CS6235 - Analysis of Parallel Programs

Introduction

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Written assignment = 1 x 10 marks.
- Programming assignments = 2 x 10 marks,
- Project = 10 marks.
- Quiz 1 = 15 marks, Quiz 2 = 15 marks, Final = 30 marks.
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a good question, answer a chosen question, make a good point! Take 0.5 marks each. Max one mark per day per person.
- Attendance requirement – as per institute norms. Non compliance will lead to 'W' grade.
  - Proxy attendance - is not a help; actually a disservice.
- Plagiarism - A good word to know. A bad act to own.
  - Will be automatically referred to the institute welfare and disciplinary committee.

Contact (Anytime) :
Instructor: Krishna, Email: nvk@iitm.ac.in, Office: BSB 352.
TA : Ramya Kasaraneni (cs19d003@smail)

What, When and Why of Program Analysis

- **What**:
  - A process of automatic analysis of computer programs regarding different program properties.
- **How?**
  - Analyze the program with or without executing!
- **Why? Study?**
  - Give guarantees about the correctness of program optimization, effectiveness of program optimization, program safety, and so on.
  - Used by compilers, debuggers, verifiers, IDEs, profilers.
  - A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer.
  - Handy, if you care about the programs you write!

Flavors of Program Analyses

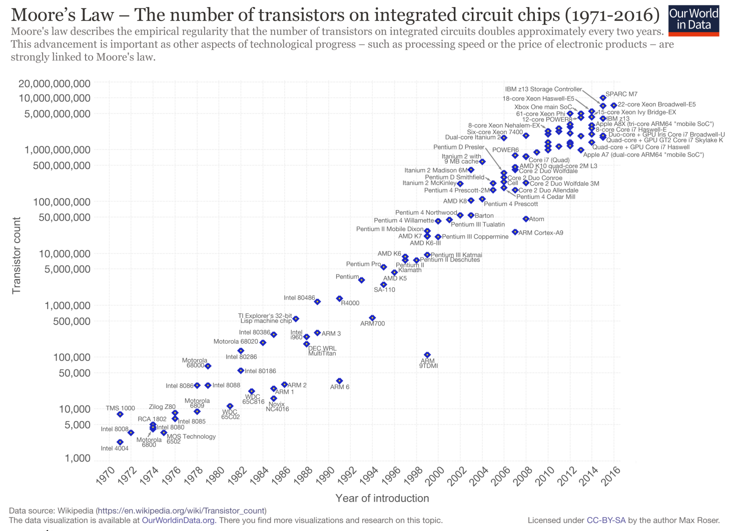
You have a goal? I have an analysis.

1.	Constant Replacement	Constant propagation
2.	Method Inlining	Points-to analysis
3.	Remove Null Pointer checks	Points-to analysis
4.	Loop parallelization	Dependence analysis
5.	Remove array out of bounds checks	Bounds check
6.	Debugging	Program slice
7.	Register allocation	Liveness analysis
8.	Find-definition (IDE)	Def-use analysis
9.	Dead-code elimination	Reaching-definition analysis
10.	Program-safety	Type checking
11.	Barrier elimination	MHP Analysis
12.	Race Detection	MHP Analysis

Why Parallel Program Analysis?

- Parallel systems have become mainstay (Why? - holdon).
- Automatic Extraction of parallelism has not been very successful.
- The community is looking at writing parallel programs.
- Analysis of parallel programs is a natural consequence.

Why Parallel Systems / Multicores?



Focus on increasing the number of computing cores.

What, When Multicores? **Why not** Multiprocessors

- **What** A multi-core processor is composed of two or more independent cores. Composition involves the interconnect, memory, caches.
- **When** IBM POWER4, the world's first dual-core processor, released in 2001.
- **Why not Multi-processors**
  - An application can be "threaded" across multiple cores, but not across multi-CPUs – communication across multiple CPUs is fairly expensive.
  - Some of the resources can be shared. For example, on Intel Core Duo: L2 cache is shared across cores, thereby reducing further power consumption.
  - Less expensive: A single CPU board with a dual-core CPU Vs a dual board with 2 CPUs.

Course outline

A rough outline (we may not strictly stick to this).

- Parallel programming constructs basics.
- Program analysis basics
- Parallel program representation
- MHP analysis and its impact on traditional analysis
- Parallel-Program Specific analysis
- Advanced Topics (depending on time).

Your friends: Languages and Tools

Start exploring

- Java - familiarity a must - Use eclipse to save you valuable coding and debugging cycles.
- JavaCC, JTB – tools you will learn to use.
- Make Ant Scripts – recommended toolkit.
- Find the course webpage: http://www.cse.iitm.ac.in/∼krishna/cs6235/

Get set. Ready steady go!

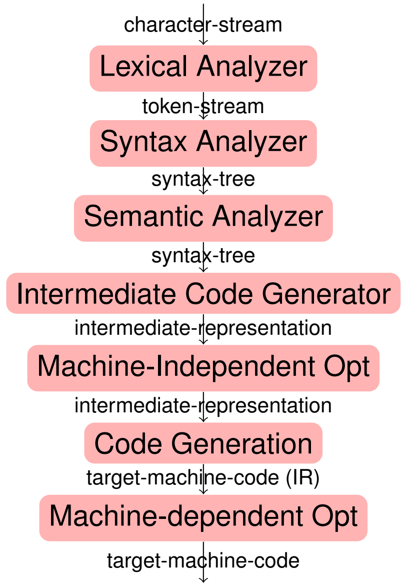
Expectations

What qualities are important in a program analysis?

1. Should identify properties correctly.
2. Should analyze all valid programs.
3. Analysis runs fast
4. Analysis time proportional to program size
5. Support for modular analysis.
6.

Each of these shapes your expectations about this course

Phases inside the compiler



Front end responsibilities:

- Recognize syntactically legal code; report errors.
- Recognize semantically legal code; report errors.
- Produce IR.

Back end responsibilities:

- Optimizations, code generation.

Program Analysis:

- After parsing.

Parallel Programs:

- Impacts both FE and BE

Examples of how Parallelism Impacts Analysis I/III

```
function int Withdraw(int amount){
     if (balance > amount) {
           balance = balance - amount;
           return SUCCESS;
     }
     return FAIL;
}
```

- Say `balance` = 100.
- Two parallel threads executing `Withdraw`(80)
- At the end of the execution, it may so happen that both of the withdrawals are successful. Further `balance` can still be 20!

Race freedom is enough?

```
void deposit(int amt) {
   acquire(m);
   balance = balance+amt;
   release(m);
}
int read_balance() {
   int t;
   acquire(m);
   t = balance;
   release(m);
   return t;
}
```

```
int withdraw(int amt) {
   int t = read_balance();
   acquire(m);
   if (t <= amt) {
      balance = 0;
   } else {
      balance = balance-amt;
      t = amt;
   }
   release(m);
   return t;
}
```

```
// Initial balance = 10.
fork  withdraw(10); ;        // Thread 1
fork  deposit(10); ;         // Thread 2
```

Example taken from Flanagan and Qadeer TLDI 2003.

Examples of how Parallelism Impacts Analysis II/III

```
for (int i = ...) {
  X[f(i)] = ... ;
  async { ... = X[g(i)]; }
}
```

$\Longrightarrow$ // Legal transformation?

```
// After loop distribution
for (int i = ...)
  X[f(i)] = ... ;
for (int i = ...)
    async { ... = X[g(i)]; }
```

Outline

Sources of speedups in Parallel Programs

- Say a serial Program P takes T units of time.
- Q: How much time will the best parallel version P' take (when run on N number of cores)? $\frac{T}{N}$ units?
- Linear speedups is almost unrealizable, especially for increasing number of compute elements.
- $T_{total} = T_{setup} + T_{compute} + T_{finalization}$
- T_{setup} and $T_{finalization}$ may not run concurrently - represent the execution time for the non-parallelizable parts of code.
- Best hope : $T_{compute}$ can be fully parallelized.
- $T_{total}(N) = T_{setup} + \frac{T_{compute}}{N} + T_{finalization} \ldots\ldots\ldots (1)$
- Speedup $S(N) = \frac{T_{total}(1)}{T_{total}(N)}$. In practice?
- Chief factor in performance improvement : Serial fraction of the code.

Amdahl's Law

- Serial fraction $\gamma = \frac{T_{setup}+T_{finalization}}{T_{total}(1)}$
- Fraction of time spent in parallelizable part = $(1-\gamma)$

$$
\begin{aligned}
T_{total}(N) &= \underbrace{\gamma \times T_{total}(1)}_{\text{serial code}} + \underbrace{\frac{(1-\gamma)\times T_{total}(1)}{N}}_{\text{parallel code}} \\
&= \left(\gamma + \frac{1-\gamma}{N}\right) \times T_{total}(1) \\
\text{Speedup } S(N) &= \frac{T_{total}(1)}{(\gamma+\frac{1-\gamma}{N})\times T_{total}(1)} \\
&= \frac{1}{(\gamma+\frac{1-\gamma}{N})} \\
&\approx \frac{1}{\gamma} \qquad \ldots\text{Amdahl's Law}
\end{aligned}
$$

- Max speedup is inversely proportional to the serial fraction of the code.

Implications of Amdahl's law

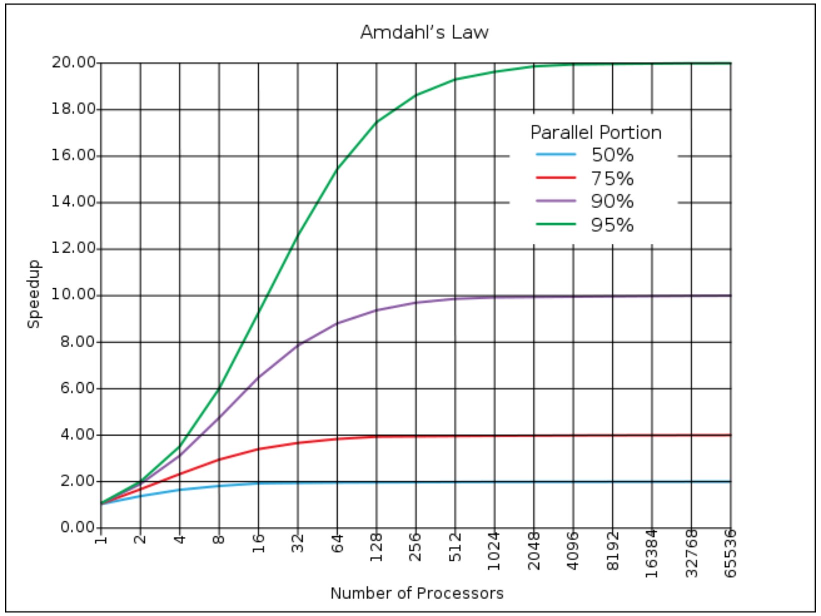
Assume: Ten processors. Goal: 10 fold speedup.

Parallel fraction	Serial fraction	Speedup = $\frac{1}{(\gamma+\frac{1-\gamma}{N})}$
40 %	60 %	2.17
20 %	80 %	3.57
10 %	90 %	5.26
99 %	01 %	9.17

Implications of Amdahl's law

- As we increase the number of parallel compute units, the speed up need not increase - an upper limit on the usefulness of adding more parallel execution units.
- For a given program maximum speedup nearly remains a constant.
- Say a parallel program spends only 10% of time in parallelizable code. If the code is fully parallelized, as we aggressively increase the number of cores, the speedup will be capped by $(\sim)$ 1.11$\times$.
- Say a parallel program spends only 10% of time in parallelizable code. Q: How much time would you spend to parallelize it?
- Amdahl's law helps to set realistic expectations for performance gains from the parallelization exercise.
- Mythical Man-month - Essays on Software Engineering. Frederic Brooks.

Peaking via Amdahl's law



Limitations of Amdahl's law

- An over approximation : In reality many factors affect the parallelization and even fully parallelizable code does not result in linear speed ups.
- Overheads exist in parallel task creations/termination/synchronization.
- Does not say anything about the impact of cache - may result in much more or far less improvements.
- Dependence of the serial code on the parallelizable code - can the parallelization in result in faster execution of the serial code?
- Amdahl's law assumes that the problem size remains the same after parallelization: When we buy a more powerful machine, do we play only old games or new more powerful games?

Discussion: Amdahl's Law

- When we increase the number of cores - the problem size is also increased in practise.
- Also, naturally we use more and more complex algorithms, increased amount of details etc.
- Given a fixed problem, increasing the number of cores will hit the limits of Amdahl's law. However, if the problem grows along with the increase in the number of processors - Amdahl's law would be pessimistic
- Q: Say a program P has been improved to P' (increase the problem size) - how to keep the running time same? How many parallel compute elements do we need?

Example of how Parallelism Impacts Analysis III/III

```
          A = new int[n]; // initialized to 0.

      T1                            T2
     ----                          ----
for (i=1;i<n;++i){            for (j=1;j<n ;++j){
  A[i] = i;                     assert(A[j] >= A[j+1]-1)
}                             }
```

- Q: Can the computation loop be parallelized?
- Is it safe?

Outline

Concurrency in programs

	Processes	Threads	Tasks
1	A program in execution	Light weight process	sequence of instructions
2	Shared mem: 1 process/run	1 or more threads per process	one more tasks can be executed by a thread
3	Distributed mem: 1 or more processes	1 or more threads per process	one more tasks can be executed by a thread
4	**C:** `fork`	**Java:**`new Thread()` **C:** `pthread_create()`	**X10:** `async S`
5.	Does NOT share heap/stack	Shares Heap	Share stack + heap
6.	Scheduled by the OS	Scheduled by the runtime	Shared by the threads

Processes - Example

```
int *y;
void main(){
  int done = 0;
  y=calloc(1,4);
  printf("1. Before forking\n");
  if (fork() == 0){
    printf("2a. In the Child\n");
    done = 1;
    while (*y == 0);
    printf("2b. Ending the Child\n");
    exit(0);
  } else {
     printf("3a. After forking\n");
     while (!done) ;
     *y = 1;
     printf("3b. Before waiting\n");
     wait();
  }
  printf("4. Bye\n"); }
```

What is the output?

Threads - Example

```
int *y;
void main(){
int done = 0;
y=calloc(1,4);
printf("1. Before forking\n");
if (create_thread() == 0){ // hypothetical call
  printf("2a. In the Child\n");
  done = 1;
  while (*y == 0);
  printf("2b. Ending the Child\n");

} else {
   printf("3a. After creating thread\n");
   while (!done) ;
   *y = 1;
   printf("3b. Before waiting\n");
   wait();
}
printf("4. Bye\n"); }
```

What is the output?

Tasks - Example

```
int *y;
void main(){
  int done = 0;
  y=calloc(1, 4);
  printf("1. Before forking\n");
  async{ // create task
    printf("2a. In the Child\n");
    done = 1;
    while (*y == 0);
    printf("2b. Ending the Child\n");
  }
   printf("3a. After creating task\n");
   while (!done) ;
   *y = 1;
   printf("3b. Before waiting\n");
   wait();

   printf("4. Bye\n"); }
```

What is the output?

Operations on or by processes / threads /tasks

- Creation.
- Execute in parallel with each other.
- May communicate with each other (data / synchronization).
- Termination.

Outline

Java Threads Vs Processes

- Each instance of JVM creates a single process.
- Each process creates one or more threads.
  - Main thread creates the others.

Java Threads

- Each Java thread is an object - instance of the Java Thread class.
- An application that creates an instance of Thread must provide the code that will run in that thread.
    - `implement Runnable` interface.
    - Provide an implementation of the `run` method.

  or

    - `extend Thread` class
    - Provide an overridden implementation of the `run` method.

  Adv/Disadv??

    - (Hint) Java allows single inheritance.
    - Extending thread class ⇒ cannot extend any other class.

What can a thread do?

- Start executing the thread body specified in the `run` method.
- Sleep - `Thread.sleep(..)`
- Wait for child threads to finish: `ch.join()`
- Communicate with other threads.

Communication via synchronized methods

- `synchronized` statements and methods. Making a methods synchronized has two effects:
    - It is not possible for two invocations of synchronized methods on the same object to interleave.
    - When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object.
        - Guarantees that changes to the state of the object are visible to all threads.
- Constructors cannot be synchronized. Why? Consequence - only the object creating threads should call the constructor.
- synchronized methods ensure that there is no thread interference.
- Q: Too many synchronized methods. Disadv?

How do synchronized methods/statements work?

- Each object has an associated intrinsic lock.
- When a thread invokes a synchronized method,
    - automatically acquires the intrinsic lock for that method's object
    - releases the lock when the method returns (normal or via exception).
- What if a thread invokes a synchronized method recursively?

Updating shared variables

Java guarantees that following actions would be atomic.

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared volatile (including long and double variables).
- Any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.
- We can also declare a variable (of some types) as atomic.

Atomic variables

- The `java.util.concurrent.atomic` package defines classes supporting atomic operations on single variables.
- Supports many types of `get` and `set` operations.
- Like `volatile` variables' write operation, the `set` operation has an happens-before relation with the corresponding `get` operation.

Atomic variables - Example

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
                // guarantees thread non-interference.
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

Atomic variables (contd)

- Supported classes:

AtomicBoolean	AtomicInteger	AtomicIntegerArray
...	AtomicLong	AtomicLongArray
...	LongAccumulator	LongAdder

- All support: `boolean compareAndSet(expectedValue, updateValue)`
- CAS operation: How to use it to realize synchronization?
- Building blocks for implementing 'non-blocking' data structures.

Happens before relation

- Sequential order: Each action in a thread happens-before every action in that thread that comes later in the program's order.
- Unlock → Lock: An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor.
- Volatile writes: A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.
- A call to start on a thread happens-before any action in the started thread.
- All actions in a thread happen-before any other thread successfully returns from a join on that thread.

Happens-before relation is transitive.

Deadlock, Livelock and Starvation

- Deadlock: two or more threads are blocked forever, waiting for each other.
- Starvation: a thread is unable to gain regular access to shared resources and is unable to make progress.
- Livelock: threads are not blocked, but are not making any progress.

Guarded blocks

- A way to coordinate with others.
- A guarded block
  - polls a condition that has to be true to proceed.
  - other threads set that condition

```
public void guardedEntry(){
    while (!flag) ;
   // flag is set. Inefficient.
}
```

Guarded blocks (contd)

```
public synchronized void guardedEntry() {
    // Check once and "wait"
    while(!flag) { // Loop is needed.
        try {
            wait(); // releases the lock
        } catch (InterruptedException e) {}
    }
    // flag is set. Efficient.
}
```

- Q: Why synchronized?
- `notify` vs `notifyAll`

Immutable Objects

- An object is considered immutable if its state cannot change after it is constructed.
- Helps write reliable code.
- cannot be corrupted by thread interference or observed in an inconsistent state.
- Creating many immutable objects Vs updating existing objects.
  - Cost of object creation, GC
  - Code needed to protect mutable objects from corruption.

Example

```
public class SynchronizedRGB {
  private int red;   // between 0 - 255.
  private int green; // between 0 - 255.
  private int blue;  // between 0 - 255.
  private String name;

  public synchronized void set(int r, int g, int b,
                                String n) {..}
  public synchronized int getRGB() {
      return ((red << 16) | (green << 8) | blue);
  }
  public synchronized String getName() { return name; }
  public synchronized void invert() {
      red=255-red; green=255-green; blue=255-blue;
      name="Inverse of " + name;
  } }
```

Example (contd)

```
...
SynchronizedRGB color =
    new SynchronizedRGB(0, 0, 0, "Black");

int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

What if another threads updates the color object after Statement 1?

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

Such issues do not arise with immutable objects

Mutable to Immutable

General guidelines:

- Don't provide 'setter' methods.
- Make all fields final and private.
- Don't allow subclasses to override methods or provide 'setter' methods.
  - declare the class as final.
  - make constructor final and provide a factory method.
- If the instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't provide methods that modify the mutable objects.
  - Don't share references to the mutable objects. Copy and share if required.

Mutable to Immutable (Example)

```
final public class ImmutableRGB {
  final private int red;   // between 0 - 255.
  final private int green; // between 0 - 255.
  final private int blue;  // between 0 - 255.
  final private String name;

  public /*synchronized*/ int getRGB() {
        return ((red << 16) | (green << 8) | blue); }
  public /*synchronized*/ String getName()
      { return name; }
  public /*synchronized*/ ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                     255 - green, 255 - blue,
                     "Inverse of " + name);
    } }
```

No synchronized methods required!

Deadlocks in Locks

```
public class Deadlock {
 class Friend {
  private final String name;
  public Friend(String name){this.name = name; }
  public String getName() {return this.name; }
  public synchronized void bow(Friend bower) {
   System.out.format("%s: %s" +" has bowed to me!",
     this.name, bower.getName());
   bower.bowBack(this); }
  public synchronized void bowBack(Friend bower) {
   System.out.format("%s:%s"+" has bowed back!",
    this.name, bower.getName()); } }
```

Deadlocks in Locks (contd)

```
public static void main(String[] args) {
    final Friend alpha = new Friend("Alpha");
    final Friend beta = new Friend("Beta");
    new Thread(new Runnable() {
        public void run() { alpha.bow(beta); }
    }).start();
    new Thread(new Runnable() {
        public void run() { beta.bow(alpha); }
    }).start();
} }
```

Avoid deadlocks in Locks

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) { this.name=name;}

        public String getName() { return this.name;}
```

Avoid deadlocks in Locks (cont.)

```
public boolean impendingBow(Friend bower) {
    Boolean myLock = false;
    Boolean yourLock = false;
    try {
        myLock = lock.tryLock();
        yourLock = bower.lock.tryLock();
    } finally {
        if (! (myLock && yourLock)) {
            if (myLock) {
                lock.unlock();
            }
            if (yourLock) {
                bower.lock.unlock();
            } } }
    return myLock && yourLock;
}
```

Avoid deadlocks in Locks (cont.)

```
public void bow(Friend bower) {
    if (impendingBow(bower)) {
        try {
            System.out.format("%s: %s has"
                + " bowed to me!",
                this.name, bower.getName());
            bower.bowBack(this);
        } finally {
            lock.unlock();
            bower.lock.unlock(); }
    } else {
        System.out.format("%s: %s started"
            + " to bow to me, but saw that"
            + " I was already bowing to him.",
            this.name, bower.getName());
    } }
```

Avoid deadlocks in Locks (cont.)

```
        public void bowBack(Friend bower) {
            System.out.format("%s: %s has" +
                " bowed back to me!%n",
                this.name, bower.getName());
        }
    }
```

Avoid deadlocks in Locks (cont.)

```
class BowLoop implements Runnable {
    private Friend bower;
    private Friend bowee;

    public BowLoop(Friend bower, Friend bowee) {
        this.bower = bower; this.bowee = bowee;}
    public void run() {
        Random random = new Random();
        for (;;) {
            try {
                Thread.sleep(random.nextInt(10));
            } catch (InterruptedException e) {}
            bowee.bow(bower);
        } } }
```

Avoid deadlocks in Locks (cont.)

```
    public static void main(String[] args) {
        final Friend alpha = new Friend("Alpha");
        final Friend beta = new Friend("Beta");
        new Thread(new BowLoop(alpha, beta)).start();
        new Thread(new BowLoop(beta, alpha)).start();
    }
}
```

Barriers in Java

- `CyclicBarrier`
  - allows a set of threads to wait for each other.
  - can be reused after the end of one phase.

```
class MainClass{
  final CyclicBarrier barrier;
  MainClass(){
    barrier = new CyclicBarrier(NumThreads,
                     new Runnable() {
                        public void run (){
                          execute-at-the-end-of-phase;
                        }});
   for(int i=0;i<NumThreads;++i){
     new Thread(new mThread().start());
   } } }
```

Barriers in Java (cont.)

```
class mThread implements Runnable {
  public void run(){
    while (not-done)
      do-some-computation;
      barrier.await();
      // may throw InterruptedExecution or
      // BrokenBarrierException
  }
}
```

- All or none breakage model.
- Memory consistency effects and happen before relations:

  `S1 Barrier{Sb} S2`
  `S3 Barrier{Sb} S4`

  `S1` and `S3` happen before `Sb` and `Sb` happens before `S2` and `S4`

Tasks and ThreadPools in Java

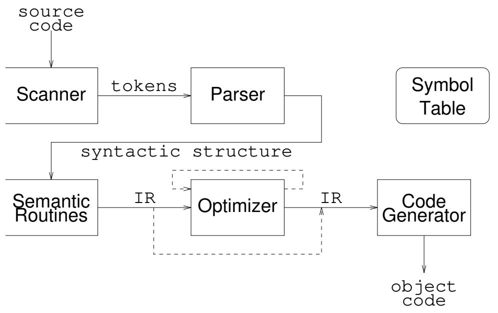
- ThreadPool: reuses previously created threads to execute current tasks
- Threads are created once and reused across all the tasks.
  - Less overhead.
  - When task/tasks arrive(s), the threads are ready.
  - Avoids resource thrashing caused by creating threads arbitrarily.

ThreadPools and Tasks example

```
class Task implements Runnable {
  public void run() {...}
}

class Main {
 public static void main(String[] args){
  Runnable r1 = new Task(..);  // Create tasks.
  Runnable r2 = new Task(..);
  ...

  // Create the pool.
  ExecutorService pool=Executors.newFixedThreadPool(max_t);

  // pass the tasks to the pool.
  pool.execute(r1);
  pool.execute(r2);
  ...
  pool.shutdown(); // shutdown - tasks cannot be added.
 } }
```

Outline

Program Analysis



- Code optimization requires that the compiler has a global "understanding" of how programs use the available resources.
- It has to understand how the control flows (control-flow analysis) in the program and how the data is manipulated (data-flow analysis)
- Control-flow analysis: flow of control within each procedure and across procedures.
- Data-flow analysis: how the data is manipulated in the program.

Example

```
int fib (int m){
   int f0=0, f1=1, f2,i;
   if (m <=1)
      return m;
   else {
      for (i=2; i<=m; ++i) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
}
```

```
1       receive m (val)
2       f0 = 0
3       f1 = 1
4       if (m <= 1) goto L3
5       i = 2
6  L1:  if (i<=m) goto L2
7       return f2
8  L2:  f2 = f0 + f1
9       f0 = f1
10      f1 = f2
11      i = i + 1
12      goto L1
13 L3:return m
```

- IR for the C code (in a format described in Muchnick book)
- `receive` specifies the reception of a parameter and the parameter-passing discipline (by-value, by-result, value-result, reference). Why do we want to have an explicit receive instruction?– Gives a point of definition for the args.
- What is the control structure? Obvious?

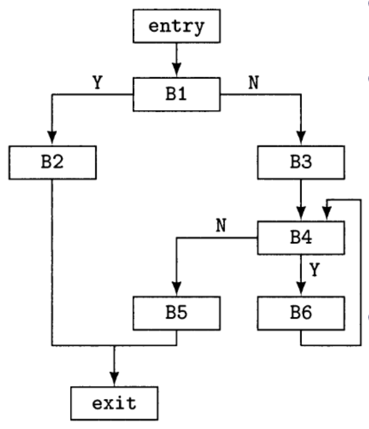
Example - flow chart and control-flow

```
int fib (int m){
   int f0=0, f1=1, f2,i;
   if (m <=1)
      return m;
   else {
      for (i=2; i<=m; ++i) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
}
```

```
      receive m (val)
      f0 = 0
      f1 = 1
      if (m <= 1) goto L3
      i = 2
  L1: if (i<=m) goto L2
      return f2
  L2: f2 = f0 + f1
      f0 = f1
      f1 = f2
      i = i + 1
      goto L1
 L3:return m
```

- The high-level abstractions might be lost in the IR.
- Control-flow analysis can expose control structures not obvious in the high level code. Possible? Loops constructed from `if` and `goto`
- A basic block is informally a straight-line sequence of code that can be entered only at the beginning and exited only at the end.

Basic blocks - what do we get?



- `entry` and `exit` are added for reasons to be explained later.
- We can identify loops by using dominators
  - a node A in the flowgraph dominates a node B if every path from `entry` node to B includes A.
  - This relations is antisymmetric, reflexive, and transitive.
- back edge: An edge in the flow graph, whose head dominates its tail (example - edge from `B6` to `B4`.
- A loop consists of subset of nodes dominated by its entry node (head of the back edge) and having exactly one back edge in it.

Deep dive - Basic block

Basic block definition

- A basic block is a maximal sequence of instructions that can be entered only at the first of them
- The basic block can be exited only from the last of the instructions of the basic block.
- Implication:First instruction can be a) entry point of a routine,b) item target of a branch, c) item instruction following a branch or a return.
- First instruction is called the leader of the BB.

How to construct the basic block?

- Identify all the leaders in the program.
- For each leader: include in its basic block all the instructions from the leader to the next leader (next leader not included) or the end of the routine, in sequence.

What about function calls?

- In most cases it is not considered as a branch+return. Why?
- Problem with setjmp() and longjmp()? [self-study]

CFG - Control flow graph

Definition:

- A rooted directed graph $G = (N, E)$, where N is given by the set of basic blocks + two special BBs: `entry` and `exit`.
- And edge connects two basic blocks b_1 and b_2 if control can pass from b_1 to b_2.
- An edge(s) from `entry` node to the initial basic block(s?)
- From each final basic blocks (with no successors) to `exit` BB.

CFG continued

- successor and predecessor – defined in a natural way.
- A basic block is called branch node - if it has more than one successor.
- join node – has more than one predecessor.
- For each basic block b:

$$Succ(b) = \{n \in N | \exists e \in E \text{ such that } e = b \to n\}$$
$$Pred(b) = \{n \in N | \exists e \in E \text{ such that } e = n \to b\}$$

- A region is a strongly connected subgraph of a flow-graph.

Data Flow Analysis

Why:

- Provide information about a program manipulates its data.
- Study functions behavior.
- To help build control flow information.
- Program understanding (a function sorts an array!).
- Generating a model of the original program and verify the model.
- The DFA should give information about that program that does not misrepresent what the procedure being analyzed does.
- Program validation.

Reaching Definitions

A particular definition of a variable is said to reach a given point if

- there is an execution path from the definition to that point
- the variable might may have the value assigned by the definition.

In general undecidable.

Our goal:

- The analysis must be conservative – the analysis should not tell us that a particular definition does not reach a particular use, if it may reach.
- A 'may' conservative analysis gives us a larger set of reaching definitions than it might, if it could produce the minimal result.

To make maximum benefit from our analysis, we want the analysis to be conservative, but be as aggressive as possible.

Different types of analysis

- Intra procedural analysis.
- Whole program (inter-procedural) analysis.
- Generate intra procedural analysis and extend it to whole program.

We will study an iterative mechanism to perform such analyses.

Iterative Dataflow Analysis

- Build a collection of data flow equations – specifying which data may flow to which variable.
- Solve it iteratively.
- Start from a conservative set of initial values – and continuously improve the precision.
  Disadvantage: We may be handling large data sets.
- Start from an aggressive set of initial values – and continuously improve the precision.
  Advantage: Datasets are small to start with.
- Choice – depends on the problem at hand.

Example program

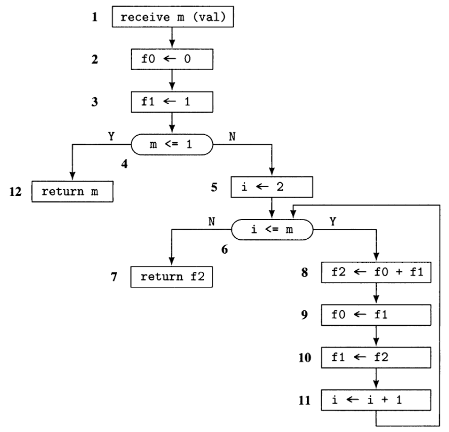
```
1 int g (int m, int i);

2 int f(int n) {
3    int i = 0, j;
4    if (n == 1) i = 2;
5    while (n > 0) {
6       j = i + 1;
7       n = g(n, i);
8    }
9    return j;
10 }
```

- Does def of `i` in line 3 reach the uses in line 6 and 7?
- Does def of `j` in line 6 reach the use in line 9?

Definitions

- GEN : GEN(b) returns the set of definitions generated in the basic block b; assigned values in the block and not subsequently killed in it.
- KILL : KILL(b) returns the set of definitions killed in the basic block b.
- IN : IN(b) returns the set of definitions reaching the basic block b.
- OUT : OUT(b) returns the set of definitions going out of basic block b.
- PRSV : Negation of KILL

Representation and Initialization



Bit Pos	Definition	Basic Block
1	`m` in node 1	B1
2	`f0` in node 2	
3	`f1` in node 3	
4	`i` in node 5	B3
5	`f2` in node 8	B6
6	`f0` in node 9	
7	`f1` in node 10	
8	`i` in node 11	

	Set rep	Bit vector
$GEN(B1)$	$= \{1, 2, 3\}$	$\langle 11100000 \rangle$
$GEN(B3)$	$= \{4\}$	$\langle 00010000 \rangle$
$GEN(B6)$	$= \{5, 6, 7, 8\}$	$\langle 00001111 \rangle$
$GEN(.)$	$= \{\}$	$\langle 00000000 \rangle$

Populating PRSV, OUT and IN

	Set rep	Bit vector
$PRSV$(B1)	= $\{4, 5, 8\}$	$\langle 00011001 \rangle$
$PRSV$(B3)	= $\{1, 2, 3, 5, 6, 7\}$	$\langle 11101110 \rangle$
$PRSV$(B6)	= $\{1\}$	$\langle 10000000 \rangle$
$PRSV$(.)	= $\{1, 2, 3, 4, 5, 6, 7, 8\}$	$\langle 11111111 \rangle$

Dataflow equations

A definition may reach the end of a basic block i:

$$OUT(i) = GEN(i) \cup (IN(i) \cap PRSV(i))$$

or with bit vectors:

$$OUT(i) = GEN(i) \vee (IN(i) \wedge PRSV(i))$$

A definition may reach the beginning of a basicblock i:

$$IN(i) = \bigcup_{j \in Pred(i)} OUT(j)$$

- GEN, $PRSV$ and OUT are created in each basic block.
- $OUT(i)$ = $\{\}$ // initialization
- But IN needs to be initialized to something safe.
- $IN(entry)$ = $\{\}$

Solving the Dataflow equations: example

Itr 1:

$OUT(entry)$	$= \langle 00000000 \rangle$	$IN(entry)$	$= \langle 00000000 \rangle$
$OUT(B1)$	$= \langle 11100000 \rangle$	$IN(B1)$	$= \langle 00000000 \rangle$
$OUT(B2)$	$= \langle 11100000 \rangle$	$IN(B2)$	$= \langle 11100000 \rangle$
$OUT(B3)$	$= \langle 11110000 \rangle$	$IN(B3)$	$= \langle 11100000 \rangle$
$OUT(B4)$	$= \langle 11110000 \rangle$	$IN(B4)$	$= \langle 11110000 \rangle$
$OUT(B5)$	$= \langle 11110000 \rangle$	$IN(B5)$	$= \langle 11110000 \rangle$
$OUT(B6)$	$= \langle 00001111 \rangle$	$IN(B6)$	$= \langle 11110000 \rangle$
$OUT(entry)$	$= \langle 11110000 \rangle$	$IN(exit)$	$= \langle 11110000 \rangle$

Itr 2:

$OUT(entry)$	$= \langle 00000000 \rangle$	$IN(entry)$	$= \langle 00000000 \rangle$
$OUT(B1)$	$= \langle 11100000 \rangle$	$IN(B1)$	$= \langle 00000000 \rangle$
$OUT(B2)$	$= \langle 11100000 \rangle$	$IN(B2)$	$= \langle 11100000 \rangle$
$OUT(B3)$	$= \langle 11110000 \rangle$	$IN(B3)$	$= \langle 11100000 \rangle$
$OUT(B4)$	$= \langle 11111111 \rangle$	$IN(B4)$	$= \langle 11111111 \rangle$
$OUT(B5)$	$= \langle 11111111 \rangle$	$IN(B5)$	$= \langle 11111111 \rangle$
$OUT(B6)$	$= \langle 10001111 \rangle$	$IN(B6)$	$= \langle 11111111 \rangle$
$OUT(entry)$	$= \langle 11111111 \rangle$	$IN(exit)$	$= \langle 11111111 \rangle$

Dataflow equations: behavior

- We specify the relationship between the data-flow values before and after a block – transfer or flow equations.
  - Forward: $OUT(s) = f(IN(s), \cdots)$
  - Backward: $IN(s) = f(OUT(s), \cdots)$
- The rules never change a 1 to 0. They may only change a 0 to a 1.
- They are monotone.
- Implication – the iteration process will terminate.
- Q: What good is reaching definitions? undefined variables.
- Q: Why do the iterations produce an acceptable solution to the set of equations? – lattices and fixed points.

Lattice

- **What** : Lattice is an algebraic structure
- **Why** : To represent abstract properties of variables, expressions, functions, etc etc.
  - Values
  - Attributes
  - ...
- **Why "abstract?** Exact interpretation (execution) gives exact values, abstract interpretation gives abstract values.

Lattice definition

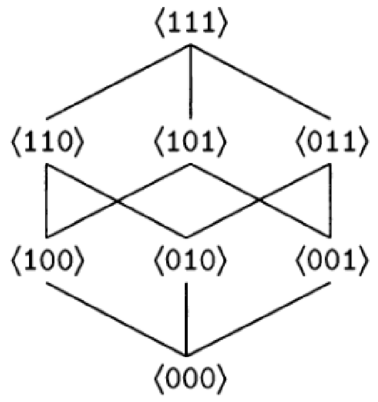
A lattice L consists of a set of values, and two operations called *meet* ($\sqcap$) and *join* ($\sqcup$). Satisfies properties:

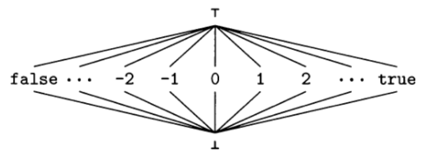
- **closure**: For all $x, y \in L$, $\exists$ a unique z and $w \in L$, such that $x \sqcup y = z$ and $x \sqcap y = w$ – each pair of elements have a unique lub and glb.
- **commutative**: For all $x, y \in L$, $x \sqcap y = y \sqcap x$, and $x \sqcup y = y \sqcup x$.
- **associative**: For all $x, y, z \in L$, $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$, and $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
- There exists two special elements of L called bottom ($\perp$), and top ($\top$).
  $\forall x \in L$, $x \sqcap \perp = \perp$ and $x \sqcup \top = \top$.
- **distributive** : (optional). $\forall x, y, z \in L$, $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$, and $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

Lattice properties

- Meet (and join) induce a partial order ($\sqsubseteq$):
  $\forall x, y \in L$, $x \sqsubseteq y$, iff $x \sqcap y = x$.
- Transitive, antisymmetry and reflexive.

Example Lattices:





Monotones and fixed point

- A function $f : L \rightarrow L$, is a monotone, if for all $x, y \in L$, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.
- Example: bit-vector lattice:
  - $f(x_1x_2x_3) = \langle x_1 1 x_2 \rangle$
  - $f(x_1x_2x_3) = \langle x_2x_3x_1 \rangle$
- A flow function models the effect of a programming language construct. as a mapping from the lattice for that particular analysis to itself.
- We want the flow functions to be monotones. Why?
- A fixed point of a function $f : L \rightarrow L$ is an element $z \in L$, such that $f(z) = z$.
- For a set of data-flow equations, a fixed-point is a solution of the set of equations – cannot generate any further refinement.

Meet Over All Paths solutions

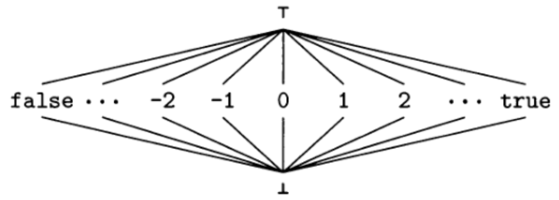
- The value we wish to compute in solving data-flow equations is – meet over all paths (MOP) solution.
- Start with some prescribed information at the entry (or exit depending on forward or backward).
- Repeatedly apply the composition of the appropriate flow functions.
- For each node form the meet of the results.

A worklist based implementation (a forward analysis)

```
procedure WorklistIterate (N: Set<Node>, entry: Node, F: Node $\times L \to L$, dfin: Node $\to L$, Init : $L$) // dfin is an output variable.
begin
    B, P: Node;
    worklist: Set<Node>;
    effect, totalEffect : $L$;
    dfin(entry) $\leftarrow$ Init;
    worklist = N $-$ {entry};
    foreach B $\in$ N do  dfin(B) $\leftarrow \top$ ;
    repeat
        B $\leftarrow$ worklist.removeOne();
        totalEffect $\leftarrow \top$;
        foreach P $\in$ Pred(B) do
            effect $\leftarrow$ F(P, dfin(P));
            totalEffect $\leftarrow$ totalEffect $\sqcap$ effect;
        if dfin(B) $\neq$ totalEffect then
            dfin(B) $\leftarrow$ totalEffect;
            worklist.add(Succ(B));
    until worklist = $\Phi$;
```

Example: Constant Propagation

Goal: Discover values that are constants on all possible executions of a program and to propagate these constant values as far forward through the program as possible
Conservative: Can discover only a subset of all the possible constants.
Lattice:



Constant Propagation lattice meet rules

- $\bot$ = Constant value cannot be guaranteed.
- $\top$ = May be a constant, not yet determined.
- $\forall x$
  - $x \sqcap \top = x$
  - $x \sqcap \bot = \bot$
  - $c_1 \sqcap c_1 = c_1$
  - $c_2 \sqcap c_1 = \bot$

Simple constant propagation

- Gary A. Kildall: A Unified Approach to Global Program Optimization - POPL 1973.
- Reif, Lewis: Symbolic evaluation and the global value graph - POPL 1977.
- **Simple constant** Constants that can be proved to be constant provided,
  - no information is assumed about which direction branches will take.
  - Only one value of each variable is maintained along each path in the program.

Kildall's algorithm

- Start with an entry node in the program graph.
- Process the entry node, and produce the constant propagation information. Send it to all the immediate successors of the entry node.
- At a merge point, get an intersection of the information.
- If at any successor node, if for any variable the value is "reduced, the process the successor, similar to the processing done for entry node.

Constant propagation - equations

- Let us assume that one basic block per statement.
- Transfer functions set F - a set of transfer functions. $f_s \in F$ is the transfer function for statement s.
- The dataflow values are given by a map: m: $Vars \rightarrow ConstantVal$
- If m is the set of input dataflow values, then $m' = f_s(m)$ gives the output dataflow values.
- Generate equations like before.

Constant propagation: equations (contd)

- Start with the entry node.
- If s is not an assignment statement, then f_s is simply the identity function.
- If s is an assignment statement to variable v, then $f_s(m) = m'$, where:
  - For all $v' \neq v$, $m'(v') = m(v')$.
  - If the RHS of the statement is a constant c, then $m'(v) = c$.
  - If the RHS is an expression (say $y\ op\ z$),

$$m'(v) = \begin{cases} m(y)\ op\ m(z) & \text{if } m(y) \text{ and } m(z) \text{ are constant values} \\ \perp & \text{if either of } m(y) \text{ and } m(z) \text{ is } \perp \\ \top & \text{Otherwise} \end{cases}$$

  - If the RHS is an expression that cannot be evaluated, then $m'(v) = \perp$.
- At a merge point, get a meet of the flow maps.

Constant Propagation - example I

```
x = 10;
y = 1;
z = 5;
if  (cond) {
   y = y / x;
   x = x - 1;
   z = z + 1;
} else {
   z = z + y;
   y = 0;
}
print x + y + z;
```

Constant Propagation - example II

```
x = 10;
y = 1;
z = 1;
while (x > 1) {
    y = x * y;
    x = x - 1;
    z = z * z;
}
A[x] = y + z;
```

Outline

Definitions

A variable is live at a program point, if it holds a value that may be needed in future

```
      a ← 0
L1 :  b ← a + 1
      c ← c + b
      a ← b × 2
      if a < N goto L1
      return c
```

- v is live on edge e if there is a directed path from SRC(e) to a use of v that does not pass through any *def*(v)
- v is live-in at node n if live on all of n's in-edges
- v is live-out at n if live on any of n's out-edges
- $v \in use[n] \Rightarrow v$ live-in at n
- v live-in at $n \Rightarrow v$ live-out at all $m \in pred[n]$
- v live-out at $n, v \notin def[n] \Rightarrow v$ live-in at n

Liveness analysis

Define:

$$\begin{aligned} \mathit{in}[n] &= \text{variables live-in at } n \\ \mathit{out}[n] &= \text{variables live-out at } n \end{aligned}$$

Then:

$$\mathit{out}[n] = \bigcup_{s \in \mathit{succ}(n)} \mathit{in}[s]$$

$$\mathit{succ}[n] = \phi \Rightarrow \mathit{out}[n] = \phi$$

Note:

$$\begin{aligned} \mathit{in}[n] &\supseteq \mathit{use}[n] \\ \mathit{in}[n] &\supseteq \mathit{out}[n] - \mathit{def}[n] \end{aligned}$$

$\mathit{use}[n]$ and $\mathit{def}[n]$ are constant (independent of control flow)
Now, $v \in \mathit{in}[n]$ iff. $v \in \mathit{use}[n]$ or $v \in \mathit{out}[n] - \mathit{def}[n]$
Thus, $\mathit{in}[n] = \mathit{use}[n] \cup (\mathit{out}[n] - \mathit{def}[n])$

Iterative algorithm

Recall the iterative forward analysis and port it to perform backward analysis. [DIY]

Flow Sensitive Vs Flow Insensitive

```
i: m = new X(); // Ri
j: n = new X(); // Rj
k: p = m;
l: p = n;
a: q = p;
b: n = m;
```

```
Flow sensitive (after 'b')       Flow insensitive:
m -> {Ri}                         m -> {Ri}
p -> {Rj}                         p -> {Ri, Rj}
q -> {Rj}                         q -> {Ri, Rj}
n -> {Ri}                         n -> {Ri, Rj}
```

Outline

Points-to/Alias Analysis

- Goal: Reason about what different variables / fields point to in the heap.

```
p = new A(); // R1
p.f = new Y(); // R2
if (cond) {
  q = new X(); // R3
  q.f = new Z(); // R4
  r1 = q;
} else {
  q = new X(); // R5
  q.f = new Z(); // R6
  r2 = q;
}
p.f = new Y(); // R7
q.f = new Z(); // R8
```

Points-to/Alias Analysis in Java

- Abstractly model Stack and Heap:
  - Vars: Set of all Variables.
  - Refs: Set of all References
  - Values: P(Refs) // P = Power set
  - Stack ρ: Vars $\rightarrow$ Values
  - Heap σ: Refs $\times$ Fields $\rightarrow$ Values
- Initialize the Stack and Heap
  - each local variable,
  - fields (of the locally allocated objects)

  $\longrightarrow$ point to the empty set.

Points-to/Alias Analysis Lattice

- L = Power set of all the abstract references.
- x $\sqcap$ y = x $\cup$ y
- x $\sqcup$ y = x $\cap$ y
- $\perp$ = Set of all the abstract references.
- $\top = \phi$

Points-to/Alias Analysis transfer functions

1.	alloc	$a : x =$ new ...	$\rho[x \leftarrow O_a]$
2.	copy	$x = y$	$\rho[x \leftarrow \rho(y)]$
3.	load	$x = y.f$	$\rho[x \leftarrow \cup_{\forall o \in \rho(y)} \sigma(o,f)]$
4a.	store (strong update)	$x.f = y$	$\forall o \in \rho(x),\ \sigma[(o,f) \leftarrow \rho(y)]$
4b.	store (weak update)	$x.f = y$	$\forall o \in \rho(x),\ \sigma[(o,f) \leftarrow \sigma(o,f) \cup \rho(y)]$

Dimensions of Analysis: Inter-/Intra- procedural

- Intra-procedural analysis:
  - Analyze each procedure independently.
  - At a call-site assume the worst-case scenario.
    - Constant propagation?
    - Points-to Analysis?
- Inter-procedural analysis:
  - Analyze the whole program, while being aware of the procedure details.
  - Need to know about who-calls-whom? - Call Graph.

Call Graph Construction

- Call Graph $G = (N, S, E, r)$.
- N = set of all the functions.
- $r \in N$ is the root (main method).
- S = set of call site labels (for example, line numbers).
- $\forall n_1, n_2 \in N$, we say $(n_1, s, n_2) \in E)$ if n_1 may call n_2 at call site s.
- In a function n_1, if there exists a call of the form `x.foo()`, then the edges to add: From n_1 to
  - every `foo` in the program (too conservative).
  - every `foo` present in the possible classes whose object `x` may contain.
    - Decide this list just based on the class hierarchy (CHA).
    - Decide this list by doing a flow analysis (CFA - not covered).

Class Hierarchy Analysis

```
class A{
  foo(){..}
  bar(){..}
}
class B extends A {
  foo(){..}
}
class C extends B {
  foo(){..}
}
class D extends C {
  foo(){..}
  bar(){..}
}
```

```
class Main {
 foo(){
    B x = ...
    x.foo();
    C y = ...
    y.bar();
  }
}
```

Call Graph?

Dimensions of Analysis

- Context insensitive (call site independent).
  - For each procedure in a program identify the subset of its parameters, such that each of the parameters will get a single "value", across all the invocations.
  - The procedure will return a single "value" across all the invocations.
- Context sensitive (call site dependent):
  - for each particular procedure called from a particular context (a particular site), the subset of parameters that have the same "value" each time the procedure is called at that site.
  - For each call site, the function may return a different "value".

What are the "values"? Depends on the analysis.

Context-insensitive points-to analysis

```
Function InterProcPointsTo(CallGraph CG)
worklist = {CG.root};
while worklist is not empty do
    $p$ = $worklist.dequeue()$;
    begin
        Process stmts in $p$ like before; // Intra-procedural analysis till fixed-point
        During the analysis, if we encounter a stmt $s$ of the form x=y.foo(..) then
            compute the actual arguments of $s$;
            foreach function $q$ that may be called at $s$ do
                Compute $meet$: current values for the formals of $q$ and the actuals;
                if the values of the arguments of $q$ have changed then
                    remember the new value and add $q$ to the worklist;
                "Update" the value of x and all the fields of the arguments to foo as
                 per the summary of $q$.
    $v$ = compute the meet of all the return values of $p$;
    Set the return value of $p$ to $v$;
    Set the summary of $p$ to include the final values of the formal arguments;
    foreach call function $q$ that calls $p$ do
        add $q$ to the worklist
```

Example I/II

```
class A{
  A f0;
  void foo(A x){
    f0 = new A();
    x.f0 = f0;
  } }
class B extends A{
  B f1;
  void foo(A x){
    f1 = new B();
    x.f0 = f1;
  }
  A getf1(A x){
    return f1;
  } }
```

```
class Main {
  public static void main (){
    B a1 = new B();
    a1.foo(a1);
    A a2 = a1.f0;
    A a3 = a1.getf1();
  // Q: a2 and a3 aliases?
    a2.foo(a3);
    A a4 = a2.f0;
  // Q: a2 and a4 aliases?
  }
}
```

Example II/II (Impact of imprecise call-graph)

```
class A{
  A f0;
  void foo(A x){
    f0 = new A();
    x.f0 = f0;
  } }
class B extends A{
  B f1;
  void foo(A x){
    f1 = new B();
    x.f0 = f1;
    if (*) x.foo(x);
  }
  A getf1(A x){
    return f1;
  } }
```

```
class Main {
  public static void main (){
    B a1 = new B();
    a1.foo(a1);
    A a2 = a1.f0;
    A a3 = a1.getf1();
  // Q: a2 and a3 aliases?
  }
}
```

Dependence Analysis - Skipped.

Outline

Symbol table information

A compiler uses symbol table to store many different types of information.

What kind of information might the compiler need?

- textual name
- data type
- dimension information (for aggregates)
- declaring procedure
- lexical level of declaration
- storage class (base address)
- offset in storage
- if record, pointer to structure table
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions
- ...

Symbol table organization

How should the table be organized?

- Linear List
  - **O**(n) probes per lookup
  - easy to expand — no fixed size
  - one allocation per insertion
- Ordered Linear List
  - **O**$(\log_2 n)$ probes per lookup using binary search
  - insertion is expensive (to reorganize list)
- Binary Tree
  - **O**(n) probes per lookup — unbalanced
  - **O**$(\log_2 n)$ probes per lookup — balanced
  - easy to expand — no fixed size
  - one allocation per insertion
- Hash Table
  - **O**(1) probes per lookup — on average
  - expansion costs vary with specific scheme

Nested scopes: block-structured symbol tables

What information is needed?

- when asking about a name, want most recent declaration
- declaration may be from current scope or outer scope
- innermost scope overrides outer scope declarations

Key point: new declarations occur only in current scope

What operations do we need?

- `void put (Symbol key, Object value)`
  bind key to value
- `Object get(Symbol key)`
  return value bound to key
- `void beginScope()`
  remember current state of table
- `void endScope()`
  close current scope and restore table to state at most recent open beginScope

Attribute information

Attributes are internal representation of declarations
Symbol table associates names with attributes
Names may have different attributes depending on their meaning:

- variables: type, procedure level, frame offset
- types: type descriptor, data size/alignment
- constants: type, value
- procedures: formals (names/types), result type, block information (local decls.), frame size

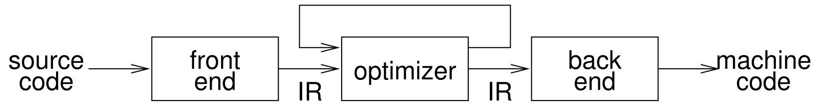
Intermediate representations

Why use an intermediate representation?

1. break the compiler into manageable pieces
   – good software engineering technique
2. simplifies retargeting to new host
   – isolates back end from front end
3. simplifies handling of “poly-architecture” problem
   – m lang’s, n targets $\Rightarrow$ $m+n$ components (myth)
4. enables machine-independent optimization
   – general techniques, multiple passes

An intermediate representation is a compile-time data structure

Intermediate representations



Generally speaking:

- front end produces IR
- optimizer transforms that representation into an equivalent program that may run more efficiently
- back end transforms IR into native code for the target machine

Intermediate representations

Representations talked about in the literature include:

- abstract syntax trees (AST)
- linear (operator) form of tree
- directed acyclic graphs (DAG)
- control flow graphs
- program dependence graphs
- static single assignment form
- 3-address code
- hybrid combinations
- Parallel Program Graphs
- Program Structure Tree/Graphs
- Concurrent Control Flow Graphs
- ...

Intermediate representations - properties

Important IR Properties

- ease of generation
- ease of manipulation
- cost of manipulation
- level of abstraction
- freedom of expression
- size of typical procedure

Subtle design decisions in the IR have far reaching effects on the speed and effectiveness of the compiler.
Level of exposed detail is a crucial consideration.

IR design issues

- Is the chosen IR appropriate for the (analysis/ optimization/ transformation) passes under consideration?
- What is the IR level: close to language/machine.
- Multiple IRs in a compiler: for example, High, Medium and Low

```
                    t1 = j + 2        r1 = [fp-4] // j
x = a[i,j+2]        t2 = i * 20       r2 = r1 + 2
                    t3 = t1 + t2      r3 = [fp-8] // i
                    t4 = 4 * t3       r4 = r3 * 20
// int a[][20];     t5 = addr a       r5 = r4 + r2
                    t6 = t5 + t4      r6 = 4 * r5
                    x = *t6           r7 = fp - 216 // a
                                      x = [r7+r6]
```

- In reality, the variables etc are also only pointers to other data structures.

Intermediate representations

Broadly speaking, IRs fall into three categories:

- Structural
  - structural IRs are graphically oriented
  - examples include trees, DAGs
  - heavily used in source to source translators
  - nodes, edges tend to be large
- Linear
  - pseudo-code for some abstract machine
  - large variation in level of abstraction
  - simple, compact data structures
  - easier to rearrange
- Hybrids
  - combination of graphs and linear code
  - attempt to take best of each
  - e.g., control-flow graphs
  - Example: GCC Tree IR.

Abstract syntax tree

An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed.

$$\begin{tikzcd}
 & - \arrow[dl] \arrow[dr] & & \\
\langle \text{id:x} \rangle & & * \arrow[dl] \arrow[dr] & \\
 & \langle \text{num:2} \rangle & & \langle \text{id:y} \rangle
\end{tikzcd}$$

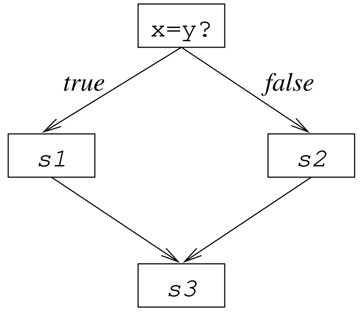
This represents "$x - 2 * y$".
For ease of manipulation, can use a linearized (operator) form of the tree.
e.g., in postfix form: $x \ 2 \ y * -$

Control flow graph

The control flow graph (CFG) models the transfers of control in the procedure

- nodes in the graph are basic blocks
  straight-line blocks of code
- edges in the graph represent control flow
  loops, if-then-else, case, goto

```
if (x=y) then
   s1
else
   s2
s3
```



3-address code

- At most one operator on the right side of an instruction.
- 3-address code can mean a variety of representations.
- In general, it allow statements of the form:
  `x ← y op z`
  with a single operator and, at most, three names.
  Simpler form of expression:
  `x - 2 * y`
  becomes
  ```
  t1 ← 2 * y
  t2 ← x - t1
  ```

Advantages
- compact form (direct naming)
- names for intermediate values

Can include forms of prefix or postfix code

3-address code: Addresses

Three-address code is built from two concepts: addresses and instructions.

- An address can be
  - A name: source variable program name or pointer to the Symbol Table name.
  - A constant: Constants in the program.
  - Compiler generated temporary:

3-address code

Typical instructions types include:

1. assignments `x ← y op z`
2. assignments `x ← op y`
3. assignments `x ← y[i]` (optional, why?)
4. assignments `x ← y`
5. branches `goto L`
6. conditional branches
   `if x goto L`
7. procedure calls
   `param` x_1, `param` x_2, ...`param` x_n
   and
   `call p, n`
8. address and pointer assignments: `x = *y, *x = y`.

How to translate:

```
if (x < y) S1 else
S2
```

?

Program Dependence Graphs (PDGs)

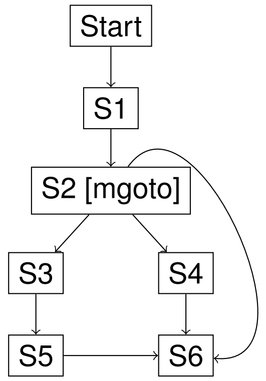
- PDG is a standard representation of control and sequential data dependencies.
- Like CFG, a PDG node represents an arbitrary sequential computation (basic-block).
- A PDG edge - can represent control or data-dependence.
- PDG is a graph (N, E_{cd}, E_{dd})
- For $a, b \in N$:
  - $(a,b) \in E_{cd}$ if control may be transferred from a to b.
  - $(a,b) \in E_{dd}$ if b is data dependent than a.
    - There exists variable v, such that $v \in Def(a)$ and $v \in Use(b)$.
    - This definition of v from a reaches b.
- Edges may have labels.
  - Control dependence edges: (i) true/false, (ii) unconditional
  - Data dependence edges: What type of dependence? Example:
    - loop carried / independent.
    - direction vector or distance vector.

Parallel Program Graph

- Parallel Program Graph (PPG) is a general intermediate representation of parallel programs.
  - includes PDG and CFG.
- PPGs include
  - control edges that represent (parallel) flow of control, and
  - synchronization edges that impose ordering constraints on the execution instances of PPG nodes.
- In PPGs, unlike in PDGs, we can have dependencies from a future iteration to a past iteration.

PPG (contd).

- A PPG is a graph (N, E_{cont}, E_{sync})
- $E_{cont} \subseteq N \times N \times \{T, F, U\}$
- $E_{sync} \subseteq N \times N \times syncConds$, where $syncConds$ represents the set of conditions.
- The nodes can $START$, END, $PREDICATE$, $COMPUTE$, or $MGOTO$



Program Structure Tree/Graph

- PST is a single-entry-single-exit (SESE) representation of a function.
- Consider a language that supports two special constructs:
  - `async S` – Creates an asynchronous task.
  - `finish S` – Waits for all the asynchronous tasks in `S`.
  - `atomic S` – ensures mutual exclusion.
- A PST (N, E) is for a procedure is rooted tree,
  - N is a set of nodes
  - A node can be root, compute-statement, if-else, if, loop, async, finish, atomic.
  - E is a set of tree edges obtained by collapsing the AST representation of the procedure into the eight nodes types.

PST Example

Extending PST to obtain Program Structure Graph

- Add another type of node: call
- Add an edge from the call-node to each possible function it may call.
- The graph no longer will be a tree.

Parallel Execution Graph (PEG)

- Obtained by combining the CFGs of each thread via special edges.
- Use cloning to resolve alias resolution:
  - Consider a code:

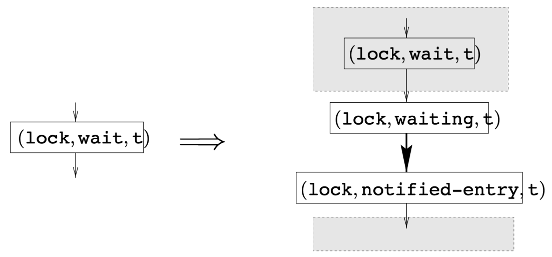
    ```
    synchronized(x) S
    ```
  - Say, `x` points to {R1, R2}.
  - Produce a structure with two branches:
    - one branch has a monitor access to `S` with R1 as the lock.
    - second branch has a monitor access to `S` with R2 as the lock.
- Methods:
  - Regular methods - inline.
    - Results in a single CFG for each thread.

Parallel Execution Graph (PEG) (contd.)

- Nodes in the graph: Nodes of the CFGs.
- Add edges from the CFG nodes of one thread to the other. Edges can be
  - Thread create
  - notify
- Each edge due to a communication methods (thread create/ wait / notify etc) is labeled:
  - `(object, name, caller)` – receiver object, method name, caller thread id.
  - Short cut: replace "`this`" with "*".
- Special nodes for entrance and exit points of synchronized blocks.
  - (lock-obj, `entry`, thread-id)
  - (lock-obj, `exit`, thread-id)
  - both outside the synchronized block.

Parallel Execution Graph (PEG) (contd.)

- Modelling the `wait` call:
  - Recall at a wait call:
    - Lock is released.
    - Reacquires the lock on notification.
  - Transform the CFG node:

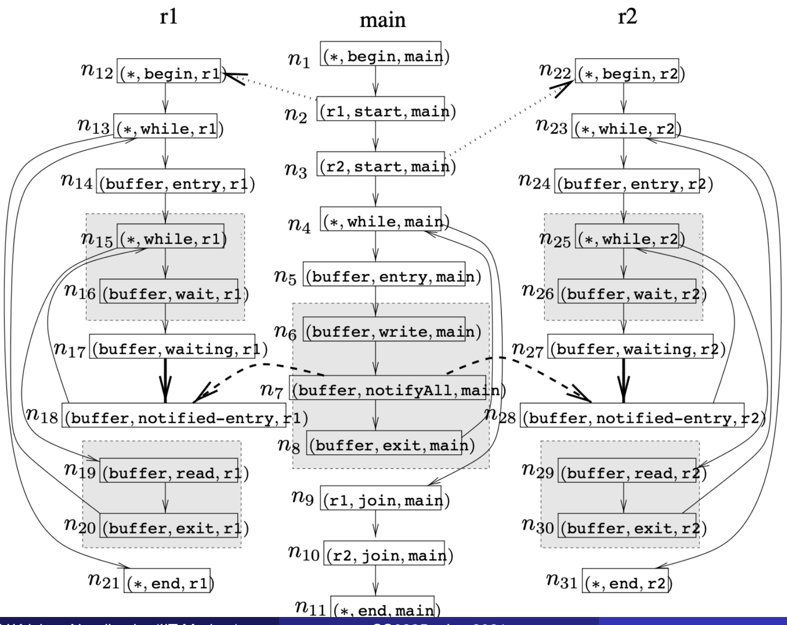


  - (lock, notified-entry, t) indicates received notification and waiting.
  - Shaded region - synchronized block.

Parallel Execution Graph (PEG) (contd.)

- Different types of edges:
  - waiting edge: between waiting and notified-entry
  - local edge: non waiting edge between two nodes of the same CFG.
  - start edge: from (t,start,*) to (*,begin,t)

Parallel Execution Graph (PEG) (contd.)



MHP Analysis

Auxiliary Challenge

Compute the MHP *map for each statement in the program (based on the key question 1).*

Auxiliary Challenge

Compute the MHP *map for each pair of statements in the program (based on the key question 2).*

Auxiliary Challenge 1	$\forall s_1$: MHP(s_1)
Auxiliary Challenge 2	$\forall s_1, s_2$: MHP(s_1, s_2)

MHP Analysis for Java Programs

- Consider the constructs, thread creation, join, wait, notify, and synchronization.
- Input: PEG
- For each PEG node n
  - $M(n)$: Nodes that may run in parallel with n.
  - $OUT(n)$: MHP information propagated to the successor(s) of n.
  - A worklist based algorithm.
    - worklist initialized to: `start` nodes of the main thread, reachable from the `begin` node the main thread.

Computing Notify Edges

Special edges have to added during the analysis.

- notify edge: from (obj, notify, r) to (obj, notified-entry, s)
- or from notifyAll node.

$$NotifySucc(n) = \begin{cases} \{m \mid m \in (\texttt{obj},\texttt{notified-entry},\star) \\ \quad \wedge WaitingPred(m) \in M(n)\} & \text{if } n \in notifyNodes(\texttt{obj}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Q: Why we cannot insert them before starting the analysis?

Computing M sets

- $N(t)$: set of all PEG nodes in the thread t.
- $thread(n)$: maps each node in the PEG to the thread to which it belongs.

$$M(n) = M(n) \cup \begin{cases} (\cup_{p \in startPred(n)} OUT(p) \\ -N(thread(n))) & \text{if } n \in (\star,\texttt{begin},\star) \\ ((\cup_{p \in NotifyPred(n)} OUT(p)) \\ \cap\ OUT(WaitingPred(n))) \\ \cup\ GEN_{notifyAll(n)} & \text{if } n \in (\star,\texttt{notified-entry},\star) \\ \cup_{p \in localPred(n)} OUT(p) & \text{otherwise} \end{cases}$$

1. Intersection: (1) the thread of n is waiting for a notification and (2) one of the notify predecessors of n executes.

Computing auxiliary maps

- `notifyAll` can awake multiple threads.
- All the corresponding notified-entry nodes may all execute in parallel.
- A node m is put in $GEN_{notifyAll}(n)$ if
  - m refers to the same lock object `obj` as n does,
  - the $WaitingPred$ nodes of m and n may happen in parallel, and
  - there is a node r labeled `(obj, notifyAll, ⋆)` that is a notify predecessor of both m and n.

$$GEN_{notifyAll}(n) = \begin{cases} \Phi, \quad \text{if } n \notin (\texttt{obj},\texttt{notified-entry},\star) \\ \{m \mid m \in (\texttt{obj},\texttt{notified-entry},\star) \wedge \\ \quad WaitingPred(n) \in M(WaitingPred(m)) \wedge \\ \quad (\exists r \in N : r \in (\texttt{obj},\texttt{notifyAll},\star) \wedge \\ \quad r \in (M(WaitingPred(m)) \cap M(WaitingPred(n))))\} \\ \quad \text{if } n \in (\texttt{obj},\texttt{notified-entry},\star) \end{cases}$$

Computing the OUT Sets

$$OUT(n) = (M(n) \cup GEN(n)) - KILL(n)$$

- $GEN(n)$ may (or) not run in parallel with n, but may execute in parallel with n's successors.
- $KILL(n)$: set of nodes that must NOT be passed to n's successors
  - n may run in parallel with nodes of $KILL(n)$.

Computing the GEN Sets

- if n is a `start` node
  - Add the corresponding the `begin` node.
  - `start` and `begin` don't run in parallel.
- n is a "notify" node.
  - Local successors of n may happen in parallel with $NotifySucc(n)$

$$GEN(n) = \begin{cases} (\star,\texttt{begin},\texttt{t}), & \text{if } n \in (\texttt{t},\texttt{start},\star) \\ NotifySucc(n), & \text{if } \exists \texttt{obj} : n \in notifyNodes(\texttt{obj}) \\ \Phi & \text{otherwise} \end{cases}$$

Computing the KILL Sets

- If n is a `join` node:
  - after n completes, no nodes from `t` may execute.
- If n is an `entry` (or `notified-entry` node:
  - after n completes, thread is inside the monitor.
  - KILL includes all nodes from this monitor.
- n is a `notifyAll`
  - no threads will wait for this object after n.

$$KILL(n) = \begin{cases} N(t), & \text{if } n \in (\texttt{t},\texttt{join},\star) \\ Monitor_{\texttt{obj}}, & \text{if } n \in (\texttt{obj},\texttt{entry},\star) \cup \\ & \quad (\texttt{obj},\texttt{notified-entry},\star) \\ WaitingNodes(\texttt{obj}) & \text{if } (n \in (\texttt{obj},\texttt{notify},\star) \wedge \\ & |waitingNodes(\texttt{obj}) \models 1) \vee \\ & (n \in (\texttt{obj},\texttt{notifyAll},\star)) \\ \Phi & \text{otherwise} \end{cases}$$

Other points: Inverse maps.

References

An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. Naumovich-Avrunin-Clarke. FSE 1999.

- Deadlock analysis for concurrent objects [Flores-Montoya et al., FORTES 2013]
- Termination and cost analysis for concurrent loops [Albert et al., ATVA 2013]
- Slicing [Krinke, PASTE 1998]
- Precise dependence analysis + Optimizing task parallel programs [Nandivada et al., TOPLAS 2013]

The speed of the MHP analysis plays a key role in the speed and effectiveness of these dependent optimizations and analyses.

MHP Analysis for task parallel programs

- May Happen in Parallel Analyses.
- For languages that support `async`-`finish`-`atomic` parallelism.
  - `async` statement creates lightweight tasks.
  - `finish` acts as a task join/termination construct.
  - `atomic` is realizes mutual exclusion.
  - Example languages: X10, HJ and so on.

Language subset under consideration: KX10

- `async S` : creates an activity to execute `S`.
- `finish S` : ensures activity termination.
  - IEF: Immediately Enclosing Finish
- `atomic S` : realizes global critical section.

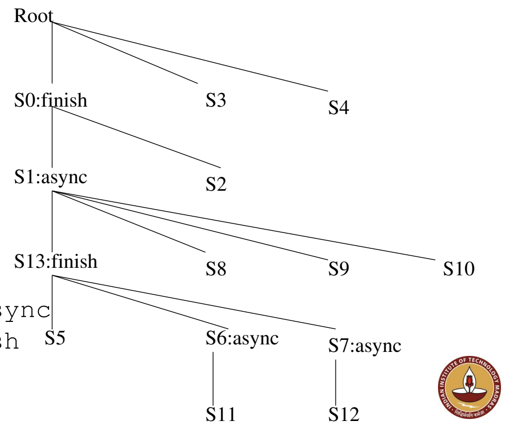
```
// Parent Activity
finish {
        S1; // Parent Activity
        async {
                S2; // Child Activity
        }
        S3; // Parent activity continues
}
S4;
```

Background: Program Structure Tree (PST)

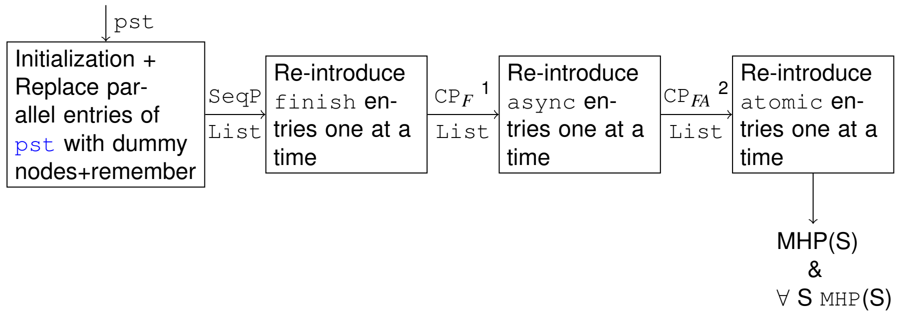
A program representation that compresses an abstract syntax tree to consider:

- root, seq-stmt, loop, async, finish and atomic.

```
S0: finish {
  S1: async {
   S13: finish {
      S5: ...
      S6: async S11
      S7: async S12
    }
    S8: ...
    S9: ...
    S10: ... } // end async
  S2: ... } // end finish
S3: ...
S4: ...
```



Incremental MHP Analysis



pst
Initialization + Replace parallel entries of pst with dummy nodes+remember
SeqP List
Re-introduce finish entries one at a time
CP_F [1] List
Re-introduce async entries one at a time
CP_{FA} [2] List
Re-introduce atomic entries one at a time
MHP(S) & ∀ S MHP(S)

[1] CP_F = `SeqP` + replaced `finish` nodes.
[2] CP_{FA} = CP_F + replaced `async` nodes.

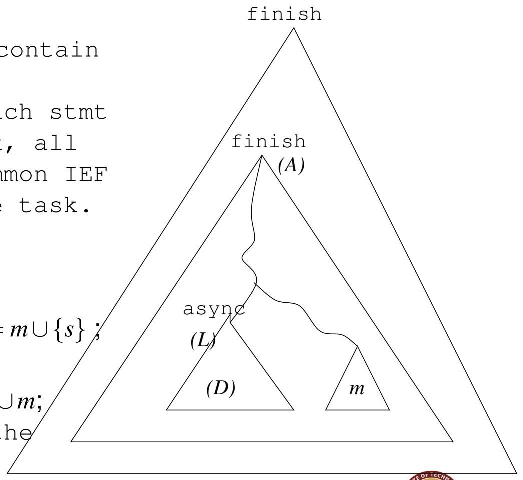
Order of re-introduction not important.

iMHP-addFinish

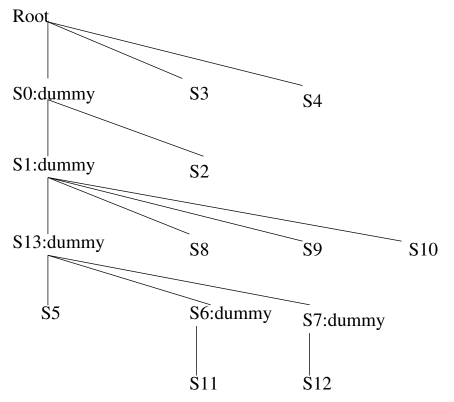
```
Function iMHP-addFinish(PST pst, Node $L$)
begin
    // Do nothing to the MHP maps!
```

iMHP-addAsync

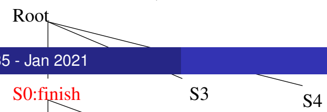
```
Function iMHP-addAsync(PST pst, Node $L$)
begin
    $A = IEF(L)$;// Tasks that may contain L
    /* Add to the MHP map of each stmt inside the proposed task, all
       the stmts within the common IEF that may start after the task. */
    $m = \{\}$;
    foreach $s \in$ Descendents($A$) do
        if $s$ is reachable from $L$ then $m = m \cup \{s\}$ ;
    $D$ = Descendents($L$);
    foreach $l \in D$ do MHP($l$) = MHP($l$) $\cup m$;
    // Update the MHP maps of the statements in $m$
    foreach $a \in m$ do
        MHP($a$) = MHP($a$) $\cup D$;
```



Illustration



Initialization:
List={S13:finish, S0:finish, S6:async, S7:async, S1:async}
∀ s, MHP(s)={



iMHP-addAtomic

```
Function iMHP-addAtomic(PST pst, Node $L_1$)
begin
    foreach $L_2 \in MHP(L_1)$ do
        if inAtomic($L_2$) AND $L_1$ and $L_2$ access the same memory location and one of them is a write then
            $MHP(L_1) = MHP(L_1) - \{L_2\}$;
    foreach $L_2 \in Nodes$ do
        if inAtomic($L_2$) AND $L_1$ and $L_2$ access the same memory location and one of them is a write then
            if $L_1 \in MHP(L_2)$ then
                $MHP(L_2) = MHP(L_2) - \{L_1\}$;
    foreach $s \in Descendents(L_1)$ do
        inAtomic ($s$) = true;
```

Complexity Argument

- iMHP-add⋆ invoked $O(C)$ times.
- iMHP-addFinish — $O(1)$.
- iMHP-addAsync
  - efficient disjoint-set union-find algorithms for union, find, delete.
  - Amortized complexity iMHP-addAsync: $O(N \times \alpha(N))$.
- Amortized complexity iMHP-addAtomic: $O(N \times \alpha(N))$.
- Overall complexity: $O(C \times N \times \alpha(N)) \approx O(C \times N)$.
- Cost of MHP(S) $= O(C \times N)$.
- Cost of ∀ S, MHP(S) $= O(C \times N)$.

```
Function iMHP-addAsync(PST pst, Node $L$)
begin
    $A = IEF(L)$; $m = \{\}$;
    foreach $s \in$ Descendents($A$) do
        if $s$ is reachable from $L$ then
            $m = m \cup \{s\}$
    $D$ = Descendents($L$);
    foreach $l \in D$ do
        MHP($l$) = MHP($l$) $\cup m$
    foreach $a \in m$ do
        MHP($a$) = MHP($a$) $\cup D$;
```

Data Race Detection

- Data Races - a common programming error.
- Two memory accesses lead to a data race.
  - If the memory accesses are performed by two “concurrent” threads.
  - At least one of the accesses is a write.

Basic idea behind static data-race detection

Q: Is there a data-race between two statements S1 and S2?

- MHP(S1, S2) = true
- If both S1 and S2 access a common memory location.
- If one of the accesses is a write.

Challenges in Static Data race detection.

- MHP analysis is imprecise.
- Leads to too many (spurious) data-race reports.
- Makes the tools unusable.

(Dynamic) Data race detection in the presence of locks

- MHP(S1, S2) = true
- If both S1 and S2 access a common memory location.
- If one of the accesses is a write.
- the accesses are made without holding a common lock.

Lockset based analysis.

- Use the Dynamic trace to determine the possible racy executions.

Insufficiency of Lockset based analysis

```
                                   Thread2
Thread1                            -------
-------
w(y)
acq(l)
w(x)
rel(l)
     ---------------------------->
                                   acq(l)
                                   r(x)
                                   r(y)
                                   rel(l)
```

- Access to `y` occurs in both the threads.
- Accesses don't happen using any shared lock.
- But, the accesses are ordered!

Happens Before Analysis for Data-race detection

- If we know that
  - statement S1 happens before (HB) S2.
  - or S2 (HB) S1
- then there can be no race between S1 and S2.

```
                                   Thread2
Thread1                            -------
-------
w(y)
acq(l)
w(x)
rel(l)
     ---------------------------->
                                   acq(l)
                                   r(x)
                                   r(y)
                                   rel(l)
```

- HB Analysis based race detectors identify event ordering communications.
- Only unordered events can lead to races.

Happens Before Partial Order

- Orders all events by a single thread - program order.
- Orders lock releases/acquires of the same lock - in the order in which they are observed.
- (Include other forms of communication)
- The remaining unordered events may potentially run in parallel.

Insufficiency HB race detection

- Locks lead to soft ordering.
  - lock based synchronized blocks can be reordered in a different execution.

Insufficiency of HB based race detectors.

```
class PolarCoord {
  int radius, angle;
  int count; // counts accesses

  static PolarCoord pc = new PolarCoord();

  void setRadius(int r) {
    count++;
    synchronized(this) { radius = r; }
   }

   int getAngle() {
     int t;
     synchronized(this) { t = angle; }
     count++;
     return t;
   }
   public static void main(String[] args){
     fork { pc.setRadius(10); }
     fork { pc.getAngle(); }
```

- The `count` field counts the number of calls to `setRadius` and `getAngle`.
- Race?

Performance of HB race detectors

```
Thread1         Thread2         Thread1         Thread2
-------         -------         -------         -------
r(count)                                        acq(this)
w(count)                                        r(angle)
acq(this)                                       rel(this)
w(radius)                       r(count)
rel(this)                       w(count)
    ------->    acq(this)                       r(count)
                r(angle)                        w(count)
                rel(this)       acq(this)
                r(count)        w(radius)
                w(count)        rel(this)
```

- HB says “race” for the second trace, and not for the first.
- Reality - Race present in both.
- First trace is considered to have a “predictable” race.

Causally Precedes as an Alternative to HB

- A race occurs if two conflicting actions are not CP-ordered.
- CP-ordering identifies “predictable” races.
- Note: considering all possible reorderings can be quite expensive.

Form of the trace

- Each trace event is of the form $[t : a]_i$,
  - t is the thread id.
  - a is the action (e.g., $w(x), r(x), acq(l), rel(l)$)
  - i is the event's index in the global trace.

Defining HB relation

- Events of the same thread program order
  - $([t : \_]_{i_1} \ll_{HB} [t : \_]_{i_2})$, if $i_1 < i_2$
- Release and acquire on the same lock - ordered as they appear
  - $[t_1 : rel(l)]_{i_r} \ll_{HB} [t2 : acq(l)]_{i_a}$, if $i_r < i_a$
- HB is closed under composition (transitivity). $\ll_{HB} = (\ll_{HB}\ o \ll_{HB})$

Defining Casually Precedes relation

Smallest relation such that

- Release-acquire relation between critical sections over the same lock with conflicting events.
  - $[t_1 : rel(l)]_{i_r} \ll_{CP} [t2 : acq(l)]_{j_a}$, if there are
    - $[t_1 : \_]_{k_1}$ conflicts with $[t_2 : \_]_{k_2}$ such that
    - $i_a < k_1 < i_r < j_a < k_2 < j_r$, where
    - (i_a and i_r) acq-rel pairs in t_1 and
    - (j_a and j_r) acq-rel pairs in t_2.
- Release-acquire relation between critical sections over the same lock that contains CP-ordered events.
  - $[t_1 : rel(l)]_{i_r} \ll_{CP} [t2 : acq(l)]_{j_a}$, if
    - acq pair of i_r in $t_1 \ll_{CP}$ rel pair of j_a in t_2.
- CP is closed under left and right composition with HB $\ll_{CP} = (\ll_{HB}\ o \ll_{CP}) = (\ll_{CP}\ o \ll_{HB})$

CP Vs HB

- CP is a subset of HB.
  - First two rules a subset of the rel-acq edges.
- Thus weaker.
- But leads to sound results.

Two events have CP-race, if (i) the events are conflicting, (ii) the event are not CP-ordered (in either direction).

Example

```
Thread1                              Thread2
-------                              -------
w(y)
acq(l)
rel(l)
     ----------------------------->
                                     acq(l)
                                     rel(l)
                                     w(y)
```

- Race exists?

More examples and details

See the example in Sections 2.2 and 3 in “Sound Predictive Race Detection in Polynomial Time”, Smaragdakis-Evans-Saadoswki-Yi-POPL-2012.

Deadlock Analysis

Skipped.

Outline

Memory Consistency Models

- A memory consistency model is
  - a set of rules governing how the memory systems will process memory operations from multiple processors.
    - Order in which memory operations will appear to execute - determines what value should a read return?
  - a contract between programmer and system.
  - Determines what optimizations can be performed for correct programs.
- Affects : Ease of programming, performance, and all the program analysis tools/techniques.

Uniprocessor Memory model

- Memory value requirement: Memory operations occur in program order: read returns the value of the last write in program order.
- Simple to reason about.
- Compiler optimizations preserve these semantics.
- Independent operations can execute in parallel.

Strict consistency

- Strictest memory model.
- Requires that the ‘read’ should get the value written by the last ‘write’.
- Requires a Global clock $\equiv$ Halting problem.

Sequential consistency

[Lamport]: A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program.

Sequential consistency

Result of an execution appears as if:

- All operations executed in some sequential order.
- Memory operations of each process in program order.
- Nothing specified about caches, write buffers.

Understanding Program Order. Dekker's Algorithm for synchronization

```
Initially Flag1 = Flag2 = 0
P1                           P2
Flag1 = 1                    Flag2 = 1
if (Flag2 == 0)          if (Flag1 == 0)
     critical section             critical section
```

```
Execution:
P1                              P2
(Operation, Location, Value)   (Operation, Location, Value)
Write, Flag1, 1                 Write, Flag2, 1

Read, Flag2, 0                  Read, Flag1, ___
```

Reads of 1 by `Flag1` `Flag2` are valid.

Problematic situation

- Write buffers with read bypassing.
- Overlap or reorder writes/reads by compiler / hardware.
- Values in registers.

Understanding Program Order. Ex 2

```
Initially A = Flag = 0
P1                       P2
A = 23;                  while (Flag != 1) {;}
Flag = 1;                ... = A;
```

```
P1                       P2
Write, A, 23             Read, Flag, 0
Write, Flag, 1
                         Read, Flag, 1
                         Read, A, _____
```

Problematic situation

- Overlap or reorder writes/reads by compiler / hardware.

Write Atomicity

```
Initially A = B = C = 0
 P1       P2       P3                P4
 A = 1;   A = 2;   while (B != 1) ;  while (B != 1) ;
 B = 1;   C = 1;   while (C != 1) ;  while (C != 1) ;
                   tmp1 = A;         tmp2 = A;
```

Q: What are the possible values of tmp1 and tmp2?

Q: Can tmp1 = 1 and tmp2 = 2 be possible? How?

- Cache coherence protocol must serialize writes to same location.
- Writes to same location should be seen in same order by all.

Atomicity Ex 2

Initially A = B = 0

```
P1              P2                      P3
A = 1           while (A != 1) ;while (B != 1) ;
                B = 1;                     tmp = A
```

```
P1              P2              P3
Write, A, 1
                Read, A, 1
                Write, B, 1
                                Read, B, 1
                                Read, A, ~~0~~
```

- if ‘read’ returns a new value before all copies see it.
- Read others’-write early optimization is unsafe.

Sequential Consistency implementation

Implementations of this model must satisfy the following:

- Program Order Requirement : The operations of same processor must be executed in program order
- Write Atomicity : All writes appear to be instantaneous (no buffer).
- All processors must see all write operations in the same order (cache coherence).
- Easier to implement in architectures with no cache, no write buffers, blocking reads, .

Sequential Consistency - issues

- Sequential Consistency constraints
  - write $\rightarrow$ read
  - write $\rightarrow$ write
  - read $\rightarrow$ read, write

  Implications (not allowed)
  - Read others’ write early.
  - Read own write early.
  - Unserialized writes to the same location.
- Simple model to reason about given parallel programs.
- Makes it very hard to modify a parallel program (automatic and manual)
  - Processor reordering for performance - write buffers, overlapped writes, non-blocking reads
  - Compiler transformations - scalar replacement, register allocation, instruction scheduling.
  - Programmer reordering code for aesthetics/SE requirements.

Sequential consistency - too strict

- Many architectures do not give SC.
- Compiler optimizations on SC are limited.
- Sofwtware engineering issues.

- Give up!
- Use weaker models - relax the program order requirement and write atomicity requirement.

Sequential consistency (English)

- Memory operations of each process happens in program order.
- any valid interleaving of read and write operations is OK.
- all processes must see the same interleaving.

Sequential consistency examples

P1	W(x)1			
P2		W(x)2		
P3			R(x)2	R(x)1
P4			R(x)2	R(x)1

Sequentially consistent - as both P3 and P4 see writes in the same sequential order.

Sequential consistency (counter) example

P1	W(x)1			
P2		W(x)2		
P3			R(x)2	R(x)1
P4			R(x)1	R(x)2

Sequentially inconsistent - as both P3 and P4 see writes in the two different sequential orders.

Sequential consistency (counter) example

P1	P2	P3
x = 1;	y = 1	z = 1
print(y,z)	print (x,z)	print (x,y)

Inconsistent execution:

1. x = 1
2. print (y, z);
3. print (x, z);
4. y = 1;
5. z = 1;
6. print (x, y);

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor and access same memory location - are seen by every node in causal order.
- Causal order is transitive.
  - memory operations that are causally related must have a total order and
  - program order for the ones issued by same processor.
- Hence such memory operations must be seen in same order by all processors.
- Here, write atomicity has been slightly weakened.
- weaker than sequential consistency, which requires that all nodes see all writes in the same order.

Causal consistency (example)

P1	W(x)1			W(x)3		
P2		R(x)1	W(x)2			
P3		R(x)1			R(x)3	R(x)2
P4		R(x)1		R(x)2	R(x)3	

Causally consistent, but not sequentially/strict consistent.

- Processors may see different order.
- All orders respect causal order (program order and read-write order).
- Has no global order, partial order for each processor.

Causal consistency (counter) Example

P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2

- Violates causal consistency.
- Removing the Read from the P2 – makes the execution causally consistent.

PRAM consistency

- All processes see memory writes from one process in the order they were issued from the process.
- Writes from different processes may be seen in a different order on different processes.
- no guarantees about the order in which different processes see writes, except that two or more writes from a single source must arrive in order, as though they were in a pipeline.

P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2

- PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict
- (Also known as, FIFO consistency, or Processor consistency)

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.
  - All data operations after synch in program order must wait for synch to complete.
  - Synchronizations are performed in program order.
  - All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
  - All other accesses may be seen in different order on different processes
- Illusion of write atomicy has to be maintained.
- Hardware implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed.

Weak Ordering

- Example 1:

```
P1   W(x)1 W(x)2 Sync
P2                     R(x)1 R(x)2 Sync
P3                     R(x)2 R(x)1 Sync
```

- Example 2:

```
P1   W(x)1 W(x)2 Sync
P2                     SyncR(x)2
```

- The programmer has to manage synchronization explicitly.
- Weak $\leq$ PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict

Weak consistency (counter) example

```
P1   W(x)1 W(x)2 Sync
P2                     SyncR(x)1
```

- P2 will observe the most recent write of the variable x, which has the value 2. Thus, it's not a valid sequence.

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :
  - Release must be executed only when all memory operations statements are complete.
  - But accesses after 'release' in program order do not have to wait for release (unless protected by another acquire).
- do "acquite" = that writes on other processors to protected variables will be known
- do "release" = that writes to protected variables are exported
- and will be seen by other machines when they do a lock (lazy release consistency) or immediately (eager release consistency)
- Total order among all synchronization instructions must be maintained.

Weak and Release comparison

- Weak: Shared data can be counted on to be consistent only after a synchronization is done.
- Release: Shared data are made consistent when a critical region is exited.

Release Consistency - example

- Example:

```
P1:  L W(x)1 W(x)2 U
P2:                      L R(x)2 U
P3:                                 R(x)1
```

- RC $\leq$ Weak $\leq$ PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict

Delta and Eventual consistency models

- **Delta consistency**: The write operations will propagate through the shared memory system and all the replicas will be consistent after a fixed time period δ.
  - if an object is modified, during the short period of time following its modification, the read may not be consistent.
  - after a fixed time period, the modification is propagated and the read will be consistent.
- **Eventual Consistency Model** : The writes propagates eventually (we cannot have a fixed bound on the delay)

Eventual Consistency

- Allow stale reads, but ensure that reads will eventually reflect previously written values (no guarantees on delays)
- Doesnt order concurrent writes as they are executed, which might create conflicts later: which write was first?
- More concurrency opportunities than strict, sequential, or causal consistency.
- Used a lot: Amazon: Dynamo, a key/value store, file synchronization

Programmer centric models

- Problem with relaxed models is that most of them are based on the performance optimization that can be performed.
- However, from a programmer's perspective, it is not clear how to use these effectively.
  - How to reason about programs for systems with relaxed memory models
  - How to use the safety nets minimally, to get the desired semantics from program
- Even Sequential Consistency is not simple enough.
- We need models which is simple for the programmer, but provides enough information about program to apply optimization and get efficiency.

Programmer centric models

Programmers understand their code:

- Different operations have different semantics

P1	P2
A = 23;	while (Flag != 1) ;
B = 37;	. . . = B;
Flag = 1;	. . . = A;

- Flag = Synchronization; A, B = Data
- Can reorder data operations
- Distinguish data and synchronization

Data Race Free 0 - DRF0

- Data-Race-Free-0 Program
  - All access distinguished as either synchronization or data.
  - All races distinguished as synchronization (in any SC execution).
- Data-Race-Free-0 Model
  - Guarantees SC to data-race-free-0 programs.
  - Others - reads return value of some write to the location.

A program is considered to be data-race-free-0 if and only if (i) for any sequentially consistent execution, all conflicting accesses are ordered by the happens-before relation, and (ii) all synchronization operation in the program are recognizable by the hardware and each accesses exactly a single memory location.

Programming with Data Race Free 0 - DRF0

- Needed information: for each operation: if it will race (in any SC execution).
- Procedure:
  - Write program assuming SC.
  - For each memory operation in the program:
    - Guaranteed to be no races? "Data access": "synchronization".

Problems with data race free model

- It does not define any semantics for programs with data races.
- A concern for safe languages like Java, which provide safety for any program and cannot let the behavior of a program to be ambiguous.
- Either define safe semantics for such programs or identify them and prevent their execution.
- Define higher abstractions for programmers which are inherently data race free
- Expensive for hardware to implement

Goals of Memory model

- Programmability? - Lost intuitive interface of SC
  - Programmer must reason about the allowed behaviors.
  - Can be subtle.
- Portability? - Many different models across different systems.
  - Code may not be portable.
- Performance? - Can we do better?
  - Optimizations must take into consideration what's allowed by the hardware/language.

Future:

- Parallel programs today are inherently non deterministic
- We need deterministic outcomes from our parallel programs.
- Deterministic Outcomes from Inherent non determinism. Possible?

Advantages from relaxed models

- Gains both in the H/W and compiler.
- Gains in H/W (during execution):
  - latency hiding - can overlap many reads and writes.
- Gain by the compiler:
  - more operations can be reordered. (compare with SC).

Impact on Compiler Optimizations

- Compilers routines move code around.
  - Instruction scheduling, code motion, register allocation, common sub-expression elimination, loop tiling, software pipelining, . . .
- Safety of many of these optimizations depends on data-dependency.
- The definition of data-dependency in parallel programs is closely tied to underlying memory consistency models.

Example impact of memory consistency model - I/III

Impact of memory consistency model/parallelism on loop distribution.

```
// Before loop distribution                // After loop distribution
for (int i = ...) {                        for (int i = ...)
  /*S1*/  X[f(i)]=...              ⟹        /*S1*/X[f(i)]=... ;
;                                          for (int i = ...)
  async { /*S2*/...X̄[g(i)];}                 async { /* S2 */ ...X̄[g(i)];}
}
```

- Loop distribution - basis?
- Flow dependence from $S1$ to $S2$ on variable X (direction vector $\leq$).
- In a sequential compiler
  - no `async` - cannot distribute; see loop-carried anti-dependence.
- In a compiler for parallel programs:
  - No anti-dependence, and hence no dependence cycle. Hence we can distribute.

Example impact of memory consistency model - II/III

Impact of memory consistency model/parallelism on loop parallelization.

```
// Before loop ||^n                         // After loop ||^n
int []A = new int[n]; //0 init             int []A = new int[n]; // 0 init
async {                                     async {
 for (int i=0;i<n;++i) {            ⟹        foreach (int i=0;i<n;++i) {
   a[i] = i;                                      a[i] = i;
} }                                         } }
for (int j=1;j<n;++j)                       foreach (int j=1;j<n;++j)
  assert (a[j] <= a[j-1]+1);                   assert (a[j] <= a[j-1]+1);
```

- Loop parallelization - basis?
- If there is no `async`
  - Safe to parallelize L1 and L2 - as iterations are independent.
- In a compiler for parallel programs (with async):
  - Safe to parallelize L2 - does not break any dependency.
  - L1: depends on the consistency model.
    - Sequential Consistency: `a[i]` and `a[i+1]` must be be seen in order at L2. **Cannot be parallelized.**
    - weak ordering: `a[i]` and `a[i+1]` need not be seen in order at L2. **Can be parallelized.**

Example impact of memory consistency model - III/III

Impact of memory consistency model/parallelism on Constant Propagation.

```
// Before const prop                 async {
int x = 0;                            x = 1
async {                               barrier
  = x;                                 = x ;
 barrier                              barrier
 x = 2;                                = x;
 barrier                              barrier
 x = 3;                                = x;
 barrier;                             x = 4;
 = x;                                }
}
```

- No asyncs?
- With async:
  - Consistency model not aware of barriers.
  - Consistency model aware of barriers.

So long. . .

Sources

- Wikipedia
- fixstars.com
- Jernej Barbic slides.
- Loop Chunking in the presence of synchronization.
- Vivek Sarkar's slides.
- Sarita Adve's slides.
- Nimit's Singhania's presentation.
- http://regal.csep.umflint.edu/ swturner/Classes/csc577/Online/Chapter06/Chapter06.html
- Java Memory Model JSR-133: "Java Memory Model and Thread Specification Revision"