CS 131 Programming Languages

October 20, 2009

World's Fastest Computer

Currently the world's speed record is the *Roadrunner*. By looking at the world's fastest computer today, you can get a glimpse at how powerful ordinary computers will be 10-15 years from today. *Roadrunner* has three major component types. First part is a wonderful interconnect – *Voltaire Infiniband*. Second part is the processor – 6,192x *Opteron DC 1.8 GHz* (dual-core). Third part is the cell processor – 12,960x *PowerXCell 8i 3.2 GHz*. Total number of cores therefore is 13,824. There are general-purpose cores (PPE) which run an instruction set that looks pretty much what Apple used to have before they switched to x86, and the special purpose (SPE) floating-point computation cores. There are 12,960 PPE cores and 103,860 SPE cores. We have a hybrid machine because there are two classic ways to do parallel computation on it. One way is the *SIMD* (*Single Instruction Multiple Date*) and another way is *MIMD* (*Multiple Instruction Multiple Date*). In MIMD each thread executes its own instruction while in SIMD each "thread" executes the same instruction simultaneously. In other words, in MIMD each thread has its own instruction pointer (IP) while in SIMD instruction pointer is shared. in *Roadrunner* there are both SIMD (PowerXCell) and MIMD (Opteron). There are 28x 244-port switches, the whole thing resides in 296 racks, weight 250 tons, and consumes a relatively small amount of energy – 2.35 MW. That's a very small number for the given scale. Its R_{max} value (linear programming benchmark) is at 1.105 Gflops (~ 435 million flops/watt). It has 98 TB of RAM (aggregate value; each node has access to a portion of the RAM). The 432 control nodes run Linux while the rest run "special-built operating system" that's a very lightweight system library designed and built specifically for this machine.

If this is the future of computing, how will we write programs for it? Will OCaml, C++, and other languages be able to take advantage of its resources? We're trying to use our old programming technology to deal with new problems, and the match isn't a very good one. We're faced with the problem from the future. We cannot expect the computers in the future to be programmed the way *Roadrunner* is programmed today – lots of programming hours, difficult performance and parallelization problems, mixture of programming languages like assembly and FORTRAN.

Let's look at a similar, simpler problem that's quite popular in the real world: the *SMP (Symmetric Multiprocessing) machine*. In SMP machine, you have multiple CPU's connected by a bus to each other and also to a big piece of RAM. The bus has I/O devices and other things attached to it. CPU's can computer on their own and store results into RAM and load results back from RAM. Furthermore, they can collaborate with each other by leaving each other messages in the RAM – their method of communication. Why won't CPU's just talk to each other directly? Wouldn't that be more efficient? The advantages of communication through RAM are: there's less waiting, one-to-many communication is cheaper for the sender, loads and stores can be used to communicate (which are needed anyway). The disadvantage is less efficiency since two copies of data are made for communication. Most of the computers have processor cores communicate the less efficient way, through RAM. This is also easier to explain and implement in terms of traditional models – it caters better to inertia, being just like the hardware originated in 1950's when most computers had only one CPU. The SMP is called *symmetric* because each CPU can access the RAM at roughly the same speed. The *Roadrunner* is not an SMP machine.

SMP machines have been around since 1980's, when they became relatively popular. *Sun Microsystems* was the first company to put SMP machines into industry. They have first tried SMP with workstations, but it wasn't a money-making business so they went into servers, where SMP proved very useful. Sun is currently trying to figure out what to do next, now that they have their server business going. Sun was the first major company to use Internet in a big way, such as their file servers talking over the internet to their workstations. They wanted to make a ton of money by taking their software and push it out to lots more places. The running joke is that they wanted to "put the internet on a toaster" – they wanted their software to run on lots of small devices connected to the internet. They were mostly worried about the software problems that one would run into when trying to run programs on a variety of devices. Sun's environment was based on their homogeneous SPARC architecture with C and Unix dominating the development. However, their desire to expand to other architectures met a challenge of the variety of instruction sets – MIPS, x86, etc. – therefore a big code porting problem. They nevertheless picked two common architectures and tried to implement their idea with C++. The disadvantage of C++ was that it's too unreliable to write solid programs that would control devices (subscript errors, bad pointers, etc). Another disadvantage of C++ was that the compiled programs were too big (executable code bloat) – shipping them over the network of 1990's would be very slow. In addition, C++ has no garbage collector, and changing a small part of the program requires recompiling an entire application ("recompile the world" problem).

At Xerox Palo Alto research center, a new system called *Smalltalk* was developed. The system generated *bytecodes* that were then interpreted, instructions for the virtual machine designed for developers to make it easier to develop applications. It also had a garbage collector and was very strongly object-oriented. There was only a few problems with Smalltalk – the syntax was "very bizarre" that Sun didn't like, and it was totally "runtime everything" – type checking, etc – which potentially decreased the reliability of applications. Sun took the ideas from Smalltalk, put C++-like syntax on it, and came up with a language called *Oak*. Sun engineers were used to SMP servers and therefore also added *threading* to the feature list of Oak. Oak was soon renamed to *Java* because of the naming conflict.

Engineers at Sun decided to create a demo using their technology, writing a browser named *HotJava* over the weekend. It was more reliable than *Mosaic*, had a smaller executable size, and had a notion of *applet* which is a downloadable application that is served by the web server. Applet contains bytecode that could run on *any* architecture because it ran inside *JVM (Java Virtual Machine)*. HotJava gave popularity to the Java language by showing off what it could do.

Java

There are no uninitialized variables – private variables are initialized by compiler to their default values (0 for int) and public variables must be initialized explicitly.

Java defines the widths of all data types that is the same for every architecture: 16 bits for char, 16 for short int, 32 for int, 64 for long, and 8 for byte. Java also standardized the data storage for integers such that all numbers as two's complement, and all floating points conform to IEEE-754 standard. The standardization resulted in one problem – it's hard to get Java to work on every architecture that isn't similar to SPARC. For example, on POWER architecture there's an instruction `multiply-add` which is equivalent to `(a*b)+c` that returns a close approximation in floating point format. Historically, Java couldn't use that instruction.

Java arrays are different from C arrays is that the only way to allocate arrays in Java is via *new* so it can only be on the heap. There's no such thing such as array that's allocated on the stack or an array that's a global variable. This makes the memory management simpler. In both languages the array size is fixed once it's allocated. Java makes a distinction between *primitive types* (int, boolean) – which are not objects but values, and *referenced types* (pointer to a piece of storage) – types of objects. Arrays in Java are referenced types. The standard way to grow arrays in Java is to allocate another one, twice the size, and copy the data to it. Garbage collector will deallocate the old array.

Java Object Hierarchy

The difference between Java classes and C++ classes is that Java classes use *single inheritance*. Every class has *exactly one* super-class except the class Object, which is the top class. The advantage of single inheritance is that it's simpler, but the disadvantage is that there's no way to create objects that have diverse functionality (by inheriting from multiple classes). Some ideas in Java were brought in to compensate for the lack of multiple inheritance. For example, the keyword *final* describes a method that cannot be overridden (similar to a *leaf* node in a tree), or a class that cannot be subclassed. In the homework assignment, *final* is used for performance because the compiler knows exactly what the code will be (it is so-called final code) and exactly what each function will do. Another keyword is just the opposite of *final – abstract*. An *abstract* method signals that there's no implementation in the method or class, and it *must be* overridden. This keyword is used for defining "placeholder methods" in parent classes that instruct the children classes to provide specific implementation for these methods.

One way of thinking about abstract methods is that it's just the opposite of ordinary inheritance methods. In ordinary inheritance, the method inherits the assed from its parent – the code that defines it. The method can override the inheritance or use it as it sees fit. An abstract inheritance is like inheriting a *debt* from a parent, an obligation to provide an implementation of the method. One can refuse to implement *abstract* method but then the `new()` constructor cannot be called on such class.

Java Interfaces

Interface supplies an API for a class that has no implementation whatsoever – every member is abstract. The second idea of interfaces is that in Java a class can inherit all the interfaces it wants to. If one wants a class that needs to implement multiple interfaces, simply inherit multiple interfaces using the `implements` keyword.

```
interface Drawable {
        Canvas container();    // everything that inherits from this interface must have container() method
}
public  class DrawableRectangle
```

```
        extends Rectangle                    // inherited class
        implements Drawable, Serializable    // inherited interfaces
{};
```

Java Threads

The `Thread` object in Java has a `run()` method that is called when thread is invoked. The default `run()` method is empty and one must extend the Thread object to define an appropriate `run()` method.

```
public  class FireworksThread
        extends Thread {       // we're a subclass of Thread – that's inconvenient
        public void run() {
                // custom implementation
        }
}
```

The above was the original way Java used threads. However, Sun engineers eventually came up with a new way to implement threads. A new interface *Runnable* was created:

```
interface Runnable {
        void run();
}
public  class FireworksThr
        extends Pirotechnics  // we're no longer a sub-class of Thread
        implements Runnable
{
        public void run() {
                // custom implementation
        }
}
FireworksThr fw = new FireworksThr();
Thread th = new Thread(fw);
th.start();                   // allocate virtual CPU and call run() method. there are now two program counters
```