

1. Arrays:

```
int[] numbers;                                // Declaration of an integer array
double[] scores = new double[5];               // Declaration and initialization of a double
array with a size of 5
String[] names = {"Alice", "Bob"};             // Declaration and initialization of a String
array with values
int[][] matrix = new int[3][3];                // Declaration of a 2D integer array (matrix)
```

2. Lists (using java.util.List):

```
import java.util.List;                         // Import the List interface

List<String> namesList;                      // Declaration of a List of Strings
List<Integer> numbersList = new ArrayList<>(); // Declaration and initialization of an
ArrayList of Integers
List<Double> scoresList = List.of(2.5, 3.8, 4.2); // Declaration and initialization of
an immutable List (Java 9+)
```

3. AtomicInteger (java.util.concurrent.atomic):

...

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

AtomicInteger count = new AtomicInteger(0); // Declaration and initialization of an
                                         // AtomicInteger
count.incrementAndGet();                  // Atomic increment (returns the updated
                                         // value)
count.getAndIncrement();                 // Atomic increment (returns the original
                                         // value)
count.get();                           // Get the current value of the
                                         // AtomicInteger
```

4. Thread Synchronization:

- synchronized keyword: Used to create synchronized blocks or methods to ensure atomic execution of code and prevent data races.
- wait() and notify()/notifyAll(): Methods used for thread communication and coordination using object monitors.

5. Thread Creation and Execution:

```
Runnable myRunnable = () -> {
    // Code to be executed in the thread
};

Thread thread = new Thread(myRunnable); // Create a new thread using a Runnable
thread.start();                      // Start the thread's execution

// Using the join() method to wait for a thread to finish:
try {
    thread.join(); // Wait for the thread to finish execution
} catch (InterruptedException e) {
    // Handle the InterruptedException
}
```

6. Thread Pools:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

ExecutorService executor = Executors.newFixedThreadPool(5); // Create a thread pool with
                                                               // 5 threads
executor.execute(myRunnable); // Submit a Runnable for execution
executor.shutdown(); // Shut down the thread pool after all tasks are completed
```

Remember, when dealing with threads, it is crucial to ensure proper synchronization and avoid data races to ensure the correct and efficient execution of multithreaded code. The AtomicInteger and other atomic classes help in achieving thread safety in concurrent programming. Additionally, using thread pools can be more efficient than creating new threads for each task when handling multiple concurrent operations.

can you explain me about recursion in context of threads and concurrent programming in java? and about ExecutorService?

CHATGPT

Certainly! Recursion in the context of threads and concurrent programming in Java can be a powerful technique to solve complex problems by dividing them into smaller subproblems and processing them concurrently.

Recursion:

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. In the context of concurrent programming, recursion can be used to divide a task into smaller subtasks and process them concurrently using multiple threads. Each recursive call represents a smaller subtask, and by breaking down the problem into smaller pieces, you can take advantage of parallelism to potentially speed up the overall computation.

Recursive algorithms can be tricky to implement correctly, especially in a multi-threaded environment. Care must be taken to handle synchronization and thread-safety properly to avoid data races and other concurrency issues.

ExecutorService:

ExecutorService is part of the `java.util.concurrent` package in Java and provides a higher-level API for managing and executing threads. It abstracts away the details of thread creation, management, and termination. With ExecutorService, you don't need to manually create threads for each task; instead, you submit tasks to the executor, and it takes care of executing them.

Key features of ExecutorService:

1. **Thread Pool:** The ExecutorService manages a pool of worker threads, and it reuses these threads to execute multiple tasks, which is more efficient than creating a new thread for each task.
2. **Task Submission:** You can submit tasks to the ExecutorService using one of the `submit()` methods. Tasks can be represented as `Callable` or `Runnable` objects.
3. **Execution Policies:** The ExecutorService provides different execution policies, such as fixed thread pool size, cached thread pool, or scheduled execution. These policies define how tasks are executed, queued, or rejected when the pool is full.
4. **Futures:** When you submit a task to the ExecutorService, it returns a `Future` object that represents the result of the computation. You can use this object to retrieve the result, cancel the task, or check if the task has completed.

Combining Recursion and ExecutorService:

By combining recursion with `ExecutorService`, you can perform parallel processing of the smaller subtasks in a recursive algorithm. For example, in a divide-and-conquer algorithm, you can submit the recursive calls to the `ExecutorService`, and it will manage the execution of these calls in parallel.

To use recursion with ExecutorService, you need to be cautious about proper synchronization and handling of shared data between threads. You should also consider the trade-offs between parallelism and overhead (e.g., thread creation, task submission) since creating and managing threads have some associated costs.

It's important to note that not all problems are suitable for parallelization, and the benefits of parallelism may vary depending on the specific problem, the size of the data, and the hardware configuration. Additionally, using threads introduces potential issues like deadlocks, race conditions, and thread contention, so careful design and testing are essential in concurrent programs.

Java provides several implementations of `ExecutorService`, such as `ThreadPoolExecutor`, `FixedThreadPool`, and `CachedThreadPool`, which you can choose based on your application's specific needs and performance considerations.

ConcurrentHashMap – תרשים של מבנה נתונים שירכב מ-**HashMap** ו-**ReentrantLock** (כגון ב-**AtomicStampedReference**) ותאפשר גישה ל-**Map** ב-**Thread Safe** ו-**Atomic**.

אֶלְעָזָר אַבְרָהָם, אֶלְעָזָר שְׁגָן - Fork Join Pool

public <T> T getElement (T [] array , int index)

וְלֹא תַּעֲשֶׂה כֵּן נִתְּנָהָר לְפָנֵינוּ כֵּן תַּעֲשֶׂה בְּבָנֶיךָ וְכֵן תַּעֲשֶׂה בְּבָנֶיךָ כֵּן תַּעֲשֶׂה בְּבָנֶיךָ וְכֵן תַּעֲשֶׂה בְּבָנֶיךָ

Fork-Join Pool

Java 7

தோர்மாக்க பிர் பிர் பா டூ

לעומת נורווגיה ופינלנד, שמיינריך נטה לארון גראן-טולסן, מנהל אוניברסיטת נורווגיה, אמר כי לא ניתן למסור מושגים ייחודיים של מדיניות כלכלית כה מתקדמת.

work Steeling מטל ועיבודים

۷۹/۸۵ ۱۲۵۴

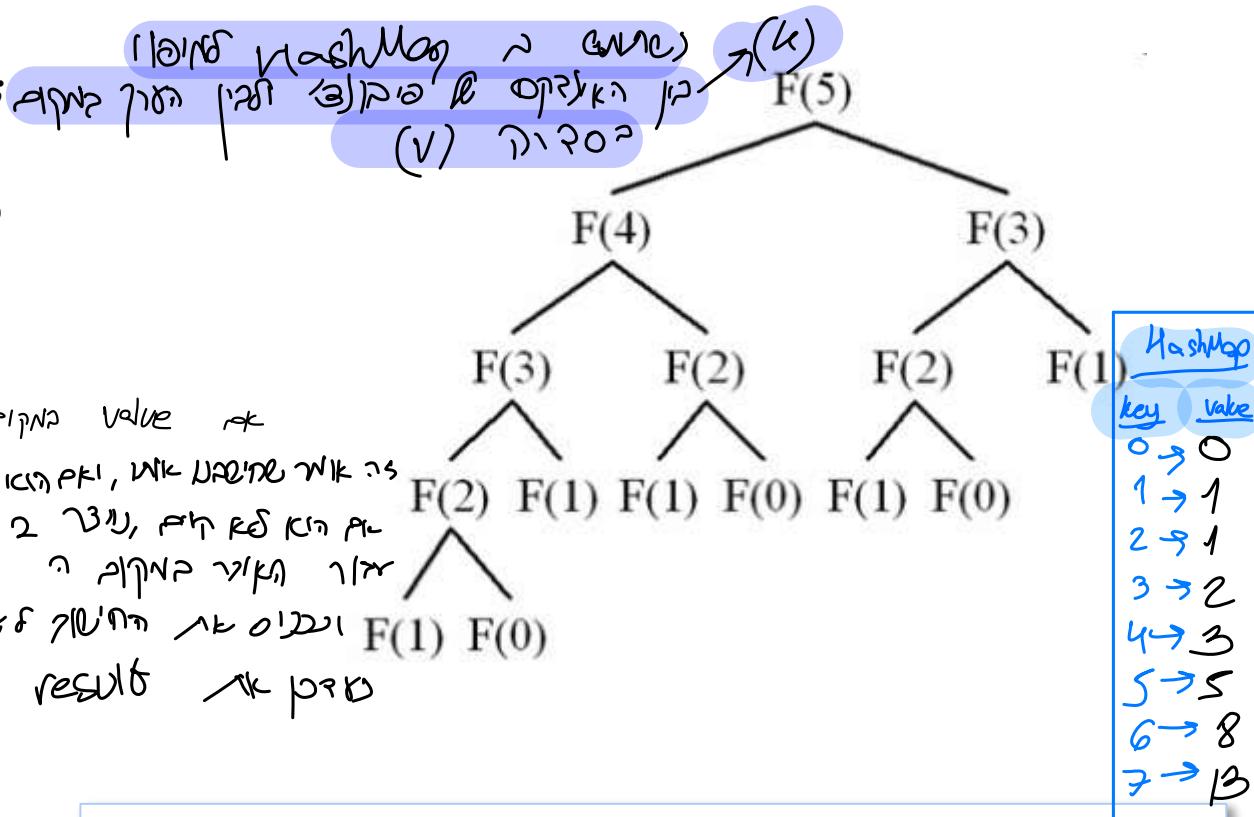
סְלִוָּר מִנְיָמִין וְעַל יְמֵינָה כְּבָנָה שֶׁבָּאָה הַכְּלָל.

Fibonacci + Dynamic Programming

```
public class Fib_DP { // without concurrency
    // but with dynamic programming
    static HashMap<Integer, Integer> fibs = new HashMap<>();
    <key , value>
    int num;
    public Fib_DP(int num) { this.num=num; } //CTOR

    public int compute(){ // a recursive task
        if(num<=1)
            return num;
        if(fibs.get(num)!=null) // null נא (num=) key גורם
            return fibs.get(num); // ות שולב נ-1 מ-2 מ-1
        Fib_DP fib1= new Fib_DP(num-1);
        Fib_DP fib2= new Fib_DP(num-2);
        int result=fib2.compute()+fib1.compute(); // result
        fibs.put(num,result);
        return result;
    }

    public static void main(String[] args) {
        System.out.println(new Fib_DP(2048).compute());
    }
}
```

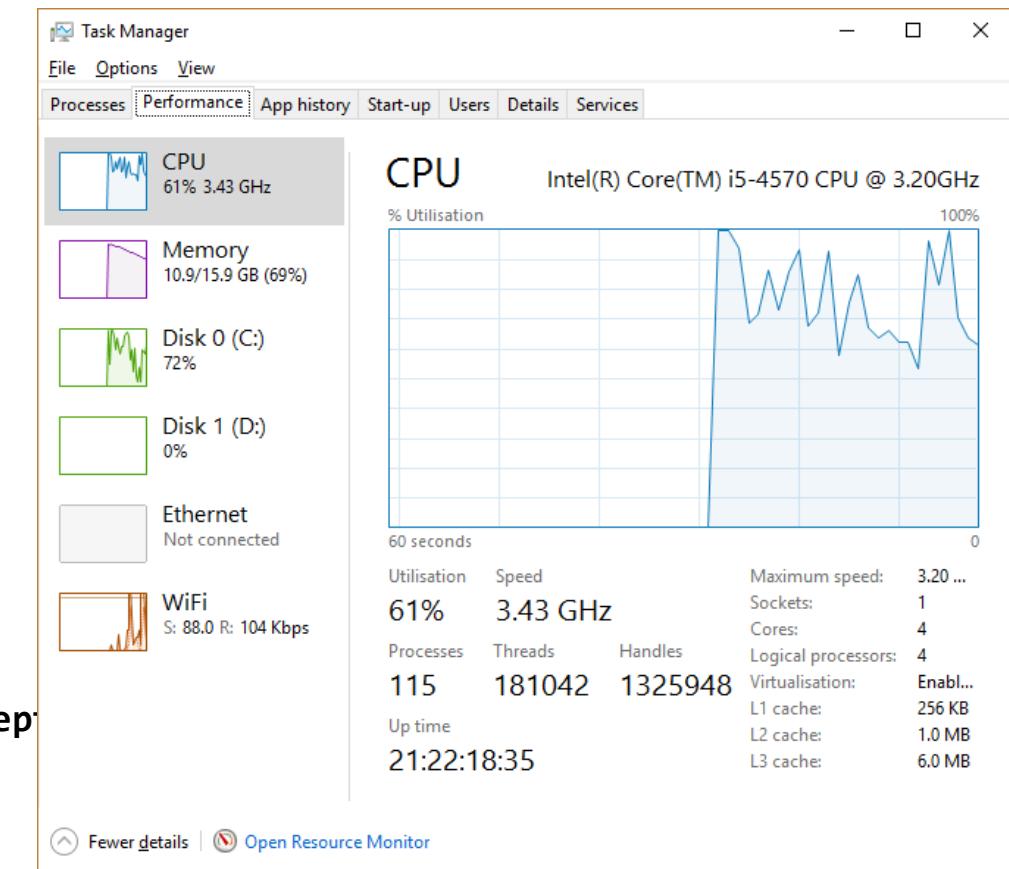


However, we wish to simulate a multithreaded task

Future
-cached Thread Pool get, submit, callable מתקבלים

Fibonacci + Thread Pool (JDK 6)

```
public class Fib_TP implements Callable<Integer>{  
  
    static ExecutorService es=Executors.newCachedThreadPool();  
  
    int num;  
    public Fib_TP(int num) {this.num=num;}  
  
    @Override  
    public Integer call() throws Exception {  
        if(num<=1)  
            return num;  
        Future<Integer> fib1 = es.submit(new Fib_TP(num-1));  
        Future<Integer> fib2 = es.submit(new Fib_TP(num-2));  
        return fib2.get()+fib1.get();  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        Future<Integer> f=es.submit(new Fib_TP(45));  
        System.out.println(f.get());  
    }  
}
```



+ רקורסיה +

Fibonacci + Thread Pool (JDK 6)

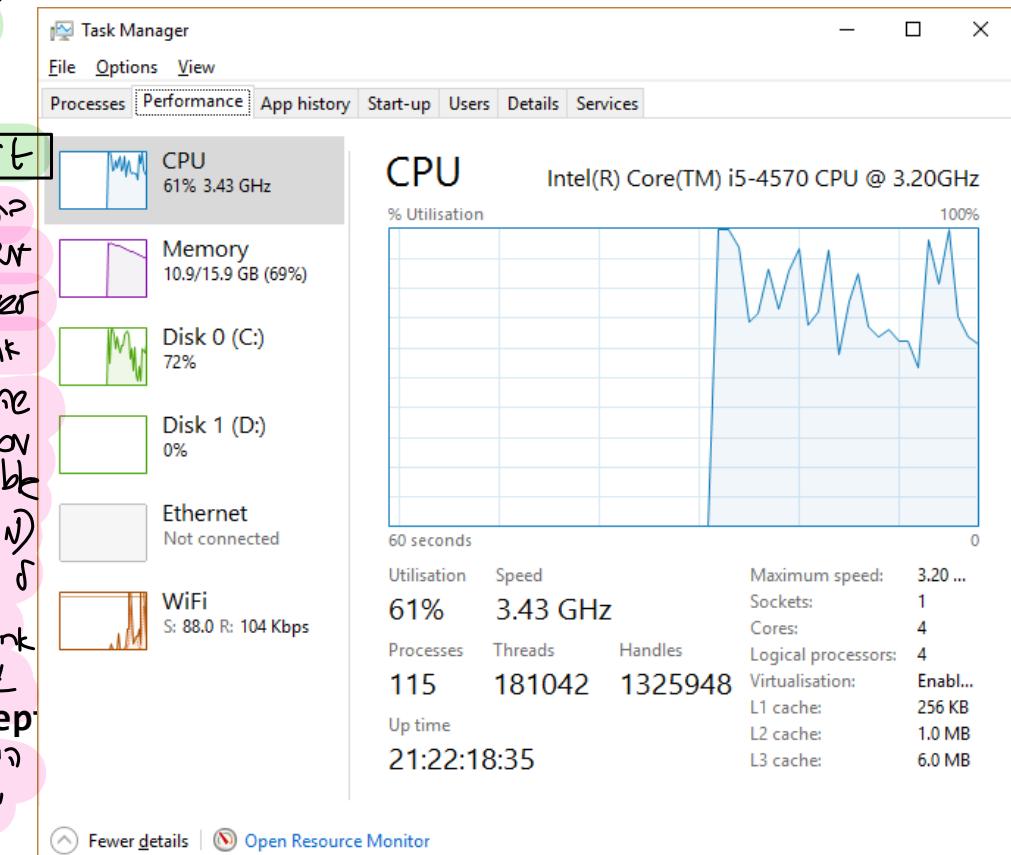
```

public class Fib_TP implements Callable<Integer> {
    static ExecutorService es=Executors.newCachedThreadPool();
    int num;
    public Fib_TP(int num) {this.num=num;}
    @Override
    public Integer call() throws Exception {
        if(num<=1)
            return num;
        Future<Integer> fib1 = es.submit(new Fib_TP(num-1));
        Future<Integer> fib2 = es.submit(new Fib_TP(num-2));
        return fib2.get()+fib1.get();
    }
    public static void main(String[] args) throws InterruptedException {
        Future<Integer> f=es.submit(new Fib_TP(45));
        System.out.println(f.get());
    }
}

```

הפעלה של פונקציית call():
 1. נזקק ל-2 Future<Integer> (fib1 ו fib2)
 2. נזקק ל-2 Callable<Integer>
 3. נזקק ל-2 submit()

הפעלה של submit():
 1. נזקק ל-1 Callable<Integer>
 2. נזקק ל-1 Future<Integer>

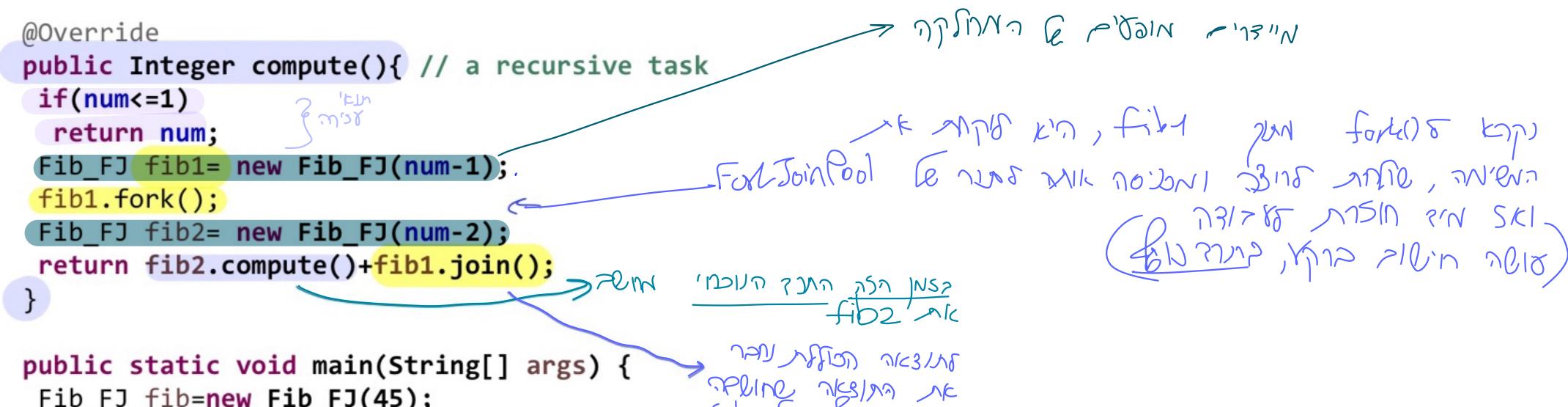


Fibonacci + Fork-Join Pool (JDK 7)

```
public class Fib_FJ extends RecursiveTask<Integer>{
    // with fork-join pool
    int num;
    public Fib_FJ(int num) { this.num=num; }

    @Override
    public Integer compute(){ // a recursive task
        if(num<=1)
            return num;
        Fib_FJ fib1= new Fib_FJ(num-1);
        fib1.fork();
        Fib_FJ fib2= new Fib_FJ(num-2);
        return fib2.compute()+fib1.join();
    }

    public static void main(String[] args) {
        Fib_FJ fib=new Fib_FJ(45);
        ForkJoinPool pool = new ForkJoinPool();
        System.out.println(pool.invoke(fib));
    }
}
```



- compute

האך גאנז. רכישת פולע תא גראונט'

fib1.fork()

fib2.
compute()

join() פון ראנץ פון fib1.fork() נאכט לילך פון הפלט הפלט

return fib2.compute()

היאו רג'ו פון
רילס הראפר
לעטן.

עלן

+

פלטן ראנץ פון פון גאנטס פון ראנץ

fib1.join()

נו פון ראנץ
.fork() פון פון פון פון

Since JDK 5 – Callable & Future

```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```



```
ExecutorService executor=Executors.newCachedThreadPool();  
  
Future<String> f = executor.submit(new Callable<String>() {  
    @Override  
    public String call() throws Exception {  
        return deepThought();  
    }  
});  
  
//...  
System.out.println(f.get()); // blocks until an answer is given
```

Cache Thread Pool
Future<String> submit
call
deepThought
Future<String> get
null or Future<String> 5
Advanced Software Development 2, Dr. Eliahu Khalastchi, 2019 ©

f.get() גורר future ו-join() מוחזק בולוק. 7
אל תגידו, blocked ו-join() מוחזק בולוק. 8.
~~בזותה... ו-join()~~ מוחזק בולוק ו-join() מוחזק בולוק. 9
מ"כ f.get() מוחזק בולוק. 10

הצורה שצ'ק אינטראקטיבית

תודה לך!

ר' יב' כ קד זכ עז'וֹן צ'וֹעַד

Since JDK 8 – lambda expressions

```
public String deepThought(){  
    // takes a really really long time...  
    return "42";  
}
```



```
ExecutorService executor=Executors.newCachedThreadPool();  
  
Future<String> f = executor.submit( ()-> {  
    return deepThought();  
});
```

ר' יב' כ קד עז'וֹן צ'וֹעַד לינ'ו
new callabk → ג'וּסָה call(C)

Still, resources are wasted because of the blocking get() call

```
//...  
System.out.println(f.get()); // blocks until an answer is given
```

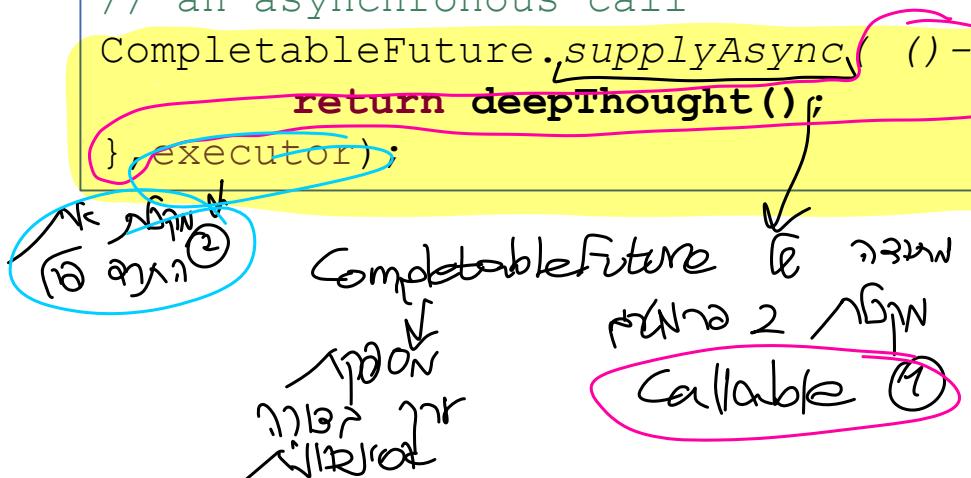
Using CompletableFuture

```
public String deepThought() {  
    // takes a really really long time..  
    return "42";  
}
```



```
ExecutorService executor=Executors.newCachedThreadPool();
```

```
// an asynchronous call
CompletableFuture.supplyAsync( () -> {
    return deepThought();
})
```



Using CompletableFuture

```
public String deepThought(){  
    // takes a really really long time...  
    return "42";  
}
```

```
// an asynchronous call  
CompletableFuture.supplyAsync( () -> {  
    return deepThought();  
});
```

Uses the default ForkJoin Pool

Fork Join Pool → the (faster) library to run the code in parallel

בפניהם זרויים
25 נק
היה בול' 31%

~6ms נק סט
(executer נו)



75% מ-15 דקות נס'
WIN, Libraries executor
Thread pool נס' מ-
75% מ-15 דקות נס'

הנ ש כל מה ש supplyAsync

Using CompletableFuture

הנ מיצירת, תפליט, CF

```
public String deepThought() {
    // takes a really really long time...
    return "42";
}
```

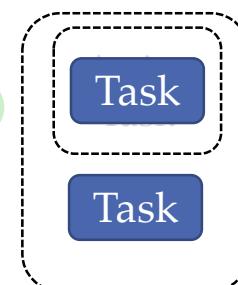
הנ מיצירת, ForkJoin Pool הינו או deepThought() הינה שום סלולר FC מילאנו בולטן

```
CompletableFuture<String> fc = CompletableFuture.supplyAsync(() -> {
    return deepThought();
});
```

```
fc.thenAccept((String answer) -> { System.out.println("answer: "+answer); });
```

Reactive pattern: This action will be taken right after deep thought is finished

הנ מיצירת, deepThought הינה מילאנו supplyAsync כוונתית למשתמש ComplettableFuture(zon) הינו fc מילאנו בולטן. The new J.K. Future-> Submit ימיהו IND גוון



- Future \sim IK UNIC SK, סר מתקבָּן SupplyAsync \sim IK UNIC SK
 : AN'EN IK B3R SK 771 get() kms IK UNIC Future \rightarrow
 IK UNIC UNIC <F>

Using CompletableFuture

```
public String deepThought(){
    // takes a really really long time...
    return "42";
}
```

```
CompletableFuture<String> fc = CompletableFuture.supplyAsync(() -> {
    return deepThought();
});
```

```
fc.thenAccept((String answer) -> { System.out.println("answer: " + answer); });
```

Reactive pattern: This action will be taken right after deep thought is finished

Fluent Programming: each method returns its object, allowing chained calls

Returns
CompletableFuture<String>

נתקע כהן גיגי היבר ווילם
 יג' נסובן, Void היבר
 .longify תרנגול
 ○○○○○ ← String → ○○○○○
 String → UNIC IK



היבר כף עט
 קוק, ווילם
 Functional
 Interface

מג אונליין
 גראף גראף

Consumer | IK IK UNIC UNIC

(לען יונק פלטינום קס, פיק גיבן גן חת-ה-זען קון)

ב' *get()* ו *set()* הם פונקציות שפועלות על אובייקט *Future*.
הן מוחזקות בפונקציית *thenAccept()*, *thenRun()*, *thenApply()* ועוד.
בנוסף להן יש פונקציית *cancel()*.

לפ' *thenAccept()* נסמן *pro'* בפ' *keepThoughts* ו*line 271*
ומ' *thenRun()* נסמן *pro'* בפ' *thenAccept()* ו*line 271*
ולפ' *thenApply()* נסמן *pro'* בפ' *thenAccept()* ו*line 271*.

Fluent programming \leftarrow पाक रहे थे शब्द String के CF का यहाँ नियन्त्रण करने का एक उपाय है।

Using CompletableFuture

```
public String deepThought() {  
    // takes a really really long time..  
    return "42";  
}
```

For string's deep thought → string & CF class will be discussed



```
CompletableFuture.supplyAsync( () -> { return deepThought(); } )
```

```
thenApply answer->Integer.parseInt(answer))
```

```
thenApply(x->x*2)
```

.thenAccept(~~answe~~

ChenAccept ✓ 218112
void main(int a)

Dear Captain Morgan, thank you

integer & cf

Functional Interface

 answer מילון ערך וDEFINE

APC \leftarrow $\text{fix } c$
 $\cup D \leftarrow \emptyset$

Fluent programming

Console X
terminated> Main [Java Application] /Applications/Eclipse.app/Contents/Eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.r
suer: 48

Using CompletableFuture

```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```



```
CompletableFuture.supplyAsync(() -> {  
    return deepThought(); }, executor)  
.thenApply(answer -> Integer.parseInt(answer))  
.the
```

- `thenAccept(Consumer<? super Void> action) : CompletableFuture<Void>` - `CompletableFuture`
- `thenAcceptAsync(Consumer<? super Void> action) : CompletableFuture<Void>` - `CompletableFuture`
- `thenAcceptAsync(Consumer<? super Void> action, Executor executor) : CompletableFuture<Void>` - `CompletableFuture`
- `thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super Void, ? super U> action) : CompletableFuture<Void>` - `CompletableFuture`
- `thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super Void, ? super U> action) : CompletableFuture<Void>` - `CompletableFuture`
- `thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super Void, ? super U> action, Executor executor) : CompletableFuture<Void>` - `CompletableFuture`
- `thenApply(Function<? super Void, ? extends U> fn) : CompletableFuture<U>` - `CompletableFuture`
- `thenApplyAsync(Function<? super Void, ? extends U> fn) : CompletableFuture<U>` - `CompletableFuture`
- `thenApplyAsync(Function<? super Void, ? extends U> fn, Executor executor) : CompletableFuture<U>` - `CompletableFuture`
- `thenCombine(CompletionStage<? extends U> other, BiFunction<? super Void, ? super U, ? extends V> fn) : CompletableFuture<V>` - `CompletableFuture`
- `thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super Void, ? super U, ? extends V> fn) : CompletableFuture<V>` - `CompletableFuture`
- `thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super Void, ? super U, ? extends V> fn, Executor executor) : CompletableFuture<V>` - `CompletableFuture`
- `thenCompose(Function<? super Void, ? extends CompletionStage<U>> fn) : CompletableFuture<U>` - `CompletableFuture`
- `thenComposeAsync(Function<? super Void, ? extends CompletionStage<U>> fn) : CompletableFuture<U>` - `CompletableFuture`
- `thenComposeAsync(Function<? super Void, ? extends CompletionStage<U>> fn, Executor executor) : CompletableFuture<U>` - `CompletableFuture`
- `thenRun(Runnable action) : CompletableFuture<Void>` - `CompletableFuture`
- `thenRunAsync(Runnable action) : CompletableFuture<Void>` - `CompletableFuture`

פרוסס

התוכנית שלנו אחרי הקימפבול יושבת על הדיסק וכשמבצעים אותה היא עוברת לראם והופכת לפרוסס שאותו מערכת הפעלה מתזמנת לרוץ על אחת הליבות. מעבד עם כמה ליבות יכול להריץ פרוססים כמספר הליבות במקביל. מעבר לזה קיים תהליך Context Switch שהוא מחליף את כל המידע שיש באוגרים לפROSSES אחר תוך כדי שהוא מקפיא את הפROSSES הנוכחי בראמ עד שיורץ בתורו שוב על המעבד. בזמן ההחלפה יש זמן מושך פרוסס לא רץ על המעבד כי צריך להחליף את המידע. בפועל ההחלפה מתבצעת מספיק מהר שהמשתמש לא ישים לב לכך.

- כדי לתזמן את כל הפROSSESים מערכת הפעלה משתמשת בתור שנקרא queue ready שבו:
- כל פרוסס שרצה על המעבד וביקש בקשת O/I נמצא בתור של O/I ואם הוא קיבל את O/I הוא יחזיר חזירה לתור queue ready.
 - אם נגמר לפROSSES שרצה הזמן במעבד מערכת הפעלה מקפיא אותו ומעבירה אותו חזירה לסוף התור.
 - אם הפROSSES רץ והוליד תהליך חדש וביקש להמתין ליד הוא יהיה במצב של waiting יסימ את התהילך שלו ואז האב יחזיר חזירה לתור.

마חר וכל תהליך מחליף תהליך אחר בזיכרון ראם איז אפשר להעביר ביניהם מידע וכל הזיכרון של תהליך מסוים הוא שלו בלבד ותהליך אחר לא יכול לגשת אליו.

user threads

התהילכנים (thread) הם חלק מתהליך (process) מסוימים וכל אחד מהם מקבל חלק זמן הריצה של התהיליך שמנמו הוא נוצר החילוף ביניהם יהי מhair יותר מאשר אין החלפה של כל הזיכרון בראמ. JVM הוא תהליך מסוים ובתוכו אפשר ליצור thread'ים ואפשר לחלק ביןיהם מידע מאחר שהם נמצאים על אותו תהליך.

- היום בעקבות שפות כמו GO שבה הבקשת לתהילון ותהליך היא זהה ומכך יש לרוב המעבדים מספר ליבות לא נדע אם קיבלנו תהילון או תהליך שבו יש תהילון אחד והוא רץ על ליבה פנוייה.
- אם אני רוצה יותר זמן CPU לתוכנית שלי אני אעדיף לכתוב בתהילכנים
 - אם אני רוצה למנוע התרסקות של התוכנית למגררי ובלי שאוכל להוציא הודעה שגיאה אז אני משתמש בתהילניים מאחר ומתקיים אחד קורס זה לא גורם ל垦יסת השאר ותהליך אחר יכול לזרזות שהטהיליך קרט
 - אם אני רוצה לחלק זיכרון בנסיבות אני משתמש בתהילכנים
 - אם אני רוצה להחליף בנסיבות בין שימושות אני משתמש בתהילכנים

מה ההבדל בין תהילכנים של מערכת הפעלה לתהילכנים של המשתמש?

כל התהילכנים שאנו פותחים ביעדרת JAVA לדוגמה אלה תהילכנים שנחשבים תהילconi משתמשים והם יכולים להתפרק ביחס מסוים בתהילכנים של מערכת הפעלה בדרך כלל יחס של 1:1 כלומר על כל תהילון שאנו פותחים מערכת הפעלה תפתח תהילון משלה.

מתי כדאי להשתמש בתהילכנים?

שאנו מוחכים ל- O/I בתהילון מסוים הוא מחייב לשובה ולכן יכול לעקם את התור השני ולפיכך אם יש לנו תהילון שבו מוחכים ל- O/I יהיה יותר מhair להריץ במקביל אליו תהילכנים נוספים שאולי יוסיפו מעט זמן בגלל כל ה啻לים שיש בין תהילכנים אבל יחסכו את הזמן שבו תהילון מסוים מוחכים ל- O/I.

נרצה לחלק את התהיליכונים בערך 50/50 לתהיליכונים שעושים O/I ותהיליכונים שעושים CPU על מנת לנצל את הזמן בצורה הטובה ביותר ובזמן שהטהיליכונים שעושים O/I מחייבים לשובהקדם את התהיליכונים שעושים CPU.

java threads

- קיטים `interface` בשם Runnable ויש בו מתודה אחת `public void run` (כלומר אפשר למשם בلمבדה).
- Thread היא מחלקה שמשמשת את Runnable ויש בה שתי מתודות אחת `run` מהר `start` והשנייה `public void sleep()` שמקחת את הקוד שנמצא בחן ושם אותו `ready` כדי שירוץ בركע.
- אנחנו צריכים ליצור מחלקה שיורשת את Thread ודורשת את חן. בכך נסמן Thread מילה משתנה Runnable כך שגם דורשים את `run` אך הפונקציה הדורשת היא זו שתירוץ בركע ואם לא דורשים את חן אז המשתנה Runnable יירוץ. (Strategy pattern)
- שיטה 1:** לרשות את Thread למחלקה חדשה ולמשם מחדש את חן עם הקוד שאנו רוצים להריץ ברכע חיסרון - בזבזנו את הירושה היחידה שהייתה לנו
- שיטה 2:** נמשם במחלקה שאנו רוצים את Runnable ובפונקציה חן שלו למת לו את הקוד שרים להריץ ברכע לאחר מכן נוכל ליצור מופע חדש של המחלקה Thread ולתת לבניי שלו את המולט Runnable שיצרנו.
- חיסרון - היינו צריכים לשנות את הקוד מקור של המחלקה שלנו ואנחנו רוצים קוד שהוא סגור לשינויים
- שיטה 3:** משתמש ב- Object adapters נשתמש את Runnable ובתוכו יש לנו מופע של interface של המחלקה שלנו.
- שיטה 4:** יצירת מופע אונימי של Thread ובתוכו נעשה באמצעות למבדה שימוש `lambda` Thread ותפקידו שמימושה את השיטה הזאת יפתח thread חדש.

controlling java threads

כאשר יש לנו תהיליכון שרצ אנחנו יכולים:

- לגרום לו ללבכת לשון
- להעיר אותו
- לגרום לו לחכות ולהודיע לנו מתי הזמן להמשיך
- אפשר לחכות שישתיים
- אפשר לשאול אם הוא עדין ח'
- ועוד

מחזור החיים של thread

thread מתחילה בקר שהוא נולד ומועבר לqueue ready שהוא תור עדיפות. המתזמן של JVM בוחר את הראשון והthead מתחילה את הריצה שלו אם thread ביקש O/I אז הוא עובר להיות Blocked והthead הבא מתחילה לרווח זמן זה thread'ים יכולים להתחלף ביניהם בגלל שנגמר להם הזמן במעבד כשהthead שביבקש O/I מקבל תשובה אז הוא חוזר לסופ queue ready והוא שביבקש sleepsleep והוא לא חוזר לטור עד שנגמר לו הזמן במצב זה או שהערנו אותו בעזרת פקודת interrupt וכשהוא מתעורר הוא חוזר לסופ הטור אם תהילך קיבל פקודת wait הוא עבר למאכזב waiting והוא חוזר לסופ הטור כל התור שביבקש waiting עבר לסופ תור queue ready כאשר אינו מסיים את החיים שלו הוא עובר למאכזב dead.

critical section problem

הבעיה

בזמן שימוש בthreads שניים שניים יכולים לבקש את אותו משאב ולדרס אותו פעמיים.

פתרונות:

- לאובייקטים במאבל יש מנעול וברגע שthread רוצה לגשת לפונקציה מסוימת שהוא מסונכרנת אם לא משתמשים בפונקציה מסונכרנת אז הוא נכנס לפונקציה והמנעל ננעל ועכשו thread אחרים לא יכולים להשתמש באך פונקציה מסונכרנת מאותו אובייקט.
השימוש בynchronized הוא איטי ופרימיטיבי והבעיה הנוסף היא שאזה שולח את thread השנייה לגשת לאובייקט מסוים ל-block ויש היום שיטות יותר טובות שנלמד בהמשך עליו על מנת לפתור את הבעיה.

Active object

אובייקטים הם בדרך כלל פסיביים כלומר מישו אחר שהוא לא האובייקט מפעיל פונקציה על האובייקט. אובייקט אקטיביומי שմבקש ממנו להפעיל משהו האובייקט יחולט האם ומתי להריץ את הפוקודה הזאת. דבר נוסף שקרה באובייקט אקטיבי זאת הפרדה בין התהיליכון שהפעיל את הפוקודה לבין התהיליכון שמריץ את הפוקודה.

אימפלמנטציה פשוטה לאובייקט אקטיבי תהיה יצירה של תור (יכול להיות גם תור עדיפויות) של ת'רדים באובייקט שאנו נרים לפ' הסדר (בלולאה) את הלולאה אפשר לעזר בעזרת משתנה בוליאני. האובייקט האקטיבי מרים את הת'רידים אחד אחריו השני אבל הוא מרים אותם בת'רד נפרד וכן אפשר לשוטם בדברים במקביל לפעולות הלאה. על מנת לעזור את הלולאה אנחנו משתמשים במתודה `close` שתסגור את הלולאה (באמצעות המשתנה הבוליאני) וכך שנוכל באמצעות ליצאת מהלולאה ולסגור את הת'ריד שמננו אנחנו מרים את ההוראות אנחנו יכולים להשתמש באחת מושתי השיטות הבאות:

1. לשים את הת'ריד של הריצה בתוך משתנה של המחלקה ולאחר מכן מנסים את המשתנה הבוליאני כדי שיעזר את הלולאה נסיף `interrupt` לת'ריד הריצה כדי שיבדק את המשתנה הבוליאני של הלולאה ויסתiem. דומה לפעולה של `kill`.
2. פתרון יותר אלגנטי - במתודה `close` אנחנו מכנים מוכנים Runnable לטור שבו הוא אומר שהמשתנה הבוליאני צריך לעזור את הלולאה וכשהת'ריד הריצה מגיע ל`Runnable` של הסירה הוא משנה את המשתנה הבוליאני כך שהלולאה תעצר ומיד לאחר מכן בודק את תנאי הלולאה ולכן היא עצרת. ולאחר מכן נכנס לסוף התור אז כל המשימות שנותרו יסתiem ורק אז הלולאה עצרת.

double checked locking

המטרה היא ליצור Singleton (מבנה עיצוב שעוזרת לנו ליצור אך ורק אובייקט אחד מסווג זהה) שהוא בטוח לת'רדים.

אפשר לפתור את הבעיה של גישה בו זמנית של כמה ת'רדים לאובייקט על ידי `synchronized`. הבעיה בפתרון זהה שאנחנו צריכים לעזור את היצירה רק בהתחלה ולאחר כך כשייש כבר את האובייקט זה רק פוגע בביבוצים ולא מעיל לנו יותר.

פתרון יותר יעיל יהיה לשימוש בлок synchronized על `this` בתוך ה'ז' שבודק האם כבר נוצר אובייקט צזה בתוך הבלוק אנחנו שמים תנאי נסיף שבודק אם נוצר האובייקט על מנת לעזור את הת'רדים שנטקו על הסינכרון קודם. לאחר היצירה של האובייקט הבעיה של הסינכרון נעלמת כי מודלים עלייה. בפתרון זהה יש באג מאוחר והקונפילר יכול לעשות אופטימיזציה ולשנות את הקוד ככה שהוא לא יעבד כמו שהשווינו. ת'ריד נסיף עלול לקבל את הכתובות לאובייקט שבתהליך יצרה למטרות שהוא עדין לא סיים להבנות. וכן הוא נמצא `cache` volatile - אומר שהמשתנה שעליו זה הוגדר קיים בזיכרון ויש רק אחד וייחיד ממנו כלומר, הוא לא נמצא `cache` וכן הוא משתנה שהוא מאד יקר.

אנחנו יכולים לתקן את הבאג באמצעות יצירת משתנה שהוא volatile ועכשו המשתנה לא יקבל את הכתובות לאובייקט לפני שהוא מוכן. הבעיה עכשו היא שהקוד הזה יקר מאוד מאחר והגישה לראמם היא יקרה.

אפשר לפתור את הבעיה הזאת באמצעות משתנה לוקאלי נסיף אליו אנחנו מכנים את המשתנה volatile בהתחלת המשתנה `init` וכן יהיה ניתן להגדיר אותו בסוף כשמחזירים את הערך של המשתנה הוא כבר נמצא במשתנה הлокאלי שהוא זול יותר וזה מספר את הביבוצעים בכ-25%.

פתרון שישים רק בשפה Java - פתרון שמשתמש במחלקות סטטיות: לשים את המשתנה סטטי מה שייגרום לJava לטען בהתחלה התוכנית ולכן אף ת'ריד עוד לא יכול לגשת אליו. הבעיה כאן זה שאנחנו יוצרים את האובייקט בלבד שהוא נדרש.

לכן אנחנו יוצרים מחלקה סטטית בתוך המחלקה שאנו רוצים ואז המחלקה הסטטית הפנימית תען בפעם הראשונה רק כאשר פנו למחלקה החיצונית וזה אין בעיה של גישה מרובה והמחלקה הפנימית נוצרת רק פעם אחת.

thread pool

מנגןן שאנו נזיר לו מושימות והוא ינהל את הת'רדים ואפשר לשלוט גם על כמות הת'רדים שהוא יכול להריץ על מנת לשלוט על הביצועים של התוכנה.

אנו רוצים להפריד בין הת'רד `Runnable` שלו על מנת שהוא יוכל להריץ מושימות אחרות ברגע שהוא מסיים משימה אחת על מנת לחסוך בתהילך בניית הת'רד שהוא תהליך יקר.

Executor Interface

מגדיר את הפונקציה `execute` שמקבלת `Runnable` וככה אנחנו מפרידים בין הת'רד לבין `Runnable` כי בפונקציה `execute` זה את אפשר לעשות עם `Runnable` מה שאנו רוצים - לדוגמה להריץ אותו בת'רד חדש או סתם להריץ אותו.

הpool `thread` בעדרת `Executor` יוצר מספר מסוים של ת'רדים ומזריק לתוכם את המשימות באמצעות תור של משימות ככה אנחנו לא צריכים לנוהל את הת'רדים ורק צריכים להעביר משימות לpool שיבצע אותן. [להיכנס דוגמאות של סוגים שונים שנמצאים תחת `Executor`]

callable and future

שאנו מרים מותודה והוא נמצא בת'רד נפרד אך היא `Asynchronous Method Invocation` (AMI) זה מה שקרה באובייקט אקטיבי ובת'רד פול. שאנו רצים לקבל ערך חוזה עברו הת'רד אנחנו משתמשים `Callable` ב-`ExecutorService` יש לנו חוץ מהמתודה `execute` שמקבלת `Runnable` גם את המותודה `submit` שמקבלת `Callable` ש杂志社 את callable לטור משימות של `thread pool` וחוזרת מיד בזמן שהמשימה נמצאת ומבצעת בת'רד פול. אך `submit` מחזירה אובייקט שיכיל בעתיד את האובייקט שאנו רצים שהת'רד יחזיר כישות'ם - `Future`.

Guarded Suspension

מנהל פעולות שמצריכות גם נעה כדי שייהו נגישים ונגישות וגם שיטקיים תנאים מסוימים שהוגדר כתנאי שציר להתקיים לפני הפעולה.

משתמשים במתודות סינכרניות בהן בפועלה שאנו רצים לבדוק עבורה את התנאי אנחנו בודקים אותו ואם הוא לא מתקיים אז מכניסים את האובייקט במצב של `wait` ובפועלה עדכון אנחנו מערימים כל פעם את הפעולה על מנת שתבדוק אם התנאי מתקיים או לא.

משתמשים בתבנית העיצוב `Future` על מנת שלא יהיה ניתן לגשת למשתנה שעוד לא הוגדר אבל הכנו עבורי מעטפת לכשייה מוקן.

הmethodות חייבות להיות סינכרניות מאחר והפעולות `wait` ו`notify` חייבות להיות מסונכרנות על אותו האובייקט כדי שלא יהיה מצב שימושו יקרה `to wait` בזמן שימושו אחר קרה `to notify`.

thread scheduling

שאנו רוצים לסנכרן בין שני ת'רדים אנחנו יכולים לכתוב תוכנית שתסנכרן בינהן באמצעות sleep או wait אבל זה לא טוב כי הסינכרון לא יהיה מדויק ואחרי הרבה זמן התוכנית תצא מסינכרון אחר והתרדים נכנסים בסוף ready queue כאשרנו מעריכים אותו ובנוסף זה לא שיטה טובה יותר וזה טוב רק למספר מועט מאוד של ת'רדים. لكن אנחנו רוצים כל אוטומטי שיעשה את זה ויפיצה על העיבוב.

לכן משתמשים בtimer זאת מחלוקת שהולכת להריצ' לנו פקודה מסוימת ואפשר לעשות את זה בלולאה ואין צורך להתעסק בת'רדים בנוסך זהה.

Replacing Synchronized

על מנת להתמודד עם הבעיה של סינכרון של משתנים פרימיטיביים יש לנו את סוג המשתנים שהם אוטומטיים והם בטוחים לשימוש בת'רדים ולא שלוחים ת'רדים למצב של lock כך שהוא יכול לחסוך לנו זמן עבודה משמעותית ועודין לאפשר שימוש בתכנות מקביל. זה חלק מהספרייה של concurrent.util.java.

Deadlock Example

טבעת חנק זה מצב שבו כמה ת'רדים תופסים משאבים על מנת לחשב משהו וכל אחד מהם תלוי באחד נוסף כך שנוצרת טבעת תלויות זהה עלול להוביל בהסתברות גבואה לכך שהמשאבים לא יושחרו וכל הת'רדים יתקעו. במקרה להשתמש בסינכרון אפשר להשתמש במנועים יותר מתקדמים שנמצאים לנו בספרייה של Java (לדוגמה TryLock) ואתם יכולים למנוע את המצב הזה.

Thread-safe containers

קיים בJava מבני נתונים שבתוכם לשימוש בת'רדים ולא מצריים דקורטור שיעזור פקודות ב synchronize אבל הם לא בטוחים לשימוש בת'רדים אם סדר הפעולות חשוב ועודין צריך לעשות את זה סינכרוני.

fork-join pool

כדי ליצור ת'רד שלא נחסם על ידי פקודות מסוימות כמו לדוגמה קרייה למידע מdatatype או הת'רד ייכה לשובה אנחנו נרצה להשתמש בת'רד פול שיעבוד בשיטת שגר ושכח ופונה לנו משאים יקרים וזמן לדברים נוספים ויעבור בצורה אסינכרונית. בסיסו fork-join יש את האפשרות לעשות work steal כלומר שת'רד מסויים עבדה של ת'רד אחר אם יש לו יותר מדי משימות בתור של ת'רד אחד (משימות מעדיות לחזור לאוותה ליבת שעיליה הם התחליו) ולכןpool join-fork מאין את הליבות ומעביר משימות נוספות התור של הליבה העומסה לליבות שיוטר פניו. ככה נקבל pool שרצ יותר מהר יותר עיל.

completeable future

שאנו משתמשים בfuture רגיל אך מתבצע לנו הרבה זמן שבו הת'רד שקרה לא עשה כלום ורק ממתין לתשובה ממנו. במקרים זאת אנחנו נרצה מבנה נתונים שמאפשר לנו להמשיך לעשות פעולות בזמן שאנו מוחכים למידע או לשחרר משאים בינתיים כדי לא לבלבב אותם וזה completeable future מאפשר לנו.

event driven programming

אנחנו לא אחראים על הסדר שבו הקלט נאוסף ואנחנו צריכים להגיב לפעולות שנעשו. הפקציית שאומרת לתוכנית איך להגיב נקראת **Event Handler**.

Java GUI technologies

קיים בעיה בAPI JavaBeans לאחר והוא רוצhim לתוכר בקוד cross platform והחלון של מערכות הפעלה שונות הוא שונה ומשתמש בקוד שונה מאשר מערכות הפעלה. על מנת לא ליצור חלון חדש ולהשתמש בחולונות המקוריים של מערכת הפעלה השתמשו הרבה זמן בספריה SWT שפיתחה IBM שבעור חלונות שקיימים במערכת הפעלה היא משתמשת בחולנות המקוריים ובעור חלונות אחרים הוא משתמש בעיצוב של התוכנה. בעבר היו צריכים לכתוב קוד שונה לכל מערכת הפעלה או להשתמש בספריה שהיתה מחייבת את החלון של מערכת הפעלה אבל לא הייתה קשורה אליה ולכן לא ניתן להשתמש בכל האפשרויות והגדרות שהוא הגדר ויהי רגיל אליו.

shared concepts

על מנת לא ללמד מהתחלת על כל ספריה זאת נלמד את הדברים שימושיים לכך. אנחנו צריכים להפריד בין הפרזנטציה לבין הלוגיקה של הפרזנטציה. זאת על מנת ליצור מחלקות שנינן להשתמש בהן מחדש גם לשימושים נוספים וכן נעשה שימוש בחומרה Design Pattern strategy של strategy.

במחינת פרזנטציה אנחנו מעצבים את **layout** ומושגים קומפוננטות. מבחןת לוגיקה אנחנו מושגים את logic pattern במערכות strategy במערכות הרצויות.

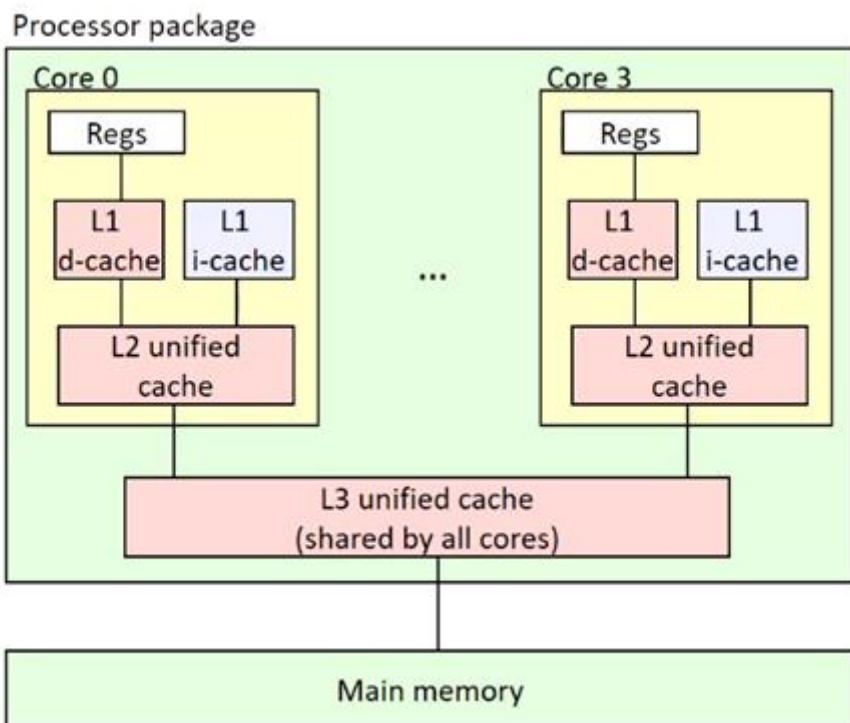
presentation logic

נרצה שאת הקובץ של העיצוב שהעורק הייזואלי ייצור ייה קובץ XML כדי שייה אפשר להשתמש בזה בכל שפה שנבחר ועם כדי שהעורק הייזואלי ידע לקרוא גם מתקנת ידע לקרוא ולעורך.

נרצה לאorgan את כל הקוד של controls במחלקה אחת, שתירש מחלוקת אחרת כללית יותר כדי לחסוך עבודה שעשו באופן כללי לדוגמה לקובץ ועם כדי שמתכנתים אחרים ידעו עם מה הם עובדים ובנוסף נרצה שאת מה שאפשר לקבל קוד מחוץ למחלוקת אז שיתקבל מחוץ למחלוקת כדי ליצור מחלוקת כמה שיותר כללית.

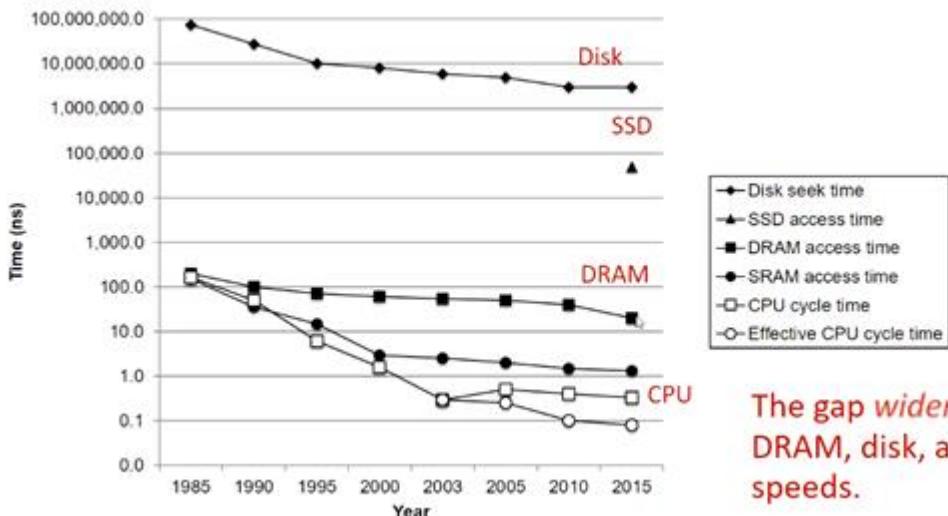
cache memory and locality

אנו נעסק בפרק זה באופטימיזציות קוד. השאלה שנסאלת היא איך נכתב קוד שיהיה ידידותי לcache, כלומר יעשה שימוש באופן שבו מערכת הפעלה שלנו מפעילה ומשתמשת(cache), כך שהמערכת תוכל לרווח הרבה יותר מהר ויעיל. נסתכל על מעבד i7 של אינטל.



לפי הציור, ניתן לראות שזהו מעבד בעל 4 ליבות. בכל ליבה יש רמות שונות של cache. לכל ליבה יש את האוגרים שלה. באוגרים יש את המידע שהכי זמין והכי נגיש למעבד. באוגרים מתבצעים כל מיני חישובים רלוונטיים - כאשר הקומpileר תרגם את הקוד שלנו לשפת מכונה, ההוראות עובדות על האוגרים הללו. **ברגע שאנו צריכים לבצע את ההוראה הבאה, אם רצים לבקש בcache level 1 - cache 1 (L1).** (שימוש לב Ci יש לנו הפרדה בין instruction ל data בכל רמה). אם לא מצאתי את המידע שם, נבקש ברמה השניה, שם המידע מעורכב - גם ה Data וגם ה instruction. אם גם שם לא מצאנו את המידע נחפש ב 3 שמות כבר גודל זמן! הגישה. אם לא מצאנו את המידע שם נצטרך להביא אותו מהRAM **וזה מצב ממש לא טוב שלוקח הזמן!** הרעיון הוא לסדר את המידע cache בצורה כזאת שהמידע הדחוס יותר יהיה/cache והמידע שאינו אצטריך אחר כך יהיה ברמות התחרונות יותר.

בגרף הבא נוכל לראות את הבדלים בזמןי הגישה הגודלים בין המעבד לבין המהיר.



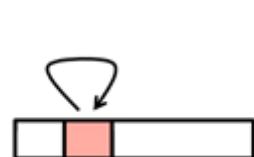
The gap widens between DRAM, disk, and CPU speeds.

אנו צריכים לטפל בפערים אלו! ולשם זה נשתמש בעקרון **Locality** (מקומיות) הראינו הוא שכשאנו מרים קוד, אם לא צריכים את כל הקוד. אם לדוגמה אנחנו בתוך לולאה, אנו צריכים רק את המידע שהלולה דרושת. למעשה התוכנית רצה בין לוקליות אחת לאחרת, ואם נתאorgan כך שמה שצריך כתע יימצא באחת מהרמות של cache אז לא נדרש להביא מידע שהוא יותר איטי.

ישנם שני סוגים של locality

локליות זמנית : temporal locality : משתמש באותו הוראה או משתנה שוב ושוב.

ההורה sum מוגדרת בפעם הראשונה ומשתמשת בה בכל הוראות sum מאוחר יותר



локליות מקומית : spatial locality : אם משתמש בהוראה מסוימת אז נדרש גם את ההוראות ש"מסביב" אותה הוראה.

ההורה sum מוגדרת בפעם הראשונה ומשתמשת בה בכל הוראות sum מאוחר יותר



זהו העקרון המנחה בניהול של cache' ואם בין אותו נוכל לתוכנת קוד ידידותי לשזה cache' שהוא קוד יותר נכון. נסתכל לדוגמא על הסיכון הבא למערך :

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Spatial or Temporal Locality?

בואו נפריד בין spatial לtemporal ונסתכל על ההוראות השונות:

1. האם הגישה לתאים במערך היא spatial או temporal ? כמוון spatial ! אנו נוגעים במערך ובתא שמנמצא לצידיו.
2. האם עדכן הוויסוס היא spatial או temporal ? כמוון temporal! אנו נוגעים אך ורק באותו משטנה שוב ושוב
3. האם ההוראה עצמה לבצע את 1 ו 2 שוב ושוב היא spatial או temporal ? היא spatial כי כמו בו אנחנו שוב ושוב בודקים תאים רבים במערך.
4. העובדה שאנו מוצאים את אותן הוראות בולולאה היא למעשה temporal כי זהה מקבצת של הוראות !

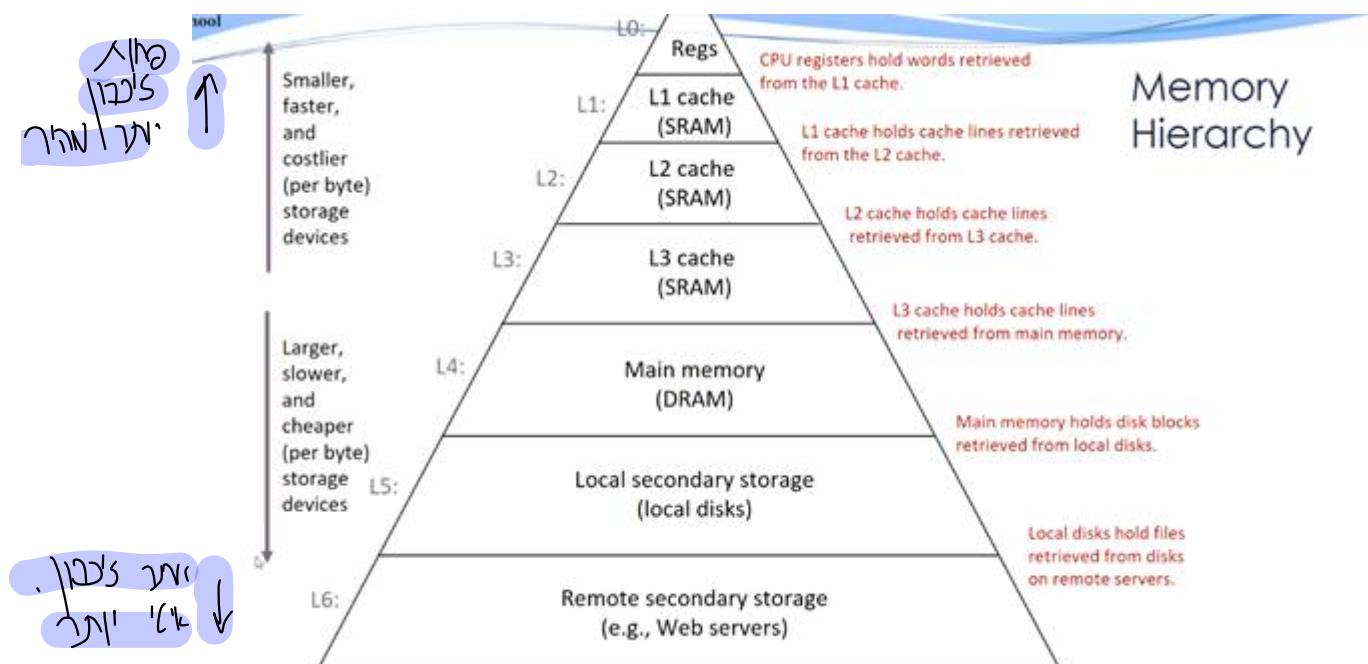
הסביר ל 3-4

בפקודת עדכן הוויסוס בקוד אסמבלי מדובר בסדרה של הוראות. (השג את הערך במערך, עדכן את וויסס וכו' ..) כולם אם אנחנו ביצענו הוראה אחת אנו צריכים גם את ההוראה שבאה אחריה. עקב העובדה שככל ההוראות חוזרות על עצמן שוב ושוב בולולאה, הינה עדיף שסט ההוראות האלה יישב בcache, על מנת שלא נדרש לגשת לRam כל פעם ולהציג בזמן איטי את אותן הוראות.

Memory Hierarchies

כפי שאמरנו, המעבדים נהיות יותר מהירים והפער ב מהירותם שלהם לעומת הram ובודאי הדיסק (זמן הגישה אליו הוא הגדול ביותר) והדילמות הגדולות היכן אנו שומרים את הinstructions והואן שומרים את data הופכות למאי צוואר בקבוק. מען זה יש לנו היררכיה של זכרון.

נראה זאת בצורה פירמידה :



כל שאמו עולים בפירמידה יש פחות זכרון אך הוא יותר מהיר, וככל שamu יורד בפירמידה יש יותר זכרון אך זמן הגישה אליו הוא הרבה יותר איטי.

איך אנו מצליחים את מבנה הפירמידה? בעזרת השימוש בлокליות! כפי שאמרנו אם נארגן את התוכנית שלנו כך שמה שאנו צריכים כרגע יהיה הכל קרוב למעבד, זהו המצביע הרצוי. בכל פעם שמה שרצינו לא נמצא שם, נטרך ללית לrama הבאה של cache. כל k level הוא catchen של hn+1.

במקרה וביקשנו מידע או הוראה מאחד הcachen -ים ולא מצאנו אותה, אנו קוראים לזה החטא - miss.

יש שלוש קטגוריות לmiss

- cold miss - אם התחילנו להריץ **תהליך חדש** שהוא עם ראשונה רצ, בזוזאות כל בקשה ראשונה צאת תהיה החטאה.
- capacity miss - במידה והלוקליות שצריך היא יותר ממה שהקAsh יכול לחתת לי לשומר.
- conflict miss - החטאות שנbowות ממשירת מידע של החומרה

The Memory Mountain

נראה דוגמא שנקראת "הר הזיכרון".
אם רצים למדוד את **read throughput** - כמה אנו יכולים לקרוא פר ייחידת זמן.
מדוד את זה בכמה megabit שטחן ממערך שנמצא בום בשנייה.
הר הזיכרון מראה את ההשפעות של מה שהוא כתובים בתוך הקוד. ככל שננצל את הלוקליות בצורה טובת ה

throughput

 יהיה יותר גבוה.

פונקציית הניסוי שלנו:

```
long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *       array "data" with stride of "stride",
 *       using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

Call **test()** with many
combinations of **elems**
and **stride**.

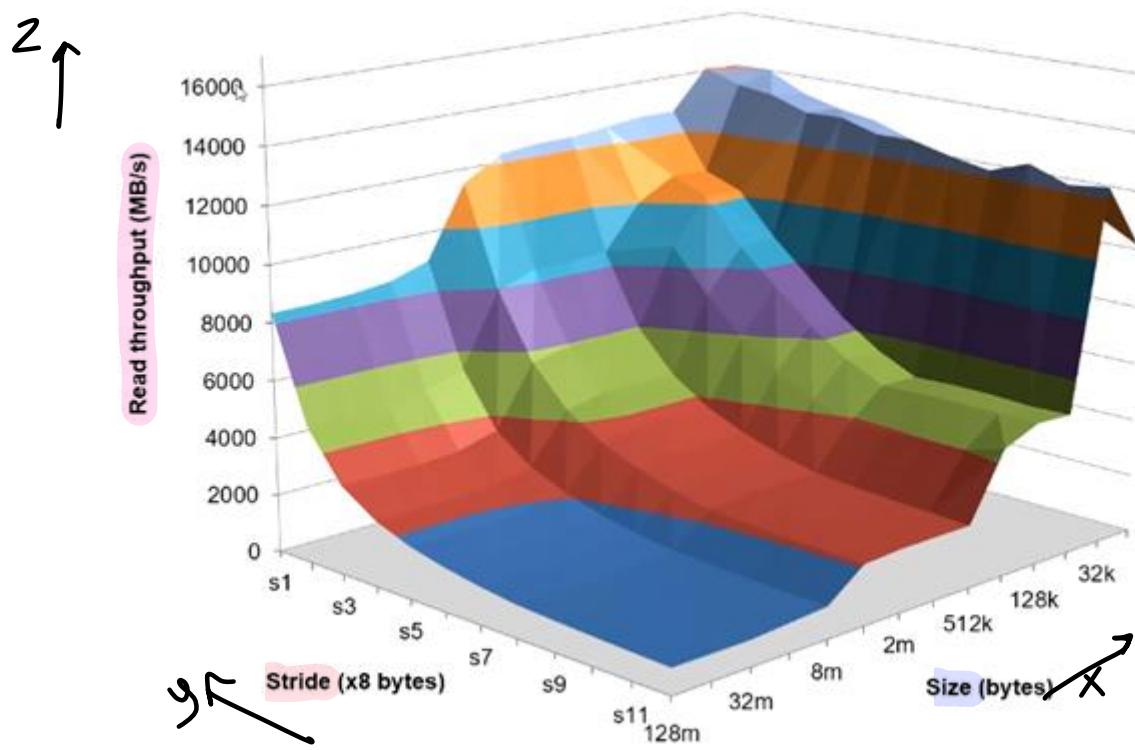
For each **elems** and
stride:

1. Call **test()** once to
warm up the caches.

2. Call **test()** again and
measure the read
throughput(MB/s)

בפונקציה זו יש את הפונקציה **test** שצריכה להחזיר **Int** - סכום של כל האיברים שנמצאים בערך **data**.
בלולאה הראשונה אנו מסכמים את ערכי המערך, אך במקומות בו לחוץ אחד אחד, אנו מקדמים אותו ב-4 צעדים, וכן הלולה ריצה פחות פעמים. מצד שני, בתוך הלולה אנו עושים יותר. זהה טכניקה ידועה שנשנה משמש בReLU להעשות אופטימיזציות בעtid. המידע של סכום כל האיברים נמצא במשתני **acc**. יש שארת לנו צרכים לטפל בה (4 התאים האחרנים) שאוותם נחשב בצורה הרגילה. בסוף מחזירים את הסכום של כלם. (לא מדובר על זה כרגע, אך שימוש לב **return** שני סכומים מופדרים בסוגרים, על מנת שנוכל לבצע שני חישובים במקביל באותה ליבה, ובכך לחסוך גם זמן אבל נשא זה ידוע בהרצתה הבאה).

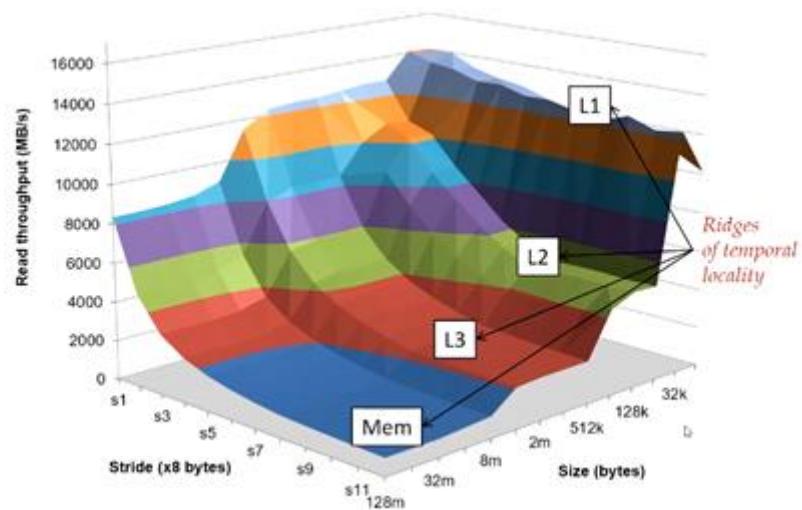
מה יש לנו עד כה - מערך, אנו מסכמים בו את האלמנטים ואנו משפיעים על מספר האלמנטים שאנו רוצים לסכום וגם על מספר הקפיצות. יש כאן בעיה, במידה וההשו **stride** יהיה 2 אנו מדלגים על איברים ויכול להיווצר מצב של Miss, כיוון שישארו איברים מחוץ לcache. נזכיר שהוא שמיינן אותנו הוא לדעת כמה איברים הפסקנו לקרוא ביחסית זמן. אם נשחק עם האלמנטים **elem** ו**stride** ונציג את זה כאיזשהו גרפ - נקבל את הר הזיכרון.



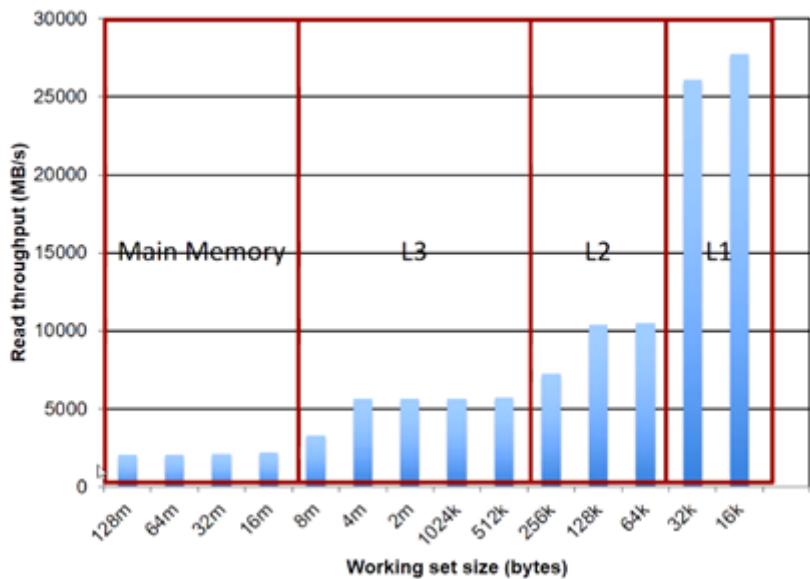
בציר הZ (ציר הגובה) יש את התהעבות throughput ככל שהוא נמצא במקום יותר גבוה - אנו מספקים לקרוא מידע בקצב גובה יותר והתוכנית רצה יותר מהר .

בציר האח - אנו רואים את גודל הבתים שהתבקשו לקרוא. בציר הע - יש את מספר הצעדים. מצד אחד אנו רואים ירידות דרישות, ומצד שני יש מדרכונות חלקים יותר. הירידות החדות (במקביל לציר הX) מייצגים את המעברים בין level level הבא אחורי בcache. ככל שמדובר בקריאה מוקבילה לcache יתיר לנו עוברים בין cache כדי שאמורנו ובכך ירדנו ב מהירות .

במקביל לציר הע יש את המדרכונות - ברגע שהגדלנו את הצעדים, ככה גדלו המס - ים - ולכן זה "עיקם" את throughput



כאן ניתן לראות את זה בצורה יותר פשוטה : נסתכל על ציר הZ - איך משפייע גודל הקובץ והCACHE שאמנו מצויים בו על התהעבות read throughput - על מהירות הקריאה ולמעשה הרצת התוכנית .



קל לראות שככל שעוברים בין ה-levelים - ימ throughput יורד משמעותית.

כעת נחזור את הגרפ' ונסתכל על ציר ה s אל מול ציר ה Z : נראה איך מספר הצעדים משפיע על ה throughput



ניתן לראות שעבור מספר הצעדים יש לנו Miss rate שホールך וגדל. ה Miss rate הוא גודל הצעד חלקי 8, (8 זה מספר הבטים שיש לנו בטור cache) – ככל שהגדלנו את הצעדים עליה miss rate עלה עד מספר הצעדים היה 8 ובוודאות אנו מפספסים כל בקשה שנבקש, ונצרך להביא אותה מה-level הבא.

במידה ונוכל לכתוב קוד שמנצל את ה cache ונהייה מודעים לגודלים שונים כתבים נוכל לשמר על read throughput גבוה.

Matrix Multiplication

נראה דוגמא נפוצה של הכפלת מטריצות. יש שתי מטריצות ריבועיות בגודל N , ואלה יהיו מטריצות של $doubles$ - כל דאבל (8 בתים). נרצה לנתח את המימושים השונים שאנו יכולים לעשות - כיצד אנו יכולים להשתמש בהם באלגוריתם, ובuzzרת שינויים קלים לקוד אנו יכולים Lagerom לו לרווח יותר מהר.

```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++) {
            sum += a[i][k] * b[k][j];
            c[i][j] = sum;
        }
    }
}
```

Variable sum held in register

cache friendly

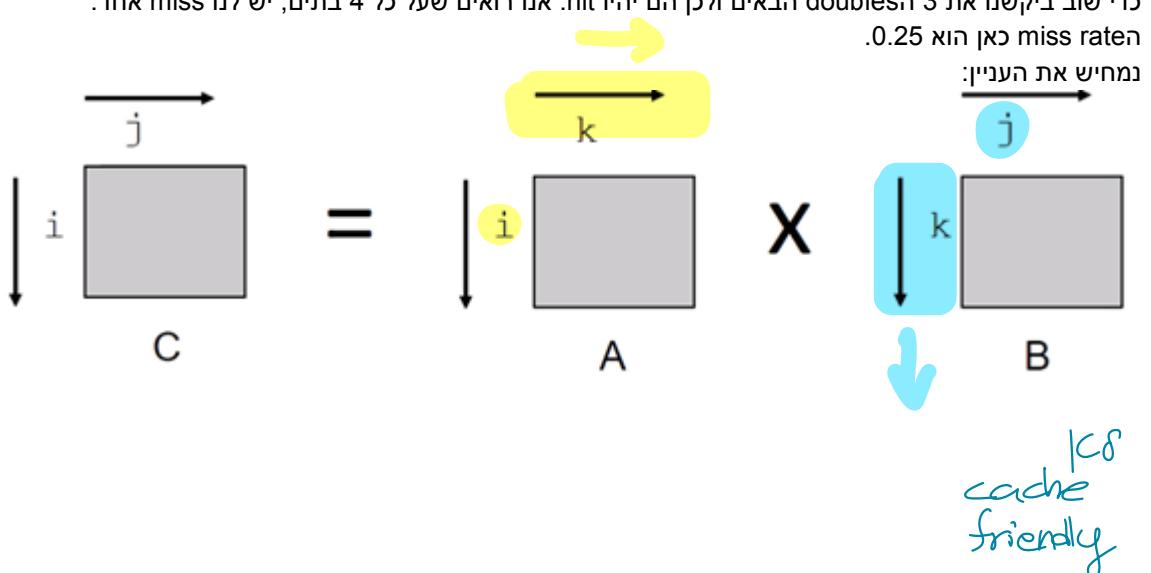
PRINT OF THIS IS OPTIMAL

PRINT OF THIS IS OPTIMAL

75 N.E.
K-311
Ann
Sor

כפועם כל העמודה K (זהו לא cache friendly - בהינתן עמודה j קפצנו למיטה בעוד העמודה מקובען - וכן תמיד יהיה miss)

ננתח את ההשפעה של הלוואות האלה ומה שקרה בפנים.
 נניח שהערך block size הוא 32 bytes (4 doubles). המשמעות היא שאם יש מקום ל 4 bytes cache - אז ברגע שנבוקש את הראשון הוא יהיה miss , אבל הוספנו גם את הבתים הבאים אחריו - את שלושת הדאבלים הבאים לאחר ה catchen .
 הם יהיו hit ! כנובע מכך האיבר הבא, ה catchen יהיה מלא (כי מלאנו כבר 4 bytes) וכך הוא יהיה Miss , אך תוך כדי שוב בזיכרון את 3 doubles הבאים וכאן הם יהיו hit . אנו רואים שעל כל 4 bytes, יש לנו miss אחד.
 rate miss 0.25icas.



בollowה - בהינתן שורה, רצמו על העמודות (במטריצה A). במטריצה B - בהינתן העמודה z את עוברים על כל ה a שלו. למעשה, אחרי שהמנו את התאים cache אפלו לא ניצלנו אותם, עברנו לתא הבא וגם הוא יהיה miss (כי האיבר הבא שנבקש כבר יהיה מעבר ל-block size miss rate). אם כל איבר שאנו בוחרים הוא miss, המה ייה !!

אז כמה יש בסך הכל ?

Miss rate for inner loop iterations:

A	B	C
0.25	1.0	0.0

בשיטת הציג שתקראת k=j
ושה"כ missrate= 1.25

פתרון יותר טוב ומהיר יותר (למרות שסיבוכיות האלגוריתם לא השנתנה):

-

```
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j]; ← הזga מושג
    }
}
```

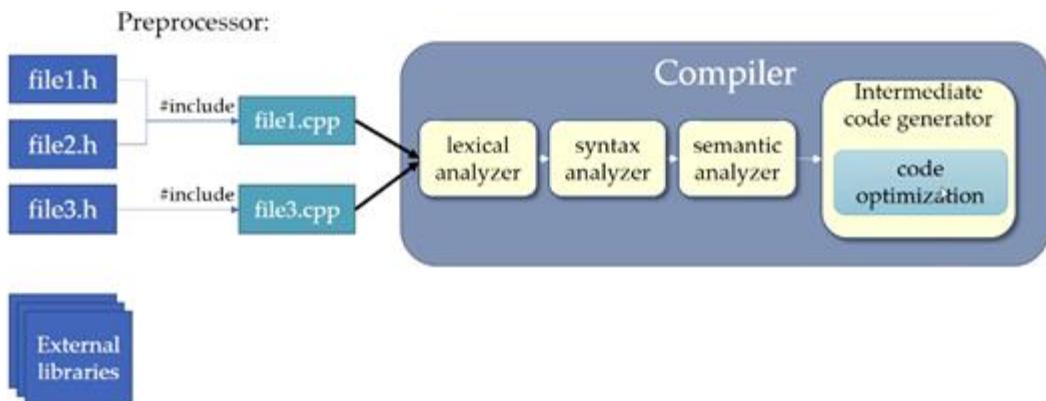
כאן הולאה ק היא החיצונית. אנו מקבעים במשתנה z את a בשורה i בעמודה k. בollowה הפנימית, c (קיבלו שורה ורצמו על העמודות z של Miss rate - 0.25 catchfriendly - 0.25) הוא שווה לעצמו ועוד לחסוך ששם גם אנו רצים על השורה זהה גם catch friendly. במילים אחרות - סך כל miss rate הוא 0.5 וזה הרבה יותר טוב !

למרות שסיבוכיות האלגוריתם לא השנתנה - תפקות האלגוריתם הרבה יותר טוב, וכך שהמערך יהיה גדול יותר מכמה ההשפעה תהיה יותר גדולה.

אופטימיזציות קוד

בפרק זה, נלמד מספר טכניקות אשר ישפרו את הביצועים של הקוד שלנו. שימושו לבי, כי החסם האוטומטי של האלגוריתם (חסם זמן הריצה) לא ישנה, אך הקבוע גדול אכן ישנה וישפי משמעותית על הקוד.

נזכר תחילה כיצד הקומפיאילר עובד. במקרה זה, הקומפיאילר של C++ יש קבצי H וקבצי CPP, וכאשר אנו מפעילים את הקומפיאילר יש תהליך של preprocessor שם הוא עובר על הseknes include -ים למיניהם ולאחר מכן מתחילה הקימפול.



לאחר הקימפול, אנו יוצרים object files ובעזרה שלinkר אנו מחברים את סיפורי הקוד החיצוניים ומקבלים את ה file executable שלנו. עכשו נשאלת השאלה, כיצד ניתן לאפשר לקומפיאילר לעשות את האופטימיזציות הקוד הללו. נסתכל על הקוד הבא:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[n*i + j] = b[j];
```

נכש לחשב מה מיותר פה. בודאי שהז'ה כי הוא קורה בכל איטרציה, נתן אותה תוצאה והוא לא תלוי בז. אך היה עדיף לנו להוציא אותו החוצה מהולולה הפנימית. הקומפיאילר היה עשה זאת בעצמו בצורה פשוטה:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[n*i + j] = b[j];
```



```
for (i = 0; i < n; i++) {
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```

הוא יצר משתנה שהחזיק את התוצאה הקבועה והשתמש בו ובכך נמנע מחישוב מיותר שוכן ושוב.

נסתכל על דוגמא נוספת:

```
void twiddle1(int *xp, *yp) {
    *xp += *yp;
    *xp += *yp;
}
```

ברור שכן יכולים לחת את התוכן של `y` פעמי אחת, להכפיל אותו בשתיים ואז לצרף ל`x`. פה דוקא הקומפיילר לא יփוך את זה לקוד שהיינו רוצים, שהוא זה:

```
void twiddle1(int *xp, *yp) {
    *xp += *yp;
    *xp += *yp;
}
```

→

```
twiddle2(int *xp, *yp) {
    *xp += 2 * (*yp);
}
```

מדוע? התשובה היא שהמצביים יכולים להצביע על אותו המשתנה. אך הקומפיילר לא עושה אופטימיזציות בצורה שיצרת memory aliasing - אם הוא לא יכול לקבוע האם שני ממצביים מצביעים באותו דבר, הוא מניח שהם כן, ובכך הוא מונע אופטימיזציה.

דוגמה נוספת:

```
int f();  
  
int func1() {
    return f() + f() + f() + f();
}
```

בקוד הזה היינו יכולים לקבל את ה`f` ממחזירה ולהכפיל אותו ב4. גם פה הקומפיילר לא יעשה את זה בשבילנו. מדוע? התשובה היא כי יכול להיות `F` מושפעת או משפיעה על משתנים גלובליים. אפילו אם `F` מדפסה משהו למסך, בלי חישוב, אם היינו עושים 4^*F היינו רואים הדפסה למסך פעם אחת.

כל פונקציה יש חשש שיש לה side effect וליקן הוא לא נכנס לתוך הפונקציה. אנו צריכים לכתוב קוד בצורה שתאפשר לקומפיילר לעשות אופטימיזציות.

code motion + reduction in strength

נלמד על אופטימיזציות שלא תלויות באך מוכנה.

code motion

אנו רוצים להזיז קטע קוד מלולאה פנימית ולהוציא אותו החוצה. כמו הדוגמא שראינו קודם.

נראה דוגמה נוספת:

```
void lower(char *s) {  
    int i;  
    for (i = 0; i < strlen(s); i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

בדוגמה הזאת קל לראות שstrlen>Dורש אופטימיזציה. מכיוון שבכל איטרציה היא רצה על מחרוזת, זה א' (o) פועלות. בדוגמה זאת, (במקרה בלבד!!) האופטימיזציה גם תשפר את החסם האסימפטוטי של הפונקציה.

למה הקומpileר לא עושה זאת בעצמו? כי שארנו, כיוון שהוא פונקציה!

```
void lower(char *s){  
    int i;  
    for (i = 0; i < strlen(s); i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

```
void lower2(char *s){  
    int i;  
    int len = strlen(s);  
    for (i = 0; i < len; i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

כאן הקוד אחרי אופטימיזציה.

reduction in strength

אנו רוצים להשתמש בפעולות קלות יותר על המעבד. למשל, חיבור הרבה יותר קל מלהשנות כפל. נחזור לדוגמא שראינו קודם:

- Recognize sequence of products

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        a[n*i + j] = b[j];
```



```
int ni = 0;  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
    ni += n; // instead of ni=n*i  
}
```

16

השתמשנו בחיבור במקום בכפל זהותי פעולה שקלה יותר על המעבד.

registers reuse + combining sub-expressions

טכניתה שלישית היא לעשות **reuse** לאוגרים. כדי כך בכל ליבת יש סט מסוים ומוגבל של אוגרים. (חולקם בשימוש של המחשב עצמו וחולקם בשימוש שלנו). מי שקובע איזה משתנה נמצא באוגר מסוים זהה הקומפיילר.

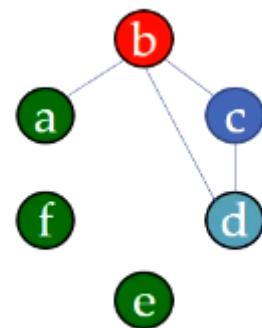
אנו יכולים לעזור לקומפיילר לעשות שימוש חוזר באוגרים הללו. נסתכל על קטע הקוד הפשוט הבא:

```
int a = c + d;
int e = a + b;
int f = e - 1;
```

נשאלת השאלה כמה אוגרים אנו צריכים כאן? התשובה היא שאנו יכולים להסתפק בשלושה אוגרים בלבד. הבעיה של שימוש באוגרים היא בעיה מאוד קשה. לכן הקומפיילר משתמש בכלל אצבע על מנת לדעת מתי להשתמש ומתי הוא יכול "לחסוך" באוגרים, נציגים זאת. בשלב הראשון אנו צריכים שלושה אוגרים d, b, c , שהם הפרמטרים של הפונקציה שלנו. בשלב הבא, אנו צריכים את a להמשך, ואת e , ואנו לא צריכים את c ו- d . בשלב לאחר מכן נצטרך את e ובשלב לאחר מכן את f .

```
// {b,c,d}
int a = c + d; // {a,b}
int e = a + b; // {e}
int f = e - 1; // {f}
```

זה שקול לגרף הבא:



הצבעים ממוקמים לפי כמות האוגרים הנדרשת לפי כל שלב. אך ניתן לראות, שלדוגמה a היה יכול לעשות reuse לאוגר של c וישנן דוגמאות נוספות. כמה שהפתרון שnitן לנו הוא לא הפתרון האופטימלי ביותר. דברים אלה הקומפיילר עושה בעצמו. ונשאלת השאלה כיצד אנו יכולים לעזור לו? אנו יכולים לטערך משתנים לוקליים ומשתנים זמינים. אם קיבלתי משתנה reference by-value ואני עובד עליו, אין ברירה לעליינו לשמור אותו באוגר שלו. לעומת זאת, אם אני עובד עם משתנים לוקליים יהיה קל לקומפיילר לעשות reuse.

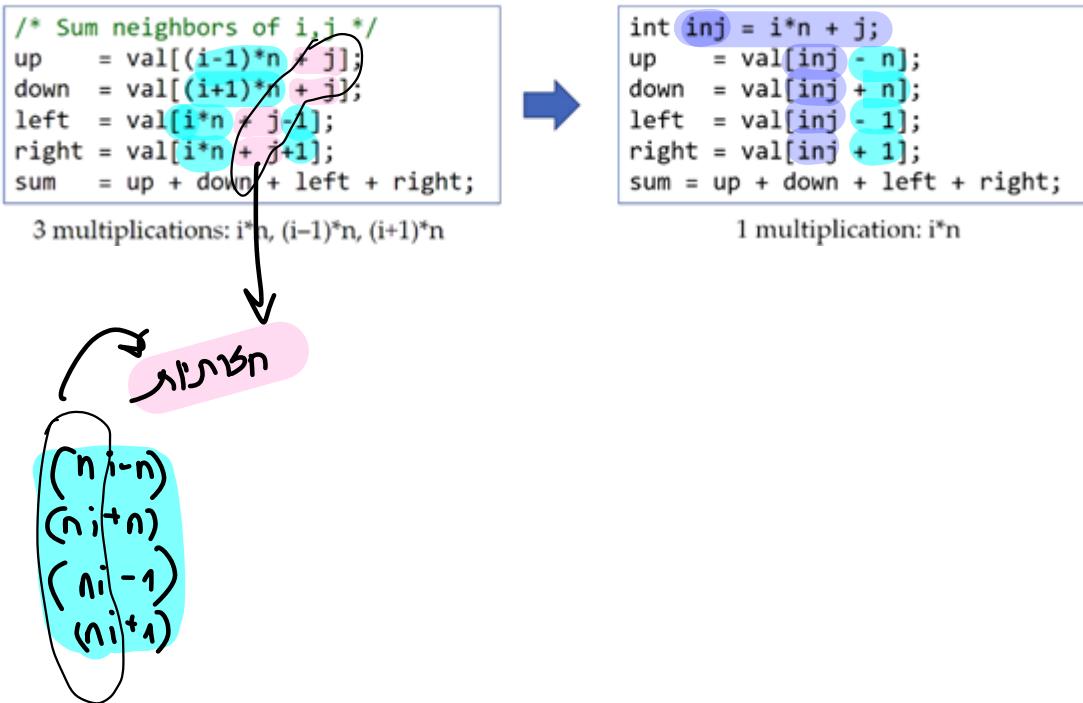
דבר נוסף הוא שnitן להשתמש במילה register להכריח את הקומפיילר לשמשנה מסוים ישב על אוגר מסוים.

SHARE COMMON SUBEXPRESSIONS

אנו רצים לאחד תת ביטויים.

```
int a = c + d;
int e = a + b;
int f = e - 1;      → int f = c + d + b - 1;
```

נראה דוגמא קצת פחות טריויאלית. בדוגמה הזאת אנו רואים שהכפלנו פעמי אחת בלבד בעזרת משתנה.



optimization example

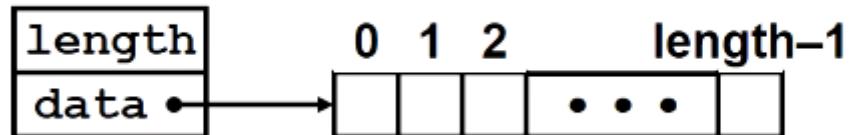
נראה דוגמא לאופטימיזציה קוד.

נראה את מבנה הנתונים שנקרא `vec`.

custom data structure
in C:

```
// data structure for vectors
typedef struct{
    size_t len;
    data_t *data;
} vec;
```

מ. ה.ל.ג



חוץ ממבנה זה יש לו גם את פונקציות העזר שלו.

- `vec_ptr new_vec(int len)`
 - Create vector of specified length
- `int get_vec_element(vec_ptr v, int index, int *dest)`
 - Retrieve vector element, store at `*dest`
 - Return 0 if out of bounds, 1 if successful
- `int *get_vec_start(vec_ptr v)`
 - Return pointer to start of vector data

אנו נרצה להחליף את `data` במשתנים מסוג אחר כמו `int`, `long` וכו'. ונראה כיצד זה משפיע על הביצועים.

פונקציית הניסוי נראה כך:

```

void combine1(vec_ptr v, data_t *dest) {
    long int i;
    *dest = IDENT; → 0/1
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}

```

אנו מודדים זמן אולט נולק

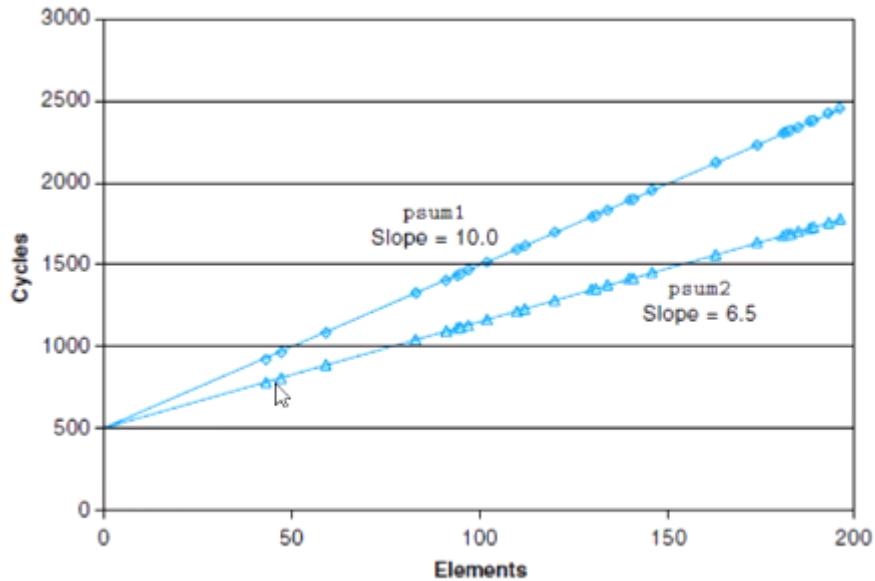
הו הינו גורם אחד מה שפונקציית העזר

כאן היא ביצורה הגרועה ביותר. בהינתן אובייקט מסווג *v*, אז יש לנו את **dest* שם אנו רוצים להשתיל את התוצאה שלו, שהוא יכול להיות או הכפלת כל האיברים או סכום של כל האיברים.

אנו מתחילה מזה ש **dest* שווה ל *IDENT* (זהו כתיבה מקבילה ל 0 או 1 בהתאם למה שנבחר בהמשך - בחיבור הוא יהיה 0, בהכפלת הוא יהיה 1) אנו רוצים על כל השזה, ואנו שותלים באמצעות פונקציית העזר את כל הערך ל *val*. את הקס אנו מחליפים עם חיבור או עם כפל.

את *t_data_t* נחליף כל הזמן ב *float,int,long,double*.

CPE - CYCLE PER ELEMENT. מה שאנחנו מודדים זה.



בגרף יש לנו מעגלי שעון בציר ה *x*, בציר ה *y* יש מספר אלמנטים. ככל שאנו מגדילים את מספר האלמנטים לוקח יותר מעגלי שעון של זמן. כמובן שאנחנו נעדיף את השיפוע הקטן יותר משום שאנו יכולים לעבד יותר אלמנטים בפחות זמן.

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

בטבלה ניתן לראות שארוחי קימפול אנו רואים שיש שיפור משמעותי בביטויים.. כאשרנו רואים הבדל כה מהותי בביטויים, אנו יכולים להסיק כי ניתן לעשות אופטימיזציות על מנת ליעל את הקוד ולהגיע לתוצאה טובה יותר מזו של הקומpileר.

הfonקציה הטובה יותר תקרא combine 4.

במקום לזרע כל פעם על vec_length נעשה זאת פעם אחת ונשמר אותה במשתנה length, וכך לא צריך להריץ את הפונקציה כל פעם. כאן ביצענו code motion.

במקום כל פעם להפעיל את הפונקציה vec_element, ששותלת את הווקטור במקום זה בתוך וע, אנו יכולים לעשות זאת עצמאו.

בנוסף, במקום לעובד על הזמן על ה포ינטר *dest, אנו יכולים לעובד על משתנה lokai ובסוף לשלווה אותו(dest).

הקוד המשופר יראה כך:

```
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

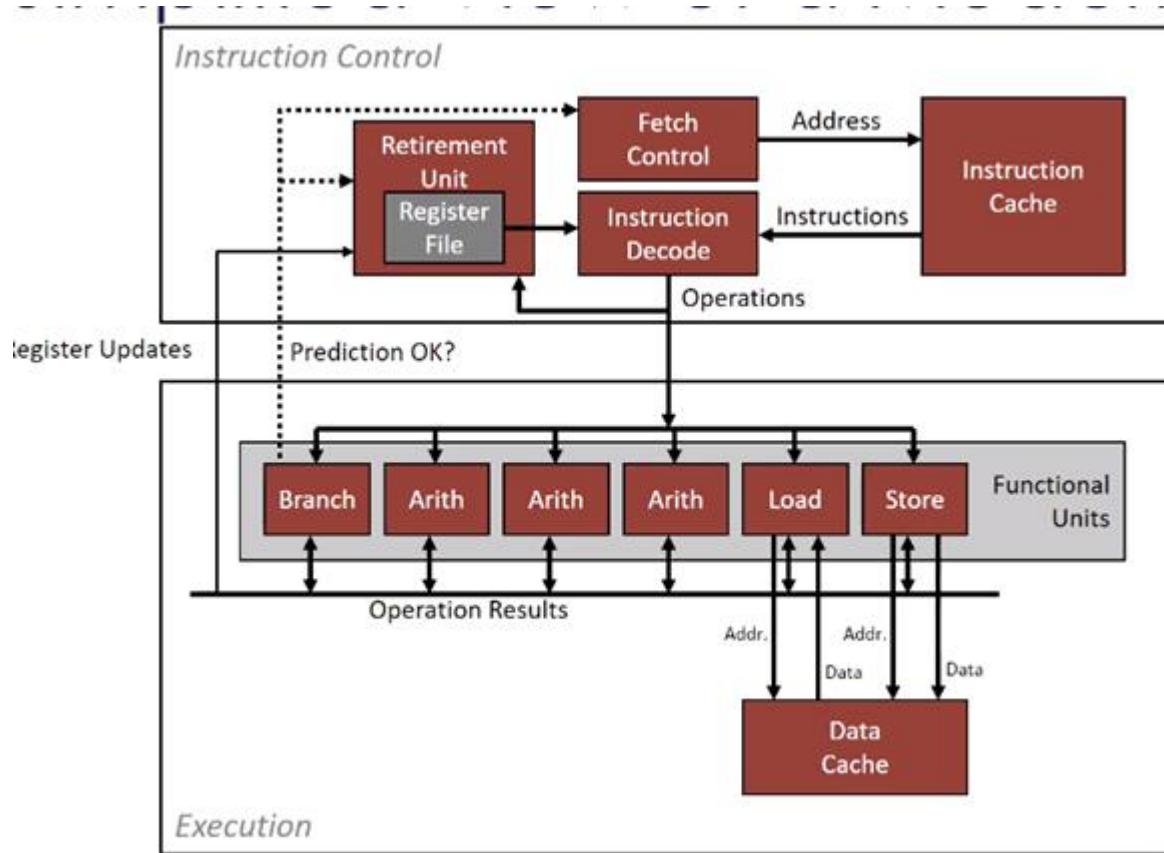
- Move vec_length out of loop
- Avoid bounds check on each cycle
 - get_vec_element()
- Accumulate in temporary

Combine1 unoptimized	22.68	20.02	19.98	20.18
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

השתמשנו כאן בכל שיטות האופטימיזציה שלמדנו עד כה ונitin לראות שיש כאן שיפור אדיר בביטויים.

loop unrolling

בחלק השני של השיעור אנו רוצים להסתכל אופטימיזציות שכל תלויות מכונה. כאשר אנו רוצים לתאר מעבד הוא יהיה מתואר בצורה הבאה:



במעבד יש לנו שני חלקים עיקריים -
יש את ה **instruction control**, ויש את **execution**.
ב**instruction control** אנו מבאים דרך הזיכרון את ההוראות שאנו צריכים לבצע, וגם את ההוראות שבאות
אחריה (מלשון **special locality**). אנו מביצים כאן למשה **FETCH**. בחלק של **decode** אנו מעבירים את
ההוראות **LAU**, שהוא רכיב של **execution** שמורכב מכמה אלמנטים שיכולים לעבוד במקביל.
(לדוגמא, רכיבים אריתמטיים, load שיכל לטען מהcache, או store אשר יודע לשומר ב**ram** וב**cache**. כל
האלמנטים הללו הם למשה ליבה אחת).

Return a+b+c+d

לדוגמא, נניח שההוראה שאנו רוצים לבצע היא

כרגע, אין ניצול מקבילי. נניח שההוראה $b+a$ מתחבצת ביחידת אריתמטית אחת, וכך הלאה. לאחר מכן הוא
יעשה את ה **store** של התוצאה הסופית. כל איטרציה תלולה בתוצאה של האיטרציה הקודמת ולפניהם אין ניצול

Return (a+b)+(c+d)

של מעגלי השעון בדוגמא הזאת. לעומת זאת,

התוצאה כמובן תהיה אותה תוצאה, אך איך זה מושפע על האופן של ה`execution`? הקומpileר מבין שיש פה שתי פעולות בלתי תלויות. כאשר עשינו fetch לכל אחת מהפעולות הללו, ככל הנראה נעשה את הפעולות במקביל בשתי יחידות אריתמטיות שונות. באיטרציה הבאה, נעשה את חיבור שתי התוצאות.

נחזיר לפוקנצייה 4 combine

```
void combine4(vec_ptr v, int *dest) {
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int p = 1;
    for (i = 0; i < length; i++)
        p *= data[i];
    *dest = p;
}
```

אם נסתכל מבחינת זמנים:

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

אנו רואים שבפעולות החיבור של integer עדין לא הגענו לזמן שאנו רוצים.
זה יהיה אפשרי רק אם במקביל.
כאשר אנו עושים את החישוב $[i]data[i]*k$, אז כל פעולה פה תלולה בתוצאה הקודמת.
לעומת זאת, אם נתחיל לשים סוגרים:

```

void unroll2a_combine(vec_ptr v, data_t *dest){
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2)
        x = (x OP d[i]) OP d[i+1];

    /* Finish any remaining elements */
    for (; i < length; i++)
        x = x OP d[i];
    *dest = x;
}

```

(כיוון שאנו קוראים בשתיים אנו מטפלים בשארית בלולאה השנייה שזה זניח)

לטכנית שאנו קוראים מספר איברים ועשים יותר פעולות בתוך הלולאה עצמה, אנו קוראים loop unrolling.
 זהה אופטימיזציה טובה לבדה! אנו פ' 2 פעוטים בזקדים את התנאי, יש לנו פחות jmp-ים וכך.
 אמנם, האם האופן שבו כתבנו את הסוגרים מבטא ומנצל את השימוש בחידות ארכיטקטורת שונות?

```

i = 0; i < limit; i+=2)
    x = (x OP d[i]) OP d[i+1];
}
nish any remaining elements */
; i < length; i++)
    x = x OP d[i];
; = x;

```

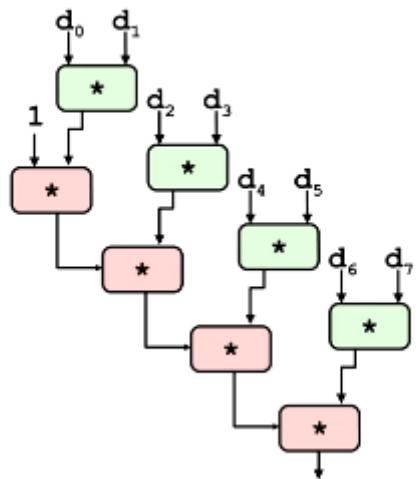
Method	Integer		Double FP		
	Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01	
Unroll 2x1	1.01	3.01	3.01	5.01	
Latency Bound	1.00	3.00	3.00	5.00	

לא! משומש לנו עדין תלויים בתוצאות שונות (קדום הסוגרים ואז החיבור הנוסף).
 $i = 0; i < \text{limit}; i+=2)$
 $x = x \text{ OP } (d[i] \text{ OP } d[i+1]);$

נשנה את מיקום הסוגרים בצורה הבאה:
 בעצם החישובים יכולים להתבצע במקביל. תחילת מבחינה תוצאה בזמןני:

Method	Integer		Double FP		
	Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01	
Unroll 2x1	1.01	3.01	3.01	5.01	
Unroll 2x1a	1.01	1.51	1.51	2.51	
Latency Bound	1.00	3.00	3.00	5.00	

שברנו את latency bound וnicלנו את המקבילות.
 נסתכל על שורת הקוד, באיטרציה הראשונה אנו עושים $d[0]$ ועוד $d[1]$, כאשר x היה אחד. באיטרציה הבאה, אנו עושים 1 פלאס למה שהחישבנו באיטרציה הקודמת, ובאותה איטרציה אנו גם מחשבים את $d[2]$ וכן הלאה...



בכל איטרציה אנו מנצחים שתי יחידות אריתמטיות ובכך מנצחים את המקבילות.
אם שברנו את ה latency bound האם אנו יכולים להגיע לחסם התפקידה? ל throughput bound?

Method	Integer		Double FP	
	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

אנו רוצים להתקerb ולשפר את התוצאות שלנו לרמה של התפקידה.
נראה את הטכניקה הבאה:

```

void unroll2a_combine(vec_ptr v, data_t *dest) {
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++)
        x0 = x0 OP d[i];
    *dest = x0 OP x1;
}

```

בב' בז' 2-2

לעדי
אל גורי

גם השתמשנו בוגר unrolling loop וכן השתמשנו בשני סופרים. x_0 ו- x_1 הם שני משתנים בלתי תלויים בכר שהם שונים. נשים לב גם שuberemo על משתנים זוגיים זאי זוגיים ובכך שמרנו על הרעיון שהקובד יהיה cache friendly. אם היינו בחורם לדוגמא לעבור חצי מערך בעזרת x_0 ובא x_1 עבר החצי השני, המשתנים לא היו קרובים ולא היינו שומרים על עקרון זה.

Method	Integer		Double FP	
	Add	Mult	Add	Mult
Operation				
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

כבר אנו רואים עוד שיפור בזמן.

נשאלת השאלה בכמה accumulator אנו צריכים להשתמש? (כאן השתמשנו בשניים - x_0 ו- x_1)
ובכמה להקפיץ את הולאה?

אם אנו מכירים את המעבד שלנו וידועים כמה יחידות אРИטמטיות יש לו, אנו יכולים לעשות זאת בצורה אנליטית ולהגיע לתשובה בלבד. במידה יש אפליקציה שרצה על כמה מחשבים, אז ניתן לבדוק איזה מעבד ספציאלי יש לכל מחשב ובכך להתאים את הולאה בעזרת תנאים שונים וכוכי.

דבר ראשון צריך להבין האם היה שווה $+2$ אז היה علينا להשתמש ב-2 משתנים. אם הוא היה שוו $+3$ אז היה علينا להשתמש ב-3 משתנים וכוכי.

כדי הגיעו לתוצאה הטובה ביותר ביותר עליינו לעשות ניסוי ותהייה. לדוגמה, במעבד הבא:

Accumulators	FP *	Unrolling Factor L								
		K	1	2	3	4	6	8	10	12
	1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	5.01	5.01
	2		2.51		2.51			2.51		
	3			1.67						
	4				1.25			1.26		
	6					0.84				0.88
	8						0.63			
	10							0.51		
	12								0.52	

כאשר K הוא מספר ה"משתנים", אנו רואים שלאחר 10 יש עלייה קלה וכאן האופטימיליות נמצאת עבור 10 אסמבליים..

branches

כל מה שראינו עד כה, שאנו עושים fetch לכמה גורמים קדימה ועשויים פועלות שונות במקביל בכמה יחידות אריתמטיות, היה ללא תנאים. אנו צריכים להבין כיצד מתמודדים עם תנאים שונים שמקפיצים אותנו למקום אחר בקוד?

כמובן, יש לנו את ה-**instruction control flow**, אך אם אנחנו ב-**fetch** ועומדים לפני הוראה של "jump" שאומרת לנו לקפוץ למקום אחר בקוד, נשאלת השאלה איך עליינו להמשיך? האם כדאי המשיך לעשותות fetch להוראות שבאותו אחריו jump, או לענות על ההוראות בכתבות שאנו קופצים אליה?

המעבד צריך לבצע "פרדיקציה". יש את הדבר שנקרא **static prediction** - מחליטים שלא מבצעים את הקפיצה. מעבר לכך, ניתן לומר שם קופצים לכתבות קדומות אז אנו כן נבצע את הקפיצה. (מתאים ללולאות מכיוון שאתה הולך לבצע את ההוראה מספר פעמים). אם אנו קופצים לכתבות קדימה, אנו מניחים שהתנאי לא מתקיים. לצורך הענן, אם נכתב אם-אזהר משפט אם אז את הפעמים היותר שכיחות נכתב בתוך ה-**if** ואלה הפעמים הנדרשות יותר נכתב בתוך **else**. נדרש לזכור שאם הפרדיקציה טוענה, המחר מבחןת הזמן יהיה גדול.

```
int sum(int[] a) {
    int resM = 1, resA = 0;
    for (int i = 0; i < 1000; ++i) {
        if (i % 2) resM *= a[i];
        else resA += a[i];
    }
    return resM + resA;
}
```

Prediction might be very wrong



```
int sum(int[] a) {
    int resM = 1, resA = 0;
    for (int i = 1; i < 1000; i += 2)
        resM *= a[i];

    for (int i = 0; i < 1000; i += 2)
        resA += a[i];

    return resM + resA;
}
```

אנו יכולים לפעול בכלל להמנע **branch-bomb**-ים.

לדוגמא, בפונקציה **Sum** יש תנאי מסוים. CAN במקום לבדוק האם המספר זוגי, שינויו את עניין התנאים ופיצלנו לשתי לולאות אשר מטפלות במספרים זוגים ובמספרים אי-זוגיים בהתאם. ולשם נשים לכתב קוד ללא תנאים כמה שניתן.

יש פקודה אסמלבית שנקראת **move conditional**. כאשר אנו לא משתמשים בה, כאשר יש תנאי בקוד, תהיה **jump**. אם אנו נבטל את ה"ביטול" של **move conditional** בкомpileר בזורה הבאה:

```
gcc -Og -S -fno-if-conversion control.c
```

הкомpileר יבצע את שני החישובים שבתוך התנאים, ולבסוף בעזרת ה-**move conditional**, הוא ישמור את התוצאה הרלוונטי באוגר הרלוונטי. ובכך הוא לא ביצע קפיצה.

מתי הקומpileר לא יכול להשתמש ב-**move conditional**? מכיוון שאנו מבצעים גם את ה-**if** וה-**else** אך לא ניתן לבצע את שניהם כאשר יש סתירה. לדוגמה:

Both values get computed

Only makes sense when computations are very simple

Possible risks and side effects:

```
val = p ? *p : 0;  
val = x>0 ? x*=7 : x+=3;
```

אנו לא רוצים לעשות x^* וגם $X+3$ כי ברור שזאת לא הייתה הכוונה של המתכנת.

דוגמא נוספת:

```
// imperative style  
void minmax1(int a[], int b[], int n) {  
    int i;  
    for (i = 0; i < n; i++) {  
        if (a[i] > b[i]) {  
            int t = a[i];  
            a[i] = b[i];  
            b[i] = t;  
        }  
    }  
}
```

```
// more "functional" style  
void minmax2(int a[], int b[], int n) {  
    int i;  
    for (i = 0; i < n; i++) {  
        int min = a[i] < b[i] ? a[i] : b[i];  
        int max = a[i] < b[i] ? b[i] : a[i];  
        a[i] = min;  
        b[i] = max;  
    }  
}
```

נעדיף שימוש בתנאים שבקדם הימני, ובתנאים כמה יותר "פרדקטיביים".

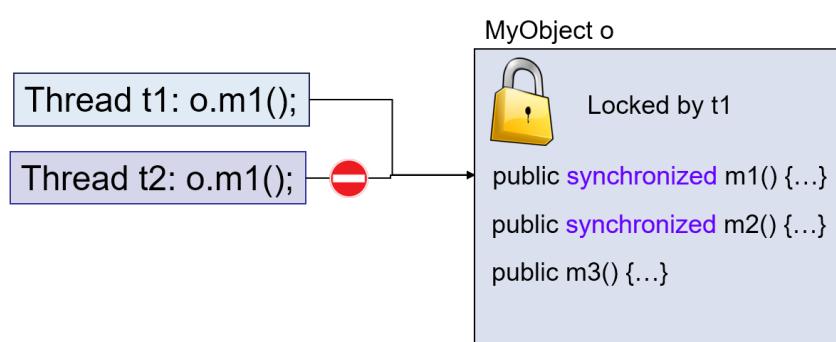
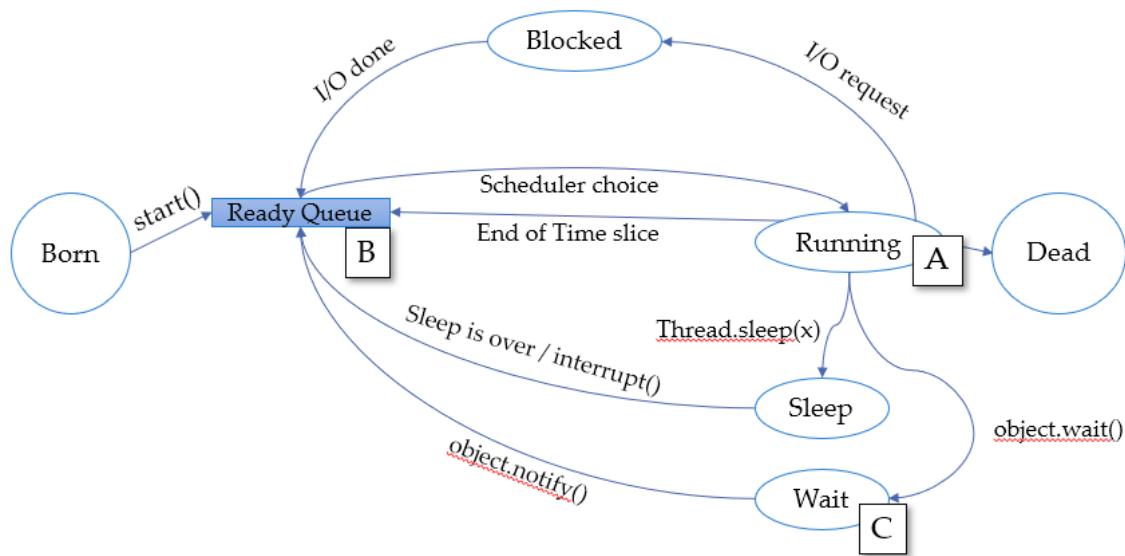
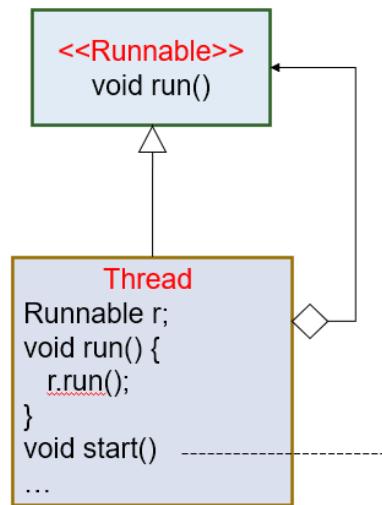
לסיכום השיעור

ראינו שקודם כל כדאי להשתמש בדגלים של compilers כדי לשימוש באופטימיזציות.

צריך לשים לב, שגם רוצים לכתוב קוד catch friendly, להמנע מחסמי אופטימיזציה וכו'.

במקרים שהם צוארי בקבוק אנו רוצים להפעיל את כל האופטימיזציות שלנו. לבסוף, אנו רוצים להמנع

מכמה שיטות breach-bench ותנאים על מנת להמנע מקפיצות.



Basic Active Object

```

class MyActiveModel implements Model {
    Maze maze;
    Solution solution;
    BlockingQueue<Runnable> dispatchQueue
        = new LinkedBlockingQueue<Runnable>();
    public MyActiveModel() {
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    try {
                        // take() blocks, so no busy waiting
                        dispatchQueue.take().run();
                    } catch (InterruptedException e) {}
                }
            }
        }).start();
    }
}

void generateMaze() throws InterruptedException {
    dispatchQueue.put(new Runnable() {
        public void run() {
            maze = MazeGenerator.generateMaze/**/;
        }
    });
}

void solve(Maze m) throws InterruptedException {
    dispatchQueue.put(new Runnable() {
        public void run() {
            solution = searcher.search(m);
        }
    });
}

```

Thread Pool

```

import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
//...
public static void main(String[] args) {
    ExecutorService executor =
        Executors.newFixedThreadPool(2);
    executor.execute (new RunnableTask1 ());
    executor.execute (new RunnableTask2 ());
    executor.execute (new RunnableTask3 ());
}

```

The Solution – Future!

```

public class MyCallable implements Callable<Worker>{
    Worker call() throws Exception{
        // after 10 minutes or so...
        return someWorker;
    }
}

```

Future <V>
V value;
set(V v);
V get();

```

ExecutorService executor = Executors. newFixedThreadPool (2);

Future<Worker> f = executor.submit (new MyCallable ());
// ...
Worker w = f.get(); // waits for the call() to return

```

Scheduling Tasks – with a simple Timer

```
import java.util.Timer;
import java.util.TimerTask;
public class ThreadTest {
    private static class Ping extends TimerTask{
        public void run(){System.out.println("ping");}
    }
    private static class Pong extends TimerTask{
        public void run(){System.out.println("pong");}
    }
    public static void main(String[] args){
        Ping ping=new Ping();
        Pong pong=new Pong();
        Timer t=new Timer();
        t.scheduleAtFixedRate(ping, 0, 1000);
        t.scheduleAtFixedRate(pong, 500, 1000);
    }
}
```

Canceling tasks:

```
int i;
while((i=System.in.read())!=13);
ping.cancel(); // canceled task
pong.cancel(); // t continues...
t.cancel(); // t is canceled
```

```
import java.util.concurrent.atomic.AtomicInteger;
public class Count {
    AtomicInteger count = new AtomicInteger(0);
    public void setCount(int x){count.set(x);}
    public int getCount(){return count.get();}
    public void update(){
        count.incrementAndGet(); // ++count
    }
}
```

```
public void run() {
    boolean w=W.tryLock();
    boolean r=R.tryLock();
    try{
        if(w && r){
            // do the writing...
            // do some reading...
            // do more writing...
        } else{
            // try again later...
        }
    }finally{
        if(w) W.unlock();
        if(r) R.unlock();
    }
}
```

Thread Safe Containers

- java.util.concurrent introduced **Thread Safe** containers,
- that also provides good performance!
 - [ArrayBlockingQueue<E>](#)
 - [ConcurrentHashMap<K,V>](#)
 - [ConcurrentLinkedQueue<E>](#)
 - etc...

```
public class Fib_FJ extends RecursiveTask<Integer>{  
    // with fork-join pool  
    int num;  
    public Fib_FJ(int num) { this.num=num; }  
  
    @Override  
    public Integer compute(){ // a recursive task  
        if(num<=1)  
            return num;  
        Fib_FJ fib1= new Fib_FJ(num-1);  
        fib1.fork();  
        Fib_FJ fib2= new Fib_FJ(num-2);  
        return fib2.compute()+fib1.join();  
    }  
  
    public static void main(String[] args) {  
        Fib_FJ fib=new Fib_FJ(45);  
        ForkJoinPool pool = new ForkJoinPool();  
        System.out.println(pool.invoke(fib));  
    }  
}
```

Using CompletableFuture

```
public String deepThought(){  
    // takes a really really long time...  
    return "42";  
}
```



```
CompletableFuture.supplyAsync( ()->{return deepThought();})  
    .thenApply(answer->Integer.parseInt(answer))  
    .thenApply(x->x*2)  
    .thenAccept(answer->System.out.println("answer: "+answer));
```

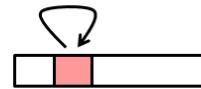
Locality

- **Principle of Locality:** Programs tend to use data and instructions

- with addresses near or equal to those they have used recently
- or same as those they have used recently

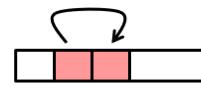
- **Temporal locality:**

- Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**

- Items with nearby addresses tend to be referenced close together in time



Machine-Independent Opt. Summary

Code Motion

- Compilers are good at this for simple loop/array structures
- Don't do well in presence of procedure calls and memory aliasing

Reduction in Strength

- Shift, add instead of multiply or divide
 - compilers are (generally) good at this
 - Exact trade-offs machine-dependent
- Keep data in registers rather than memory
 - compilers are not good at this, since concerned with aliasing

Share Common Subexpressions

- compilers have limited algebraic reasoning capabilities
- Help compiler overcome aliasing, use local variables

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

What changed:

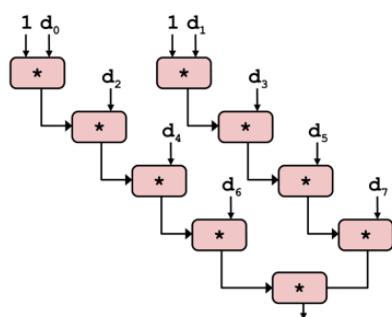
Two independent "streams" of operations

Overall Performance

N elements, D cycles latency/op
Should be $(N/2+1)*D$ cycles:

$$\text{CPE} = D/2$$

CPE matches prediction!



Sometimes branches can be avoided

```
int sum(int[] a) {  
    int resM = 1, resA = 0;  
    for (int i = 0; i < 1000; ++i) {  
        if (i % 2) resM *= a[i];  
        else resA += a[i];  
    }  
    return resM + resA;  
}
```

Prediction might be very wrong



```
int sum(int[] a) {  
    int resM = 1, resA = 0;  
    for (int i = 1; i < 1000; i += 2)  
        resM *= a[i];  
  
    for (int i = 0; i < 1000; i += 2)  
        resA += a[i];  
  
    return resM + resA;  
}
```

Write Code Suitable for Implementation with Conditional Moves

GCC is able to generate conditional moves for code written in a more “functional” style

```
// imperative style  
void minmax1(int a[], int b[], int n) {  
    int i;  
    for (i = 0; i < n; i++) {  
        if (a[i] > b[i]) {  
            int t = a[i];  
            a[i] = b[i];  
            b[i] = t;  
        }  
    }  
}
```

CPE of around 14.50 for random data, and 3.00–4.00 for predictable data
a clear sign of a high misprediction penalty



```
// more “functional” style  
void minmax2(int a[], int b[], int n) {  
    int i;  
    for (i = 0; i < n; i++) {  
        int min = a[i] < b[i] ? a[i] : b[i];  
        int max = a[i] < b[i] ? b[i] : a[i];  
        a[i] = min;  
        b[i] = max;  
    }  
}
```

CPE of around 5.0 regardless of whether the data are arbitrary or predictable

JVM

Optimization Tip 1: call native methods

Optimization Tip 2: use intern() to compare strings

Optimization Tip 3:

- make a smart use of *final*, *private*, and *static* methods
- Try to keep the number of subclasses low
- Favor composition over inheritance

Optimization Tip 4:

- put the more likely branch first
- Sort your data to make it more predictable

Optimization Tip 5: Use local objects in small scopes