct.c
19   Function: main
20   Line: 15
21   Variable: Q
22   Scope: Structure
23   Extent: 506
24   PartialAliases:
25     - K
26 - File: struct.c
27   Function: main
28   Line: 15
29   Variable: P
30   Scope: Structure
31   Extent: 128
```

(a) Pointer analysis information

```
1 #include"myheader.h"
2 int main()
3 {
4     int i;
5     int no1=128, no2 = 256,K[512];
6     struct mystruct A,*G;
7     G=&A;
8     A.P = (int*)malloc(no1 * sizeof(int));
9     A.Q = &K[6];
10    G -> R = (int*)malloc(no2 * sizeof(int));
11    G -> S = &K[10];
12 #pragma scop
13    #pragma species kernel par(4) Q[0:3]|element -> P[0:3]|element
14    for(i=0;i<4;i++)
15    {
16        A.P[i] = A.Q[i];
17    }
18    #pragma species endkernel
19 #pragma endscop
20    theircode(G,no2);
21    free(A.P);
22    free(A.R);
23    return 0;
24 }
```

(b) annotated struct.c

Figure 35: struct.c

It can be observed that the pointer analysis information in Fig. 35a correctly represents the details of the memory accesses performed in Fig. 34a. The extent value for pointers P was identified as 128 and for pointer Q as 506 (after adjusting the offset by 6). It was identified that pointer Q and array K Partially Alias. The memory accesses performed on line 15 do not alias and were found to be independent. So annotations were produced as shown in the example in Fig. 35b.

Fig. 34c represents the trc.c file which involves memory accesses at line 9. Fig. 36a represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 34c.

Pointer based memory assignment	Pointer based memory access
ptr = malloc()	structptr -> ptr[offset]
-	ptr[offset]

Table 11: Pointer constructs treated in the example

```
1 - Patch: True
2 - File: trc.c
3   Function: theircode
4   Line: 9
5   Variable: X->R
6   Scope: Argument
7   Extent: 256
8   MayAliases:
9     - B
10    - X->S
11 - File: trc.c
12  Function: theircode
13  Line: 9
14  Variable: X->S
15  Scope: Argument
16  Extent: 502
17  MayAliases:
18    - B
19    - X->R
20 - File: trc.c
21  Function: theircode
22  Line: 9
23  Variable: B
24  Scope: Local
25  Extent: 256
26  MayAliases:
27    - X->R
28    - X->S
29
```

(a) Pointer analysis data

```
1 #include"myheader.h"
2 void theircode(struct mystruct *X, int No)
3 {
4     int i, *B;
5     B=(int*)malloc(No*sizeof(int));
6 #pragma scop
7     #pragma species kernel par(4) S[0:3]|element ^ R[0:3]|element -> B[0:3]|element
8     for(i=0;i<4;i++)
9     {
10        B[i] = X->R[i] + X->S[i];
11    }
12    #pragma species endkernel
13 #pragma endscop
14    yourcode(B,X->P);
15    free(B);
16 }
```

(b) annotated trc.c

Figure 36: pointer analysis information for file trc.c

It can be observed that the pointer analysis information in Fig. 36a correctly represents the details of the memory accesses performed in Fig. 34c.The extent value for pointers B was identified as 256, for R as 256 and for S as 502 (after adjusting the offset by 10). Pointers R and S are structure members. They were accessed with structure pointer X which is a formal argument. As a result R and S were identified as May Aliases. We have performed read and write operations on May Aliases B, R and S. So the *block* in Fig. 34c was not annotated. We have resolved these May Aliases by applying the method suggested in Sec. 4.6 to produce annotations as shown in Fig. 36b.

Fig. 34d represents the yrc.c file which involves memory accesses at line 10. Fig. 37a represents the pointer analysis information produced by our infrastructure for the code snippet in Fig. 34d. Fig. 37b represents the annotated code generated by ASET by processing the pointer analysis information in Fig. 37a.

Pointer based memory assignment	Pointer based memory access
ptr = &Array[]	ptr[offset]

Table 12: Pointer constructs treated in the example

```
- Patch: True
- File: yrc.c
  Function: yourcode
  Line: 6
  Variable: Z
  Scope: Local
  Extent: 256
  PartialAliases:
    - L
- File: yrc.c
  Function: yourcode
  Line: 10
  Variable: U
  Scope: Argument
  Extent: 256
  MayAliases:
    - V
- File: yrc.c
  Function: yourcode
  Line: 10
  Variable: V
  Scope: Argument
  Extent: 128
  MayAliases:
    - U
- File: yrc.c
  Function: yourcode
  Line: 10
  Variable: L
  Scope: Local
  Extent: 251
  PartialAliases:
    - Z
- File: yrc.c
  Function: yourcode
  Line: 10
  Variable: Z
  Scope: Local
  Extent: 256
  PartialAliases:
    - L
```

(a) Pointer analysis data

```
#include"myheader.h"
void yourcode(int *U, int *V)
{
    int i,*L;
    int Z[256];
    L=&Z[5];
#pragma scop
    #pragma species kernel par(4) U[0:3]|element ^ L[0:3]|element ^ V[0:3]|element -> Z[0:3]|element
    for(i=0;i<4;i++)
    {
        Z[i] = U[i]*4+8*V[i]+L[i];
    }
    #pragma species endkernel
#pragma endscop
}
```

(b) annotated yrc.c

Figure 37: pointer analysis information for file yrc.c

It can be observed that the pointer analysis information in Fig. 37a correctly represents the details of the memory accesses performed in Fig. 34d. The extent value for pointers U, V, L and array Z were correctly identified. As pointers U and V are arguments they were identified as May Aliases. Pointer L and array Z were identified as Partial Aliases. We have performed read operations on May Aliases U and V, so they were independent. Also the rest of the accesses on line 10 were found to be independent. So annotations were produced for the memory accesses on line 10 as shown in Fig. 37b.

The results shown in Figs. 35, 36, 37 demonstrate that our pointer analysis infrastructure can handle the structure based pointer constructs that have been employed for performing memory accesses in PAINt and the Data Path algorithms.

5.2 Summary

We have conducted three tests for validating the behavior of our pointer analysis infrastructure.

1. The first test demonstrated that our pointer analysis infrastructure can be integrated into ASET + BONES approach for deriving *species* based on all five access patterns for pointer based code.

2. The second test demonstrated that our pointer analysis infrastructure can handle the system defined pointer constructs used for performing memory accesses in PAINt and the Data Path algorithms.

3. The third test demonstrated that our pointer analysis infrastructure can handle the structure based pointer constructs used for performing memory accesses in PAINt and the Data Path algorithms.

6 Limitations

In the following section we discuss the limitations of our pointer analysis infrastructure. The pointer analysis infrastructure has been designed keeping in mind the requirements of the ASET + BONES approach. We have treated pointer constructs that are used for performing memory accesses in PAINt and the Data Path algorithms. We do not treat general pointers as part of our work.

We found the following limitations in our work.

1. Our pass is designed to handle structure pointers and structure members as pointers. But the approach in (algorithm 6) of back tracking arguments fails if structures are passes by value.
2. We have restricted ourselves at treating single level of pointers (pointer to variables). We do not treat hierarchical pointers (pointer to pointer).
3. We have implemented a flow insensitive pass. We do not produce a solution based on the control flow information of the code. So our approach is less accurate and is prone to false positives.
4. We rely on GEP instructions in LLVM for identifying pointer constructs. GEP does not handle unions. As a result our implementation is insensitive to unions in source code.
5. We identify pointer based memory accesses if and only if they are performed as array subscripts. We do not treat *value at address operator* in our work. So a memory access at an address should be done as address[offset] rather than *(address + offset).

Following are the limitations imposed by ASET.

1. ASET cannot perform constant propagation. As s result the *block* to be annotated must be free from constant values or there values must be suitable propagated.
2. Memory accesses in ASET are classified based on a fixed set of *formatting functions*. The current set of *formatting functions* are inadequate in classifying complex memory operation performed using structure members in real world programs.

7 Conclusion

Océ wants to exploring the feasibility of a semi-automatic code parallelizing tool to reduce the overall effort otherwise needed in manually parallelizing sequential code.

It was observed that tools like Par4all and ASET + BONES offer limited or no support for pointers due to lack of alias analysis capability. In this thesis we have developed a pointer analysis infrastructure to enables the ASET to handle pointer constructs used for performing memory accesses in PAINt and the Data Path algorithms.

Our results show that the developed infrastructure can handle the pointer constructs used for performing memory accesses in PAINt and the Data Path algorithms. We have integrated the developed infrastructure into ASET and have tested the ASET + BONES approach for parallelizing these algorithm.

The ASET + BONES approach is under development and has several limitations. Because of these limitations it is currently unsuitable for parallelizing real world algorithms. Although the approach is promising and can be developed as a full fledged semi automatic code parallelizing tool in the future.

8 Future Work

In this thesis we have developed a pointer analysis infrastructure to enable pointer support for ASET + BONES approach. The pointer analysis infrastructure is under constant development and can be considerably improved in the future.

1. In future we would like to incorporate support for additional constructs to increase the applicability of our pointer analysis infrastructure.
2. The theory behind ASET and BONES can be extended to handle real world programs composed of pointers and structures.
3. Bones is now interfaced with A-Darwin a *species* generation tool based on *theory of interprocedural analysis of array regions* [13]. Our pointer analysis infrastructure can be explored for getting pointer support into A-Darwin + BONES approach.

References

[1] http://www.freescale.com/webapp/sps/site/overview.jsp?code=PRDCT_LONGEVITY_HM.

[2] http://llvm.org/.

[3] http://llvm.org/releases/2.3/docs/GetElementPtr.html.

[4] http://llvm.org/docs/SourceLevelDebugging.html#llvm-dbg-declare.

[5] http://llvm.org/docs/SourceLevelDebugging.html#llvm-dbg-value.

[6] http://llvm.org/docs/doxygen/html/BasicAliasAnalysis_8cpp.html.

[7] http://llvm.org/docs/AliasAnalysis.html.

[8] http://llvm.org/docs/doxygen/html/GlobalsModRef_8cpp_source.html.

[9] Mehdi Amini, Batrice Creusillet, Stphanie Even, Ronan Keryell, Onil Goubier, Janice Onanian Mcmahon Serge Guelton, Franois Xavier Pasquier, Grgoire Pean, and Pierre Villion. Par4all : From convex array regions to heterogeneous computing. In *In 2nd International Workshop on Polyhedral Compilation Techniques (IMPACT 2012)*, 2012.

[10] Mehdi Amini, Batrice Creusillet, Onil Goubier, Serge Guelton, Ronan Keryell, Janice Onanian-McMahon, and Grgoire Pan. Par4all user guide. January 2014.

[11] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[12] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Béatrice Creusillet and Francois Irigoin. Interprocedural array region analyses. *Int. J. Parallel Program.*, 24(6):513–546, December 1996.

[14] P. Custers. Algorithmic species: Classifying program code for parallel computing. 2012.

[15] Michael Hind. Pointer analysis: Havent we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[16] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[17] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. A script-based autotuning compiler system to generate high-performance cuda code. *ACM Trans. Archit. Code Optim.*, 9(4):31:1–31:25, January 2013.

[18] X. Kong, D. Klappholz, and K. Psarris. The i test: An improved dependence test for automatic parallelization and vectorization. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):342–349, July 1991.

[19] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[20] Paul E. Mckenney and Maged M. Michael. Is parallel programming hard, and if so, why?, 2009.

[21] C. Nugteren and H. Corporaal. A modular and parameterisable classification of algorithms. Electronics Systems Group,Eindhoven University of Technology, July 2011.

[22] C. Nugteren, H. Corporaal, and B. Mesman. Skeleton-based automatic parallelization of image processing algorithms for gpus. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 25–32, July 2011.

[23] Cedric Nugteren and Henk Corporaal. Introducing 'bones': A parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM.

[24] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Trans. Archit. Code Optim.*, 9(4):40:1–40:25, January 2013.

[25] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Automatic skeleton-based compilation through integration with an algorithm classification. *APPT13: Advanced Parallel Processing Technology*, 2013.

[26] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[27] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[28] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. *Second International Workshop on Polyhedral Compilation Techniques (IMPACT12)*, January 2013.