A Survey of Data Dependence Analysis Techniques for Automated Parallelization

by
Joseph C Libby and Kenneth B. Kent

TR07-188, December 7, 2007

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
www: http://www.cs.unb.ca

Abstract

Finding parallelism that exists in a software program depends a great deal on determining the dependencies that exist between statements that exist in that program. Instructions that are found to be independent of one another can be executed in parallel with one another with the hope of increasing the execution speed of a software program. Determining dependence relationships between statements in software programs, however, is not an easy task. There exist many different methods for determining dependence relations, filling different requirements from speed to accuracy. This paper will discuss a number of these techniques, detailing several different classes of dependence analysis techniques as well as several different techniques within these classes.

Table of Contents

1. Introduction

In recent years the realms of software development and hardware design have began to converge. Hardware design, once the domain of electrical engineers, has recently become more accessible to software developers through the rise of hardware description languages such as Handel-C and VHDL and reconfigurable hardware such as Field Programmable Gate Arrays. This shift allows software developers to leverage the potential of building customized hardware circuits which can improve the performance of their software systems. This move to high level hardware design does not come without problems. The hardware systems used to implement these high level hardware designs are well suited for parallel execution. Hardware designs do not need to leverage this potential parallelism but in doing so sacrifice much of the potential for increasing the performance of the system. In order to take advantage of this potential, however, the hardware designer needs to be aware of how to best parallelize the actions performed by their hardware systems. This can pose a problem to hardware designers using a high level description language such as Handel-C. These description languages enable developers to easily create a hardware system by using a language that is similar to a software programming language. In doing so, however, many developers fail to use the full potential of the hardware platform they are targeting.

Even with the rise in parallel computation, finding developers that are experienced with parallel programming is challenging. Translating this lack of parallel programming skills to software developers who have moved into high level hardware design leads to an even smaller number of developers who are capable of creating parallelized hardware systems. In order for these developers to properly leverage the technology available to

them it is important to consider the creation of a tool set that is capable of automatically detecting and exploiting parallelism in a hardware design. The purpose of this paper is to investigate a number of different techniques that are currently being used for automated parallelism detection as well as any background material that is necessary to implement these techniques.

2. Dependency Analysis

One of the key requirements for determining the presence of parallelizable programming code, whether it hardware or software, is determining the data and control dependencies that exist between the different statements or actions in the source file. This section will discuss the types of dependencies that arise in computer programs and then discuss several techniques that have been proposed for determining where dependencies exist in a given program.

2.1 Control Dependency

The first type of dependency that will be discussed is the control dependency. Control dependencies arise in almost every software program written due to the inclusion of control structures such as conditional branching. One method for representing the control dependencies for a given program is the Control Flow Graph (CFG). Figure 1 shows an example of a control flow graph based on the pseudocode presented in Example 1. A control flow graph is comprised of nodes which form a graph of all paths that can be traversed through a program. Each node in a CFG represents a basic block which is a sequential piece of code or jump target. Directed edges are used in a CFG to denote the target of branches and sequential execution.

```
Array x[10]
for i <- 0 to 10
     if i mod 2 = 0 then
          x[i] = x[i]+i
     else
          x[i] = x[i]+i/2
     end if
end for
```

Example 1: Example Pseudocode

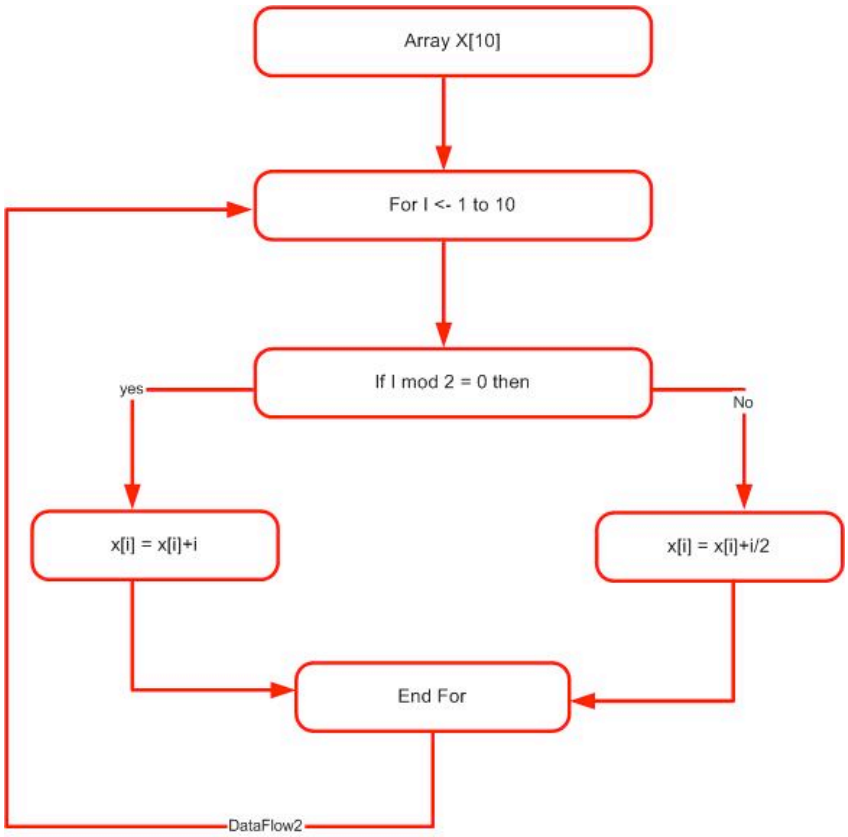

Figure 1: Example of a Control Flow Graph

Determining control dependencies is vital in the search for parallelism as some basic blocks of code may not contain a data dependency but still may not be executable in parallel due to a control dependency. Unlike data dependencies, control dependencies are much easier to resolve from program source code. Control flow graphs can be generated without any knowledge of the data that is being used in the system and thus require only simple parsing of the source file to locate branch targets [1].

2.2 Data Dependency

As discussed in Section 2.1 data dependencies are not as trivial to resolve as control dependencies. In order to discuss data dependencies it is important to first discuss the different types of data dependencies that can arise in programs. Once this is complete several different techniques for handling data dependencies will be presented.

2.2.1 True Data Dependencies

A true data dependency arises in a program if one operation in the program creates a value that is used by another operation in the program. True data dependencies are also known as Read After Write or RAW dependencies. This type of dependency forces execution to take place in such a way that source values are created before they are needed by subsequent operations. True data dependencies cannot be removed from a program without extensively rewriting the program using different techniques.

2.2.2 Storage Dependencies

When writing a software program little thought is normally given to the underlying architecture that a program will be executing on. Variable names are used to abstract away the underlying storage structures that are used by the hardware such as registers and

other types of memory. In order for a program to execute on physical hardware it is necessary to be able to make use of limited storage resources during execution. This limited storage capability leads to storage dependencies where execution of some instructions must be delayed until the storage location for the result is no longer needed by previous instructions. Storage dependencies can be broken down into two separate categories: Write After Read (WAR) and Write After Write (WAW). Write After Read dependencies arise when an instruction needs to write a value to a storage location but the target storage location needs to be read by a previous instruction. This means that the instruction that needs to perform the write must now wait until the read has completed before performing its write. Write After Write dependencies occur when subsequent instructions attempt to write to the same storage location. One instruction must now wait until the other instruction has completed writing to storage before it can perform its write to storage.

Unlike true data dependencies, however, storage dependencies can be removed by using a technique known as register or storage renaming. This technique is applied by using temporary storage locations in cases where WAR and WAW storage dependencies are found. This allows instructions that previously were blocked by the storage dependency to execute and then finalize their data storage at a later time.

2.2.3 Data Flow Graph

One common method for representing the dependencies in a software program is through a Data Flow Graph (DFG). A DFG utilizes the set of dependencies generated by different dependency testing techniques to construct a graph that illustrates the dependencies that exist in a program. A DFG is very similar to the CFG's discussed

earlier in that they are a directed graph with each node representing some statement in a software program and each edge representing a dependency between two nodes. Figure 2 shows an example DFG based on the example pseudocode shown in Example 1.

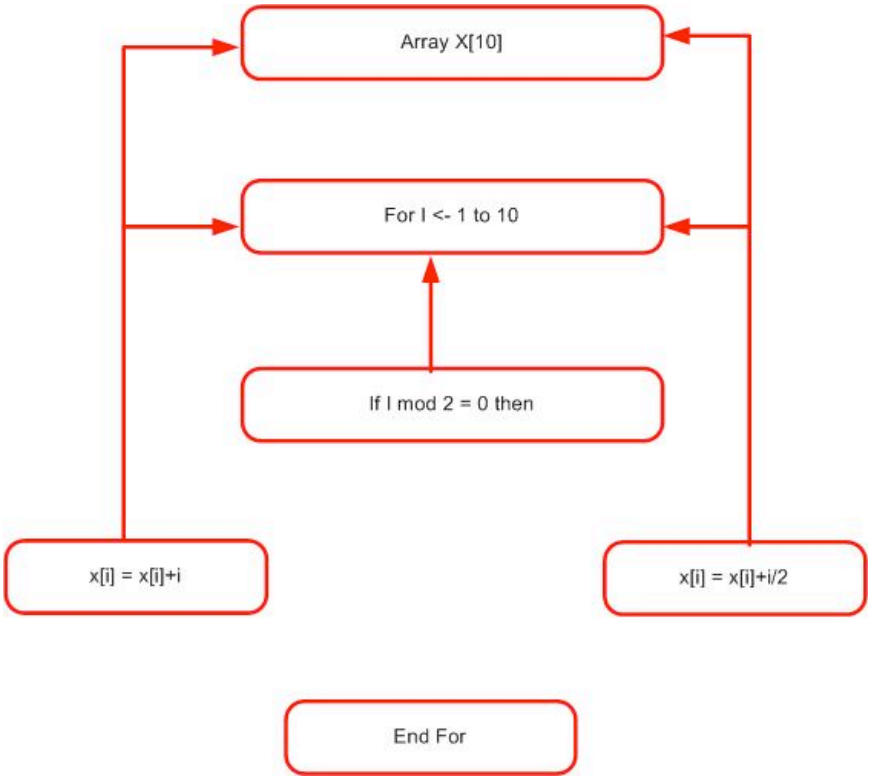

Figure 2: Data Flow Graph

2.3 Control-Data Flow Graph

Given a set of data flow dependencies as calculated by the methods that will be discussed later in this paper, along with a control flow graph generated from the program source code, a Control-Data Flow Graph (CDFG) can be constructed. A CDFG is created by merging the data flow dependency graph with the control flow dependency graph into one combined graph. Figure 3 shows an example of a CDFG based on merging the graphs from Figures 1 and 2.

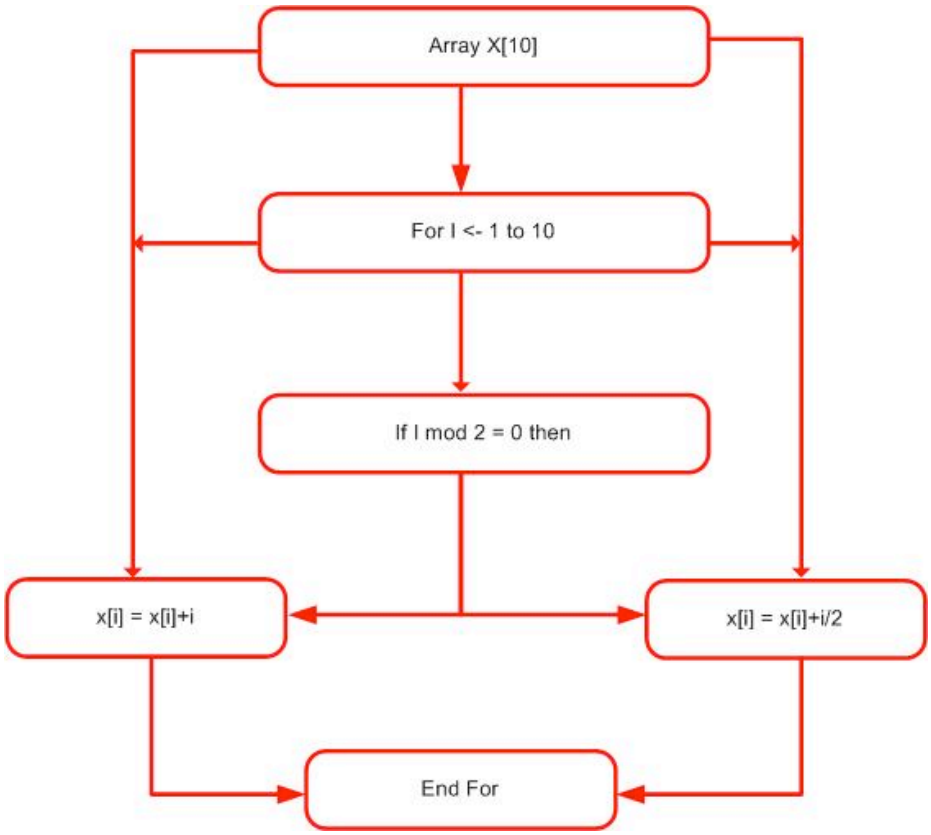

Figure 3: Control-Data Flow Graph Example

2.3 Dependence Classification

This section will discuss several criteria that is used for classifying the subscripts in pairs of array references. Complexity refers to the number of indices appearing within a subscript and separability describes whether a given subscript interacts with other subscripts for the purpose of dependence testing.

Before continuing with this section it is important to introduce the definitions for indices and subscripts as they will be used for the purpose of this paper.

Index (Indices) – The index variable for some loop surrounding both of the array references being tested for dependence.

Subscript – Subscript refers to the subscripted positions in a pair of array references. For example in the array reference A(I,J) = A(I,K) + X the index I appears in the first subscript and J and K appear in the second.

2.3.1 Index Complexity

When performing dependence testing it is possible to classify the subscript positions by the total number of loop indices that they contain. There are three possible types of complexity.

Zero Index Variable (ZIV) – A subscript position contains no index in either reference.

Single Index Variable (SIV) - A subscript position contains only one index in either array reference.

Multiple Index Variable (MIV) – A subscript position contains more than one index in its array references. A special case of the MIV subscript exists which is known as the Restricted Double Index Variable (RDIV). An RDIV subscript has the form $<a_1i+c_1, a_2j+c_2>$. RDIV subscripts [2] are similar to SIV subscripts except that `i` and `j` are distinct indices. SIV tests may be extended to handle RDIV subscripts by observing the loop bounds for `i` and `j`.

The following example demonstrates the three types of complexity classification.

```
DO 5 i
     DO 5 j
          DO 5 k
               A(5, i+1, j) = A(N, I, K) + X
```

Example 2: Three types of complexity classification

When performing dependence testing between the two array references to A in the code above the first subscript is ZIV, the second is SIV and the third is MIV. [3]

2.3.1 Index Separability

When performing dependence testing on multidimensional arrays a subscript position is said to be separable if its indices do not occur in the other subscripts [4,5]. Two different subscripts are said to be coupled when they contain the same index [6]. In the following example the first subscript is separable, but the second and third are coupled because they both contain the index J. ZIV subscripts are by nature separable because they have no indices.

```
DO 5 I
      DO 5 J
            DO 5 K
                  A(I,J,J) = A(I,J,K) + X
```

Example 3: Subscript Examples

When dealing with multidimensional arrays separability becomes a key issue because it can introduce imprecision in the dependence testing. One approach, suggested in [7], is called the subscript-by-subscript approach and tests each subscript separately and then intersects the resulting sets of direction vectors (Section 2.3.1). One issue with this technique is that it may produce direction vectors that do not exist. The following example highlights this problem.

```
DO 5 I
      A(I+1, I+2) = A(I,I) + X
```

Example 4: Separability Example

Performing the subscript-by-subscript test on the array reference in this example would produce a direction vector of the form (<) which can easily be shown to be incorrect because the above example contains no actual data dependence.

While this technique may cause problems when the subscripts are separable, if the subscripts are separable it can be used to compute the direction vector for each subscript independently, merging the direction vectors on a positional basis without losing any precision. The example below demonstrates this by producing a direction vector of the form (<.=,>) or the distance vector (Section 2.3.2) of the form (1,0,-1).

```
DO 5 I
      DO 5 J
            DO 5 K
                  A(I+1,J,K-1) = A(I,J,K) + X
```

Example 5: Separable subscripts example

2.3 Dependence Analysis Techniques

Locating dependencies between variables with static addresses is trivial and is not the focus of much work in the field of performing dependency analysis. Problems arise, however, when attempting to determine data dependencies between storage locations referenced by variable pointers. Without taking variable pointers (indices) into consideration potential flow dependence between two statements S_v and S_w can be identified if both statements exist in the body of the same, possibly nested loop and if both statements refer to the same array. If statement S_v refers to the array on the left hand side of the statement and S_w on the right there is the potential for a flow dependence between these statements. The following method describes how to determine if a

potential flow dependence will resolve to an actual flow dependence or if no dependence between the two statements exist. Whenever a potential flow dependence is identified it is necessary to invoke the dependence tests that will decide if an actual dependence exists.

When describing analysis techniques most of the focus is on programs that contain array references (variable pointers) and these references are contained within (possibly nested) loops. In order to demonstrate this problem Figure 4 shows a n-dimensional array X, with array indices provided by functions f_i and g_i which map from Z_d to Z (where Z is the collection of all integers). The iteration space is the set of all possible values the vector of loop indices can assume, or $\bar{I} = (I_1, I_2, \ldots, I_d)$.

```
DO I1 = L1,U1
      …
      DO Id = Ld, Ud
            Sv:   X(f1(I1,…,Id),…,fn(I1,…Id)) =
            Sw:   X(g1(I1,…,Id),…,gn(I1,…Id)
      END DO
      …
END DO
```

Figure 4: Generic Nested Loop [1]

It must now be decided whether or not some value computed by S_v is used by S_w. In this case a flow dependence would exist between the statements S_v and S_w. In order to determine if a flow dependence exists between S_v and S_w it is necessary to show the following conditions are met:

1) The execution of S_v takes place in the iteration $\bar{I}' = (I'_1, I'_2, \ldots, I'_d)$ and the execution of S_w takes place in the iteration $\bar{I}'' = (I''_1, I''_2, \ldots, I''_d)$. Both $\bar{I}'$ and $\bar{I}''$ are in the iteration space so $(L_j <= I'_j, I''_j <= U_j, 1 <= j <= d)$.

2) $f_i(\bar{I}') = g_i(\bar{I}'')$ for all $(1 <= I <= n)$

3) $\bar{I}' <= \bar{I}''$

What this set of conditions states is that for each pair of statements in a loop, a flow dependency exists between them if S_w is either located in the same loop iteration as S_v or a future loop iteration (both must be in the same loop or loop space), the index produced by index function f_i must equal the loop index produced by index function g_i and finally the loop indices $\bar{I}'$ must be less than or equal to the loop indices $\bar{I}''$.

If it is necessary to determine a flow dependence between S_v and S_w where S_v is located lexically after S_w the same conditions 1 and 2 hold while condition 3 must be modified as such: $\bar{I}' < \bar{I}''$. If these conditions are satisfied then statements S_v and S_w cannot be executed in parallel due to the existence of a data dependency.

This also applies to the two other types of data dependencies that were discussed in the background section: anti-dependency and output dependency. In order for an anti-dependency to exist it is necessary for the element of array X to be on the right hand side in statement S_v and on the left hand side in S_w. For an output dependency to exist it is necessary for both references to X to be on the left hand side of the statements S_v and S_w.

2.3.1 Direction Vectors

One method that is often used for determining flow dependence is decomposing the relationship between the components of vectors $\bar{I}'$ and $\bar{I}''$ into several sub-problems. Each of these sub-problems represents each possible ordering relationship between the components of $\bar{I}'$ and $\bar{I}''$. For example, if $\bar{I}''$ and $\bar{I}'$ both exist in Z^2 then the condition $\bar{I}' < \bar{I}''$ is decomposed into four cases [8].

Usually these cases are specified using direction vectors. These direction vectors take the form of $Y = (y_1, y_2, \ldots, y_d)$ where each y_k is either <, >, or = and represents the ordering relation between $\bar{I}'_k$ and $\bar{I}''_k$. The direction vectors derived from Example 1 would consist of: (<.<), (<,=) and (=,<). Direction vectors were first introduced by Wolfe [7] and are useful for calculating the level of loop carried dependences [4,9].

2.3.2 Distance Vectors

Distance vectors were first used by Kuck and Muaoka in [10, 11] and are more precise versions of the Direction vectors discussed in Section 2.3.1. Distance vectors are used to specify the actual distance between two accesses to the same memory location and have been used to guide optimizations which exploit parallelism [12,13,14,15,16] and memory hierarchy [17,18,19]. The following example demonstrates how a distance vector is computed from a set of array references.

```
DO 5 I
     DO 5 J
          DO 5 K
               A(I+1,J,K-1)  = A(I,J,K) + X
```

Example 6: Computing a Distance Vector

In this example the distance between I+1 and I is 1, J and J is 0 and K-1 and K is -1. This produces a distance vector of the form (1,0,-1). In some cases a direction vector or distance vector alone may be insufficient to completely describe a dependence and so both the distance and direction vector may be required.

2.4 Dependence Tests

There exist many different types of dependence tests that can be used to determine if dependencies exist between statements in a piece of software. This section will discuss several of these dependence tests and highlight their strengths and weaknesses. These tests will be broken down into two main categories: integer programming and simple or approximate based dependence tests.

2.4.1 A Partition Based Algorithm

In [3] an algorithm is presented that demonstrates a natural partition-based approach for determining dataflow dependencies. The algorithm is as follows:

1. Partition the subscripts into separable and minimal coupled groups.
2. Label each subscript as ZIV, SIV or MIV,
3. For each separable subscript, apply the appropriated single subscript test (ZIV,SIV,MIV) based on the complexity of the subscript. This will produce

independence or direction vectors for the indices occurring in that subscript. Several of these tests are discussed in further detail later in this paper.

4. For each coupled group, apply a multiple subscript test to produce a set of direction vectors for the indices occurring within that group

5. If any test yields independence, no dependences exist.

6. Otherwise merge all the direction vectors computed in the previous steps into a single set of direction vectors for the two references.

This algorithm is implemented in both PFC, an automatic vectorizing and parallelising compiler, as well as ParaScope, a parallel computing environment [13,20,21].

The algorithm presented above takes advantage of the properties of separability by classifying all subscripts in a set of array references as separable or part of a minimal coupled group. A minimal coupled group cannot be partitioned into two non-empty subgroups with distinct index sets. Once this partitioning is complete each separable subscript or group of coupled subscripts have disjoint sets of indices. With the disjoint sets of indices it is now possible to test each partition separately, merging the resulting distance or direction vectors without the fear of losing precision.

2.4.1 Simple and Approximate Dependence Tests

When searching literature on dependence testing one of the most common names referenced is Banerjee. Banerjee developed several different approximate dependence testing techniques [22,23] that have been widely adopted in both commercial as well as experimental compilers. Banerjee's work has also given rise to much other work [6,24,25] which will be discussed later in this section.

Simple and approximated dependence testing has a main goal of being capable of breaking potential data dependencies without the large processing cost of integer programming techniques that will be discussed in Section 2.4.2. The main difference between these tests and the integer programming tests is that simple and approximate tests analyze only one subscript at a time. This means that the test will only show a dependence as being broken if for some subscript *i*, there exist no index vectors $\bar{I}'$ and $\bar{I}''$ within the iteration space that satisfy the equation $f_i(\bar{I}') = g_i(\bar{I}'')$ where f_i and g_i are the array index function as in Figure 1. This test is considered to be conservative because if there exist coupled subscripts the system of equations may not have a solution even if all of the individual equations have solutions.

2.4.1.1 Simple Dependency Tests

The first test that will be discussed in this section is the constant test. The constant test is capable of breaking potential dependencies as well as proving that dependencies do indeed exist. Using the constant test it can be shown that if all subscripts in two array references are loop invariant and have the same value, then there exists a data dependence for all possible direction vectors. If any pair of corresponding subscripts are constant and are not equal then there exists no data dependence between them. Loop invariant terms that exist in both subscripts in the potential dependence can be cancelled before the comparison is performed.[8]

Another simple test that is used to determine flow dependencies is the Generalized Greatest Common Divisor (GGCD) test [23]. This test can be used to find criterion for the solution to the equation $f_i(\bar{I}') = g_i(\bar{I}'')$ as shown earlier. This test can be

used to show that if both fi and gi are linear that a solution to the equation exists if the greatest common divisor of the coefficients multiplies $\bar{I}'$ and $\bar{I}''$ and also divides the constant term. If the GCD does not divide the constant term then no solution to the equation can exist.

2.4.1.2 Banerjee's Inequalities for Loops with Known Limits

This section will discuss Banerjee's inequalities for loops with known limits as well as some work that has been done based on these inequalities. In order to describe Banerjee's inequalities we first assume that

$$f_i(\bar{I}) = a_1I_1 + a_2I_2 + \ldots + a_dI_d + a_0$$

and

$$g_i(\bar{I}) = b_1I_1 + b_2I_2 + \ldots + b_dI_d + b_0$$

This allows the equation $f_i(\bar{I}) = g_i(\bar{I}'')$ to be written as $(a_1I'_1 - b_1I''_1) + \ldots + (a_dI'_d - b_dI''_d) = (b_0 - a_0)$. The function $F(\bar{I}) = (a_1I'_1 - b_1I''_1) + \ldots + (a_dI'_d - b_dI''_d)$ is continuous in R^{2d}. Let B_{min}, B_{max} denote any two values of F in a connected set R (subset) R^{2d}, which contains all possible values of the iteration space. Suppose also that $B_{min} <= b_0 - a_0 <= B_{max}$. Then, from the intermediate value theorem, we know that the equation $F(\bar{I}) = b_0 - a_0$ has a solution $\bar{u} = (u_1, u_2, \ldots, u_{2d})$ exists in R and Banerjee's test assumes that a dependence exists. Since $\bar{u}$ only belongs to the iteration space when it is an integer vector it can be said that this is a conservative assumption to make.

If we define B_{min} and B_{max} as the minimum and maximum values of F in R the relation shown above, $B_{min} <= b_0 - a_0 <= B_{max}$, does not hold so no solution can

exist and potential dependencies are broken. It has been shown in [22,23] that dependence tests based on this formulation of Banerjee's dependence inequalities apply in two general cases.

Loop limits are either constant or linear functions creating an iteration space in the shape of a trapezoid. In this case a constant limit would be a constant value that can be determined by the compiler at compile time.

2.4.2 Symbolic Tests

The basis of symbolic testing for resolving data flow dependencies is the idea that c_2-c_1, the difference between the constant terms in two array reference subscripts, can be symbolically formed and simplified. The result of this simplification can then be used like a constant in order to break possible dependencies.

The test that will be discussed in this section deals with two array references that are held at two different levels of nested loops as shown in Example 7.

```
Do 5 I = 1, N1
     A(a1i+c1) = …
     DO 10 J = 1, N2
          … = A(a2j+c2)
```

Example 7: Two Array References at Different Levels

Based on the above example, and assuming for simplicity that a_1 is greater than or equal to zero. A dependence exists if the following dependence equation is satisfied:

$$A_1 i - a_2 j = c_2 - c_1$$

For some value of I, $1 <= I <= N_1$ and j, $1 <= j <= N_2$.

In this test there are two cases that must be considered. a_1 and a_2 may have the same sign and in this case $a_1i - a_2j$ assumes the maximum value for $i=n_1$ and $j=1$ and its minimum value for $i=1$ and $j=n_2$. Based on this a dependence exists if and only if:

$$A_1-a_2N_2 <= c_2-c_1 <= a_1N_1 - a_2$$

In this case, if either inequality is violated then the dependence cannot exist. In the second case a_1 and a_2 now have different signs. $a_1i - a_2i$ now assumes its maximum for $I = N_1$ and $j = N_2$ so a dependence only exists if:

$$A_1 - 2a <= c_1 - c_2 <= a_1N_1 - a_2N_2$$

Again, if either of these inequalities are violated the dependence does not exist.

2.4.2 The Delta Test

The main idea behind the delta test [6,26] is that constraints derived from SIV subscripts may be efficiently propagated into other subscripts in the same coupled group without losing any precision. The delta test can find independence if any of its ZIV or SIV tests determine independence. If no independence is found using the ZIV and SIV tests the delta algorithm then converts all SIV subscripts into constraints and propagates them into MIV subscripts when possible. This conversion is repeated until no new constraints can be found then constraints for RDIV subscripts are propagated. Once this is completed remaining MIV subscripts are tested. The results are then intersected with existing constraints. The algorithm for the Delta test is outlined in Figure 4.

```
INPUT: coupled SIV and/or MIV subscripts
OUTPUT: hybrid distance/direction vector,
constrained MIV subscripts
initialize elements of constraint vector C⃗ to <none>
while ∃3 untested SIV subscripts do
     apply SIV test to all untested SIV subscripts,
          return independence or
          derive new constraint vector C⃗'
     C⃗' ← C⃗ ∩ C⃗'
     if C⃗' = 0 then
          return independence
     else if C⃗ ≠ C⃗' then
          C⃗ ← C⃗'
          propagate constraint (C⃗ into MIV subscripts,
          possibly creating new ZIV or SIV subscripts
          apply ZIV test to untested ZIV subscripts,
          return independence or continue
     endif
endwhile
while ∃ untested RDIV subscripts do
     test and propagate RDIV constraints
endwhile
test remaining MIV subscripts, then
     intersect resulting direction vectors with C⃗
return distance/direction vectors from C⃗
```

Figure 4: The Delta Test [3]

Some key points from the above algorithm will now be discussed briefly:

Constraints: Assertions on indices derived from subscripts. A dependence distance is an example of a simple constraint. A constraint vector contains one constraint for each of the n indices in a coupled subscript group. The Delta test

uses constraint vectors to store constraints generated by the SIV tests. These vectors can be easily converted into distance or direction vectors. A given constraint in the constants vector can take the following form:

- Dependence line: A line $<a_x+b_y = c>$ representing the dependence equation.
- Dependence distance: the value <d> of the dependence distance. Equivalent to the dependence line <x - y = -d>.
- Dependence point: a point <x,y> representing dependence from iteration x to y.

Intersecting Constraints: Dependence equations from all subscripts must be solved simultaneously for a dependence to exist, intersecting constraints from each subscript results in greater precision. Constraints are intersected by performing the SIV tests and then performing a comparison of the resulting constraint vectors. If the vectors do not contain a common point, distance or dependence line then no dependence exists.

Restricted DIV Constraints: Restricted DIV constraints discussed in Section 2.3 are a special case when dealing with propagating restraints. Given the following set of dependence equations propagation of RDIV constraints will be demonstrated.

$$i+c_1 = j' +c_3$$

$$j+c_2 = I'+c_4$$

While each of these dependence equations can be tested for dependence separately without losing precision they must be considered together when computing the distance and direction vectors. The constraints may now be propagated by replacing the instances of index I and J with $i+\Delta I$ and $j+\Delta J$, where ΔI and ΔJ is the distance between the two occurrences of I and j respectively. This yields the following set of dependence equations:

$$i + c_1 = j + \Delta j + c_3$$

$$j+c_2 = i+ \Delta I + c_4$$

These equations can now be rewritten as:

$$\Delta I + \Delta j = c_1 + c_2 + c_3 + c_4$$

This equation can now be used to test dependence when given a specific distance or direction vector.

In practical usage the precision of the Delta test depends greatly on the type of coupled subscripts being tested. The SIV tests applied in the first phase of the Delta test algorithm as well as the constraint intersection algorithm are exact and suffer from no loss of precision. The Delta test does, however, have several sources of imprecision:

- Constraint propagation of dependence lines and distances may be imprecise if an index cannot be eliminated from both references.
- Complex iteration spaces such as triangular loops may impose constraints between subscripts not utilized in the Delta test.
- The Delta test does not propagate constraints from general MIV subscripts. This may cause coupled MIV subscripts to remain at the

end of the delta test. In this case more expensive tests such as the λ-test or Power [6,27] test must be used.

2.4.3 Multiple Subscript Tests

This section discusses a number of multiple subscript dependence tests that have been developed. Multiple subscript tests provide precision at the cost of performing tests on all subscripts simultaneously.

Fourier-Motzkin Elimination: One of the earliest methods used for handling multiple subscript tests utilized the Fourier –Motzkin elimination. The Fourier-Motzkin elimination is a linear programming method based on the pair wise comparison of linear inequalities. Some work done in dependence analysis using the Fourier-Motzkin elimination included Kuhn [28] and Triolet et al. [29] who attempted to represent array access in convex regions. These regions may be intersected by using Fourier-Motzkin elimination and may also be used to summarize memory accesses for entire segments of a program. One downside to these techniques is their processing time taking 22 to 28 times longer than conventional dependence testing [29].

Multidimensional GCD: Using Gaussian elimination modified for integers, Bajerjee's multidimensional GCD test checks for simultaneous unconstrained integer solutions in multidimensional arrays [23]. This creates a compact system where all integer points provide integer solutions for the original dependence system. The multidimensional GCD test has also been extended in [27] to provide an exact test for distance vectors.

λ Test: Another multidimensional version of Banerjee's inequalities is the λ test, presented in [6] by Li et al. The λ test checks for simultaneous constrained real-valued solutions by forming linear combinations of subscripts that eliminate one of more instances of indices. The result is then tested using Banerjee's inequalities. Simultaneous real valued solutions exist if and only if Banerjee's inequalities find solutions in all linear combinations generated.

The λ test can also be used to test direction vectors as well as triangular loops. The precision of the λ test can be improved by also using the GCD or single-index exact tests. One issue with the λ test is that it is not possible to extend the test to prove the existence of simultaneous integer solutions. The λ test is exact in two dimensions if an unconstrained integer solution exists and the coefficients of the index variables are all either 1, 0 or -1 [30]. Even with these restrictions the λ test is not exact for three or more coupled dimensions.

Constraint-Matrix: In [26] Wallace developed the Constraint Matrix test which is a simplex algorithm modified for integer programming. Its precision and expense are difficult to determine since it halts execution after an arbitrary number of iterations to avoid cycling. The simplex algorithm itself has a worst case exponential time complexity but in reality takes linear time for most linear programming problems. In [31], however, it is shown that in combinatorial problems where coefficients tend to be 1,0 or -1 the simplex algorithm is very slow and may cycle for certain pivot rules.

Power Test: The Power Test, developed by Wolfe and Tseng [27], gains great precision by applying loop bounds by using Fourier-Motzkin elimination to the dense system resulting from the multidimensional GCD test [27]. In practice the Power test is quite expensive as compared to other methods for determining dependence, but it is also very flexible and is well suited for providing precise dependence information such as direction vectors in imperfectly nested loops, loops with complex bounds and non-direction vector constraints.

One of the benefits of the Power test, as well as the Delta test is their ability to detect and discard linearly dependent subscripts as part of their basic algorithm. Constraint-Matrix and λ tests, on the other hand, both require a pre-test be performed to eliminate the linearly dependant subscripts.

2.5 Integer Programming Based Dependence Tests

Data dependence analysis of linear array references is equivalent to deciding if there exists an integer solution to a set of linear equalities and inequalities. The problem of integer programming can be stated as follows: Does there exist an $\bar{x}$ such that $A\bar{x} = \bar{b}$, $B\bar{x} >= 0$, $\bar{x} >= 0$ for integer $\bar{x}$ [32].

When compared to the approximate tests described above, general integer programming techniques show several advantages. Integer programming techniques have the ability to simultaneously consider all of the subscripts of an array reference which allows this type of dependency test to correctly analyze coupled subscripts. Another benefit to integer programming solutions for dependence testing is the ability to incorporate both affine loop limits for trapezoidal loops, as well as covering conditionals,

into the equations. Since these tests work on the integer domain the existence of a non-integer solution does not cause the test to assume that a dependence exists while Banerjee's test would.

2.5.1 Simplex Based Integer Programming Test

One type of integer programming test is known as the simplex based integer programming test and is based around using a branch and bound algorithm which first applies a linear programming algorithm to find a real-valued solution. This solution is then checked to see if all of its components are integers. The first non-integer components is then selected and is used to create two new problems with constraints. The first problem is the same as the addition problem with the added constraint $x_i <= \lfloor xi \rfloor$ and the second problem is the same as the first problem with the constraint $x_i <= \lceil xi \rceil$. This process generates a binary tree of problems that repeatedly divide the problem domain.

The branch-and-bound algorithm does an optimized exhaustive search of the problem domain. If a region of the problem domain does not have a real valued solution that region will not be searched for an integer solution. Once an integer solution is found, the process stops and reports success. If all branches of the search lead to empty sets the process reports that no solution exists.

2.5.2 The Omega Test

The Omega test is based on the Fourier-Motzkin algorithm, discussed earlier in this paper, and provides an extension to this algorithm by introducing integer constraints on the solution vector [33]. The Omega test is a very powerful method that subsumes the Simplex based methods discussed in Section 2.5.1 and includes a number of additional

capabilities. Although the Omega test is only used to determine if there exists an integer valued solution to a dependence equation it can also be used to eliminate value based transitive dependences and accurately compute distance vectors.

In cases where simple subscripts are used, like the kind handled by Banerjee's test, the Omega test is usually a small constant factor slower than Banerjee's test [22]. As the subscripts being tested increase in complexity the execution time will increase for the Omega test until it reaches a worst-case exponential runtime in the number of variables.

3.0 Summary

This literature survey was conducted as part of preparation for a project that centered around extracting parallelism from a Handel C hardware specification. Based on the research completed for this survey it was decided that the method to be used for this project would be Banerjee's Inequalities for Loops with Known Limits. This method was chosen due to its simple implementation and the wide body of software projects that have been completed using this technique. This will make implementation of the project easier by providing implementation examples. The other dependence analysis techniques are either too slow (integer based solutions) or rely heavily upon an existing implementation of the Banerjee method. This means that creating a solution from the ground up would require implementing Banerjee's methods so it was decided that this would be the best starting point. If in the future a more comprehensive method is required, the Banerjee method can be augmented with another method.

3.1 Conclusion

This survey discussed many of the methods that are available for determining the dependence relationships between statements in computer programs. While this paper is by no means exhaustive it does provide a good basis for beginning a project that includes the use of dependence analysis techniques. Multiple dependence analysis techniques were presented, from slow but accurate integer based solutions to fast techniques that rely heavily on constraining the dependencies being analyzed and may introduce some inaccuracies in the dependencies that are found.

The survey also follows the development of modern dependence techniques from Banerjee's method to the multitude of techniques that build upon it as well as techniques that use a combination of both Banerjee's method with other techniques to gain added accuracy or processing speed. After researching this topic it can be said that there is still much room for growth in this field as there exist few fast and accurate methods capable of determining dependence relationships. Research must be done to determine how best to leverage the speed of simple dependence tests with the accuracy of integer based tests to create a dependence testing method that can satisfy all dependence analysis problems.

4.0 Acknowledgements

The authors would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council as well as the Canadian Microelectronics Corporation for equipment used in this research.

5.0 References

1. GNU Compiler Collection (GCC) Internals - **Control Flow Graph**,

http://gcc.gnu.org/onlinedocs/gccint/Control-Flow.html, Accessed on November 30, 2007.

2. M.J. Wolfe, C. Tseng, **Optimizing Supercompilers for Supercomputers**, The MIT Press, Cambridge, MA, 1989.
3. G. Goff, K. Kennedy, C. Tseng, **Practical Dependence Testing**, Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, June 26-28, 1991.
4. J.R. Allen, **Dependence Analysis for Subscripted Variables and Its Application to Program Transformations**, PhD thesis, Rice University, April 1993.
5. D. Callahan, **Dependence Testing in PFC: Weak Separability**, Supercomputer Software Newsletter 2, Dept. of Computer Science, Rice University, August 1986.
6. Z. Li, P.Yew, C. Zhu, D**ata dependence analysis on multi-dimensional array references**, Proceedings of the 1989 ACM Internaional Conference on Supercomputing, Crete, Greece, June 1989.
7. M.J. Wolfe, **Optimizing Supercompilers for Supercomputers**, PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaigne, October, 1982.
8. P.M. Petersen, D.A. Padua, **Static and Dynamic Evaluation of Data Dependence Analysis Techniques**, ICS93, 1993.
9. J.R. Allen, K. Kennedy, **Automatic translation of Fortran programs to vector Form**, ACM Transactions on Programming Languages and Systems, October 1987.
10. D. Kuck, **The Structure of Computers and Computations**, Volume 1, John Wiley and Sons, New York, NY, 1978.
11. Y. Muraoka, Parallelism Exposure and Exploitation in Programs, PhD thesis, Dept of Computer Science, University of Illinois at Urbana-Champaigne, February 1971.
12. R. Heuft and W. Little, **Improved time and parallel processor bounds for Fortran-like loops**, IEEE Transactions on Computers, January 1982.
13. K. Kennedy, K.S. McKinely, C. Tseng, **Analysis and transformation in the ParaScope Editor**, IEE Transactions on Parallel and Distributed Systems, July 1991.
14. L. Lamport, **The Parallel Execution of DO Loops,** Communications of the ACM,

February 1974.

15. M.J. Wolfe, M. Lam, **Maximizing Parallelism Via Loop Transformations**, Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing, August 1990.
16. M.J. Wolfe, **Loop Skewing: The Wavefront Method Revisited**, International Journal of Parallel Programming, August 1986.
17. D. Callahan, S. Carr, K. Kennedy, **Improving Register Allocation for Subscripted Variables**, Proceedings of the ACM SIGPLAN 1990 Conference on Program Language Design and Implementation, June 1990.
18. D. Gannon, W. Jalby, K. Gallivan, **Strategies for Cache and Local Memory Management By Global Program Transformations**, Proceedings of the First Internation Conference on Supercomputing, June 1987.
19. A. Porterfield, **Software Methods for Improvement of Cache Performance**, PhD thesis, Rice University, May 1989.
20. D. Callahan, K. Cooper, R. Hood, K. Kennedy, **ParaScope: A Parallel Programming Environment**, The international Journal of Supercomputer Applications, Winter 1988.
21. K. Kennedy, K.S. McKinely, C. Tseng, **Interactive Parallel Programming Using the ParaScope Editor**, IEE Transactions on Parallel and Distributed Systems, July 1991.
22. U. Banerjee, **Speedup of Ordinary Programs**, PhD thesis, University of Illinois at Urbana-Champaigne, October 1979.
23. U. Banerjee, **Dependence Analysis for Supercomputing**, Kluwer Academic Publishers, 1988.
24. X. Kong, D. Klappholz, K. Psarris, **The I Test : A new Test for Subscript Data Dependence**, Proceedings of the 1990 Internationl Conference on parallel Processing, August 1990.
25. M. Wolfe, C.W. Tseng, **The Power Test for Data Dependence**, IEEE Transactions on Parallel and Distributed Systems, September 1992.
26. D. Wallace, **Dependence of Multi-dimensional Array References**, Proceedings of the

Second International Conferences on Supercomputing, July 1988.

27. M.J. Wolfe, C. Tseng, **The Power Test For Data Dependence**, Technical Report, Dept. of Computer Science and Engineering, Oregon Graduate Institute, August 1990.
28. R. Kuhn, **Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks and Decision Trees**, PhD Thesis, Dept of Computer Science, University of Illinois at Urbana-Champaigne, February 1980.
29. R. Triloet, F. Irigoin, P. Feautrier, **Direct Parallelization of CALL Statements**, Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, July 1986.
30. Z. Li, P. Yew, **Some Results On Exact Data Dependence Analysis, Languages and Compilers for Parallel Computing**, The MIT Press, 1990.
31. A. Schrijver, **Theory of Linear and Integer Programming**, John Wiley and Sons, 1986.
32. H.M. Salkin, K. Mathur, **Foundations of Integer Progamming**, North Holland, 1989.
33. W. Pugh, **The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis**, Communications of the ACM, August 1992.