Notes about Interference Dependence

Venkatesh Prasad Ranganath
rvprasad@microsoft.com

November 22, 2007

This note examines validity of the statement — "*interference dependence is intransitive.*".

Basics

For purpose of simplicity, we shall consider programs written in *a simple WHILE language that supports shared variable concurrency*. Further, we allow only integer variables and values in the language and preclude the support for aliasing and dynamic thread creation from the language.

Each program in the above language is composed of one or more threads and a set of *shared variables* that are accessible from every thread in the program. All shared variables are initialized to zero before the execution of the threads in the program. A *thread* is an executable entity that is composed of a set of *local variables* along with a sequence of commands involving local variables and shared variables. Every variable in the program has a unique name. During program execution, each thread in the program is executed once and only once, and the threads are executed concurrently. For simplicity, we preclude any compiler optimizations that may reorder the commands in a thread.

Data Dependence

Consider the program in Figure 1. As it is sequential (composed of a single thread), there can be only one execution trace and it is dictated by the sequencing of the commands in thread `main`.

Based on the only possible execution sequence, the value assigned to `y` depends on the value of `a`. Hence, line 7[1] depends on line 6. More specifically, the data used at line 7 depends on the data defined at line 6. Similarly, line 6 depend on (the data defined at) lines 5. This form of dependence between definition and use of data is referred to as *data dependence* [3].

Definition 1 In a program P, a program point p_t is *data dependent* on a program point p_e if

[1]When there is no ambiguity, we shall use line x as a shorthand for the command at line x.

```
1 int x,y;
2
3 Thread main {
4   int a;
5   x=random();
6   a=x*2;
7   y=sqrt(a);
8 }
```

Figure 1: Example to illustrate data dependence

1. p_e defines a local variable v,
2. p_t uses the local variable v,
3. p_t and p_e execute in the same thread, and
4. there exists an execution of P in which p_t executes after p_e and uses the definition of v at p_e.

Interference Dependence

Consider the program in Figure 2. Unlike the program in Figure 1, this program is concurrent as it has multiple threads. Hence, due to interleaving of commands from various threads, there are three possible execution traces: [4,5,9], [4,9,5], and [9,4,5] where each number in the sequence is a line number from the program and the lines are executed in the order given by the sequence.

```
1 int a,b,c;
2
3 Thread one {
4   a=b+1;
5   c=10;
6 }
7
8 Thread two {
9   b=c+1;
10 }
```

Figure 2: Example to illustrate interference dependence

In the execution trace [4,5,9], line 9 depends on line 5 as line 9 is executed after line 5 and the definition of `c` at line 5 is used at line 9. The definition of `b` at line 9 does not reach the use of `b` at line 4 as line 9 is executed after line 4; hence, there is no dependence relation between line 4 and line 9.

In the execution trace [4,9,5], all dependences in trace [4,5,9] are valid except that line 9 is not depend on line 5 as line 9 is executed before line 5.

In the execution trace [9,4,5], all dependences in trace [4,9,5] are valid in addition to line 4 depending on line 9 as the value assigned to b at line 9 is used at line 4.

This form of data dependence stemming from the definition and use of shared data in different threads is referred to as *interference dependence* [1].

Definition 2 In a program P, a program point p_t is *interference dependent* on a program point p_e if

1. p_e writes to a shared variable v,
2. p_t reads the shared variable v,
3. p_t and p_e execute in different threads, and
4. there exists an execution of P in which p_t executes after p_e and reads the value of v written at p_e.

Upon summarizing the interference dependences in the program (based on the possible execution traces) in Figure 2, we can conclude line 9 is interference dependent on line 5 and line 4 is interference dependent on line 9.

Intransitivity

Given the data dependence relation between line 5, 6, and 7 in the program in Figure 1, we can conclude that line 7 is *transitively* data dependent on line 5. This conclusion is valid as there exists an execution that contains every dependences required to establish transitivity.

Despite the similarities between data and interference dependence, we cannot conclude that line 4 is transitively interference dependent on line 5 in the program in Figure 2. This is due to the absence of an execution of the program in which the data definition at line 5 can influence the data used at line 4 (as line 5 will always execute after line 4).

Based on this observation, few efforts [1, 2] related to concurrent program slicing have concluded that *"interference dependence is intransitive."*[2] We shall refer to this as the *intransitivity conclusion*.

This conclusion would be valid if intransitivity is defined as *a binary relation R is intransitive when* $\exists a, b, c. \neg((aRb \wedge bRc) \implies aRc)$. Interestingly, the intransitivity conclusion would be invalid if intransitivity is defined as *a binary relation R is intransitive when* $\forall a, b, c. (aRb \wedge bRc) \implies \neg(aRc)$. This definition of intransitivity is also referred to as *antitransitivity*. Interestingly, in mathematics, this latter definition is commonly associated with intransitivity. In the rest of the discussion, we assume that the former definition of intransitivity was used in the conclusion.

To understand the validity of the conclusion, consider the program in Figure 3. The program is identical to the program in Figure 2 except that the definitions of a and c in thread one occur within a loop. The possible execution traces for the above program (considering only the relevant commands) are [6,7,6,7,13], [6,7,6,13,7], [6,7,13,6,7],

[2] [1] mentions interference dependence is not transitive.

[6,13,7,6,7], and [13,6,7,6,7].[3] In the trace [6,7,13,6,7], the line 6 is transitively interference dependent on line 7 (via line 13). This observation would also be true if thread `one` in the program in Figure 2 was instantiated and executed twice (with relaxed semantics of the language).

```
1  int a,b,c;
2
3  Thread one {
4    int i=0;
5    while (i<2) {
6      a=b+1;
7      c=10;
8      i=i+1;
9    }
10 }
11
12 Thread two {
13     b=c+1;
14 }
```

Figure 3: Example to illustrate transitivity of interference dependence

In the trace [6,7,13,6,7], we can make a finer observation — *the second execution instance of line 6 is interference dependent on the first execution instance of line 13*. If interference dependence is stated in such finer terms and context — execution instance of a program point in a specific execution trace, then the intransitivity conclusion would be valid. In coarser contexts (or terms) based only on either program points, execution instances of program points, or execution traces, the intransitivity conclusion would be invalid.

As most sound program analyses (such as program slicing) operate on abstractions of concrete executions (e.g., a control flow path abstracts multiple execution traces, a program point abstracts multiple execution instances), contexts and terms based on such abstractions would be coarse; hence, *interference dependence should be considered to be transitive*.

Conclusion

Contrary to common perception, interference dependence can be either transitive or intransitive depending on the program structure and the stated context and terms. In the realm of sound program analyses that operate on abstractions of concrete executions, interference dependence should be considered to be transitive.

[3] Lines 6, 7, and 13 correspond to lines 4,5, and 9, respectively, in the program in Figure 2.

References

[1] J. Krinke. Static slicing of threaded programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7).

[2] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 180–190, 2000.

[3] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.