A Study of the Aliasing Problem in Relation to the Automatic Determination of Parallelism

Howard Bowman and Gordon S. Blair

Abstract (Contents and Objectives). The potential presence of aliasing in source programs presents a significant hindrance to automatic parallelization. There are many ways in which aliasing can arise in programs, and various categorisations have been made by researchers interested in programming methodologies and especially the design of programming languages more suitable for the programming of reliable and verifiable software. It is the view of the authors that these categorisations are insufficient when applied to the automatic parallelization domain. This paper presents a new categorisation of aliasing which is specifically tailored to automatic parallelization. This categorisation is shown to be valuable in defining solution sets to aliasing for automatic parallelization.

Abstract (Bibliography 1). Howard Bowman is a lecturer at the University of Kent at Canterbury. He received his Ph.D. from the University of Lancaster in 1991 for a thesis titled 'A Unifying Language Study of Automatic Parallelization'. His current research in this area focusses on the release of parallelism from the University of Kent’s functional programming language: Miranda. He has published extensively in his field of research. His current research interests include the development of Automatic Parallelization Techniques and the application of formal techniques to the development of Distributed Multimedia Systems.

Abstract (Bibliography 2). Gordon S. Blair completed his Ph.D. in Computer Science in 1983 at Strathclyde University. In 1986, he became a lecturer in the Department of Computing at Lancaster, and became a senior lecturer in 1990. He has published over seventy papers in his field and is co-author of two books (one on the Unix operating system and the other on Object Oriented systems). His current research interests include distributed multimedia computing, object-oriented systems and the automatic parallelization of sequential programming languages.

Keywords. Automatic Parallelization, Aliasing, Imperative Languages, Dataflow Languages, Functional Languages.
Abstract (Contents and Objectives). The potential presence of aliasing in source programs presents a significant hindrance to automatic parallelization. There are many ways in which aliasing can arise in programs, and various categorisations have been made by researchers interested in programming methodologies and especially the design of programming languages more suitable for the programming of reliable and verifiable software. It is the view of the authors that these categorisations are insufficient when applied to the automatic parallelization domain. This paper presents a new categorisation of aliasing which is specifically tailored to automatic parallelization. This categorisation is shown to be valuable in defining solution sets to aliasing for automatic parallelization.

1. Introduction

Automatic parallelization techniques [Allen85] are tools which determine the concurrency present in source programs without programmer involvement [Bowman93]. There have been certain notable successes in the field of automatic parallelization. For example progress has been made in the areas of vectorization [Allen87], concurrentization (also called simply parallelization) for either shared memory multiprocessors [Padua86] or distributed memory multicomputers [Hiranandani92], dataflow [Gao93] and reduction [Treleaven82]. However, the issue of aliasing [Ghezzi82] can be identified as a significant hindrance to this progress. Aliasing is defined as follows:-

**definition 1**

**Aliasing**

Two or more distinct identifiers are said to be **aliases** if they name the same data object in the same scope of the program.

There are many ways in which aliasing can arise in programs and researchers have documented most of these ways [Ghezzi82]. These categorisations have been made by researchers interested in programming methodologies and especially the design of sequential languages more suitable for the programming of reliable and verifiable software [Popek77]. For workers in this field, aliasing is seen as leading to confusion for programmers, program analysers and verifiers and thus hindering the generation of reliable code.

Workers in the field of automatic parallelization have discovered that aliasing presents a similar hindrance to the determination of concurrency. The following example is offered as an initial indication of the problem aliasing causes automatic parallelization.

**example**

Given the procedure,

```plaintext
PROCEDURE REVERSECOPY(VAR A, B:ARRAY[1..10] OF REAL);
VAR
  i:INTEGER;
BEGIN
  FOR i:=1 TO 10 DO A[i]:=B[11-i]
END;
```

It would seem that an invocation of this procedure (such as `REVERSECOPY(X, Y)`) would enable the loop to be parallelized and all iterations performed concurrently. Unfortunately a preventing case exists. If the procedure is invoked
with the call, \texttt{REVERSECOPY(z,z)}, the formal parameters \(A\) and \(B\) become aliases of one another, data dependencies develop between certain iterations of the loop and the parallelization is no longer valid.

The nature of the problem is such that this one special case may well prevent parallelization of the loop for all other cases.

We view the aliasing categorizations developed within the programming methodologies community as insufficient when applied to automatic parallelization, as aliasing manifests itself differently in this new problem domain. It is therefore the objective of this paper to highlight a new categorisation of aliasing, which we have developed specifically with regard to automatic parallelization. The next section describes our new categorisation of aliasing. We will then, in order to show the value of this new work, apply our categorization to the field of automatic parallelization. This will entail relating our categorisation to the aliasing solutions that previous researchers into automatic parallelization have developed. It will be demonstrated that classes of solutions can be more easily uncovered from our categorisation.

2. **A New Categorisation of Aliasing**

There are two basic means of division within our categorisation:-

* Firstly, there is a division according to the granularity of the underlying data object, this divides aliasing into two separate forms: \textit{whole object} and \textit{sub-object aliasing}. This is really a generalized categorisation of the circumstances by which aliasing arises.

* Then, secondly, there is a division according to the mechanism underlying the appearance of these aliases. This categorisation divides the problem into \textit{value dependent} and \textit{non-value dependent} aliasing. The distinction here is between aliasing which arises from the inability to determine the value of program identifiers and aliasing which is simply due to the repetition of program identifiers.

Section 2.1 introduces some basic terminology. Following this, section 2.2 will survey examples of whole object aliasing and then section 2.3 will survey examples of sub-object aliasing. In the latter two sections the mechanisms by which aliasing arises in each example will be highlighted by reference to value and non-value dependent aliasing.

### 2.1 Definitions

The following are basic definitions required for the analysis in this section:-

**Whole and Sub-object Identifiers**

An identifier which names the whole of a data object is said to be a \textit{whole object identifier}. In contrast, an identifier which names a sub data object (an array element or a field of a record) is said to be a \textit{sub-object identifier}.

**Example**

Given the declarations,

```plaintext
TYPE
  POINTER = ^CELL;
CELL = RECORD
    VAL : INTEGER;
    NEXT : POINTER;
END;
```

1073
H. Bowman and G.S. Blair

the identifiers i, A, Q1 and Q2 are whole object identifiers. In contrast, identifiers such as:
A[9], A[i], Q1^.VAL, and Q2^.NEXT^.NEXT

are sub-object identifiers.

note: We see composite names such as A[i] and telephone_number[5].NUMBER as identifiers even though they are themselves composed of identifiers. Some authors distinguish these, calling such objects designators or descriptors.

2.2. Whole Object Aliasing

Whole object aliasing is defined as follows:

definition 3
Whole Object Aliasing
If aliased identifiers X and Y (this definition is trivially extendible to more than two aliased identifiers) are whole object identifiers then they are called whole object aliases.

Whole object aliasing can arise in two basic ways, i.e.:

(i) Explicitly Defined Whole Object Aliasing

(ii) Whole Object Aliasing due to Parameter Passing

The two types are considered separately.

2.2.1 Explicitly Defined Whole Object Aliasing

The FORTRAN equivalent construct [Balfour79] enables aliases to be explicitly defined.

example
The following FORTRAN equivalence statement will create two sets of aliased identifiers {A,B,C} and {x,y}:

    INTEGER     x,y
    REAL       A(20), B(20), C(20)

    EQUIVALENCE (A,B,C), (x,y)

Thus, as a result of this statement A, B and C and x and y become interchangeable in the relevant program block. Since, A, B and C and x and y are whole object identifiers, they become whole object aliases. In addition, the mechanism through which these aliases arise is non-value dependent.

Such explicit specification of aliases only arises in a small number of languages and furthermore has been largely discredited. For these reasons only the above example will be presented. Veen [Veen85] contains an extensive investigation of explicitly defined aliasing, as found in the research language SUMMER.

note: It should also be highlighted that sub-object aliasing can also be explicitly generated.

2.2.2 Whole Object Aliasing due to Parameter Passing

Specific Examples. The first, introductory, example presented at the start of this paper falls directly into this category of aliasing. Variables A and B, in the example, become aliases when they name the whole array data object identified by Z.
addition, this aliasing example is non-value dependent since the aliases arise from the repetition of the identifier \( z \) in the actual parameter list and are not related to the values of any particular identifier.

**Scalar whole object aliasing** can also be found:-

**example**

Given the rather trivial procedure definition:-

```pascal
PROCEDURE SIMPLEASSIGNMENT(VAR I,Y,Z:INTEGER);
BEGIN
    Y:=SQRT(999);
    I:=Z+28
END
```

The procedure invocation,

```
SIMPLEASSIGNMENT(B,X,X)
```

causes the formal parameters \( Y \) and \( Z \) to become aliases of one another and data dependencies appear between the procedure's assignment statements. Thus, the validity of concurrently executing the two statements comprising the procedures body is uncertain.

More complicated forms of scalar whole object aliasing can be found. These arise from array element actual parameter passing:-

**example**

The `SIMPLEASSIGNMENT` procedure presented in the previous example could be invoked with any of the following calls:

```
SIMPLEASSIGNMENT(B,A[2],A[2]);
SIMPLEASSIGNMENT(B,A[i],A[i]);
SIMPLEASSIGNMENT(B,A[i],A[2]);
```

or,

```
SIMPLEASSIGNMENT(B,A[i],A[j]);
```

where \( A \) is an array of 10 integers say.

Notice that, although all these examples involve sub-objects at the actual parameter level, it is whole data objects at the formal parameter level which become aliases. Therefore, these are examples of whole object aliasing.

The first two calls are non-value dependent because, although the first contains literal values and the second contains non-determinate identifiers, neither require any evaluation to demonstrate they are aliases. The alias arises directly from the repetition of identifier names. The last two calls are value dependent though as the first only produces an alias if \( i=2 \) and the second if \( i=j \).

**Whole object aliasing due to the passing of pointer parameters** can also arise:-

**example**

Consider the following procedure declaration, which assumes two linear linked lists have been built using the pointer type declaration presented earlier.

```pascal
PROCEDURE count_front (x : INTEGER; VAR PNT1,PNT2 : POINTER);
VAR
    CNT1, CNT2 : INTEGER;
BEGIN
```

```pascal
    // Body of count_front procedure
```

This procedure takes two linked lists (the start of which is pointed to by PNT1 respectively PNT2) and counts the front of the lists before a cell containing x is reached. Finally, the procedure outputs these results in a writeln statement.

Notice that the simple way chosen to perform this task is to repeat the while loop segment that counts the list elements.

![Dependence Graph of count_front](image)

Fig. 1: Dependence Graph of count_front

It would seem that the two instances of the while loop are independent and can be executed concurrently, as expressed in the dependence graph of the procedure shown in figure 1.

However, this parallel execution would be invalidated if PNT1 and PNT2 were whole object aliases of one another, as would result from the invocation:

\[
\text{count_front}(y, P, P);
\]

This is because dependencies arise between the two while loops, i.e. between statements altering and using the two pointers (PNT1 and PNT2).

Pointer aliasing can also arise through call by value parameter passing, but this type of aliasing will be investigated in the sub-object category. In this example though the whole objects PNT1 and PNT2 become aliases. In addition, the procedure invocation \(\text{count_front}(y, P, P)\) is clearly an example of non-value dependent aliasing. However, the procedure call \(\text{count_front}(y, P, Q)\) (where \(P\) and \(Q\) are pointers) produces a value dependent point of conflict as the formal parameters \(P_1\) and \(P_2\) would become aliases only if \(P = Q\). Thus, this form of whole object aliasing can result from a value dependent or non-value dependent mechanism.

Formal Categorization. An exact categorisation can be given of when whole object aliases due to call by reference parameter passing arise. Firstly, the following lemma categorises non-value dependent whole object aliasing.
H. Bowman and G.S. Blair

**Lemma 1** Whole Object Non-Value Dependent Aliasing (CASE 1)

Given a procedure P with by reference formal parameters X and Y. For any call to P which has an identifier Z appearing twice in the actual parameter list such that Z corresponds once to X and once to Y, X and Y will be whole object non-value dependent aliases.

Thus, identifiers can be determined to be whole object non-value dependent aliases independently of their type, and there exists a certain lexical format from which they arise.

The next lemma categorises whole object value dependent aliasing:

**Lemma 2** Whole Object Value Dependent Aliasing (CASE 1)

Given a procedure P with by reference formal parameters X and Y and a call to P which contains distinct actual parameters A and B corresponding to X and Y respectively. Then on this invocation of the procedure P, X and Y will become whole object value dependent aliases if either,

- X, Y, A and B are pointer identifiers and A=B.
- A and B are array elements with subscripts i and j and i=j.

**Refinement for Global Variables.** There is one other possible means by which whole object aliases may arise and this is through the combination of call by reference parameter passing and global variables.

**Example**

Given the header,

```
PROCEDURE Q(VAR X : ..........);
```

of a procedure which contains references to a global variable Y, then the procedure invocation, Q(Y), will cause the formal parameter X and the global variable Y to become whole object aliases.

Both non-value and value dependent whole object aliasing can be generated with global variables. Thus, extensions to both the above lemmas must be made to incorporate this new aliasing type.

**Lemma 3** Whole Object Non-Value Dependent Aliasing (CASE 2)

Given a procedure P with by reference formal parameter X and a global variable Y. For any call to P such that the variable Y appears as an actual parameter corresponding to X, X and Y will be whole object non-value dependent aliases.

**Lemma 4** Whole Object Value Dependent Aliasing (CASE 2)

Given a procedure P with by reference formal parameter X and a global variable Y, and a call to P which contains an actual parameter A corresponding to X. Then on this invocation of the procedure P, X and Y will become whole object value dependent aliases if either,

- X, Y and A are pointer identifiers and A=Y.
- A and Y are array elements with subscripts i and j and i=j.

2.3 Sub-object Aliasing

The second category has not been as extensively considered as the previous form of aliasing. This is because sub-object aliasing only arises as a by-product of the utilization of concurrent program execution paths.
**Sub-object Aliasing**

Given aliased identifiers $X$ and $Y$ (this definition is also trivially extendible to more than two aliased identifiers), if $X$ and $Y$ name sub data objects then they are called **sub-object aliases**.

Unlike whole object aliasing, this form of aliasing is not caused by the call by reference passing of procedure parameters, but is independent of subprogram control flow. The following is an initial example of sub-object aliasing, in which array elements become aliased.

**Example**

Consider the code segment,

(S1) \[ \text{A}[i] := 12; \]
(S2) \[ X := \text{A}[j]; \]

Concurrent execution of these two statements is uncertain since, if $i = j$ the second statement is data dependent upon the first. In terms of the categorisation, the above eventuality coincides with $\text{A}[i]$ and $\text{A}[j]$ becoming aliases. The aliasing is value dependent since the values of $i$ and $j$ are required in order to determine aliasing arises.

It should be noted that this form of aliasing results directly from the enforcement of parallelism in program execution paths. This is because, $\text{A}[i]$ and $\text{A}[j]$ only name the same data object at the same time if the two statements S1 and S2 are being executed concurrently. Under sequential execution models, this example is not aliasing, so it has not been considered by workers in the field of programming methodologies.

**Sub-object aliasing due to pointer data structures** is demonstrated in the following example:

**Example**

Given a linked list with pointers $P1$ and $P2$ as earlier declared, then consider the following two statements:

(S1) \[ P1^.VAL := 12; \]
(S2) \[ X := P2^.VAL; \]

The aliasing problem in this example manifests itself in an equivalent way to the array element aliasing example before last. If S1 and S2 are seen to be data independent and are executed concurrently then if $P1 = P2, P1^.VAL$ and $P2^.VAL$ become aliases and the concurrent execution of the two statements collides. The existence of sub-object aliases in this example is value dependent upon $P1$ and $P2$.

Pointer sub-object aliasing is complicated by procedure invocation. This is because parameter passing can alter the naming of aliased identifiers:

**Example**

Consider the `count_front` procedure presented in section 2.2.2.1 with call by value rather than call by reference parameter passing, i.e. the procedure header,

```
PROCEDURE count_front (x : INTEGER; PNT1,PNT2 : POINTER);
```

and invoke the procedure with the call,

```
count_front(x,Q1,Q2);
```

where $Q1 = Q2$. This last condition means that there is pointer sub-object conflict in the calling block and the invocation has the effect of redirecting this conflict into the procedure block. In the procedure, any two references to the linked list
differing only in the pointer name (PNT1 or PNT2) will be aliases, for example PNT1^.NEXT and PNT2^.NEXT would become aliases through this invocation.

Thus, value parameter passing can cause the redirection of points of conflict to different identifiers, in different program locations.

The following lemmas characterise the occurrence of sub data object aliasing:

**Lemma 5**

*Points of Sub-object Conflict*

Given two statements S1 and S2 with sub data object identifiers X and Y scheduled to be concurrently executed, then a point of sub-object conflict exists between the two statements if either,

- X and Y are pointer sub-object data objects with pointer identifiers P and Q such that P and Q are declared as pointing to the same structure (i.e. could be equal).

or

- X and Y are elements of the same array (i.e. are of the form Z[i] and Z[j]) and the subscripts i and j are not both literals.

**Lemma 6**

*Sub-object Aliasing*

If two statements satisfy the above lemma. Such that in the first case P=Q and in the second i=j, then X and Y are sub-object aliases.

**Lemma 7**

Sub-object aliasing is always value dependent.

### 2.4 Summary

This section has categorised aliasing into two basic types: whole object and sub-object aliasing. This is a categorisation in terms of the coarseness of the underlying aliased data object.

- **Whole Object Aliasing**
  
  Whole object aliasing arises through either specific language constructs which are designed to explicitly create aliases or, more interestingly, from side-effects resulting from call by reference parameter passing across subroutine boundaries. It is the most widely understood and documented of the aliasing types. In fact, whole object aliasing has by many workers been seen as aliasing in its entirety.

- **Sub-object Aliasing**
  
  Sub-object aliasing occurs independently of program control flow constructs (such as mechanisms of procedure invocation) and can appear free in any block of sequential code. The presence of sub-object aliasing can be directly linked to the concurrent execution of statements assigning values to structured data objects (array elements or record fields).

This section, has also investigated a second categorisation which is intended to divide in terms of the mechanisms underlying the appearance of aliases. The two mechanisms highlighted are value and non-value dependent aliasing. It was discovered that whole object aliasing can be either value or non-value dependent, while sub-object aliasing is specifically value dependent.
3. Applying the Categorisation

We propose that our new categorisation not only highlights more completely the full range of aliasing types, but also facilities the location of particular classes of aliasing solution. In order to demonstrate this view the following two sections highlight the solution sets which we believe arise naturally from our categorisation. Throughout we will relate these solution sets to existing aliasing solutions.

3.1 Solutions to Whole Object Aliasing

3.1.1 Solutions in Imperative Languages

As indicated in section 2.2 whole object aliasing in imperative languages can be both value and non-value dependent. Since comparison of identifier values is not required, the resolution of non-value dependent whole object aliasing in imperative languages simply entails compile time searching of programs for code segments satisfying either of the two lemmas, 1 and 3, and forcing reprogramming or sequential execution of such segments ([Bowman92] contains such an approach).

Value dependent whole object aliasing presents a somewhat more difficult problem, since the value of data objects is required in order to resolve such points of conflict. Thus, compile time analysis cannot be relied upon to definitely resolve this category of whole object aliasing. Whole object value dependent aliasing has one significant advantage over arbitrary sub-object aliasing though, i.e. it only arises from procedure invocation and not in any arbitrary section of code.

Solutions to whole object aliasing within existing imperative languages can then only resolve non value dependence whole object aliasing easily (at compile time). However, the presence of whole object aliasing of any form can be seen to be inherently dangerous (for all the reasons highlighted by the software methodologies community). Therefore, many new language designers of the 70's and 80's have sought to ban whole object aliasing altogether. Adapted imperative languages, such as Euclid ([Popek77]) have been able to ban whole object aliasing within an imperative framework. Interestingly though, although the compiler can prevent any non-value dependent whole object aliasing, the system is forced to include run time checking, in the form of legality assertions, for value dependent whole object aliasing.

3.1.2 Solutions in Non-Imperative Languages

Functional languages ([Field89]) and Dataflow languages ([Ackermann82]) have been able to banish whole object aliasing in it's entirety without resorting to any run time checks. Functional languages avoid all classes of aliasing by employing a 'by value' expression based data paradigm, while dataflow languages have banned whole object aliasing as a by product of the incorporation of a single assignment rule for both scalar and structured data ([Ackermann82]). This is one of the major reasons why these two classes of languages have been seen as so suitable for automatic parallelization ([Glauert82] [Keenaway84]).

3.1.3 Summary

Whole object aliasing can then not only be seen as complicating the automatic determination of concurrency (especially the non-value dependent variety, which cannot be fully analysed at compile time), but also as a hindrance to unambiguous, well structured programming. It is therefore important that future language designs address this issue and remove all features which can lead to whole object aliasing.
4.2 Solutions to Sub-object Aliasing

### 4.2.1 Solutions in Imperative Languages

The question arises whether the sub-object aliasing in imperative languages can be solved in the same way as whole object aliasing was (by banning it in some way). This is unfortunately not possible because there is no sense to which the occurrence of sub-object aliasing corresponds to a bad programming style; rather it arises out of fundamental notions of imperative data structuring. Sub-object aliasing arises as a result of a lack of information at compile time; array subscript values and pointer values need to be known in order to determine valid parallel execution paths.

The most extensive consideration of sub-object aliasing in traditional imperative languages has been carried out by researchers in to vectorization techniques [Allen87] and concurrentization techniques [Padua86]. Workers in these fields have developed complex techniques for analysing array sub-object aliasing at compile time. However, these techniques will only ever successfully resolve a subset of all the points of sub-object conflict at compile time.

**example**

The following sub-object conflicting code segment,

```plaintext
read (j);
FOR i:=1 to m DO A[i] := A[i+j];
```

cannot be resolved at compile time as it is directly dependent upon the value of j, which is input at run time.

Thus, since it is unlikely that all subscript and pointer values can be extrapolated at compile time, in order to fully analyse sub-object aliasing in existing imperative languages run time checking would have to be employed.

One legacy of sequential computing is that dynamic analysis is seen as almost totally unacceptable. The potential redundancy available in parallel computing though to some extent alters this. This is especially the case in the field of automatic parallelization, which has to date been constrained by the amount of parallelism that can be obtained from sequential algorithms (via compile time analysis) and not by the potential parallelism of the hardware. An algorithm which scheduled a run time checker to redundant processing agents could be incorporated, with the run time checker working its way systematically through resolving conflict points during the programs execution.

### 4.2.2 Solutions in Non-imperative Languages

New languages developed for automatic parallelization have also tackled sub-object aliasing. Application of the single assignment principle to structured data [Ackermann82] prevents sub-object aliasing arising in dataflow languages. Complex data structures are seen as immutable and indivisible. Thus, the same array cannot be assigned to more than once. While this is conceptually an elegant alternative, it has an effect on the type and amount of concurrency uncovered.

**example**

Consider the following equivalent pascal and val [McGraw82] (an archetypal dataflow language) segments of array manipulating code,

**pascal segment**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>A[i]:=12;</td>
</tr>
<tr>
<td>S2</td>
<td>X:=A[j];</td>
</tr>
<tr>
<td>S3</td>
<td>A[k]:=14;</td>
</tr>
</tbody>
</table>

**val segment**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>B:=A[i:12];</td>
</tr>
<tr>
<td>S2</td>
<td>X:=B[j];</td>
</tr>
<tr>
<td>S3</td>
<td>C:=B[k:14];</td>
</tr>
</tbody>
</table>
where array assignment in the val segment has the effect of creating a new array with a new data value at the specified location and unchanged at all other locations. They produce the following data dependence graphs:

Graph of pascal segment

\[
\Sigma_1 \xrightarrow{(i,j)} \Sigma_2 \xrightarrow{(j,k)} \Sigma_3
\]

Graph of val segment

\[
\Sigma_1 \xrightarrow{(i,j)} \Sigma_2 \xrightarrow{(j,k)} \Sigma_3
\]

**Fig. 2: Data Dependencies for Code Segments containing Sub-object Aliasing**

In this diagram, solid arrows indicate standard data dependencies and broken arrows potential data dependencies, which are conditional upon the value of the bracketed identifiers.

The indivisibility rule for structured data employed in val reduces the dependencies generated and thus releases added concurrency over the worst case dependency graph for the pascal segment. However, if \(i, j\) and \(k\) are not equal, no dependencies would arise at all in the Pascal segment.

Thus, the single assignment principle prevents the concurrent execution paths from which sub-object aliasing would arise from being sought. Therefore, dataflow languages are an improvement on the worst case sub-object aliasing situation in unconstrained imperative languages, but prevent the best case analysis of sub-object aliasing.

The alternative dataflow array structure proposed by Arvind (I-structures [Arvind89]) seeks to increase the level of concurrency made available from data structures. In I-structures immutability of complete data structures is replaced by immutability of structure elements. Thus, multiple assignments can be made to a particular I-structure as long as multiple assignments are not made to specific elements of the structure (i.e. sub-object aliasing is still banned).

This weakening of the single assignment property seems initially to be highly desirable however, conflict between this new principle and the functionality properties of single assignment languages does arise. In particular, occurrences of multiple assignments to specific I-structure elements causes a run time failure of the program. Thus, it is left to the programmer to explicitly prevent such sub-object aliasing situations from arising; this is far from simple to guarantee.

4.2.3 Summary

To summarise then, it should be apparent that while new languages can ban whole object aliasing, restriction of sub-object aliasing may reduce conflict (and simplify concurrency analysis), but it will also reduce the parallelism made available.

4. Conclusions

This paper has presented a new categorisation of the aliasing problem. This categorisation has been motivated by the problems aliasing cause automatic parallelization. These problems arise out of the need to be conservative in automatic parallelization. In particular, they result in added program dependencies and loss of concurrency in order to prevent invalid parallelization in the rare case in which aliasing occurs. The new categorisations presented here seek to break aliasing down in a manner that will aid the determination of solutions.
Solutions that researchers have found to aliasing were then briefly reviewed. This survey related the different solutions using the new categorisation presented in this paper. Two major results can be discerned from this survey:

- **Aliasing Solutions in Existing Imperative Languages**

  Solutions to aliasing within existing imperative languages can be divided into aliasing that can be resolved at compile time and that which cannot. It turns out that the categorisation into non-value and value dependent aliasing corresponds directly to this division and categorises exactly the aliasing which is run time dependent. We suggest that the topic of dynamic analysis is particularly worthy of future investigation.

- **Aliasing Solutions in New Languages**

  The categorisation into whole and sub-object aliasing indicates how new languages not exhibiting aliasing should be designed. This is because whole object aliasing arises specifically from poor and ambiguous programming and can thus be simply banned from new languages. However, sub-object aliasing has no such implication and can not be naively solved by enforcement of constant data structuring paradigms. Thus, new languages for automatic parallelization have been forced to re-evaluate traditional imperative data structuring and include constrictive rules to prevent interference across statements manipulating structured data.

  It is suggested that the classification presented in this paper will aid the resolution of the aliasing problem in automatic parallelization, by providing a comprehensive framework for aliasing analysis. The development of aliasing solutions using this categorisation can be found in [Bowman92] and [Bowman93].

**Acknowledgements.** This work was supported by the British Science and Engineering Research Council.

**References**


