CS6843 (EVEN16) : Program Analysis

Assignment A4 : Specification (v1.0)

Submission Deadline : `17-Apr-2016 23:55`

1 Introduction

Single Program Multiple Data. In SPMD style parallel programs, a copy of the program (or some specified part of it, which we call as a *parallel region*) is given to each thread for parallel execution. Each thread may execute the program on different input, to obtain parallelism.

Memory accessibility. Each thread has access to a portion of memory that is shared across all the threads. In shared memory programs, this shared memory is used for communication and synchronization among different threads. Furthermore, each thread also owns a private memory that cannot be accessed by other threads. Private memory is faster to access than shared memory.

Race condition. When different threads concurrently access same location in shared memory, where at least one of these accesses is a write, then a race condition occurs. (Not all race conditions are harmful, and some can be intentional/useful in nature.)

Barrier. To ensure a deterministic ordering among these various accesses from multiple threads, we use synchronization constructs, like barriers, that help us prevent race conditions. When a thread encounters a barrier instruction, it waits for all the other threads to encounter this, or some other, barrier instruction. Once all the threads encounter a barrier instruction, they can proceed with the parallel execution of other instructions that follow the encountered barrier(s).

(a) A racy program		**(b) A race-free program**	
Thread T1	Thread T2	Thread T1	Thread T2
Initially, x = 0;		*Initially, x = 0;*	
`x = 5;`	`print x;`	`x = 5;`	
		`barrier();`	`barrier();`
			`print x;`
Expected output : 0 *OR* 5		*Expected output* : 5	

Figure 1: Example of using a barrier

Figure 1 shows an example of how barriers can help prevent a race condition. In part (a), the value of x that can be printed is either 0 or 5. In part (b), the barriers ensure that instruction `print x;` can not be executed by thread T2, unless the instruction `x=5;` has been executed by thread T1. Therefore, the output becomes deterministic in nature (i.e., will always be 5).

May Happen in Parallel (MHP). Given an MHP query for a pair of statements at compile time, say ($S1$, $S2$), we conservatively return `true` if there exist any runtime instances of $S1$ and $S2$ that may get executed by different threads in parallel, i.e., the order in which these instances may get executed across different executions of the program is not known to be deterministic. If it is possible to give a proof at static (compile) time that no instances of $S1$ and $S2$ can be executed in parallel by different threads (or in other words, $S1$ must happen before/after $S2$ in all the executions), then we return `false` for the MHP query on $S1$ and $S2$.
Clearly, $S1$ and $S2$ can **never** be a race condition pair, if `MHP(`$S1$`, `$S2$`)` = `false`.

Phase. Starting with the first statement of a parallel region, threads start execution of various statements they encounter unless they hit a barrier (may be different for different threads). After synchronization at the first set of barriers, threads execute further statements unless they each hit a barrier(s) again, synchronize, and proceed in a similar fashion. We denote each such set of statements that may get executed by different threads, starting at a set of barriers (*start-set*) and ending at a set of barriers (*end-set*), as a *phase*.
A statement may be a part of multiple phases, and each phase may contain multiple statements. All the statements within a phase may get executed in parallel across different threads. (Of course, intra-thread dependences are preserved.)
If there exists any phase in which statements $S1$ and $S2$ may get executed, then MHP query over these statements should conservatively evaluate to `true`.
More formally, let ϕ(S) represent the set of phases in which statement S may get executed. Then,

$$(\phi(S1) \cap \phi(S2) = \emptyset) \implies \texttt{MHP}(S1,\, S2) = \texttt{false}$$

2 Aim of the assignment

In A4, we are expected to perform the following two analyses:

1. Two versions of MHP analysis using phase information:
   (a) Path-insensitive MHP analysis (explained in Section 4.1).
   (b) Path-sensitive MHP analysis using the concept of single-valued expressions (explained in Section 4.2).
2. Detect and report race condition pairs as obtained individually (explained in Section 5) from both the versions of MHP analysis.

In Section 3, we look into the structure and semantics of the input programs.
Analyses' dimensions: Intra-procedural, flow-sensitive, path-sensitive (as well as path-insensitive).

3 Input program domain

Each input testcase will consist of a special function named `parRegion`, and some global integer variables. This function may take one or more `const` parameters, and will have `void` return type. We assume that a copy of this function will be executed by different threads in parallel, with **different values** of the arguments. In other words, this function semantically denotes a parallel region.
`parRegion` may contain any valid C code, with an exception that there can not be calls to any function, except for a special function:

- `barrier()`: We assume that this (empty) function semantically represents the concept of a barrier, as explained above.

Ignore all the other functions, if any, in the testcases.
The global variables should be semantically assumed to be stored in shared memory, i.e., all the threads share a single copy of these variables. All the parameters and local variables of `parRegion`, should be assumed to be stored in the private memory, i.e., each thread will have an individual private copy (a different memory location per thread) for these variables.
Note that there won't be any writes to parameters within `parRegion`. Furthermore, to keep the assignment simple and focused, we will not use any pointers, arrays or structures. In fact, all the local variables, parameters and global variables will be of type `int`.

4 May Happen in Parallel (MHP) analysis

4.1 Path-insensitive MHP analysis

In this section, we discuss a naive approach to perform MHP analysis on the input program. To obtain the MHP information, it is sufficient to obtain the set of all the unique phases that are present in the parallel region.
We represent each phase with a 3-tuple ($\mathcal{S}$, $\mathcal{E}$, $\mathcal{I}$), where

$\mathcal{S}$ is the set of barriers at which this phase starts,
$\mathcal{E}$ is the set of barriers at which this phase ends, and
$\mathcal{I}$ is the set of instructions that belong to this phase.

Please note that we assume **one implicit barrier each at the begin and end of the body** of the `parRegion`. Let's denote them by B_s and B_e, respectively, for our discussion.
We mark the instructions with phases, one phase at a time.

4.1.1 Algorithm

Create a new phase, say P. Populate $P.\mathcal{S}$ with B_s. Starting with the first instruction in `parRegion`, collect all the instructions which are reachable on a barrier-free path (i.e., all the nodes that are reachable until we hit a barrier) in a forward traversal of the CFG. These instructions will become the set $P.\mathcal{I}$ of those instructions that may get executed in phase P. Collect all the barriers where we stop the traversal on different paths, as the set $P.\mathcal{E}$ (the ending set of barriers). If we reach the end of the body of `parRegion` via some barrier-free path, add B_e as well to $P.\mathcal{E}$.

Now, we proceed with the marking of next phases. Each new phase will have its starting set, $\mathcal{S}$, same as the ending set $\mathcal{E}$ of the previous phase. For each barrier b in the starting set of a new phase, we collect all the instructions that are reachable in the forward traversal (on CFG) starting at b into the set $\mathcal{I}$, until we hit a barrier (or an infinite loop). These barriers which terminate the traversal are added to the ending set $\mathcal{E}$, which will become the starting set of the next phase; and so on.

When do we stop creating new phases? A phase simply represents the fact that the instructions in it may happen in parallel with each other. So having two different phases with the same set of statements won't add to any new information. Now, given a set of starting barriers, the set of reachable statements on a barrier-free path is fixed. So we will stop creating any new phase from a given phase if the ending set of barriers in the given phase is already present as the starting set of barriers of some other existing phase.

Will this process of marking the instructions with phases terminate? Yes. Think about how many distinct phases are possible for a given program. Then the proof of termination is straightforward.

4.1.2 Example

Consider a sample code given in Figure 2. Let us perform MHP analysis on this sample code. For simplicity, we represent each barrier (and instruction) by the line number where it is present in the source file. Let us represent the implicit barriers present at the beginning and end of `parRegion` with special line numbers 0 and -1, respectively.
Important: Please note that we are only interested in those instructions that read or write any local variable, parameter or global variable. We won't mark other line numbers (e.g., those which perform `alloca` instruction without any initializations).

```
1: void barrier() {}
2: int g1 = 10;
3: int g2 = 20;
4: void parRegion(const int p1, const int p2) {
5:      int l1;
6:      int l2;
7:      l1 = 5;
8:      l2 = 15;
9:      barrier();
10:     if (l1 < g1) {
11:             g2 += l2;
12:             barrier();
13:             while (l2 < p2) {
14:                     l2++;
15:                     barrier();
16:                     l1 = g1 + g2;
17:             }
18:     } else {
19:             g2 += p2;
20:             barrier();
21:             while (g2 < l1) {
22:                     l2++;
23:                     barrier();
24:                     l2 = p2 + p1;
25:             }
26:     }
27:     l2 = g1 + l1 + l2*p2 - p1;
28:     barrier();
29:     l1 = g2 - g1;
30:     return;
31:}
```

Figure 2: A sample code

Phase A.

$$P.\mathcal{S} = \{0\}; \text{ 0 represents the implicit barrier at the beginning of } \texttt{parRegion}$$
$$P.\mathcal{I} = \{7, 8\};$$
$$P.\mathcal{E} = \{9\};$$

Phase B.

$$P.\mathcal{S} = \{9\};$$
$$P.\mathcal{I} = \{10, 11, 19\};$$
$$P.\mathcal{E} = \{12, 20\};$$

Phase C.

$$P.\mathcal{S} = \{12, 20\};$$
$$P.\mathcal{I} = \{13, 14, 27, 21, 22\};$$
$$P.\mathcal{E} = \{15, 23, 28\};$$

Phase D.

$$P.\mathcal{S} = \{15, 23, 28\};$$
$$P.\mathcal{I} = \{16, 13, 14, 27, 24, 21, 22, 29\};$$
$$P.\mathcal{E} = \{15, 23, 28, -1\}; \text{ -1 represents the implicit barrier at the end of } \texttt{parRegion}$$

Phase E.

$$P.\mathcal{S} = \{15, 23, 28, -1\};$$
$$P.\mathcal{I} = \{16, 13, 14, 27, 24, 21, 22, 29\};$$
$$P.\mathcal{E} = \{15, 23, 28, -1\};$$

Note: Please verify your understanding with the help of this complicated example. In case of doubts, please use forums/meet the TAs.

4.2 Path-sensitive MHP using single-valued expressions

Now, let's try and improve the precision of the MHP analysis, by making it path-sensitive in nature.

Single-valued Expressions. If an expression is guaranteed to be evaluated to the same value for all the threads, then the expression is termed as a single-valued expression (SVE). Some trivial examples are $(2 + 5)$, $(2 < 3)$, etc.
If an expression is not an SVE, we call it a multi-valued expression (MVE).
In Section 4.2.1, we will see how to find whether an expression is an SVE or not. Then, in Section 4.2.2, we will see how we can use SVE analysis to make MHP path-sensitive in nature.

4.2.1 SVE Analysis

Note: There are many ways to improve upon the precision of SVE analysis discussed in this section. No matter howsoever tempting it may be, please don't use any different technique/heuristic. That way we all can be on the same page.

Given an expression, move backward in the CFG. Check whether there exists any term in the expression that receives its value, directly or indirectly, from a variable (via any path) that may evaluate to different values for different threads. If that is indeed the case, then the expression is an MVE, else an SVE.
Do not try to simplify the expressions in the process! As an example, in the code snippet {`x = 0*a; if(x<3) {...} else {...}`}, if we want to check whether the predicate of the `if` statement is an SVE or not, and if it is known that `a` is a variable that may evaluate to different values for different threads, then we should mark the predicate as an MVE. We should not try to simplify `x=0*a;` to `x=0;`

Now, what are the variables that may evaluate to different values for different threads? Let's think about it.
In our semantics, we have specified that the parameters may obtain different arguments for different threads, and are `const` in nature. So reads of all the parameters are MVEs.

If a global variable is not being written to *anywhere* in `parRegion`, all the reads of that global variable will be SVEs. To keep the assignment simple, we will conservatively assume that if there exists even a single write to a global variable in `parRegion`, then all the reads of that global variable will be MVEs.

Reads to local variables may be SVE or MVE, depending upon what they have been written with. Given any expression of concern, we will traverse backwards in the CFG (just like in A3), and on each path, one of the following possibilities may occur :

1. We hit a loop.
2. We find that the expression, directly or indirectly, obtains its value from a constant, parameter, or a global variable.

If we hit a loop on any of the paths, or if the value of the expression is dependent on a variable whose reads are MVEs, then the expression is an MVE. Otherwise, we mark the expression as an SVE.

As an example, following is the only predicate expression which is an SVE in Figure 2 :

```
(l1 < g1)
```

4.2.2 Precise MHP using SVE analysis

Now, let's see how we can make the MHP analysis path-sensitive (thereby, more precise in general) using SVE analysis.

We perform SVE analysis for all the predicates of different branch instructions. While performing a forward traversal on the CFG during the marking of phases, when we encounter a branch instruction, if the predicate of the branch instruction is an SVE, we can have a guarantee that all the threads will take the same branch, and therefore, instructions in one branch can not run in parallel with the instructions in the other branch. To represent this fact in our phase information, we "split" the marking phase, say P, into as many new phases as there are branches out of the branch instruction. Furthermore, we need to add all the instructions from $P.\mathcal{I}$ to $\mathcal{I}$ sets of all the new phases that have been created during the splitting of the phase P at the branch node. Once this is done, **phase P should be removed from the set of all the phases**, since the information computed by P is already derivable from the new phases.

4.2.3 Example

Following are the set of phases that we obtain upon performing a path-sensitive MHP analysis on the code given in Figure 2.

Phase 1.

$P.\mathcal{S} = \{0\}$; 0 represents the implicit barrier at the beginning of `parRegion`
$P.\mathcal{I} = \{7, 8\}$;
$P.\mathcal{E} = \{9\}$;

Phase 2.

$P.\mathcal{S} = \{9\}$;
$P.\mathcal{I} = \{10, 11\}$;
$P.\mathcal{E} = \{12\}$;

Phase 2′.

$P.\mathcal{S} = \{9\}$;
$P.\mathcal{I} = \{10, 19\}$;
$P.\mathcal{E} = \{20\}$;

Phase 3.

$P.\mathcal{S} = \{12\}$;
$P.\mathcal{I} = \{13, 14, 27\}$;
$P.\mathcal{E} = \{15, 28\}$;

Phase 3′.

$P.\mathcal{S} = \{20\}$;
$P.\mathcal{I} = \{21, 22, 27\}$;
$P.\mathcal{E} = \{23, 28\}$;

Phase 4.

$P.\mathcal{S} = \{15, 28\}$;
$P.\mathcal{I} = \{16, 13, 14, 27, 29\}$;
$P.\mathcal{E} = \{15, 28, -1\}$; -1 represents the implicit barrier at the end of `parRegion`

Phase $4'$.

$$P.\mathcal{S} = \{23, 28\};$$
$$P.\mathcal{I} = \{24, 21, 22, 27, 29\};$$
$$P.\mathcal{E} = \{23, 28, -1\};$$

Phase 5.

$$P.\mathcal{S} = \{15, 28, -1\};$$
$$P.\mathcal{I} = \{16, 13, 14, 27, 29\};$$
$$P.\mathcal{E} = \{15, 28, -1\};$$

Phase $5'$.

$$P.\mathcal{S} = \{23, 28, -1\};$$
$$P.\mathcal{I} = \{24, 21, 22, 27, 29\};$$
$$P.\mathcal{E} = \{23, 28, -1\};$$

5 Race condition detection

Provided that we have calculated the phase information, it is easy to perform race condition detection. We don't need to look into the CFG anymore, for this step. If a phase contains a pair of instructions, such that both the instructions access a common *shared* location, and at least one of the accesses is a write, then such an instruction pair may contribute to a race condition. Our aim is to individually report all such race condition pairs, for **both** path-insensitive as well as path-sensitive MHP analyses that we have performed in the previous sections.

5.1 Algorithm

For each phase, say P, and for each pair of instructions, say I_1 and I_2 that may get executed in P, if there exists a shared location (a global variable), such that both I_1 and I_2 access this shared location and at least one of the accesses is a write, then we add the pair (I_1, I_2) to the set of race condition pairs.

Note: It is sufficient to put either of (I_1, I_2) or (I_2, I_1) in the set of race condition pairs, if I_1 and I_2 may contribute to a race condition.

5.2 Example

In case of path-insensitive MHP analysis of the code in Figure 2, the race condition pair is obtained from **Phase B**, where instructions at line #11 and #19 both read and write to a shared location `g2` in parallel.

In case of path-sensitive MHP analysis for the code in Figure 2, we obtain **no** race condition pairs from any of the phases.

6 Expected output

With so much information to derive and report, the output format may get very complicated to follow. So in A4, you are not expected to print *anything!*

We will be shortly providing you with a sample LLVM pass file that will already contain some data structures to represent the phase information and race condition information for both types of MHP analyses. We will also provide a function, named `dumpInformation()` that will print these data structures. The sample pass file will contain a call to `dumpInformation()` at the end of the pass. Your task is to write the code before this call-site, to populate the various global data structures that are pre-declared in the sample file. Important: Do NOT make any changes in the definition of `dumpInformation()`!

That's all folks! All the best!