PROFESSOR: Any issues with the project so far? Have you gotten your repositories set up and the compiler works and all this?

So I'm glad to see a lot of people are getting a early start. In fact, if you haven't, a lot of other people in the class are getting an early start so you're behind already. So go there, get the project started, because these performance projects can run for long time. And in fact, last year a lot of people asking, when is my project done? Because a lot of times when you have a project there's a certain set of-- we ask you to implement. Once that's implemented, the project is over.

But when is a performance project done? And the only answer I L when the deadline comes because you can always shave 10% here, you can get a better data structure, you can keep like kind of tweaking for a long time. And a lot of you will do.

In fact, to give you a warning, last year, the performance difference between the best project won, versus the worst one, even the worst one got about 50% better than the code we gave. But the best one got 1,000x better than the worst one because they figured out right data structures, they changed algorithms, they did some precomputation, they did a lot of cool stuff, and so there's a lot to get if you do this project right.

So just don't be happy if you just got 10%, 50%, and say, yeah, I got some performance. Think carefully. There's a lot of room in this project for you to gain, so work hard. I'm not saying this yields 1,000x. I don't know. I haven't exactly seen all the things you can do, but there could be a huge thing you can win.

So think through not just small, low hanging fruit, but there could be interesting things you can do that can have impact.

So today what I am going to do is talk a little bit about basics of performance engineering. Kind of things you guys would probably know, rules to follow.

So to start that if you look at performance engineering, it can fit into a couple of classes. One is you want to get a maximum out of the compiler you have, the processor you have, the system you have, you want them running efficiently.

So if you look at the matrix multiple examples we did this in the first class it's saying, OK, we have a system, we have this very well known algorithm, very long piece of code, and we want to get the best out of that thing, how do you go about doing that? That's mainly the focus there.

Then when you we looked at the bithacking stuff that Professor Leiserson did last time, it's mainly about really changing the algorithms. We basically said, OK, look, if we're thinking about this as bits there might be fundamental different ways to approach this problem. By doing that we can actually get really good performance. OK, we'll be doing more of that.

Today we are going to focus on this two middle criteria. What we are going to do is we are going to stay within the spirit of the program that was given. Even though we are going to change the program a little bit, we are not going to fundamentally change what the program does. So the same computations would be done basically on the same day but what we will do is we will change certain aspects how it's being done to gain some performance advantages.

So the nice thing about this one is you don't have to really understand the algorithm, you don't have to understand lot of what's going on in, what's implementation. This is more-- you can do this as a mechanism sub out.

The unfortunate thing is there's not theory in these type of engineering. There's not something we can prove, here's the optimal you can do. [UNINTELLIGIBLE] nice there.

So normally performance programming basically knows that you need to understand everything that's in modern performance. That means all the way from high level algorithms, every layer in the systems, the compiler, the hardware, the disks, all those things can impact performance. So it's normally when in performance to understand everything that [UNINTELLIGIBLE].

This is very different from a lot of what you guys are used to. It is staying within one layer. You say, OK, this is my layer, I don't care what's on any of it. I don't know care much about-- I'm going to work on that. And the thing that normally comes and bite you is the other layers that you just take for granted. In performance it's basically going through end to end. That's the normal thing.

And normally, if you are an experienced performance engineer, you know when the problems show up. And you can kind of have a basic understanding, here are the kind of places where they have performance problems. And so a good performance this is where experience comes in.

So instead of saying looking at the code and say I have no idea what's going on, you can say, let me think about this, this is where normally something goes wrong. And then you have to have the skill to zoom in and identify the problem fast. And most of performance engineering is basically a good dose of common sense.

So have kind of a good person who can do really good debugging. Is you have kind of a built in algorithm how do you go about doing something. And a lot of times you know your strengths, kind of mistakes you make, you look at those things first, and the same way performance engineering there has to be kind of a back of your head algorithm. And hopefully at the end of this class you're going to develop that.

And in this lecture what I'm trying to do is come up with a set of rules, that kind of patterns that occur regularly, mistakes lot of people make, and that can have a substantial performance impact, or performance gain if you do that. If you have done the design patterns lectures in 005, yes, you do, you will see some similar ideas. What you're doing is kind of say, OK, here's a bunch of patterns in there,

what we can do.

So what I'm doing is I am following some very interesting set of rules that was developed by this guy named John Bentley. He wrote a book called *Programming Pearls*, I guess. It's a very old book, last published in 1990. It talks about a lot of the small, interesting things you can do to get really good performance. It's out of print.

This is probably one reason to probably go to library probably. Libraries have this copy but you can't get it on Amazon or online so that's unfortunate. But what I'm going to do is I'm going to kind of read through it and some of these rules, modified to suit the current program and practices.

So he looked at it and said there are basically two kinds of things you can do, we can modify data, you can modify code. And then went down more into details, if you modify data you can do three things. You can pay one against other, you can pay by space and gain time, you can pay from time and gain space, and there's sometimes, if you're lucky, you have a win-win situation when you can get both space and time advantage.

So let's go through a couple of those things. So if you [UNINTELLIGIBLE] by space for time, there are four things. I'm going to go through all this four data structure augmentation, storing precomputed results, caching, and lazy evaluation.

So what's data structure augmentation? So here what you want to do is you have a data structure that you keep the data. You add some extra information to your data structures. That can mean the common operations run much quicker. OK? And when this is good, because what do you want to do is this adding this information has a clear benefit.

Calculating additional information has to be cheap and easy. So you don't want a situation where we have this information, you've spent all this time calculating this additional information and you're not using it that much, or the use doesn't give you that much advantage. So you have amortize the cost in here.

And then also keeping that information current can't be too difficult, because

sometimes you can add information into a data structure that makes it very hard to make sure that that information is kept up to date, and then you're spending more time and you're [UNINTELLIGIBLE] that.

So a couple of quick examples in here. Something like if you have a single link list, if you are doing a lot of deletion it's a very expensive proposition because you might had to walk through the entire list to delete an element.

But if you are in a doubly linked list, a division operation is basically order one instead of [UNINTELLIGIBLE]. So suddenly you realize, I'm doing a lot of deletions, suddenly I can say I'm adding this additional information to my link list, I'm maintaining point as to both directions, make this one operation run much faster. This is a very classic example of adding additional information.

Another one is what we call reference counting. So assume I have objects somewhere and I want to figure out who points to me. In terms of garbage collection you want to see if anybody points to me. The normal way to do that is look for every other thing saying is anybody pointing to that person, had to walk through every possible object out there to find if anybody's pointing to this object. That's what normal garbage garbage collectors do.

But there's something easier, what you call reference counting. That means if each object keep a count on the amount of pointing coming to itself then to ask that question, is this really true? And you look and say, OK, what's my count? If there's count to zero nobody's pointing to me. But of course every time you point to that object to update the counter, every time you point away or take away an object that points you have to subtract the counters, you had to keep that information additional. But by doing that discussion about asking whether anybody points to you becomes a very trivial question.

So here's a case you add a little bit more information and that makes some questions, some things you want to know trivial and very easy to do, and you have to figure out where the [UNINTELLIGIBLE] is. So here's one interesting thing.

Go a little bit further, one thing you can do is precompute, result and storage. What you are doing is you want some calc computation and instead of every time you want that in recalculating it, say, OK, I'm going to compute them early and I'm going to store it somewhere, and then every time I want then I just do a look up.

So when is this going to be useful? So this is-- the rest of my slides are going to have this kind of form, I'm going to give you a problem, tell you when it's going to be useful, and use some examples.

First of all, the function you are calculating has to be somewhat expensive, otherwise if it is just a simple single operation look up is probably too slow to compile. And the function has to be very heavily used, because if you are calling it many, many times it's worth storing it.

Another very important thing is the argument space has to be small. You might be asking for calling the function millions and millions of time, but every time you call you give a different set of arguments. If the arguments are very large it can be many, many things here, and then it's not easy to store everything. And it might take more time to find all the solutions for all the arguments.

So if the arguments have to be small. And of, course, a couple additional things, results only depends on the arguments. So assume the function results actually look time of day to the computation. OK, that doesn't work, because then every time you call the function, you get different result. That doesn't work that much. And functions shouldn't have side effects. What that means is if you call the function, if it is modifying some global variable, global state.

OK. Then if you keep the call, if you don't call the function that might not happen, so you want to make sure that this function is something can not call and give the results and the state of the system won't change. And another interesting, nice thing to have is function determinants, that means every time you call the function you get the same four arguments, you get the same results.

So this is a long list. Some of the things I will go, so for an example, OK, so instead

of me getting there getting all the things I want, you [INAUDIBLE] when I give a prescription like the thing, what kind of things can I do in this way that might be helpful? Can anybody come up with an interesting example? [UNINTELLIGIBLE] would precompute and save.

AUDIENCE: Anytime you do dynamic programming.

PROFESSOR: Anytime you dynamic program. This is dynamic programming, it's a very good way, later we probably go get a little bit into that. Where you precompute and save results. OK, that's very sophisticated answer. Good. Any other things you can think of? That's almost a meta answer that's in that class of things you can do.

Any problem there that you have encountered that you'd rather precompute and save that would be a big, big win.

AUDIENCE: Operations with large primes that [INAUDIBLE].

PROFESSOR: Operations with?

AUDIENCE: Large primes.

PROFESSOR: Large primes, operations in large primes, yes. If you are doing that, especially if this fits into this argument space small apart. If the large primes are, any large primes then you might not call it multiple times.

So here I want to give you kind of a template that what happens here. So what you want to do is at initialization time I get this function initialized, what I do is I precompute for all arguments what the results are stored at, and then every time you go apply the function I just basically, instead of calling the function, I just look at this table. OK.

So then would this be a bad idea?

AUDIENCE: Because you know this early ahead of time you know that you're going to be calling the function for all of your arguments.

PROFESSOR: So if the argument space is very large or if you don't call the function that many times, or as I said, it might only call for very small number of arguments, [UNINTELLIGIBLE] this might not be a good idea. OK?

So we will go. As we improve, you'll see some other things that actually address that. So here's an interesting way of doing a use case in here. So what I have here is Pascal's Triangle. How many people know Pascal's Triangle? OK, good, so I don't have to explain that.

So here is a simple computation for Pascal's Triangle. So what I'm doing is in order to calculate this value I am calling these two values and adding that. OK?

But the problem is that this will call- so it kind of call exponentially more and more doing that. So you'd say exponential rhythm. However, what I can do in Pascal's Triangle is I can basically compute it at the beginning by storing the previous role, or previous set of values in and then previous role, and every time I'm calling I just basically have to look up the two elements in the previous role.

Trying to get my mouse, OK. Only have to look at the two elements on the previous role. I added up, I will never have to compute beyond that. So, in fact, by just doing this one I have taken exponential algorithm to basically lead a path through the data, basically. This actually become a square, because if you have some depth, I had to do i squared-- become a square, pass through the data. If I'm looking for n, I don't know how to do exponential look up.

Everybody saw this? So this is a nice way, simple way I'm just doing a simple storing this data, however in here I had to calculate, I need to know how much maximum of that, and calculate and I just look up.

And here the [UNINTELLIGIBLE] if you calculate a little bit extra, who cares, I'm OK. And another interesting thing is Fibonnaci. And again, here, if you do keeping the previous value, in here, you're actually changing from exponential algorithm, to in here basically a linear calculation. OK?

So this is almost an algorithmic change here. I kind of lied when I said it's just a data

structure change. But just a simple data structure change made an algorithm difference.

So the other interesting thing you can do is caching. So what that means is if you have a heavily used function you can keep some number of previous basically results from calling that function. This has a lot of caveats so let's go through this carefully. Again the function has to be expensive, otherwise it's not worth doing all that work. Functions have to be heavily used, otherwise you're calculating and if you don't reuse it it's not worth doing all the worth. Here the arguments space can be large, OK, because you are not calculating everything, you can have a large argument space, and that's OK.

And then there's another thing called temporal locality. Anybody knows what temporal locality means? OK what.

AUDIENCE: The same arguments are called [UNINTELLIGIBLE].

PROFESSOR: Yeah, so a lot of times we realize that if you go into the caching and stuff like that. We really get into that in hardware when you talk about hardware lecture. So what happens is yes, we have observed that if I have a set of arguments and if I calculate most of the times I will have the same arguments very quickly time-wise. So if I calculate something, then for a short time I might reuse those arguments again and again, so that's called temporal locality.

So caching works when there's a very good temporal locality. And then there's two other things we want to do because we won't have hash function that means because a lot of arguments from that we only get one value to look up where I store my cache, have to have a hash function, but to calculate it by argument. Sometimes if the arguments are too complicated you might not be able to calculate a good hash function and the hash function has to be good. What do you mean by a good hash function?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, no [UNINTELLIGIBLE]. That means they tend to be a good distribution. So

given most of the arguments it shouldn't be ending up as the same value again, so you should have good distribution when you call with different arguments. And the final results only depend on the arguments that's again otherwise you have a problem and the function [UNINTELLIGIBLE] if your [UNINTELLIGIBLE] this important. And there's another thing called coherence.

So what that means is I keep some precalculated values in here. And sometime this if results are only dependent on the arguments you don't have that much of an issue. But sometimes you might in the situation that the results are dependent on the arguments and some other things like the time of the day or whatever. And if you want full coherence, what you want to make sure is-- many of you know that the results might be varied. I need to understand that, I need to invalidate all the cache.

And suddenly the data or the user changed so that thing I memorized only works for User A, now went to User B, the results are wrong and I had to get rid of all the results. Or there are some cases where you could say, yeah, little bit of things I can tolerate, like for example, a really far fetched example is something like if you're looking at a web page on a search. OK, you don't have a good guarantee that all the web pages in the world are in that search. If you're a little bit late getting a new web page, then you're OK.

So you can do tolerate sometimes, a lot of times, if you can tolerate these kind of things you can have huge performance impact. You don't have to be exact. So some of these projects we look we look at these kind of ideas, of tolerate a little bit of a slack and getting a good performance out of that.

So here's the kind of code, I put a lot of code in my slides because not I won't explain everything but since these online if you want to figure out what the right template is you can look at it. I have no idea why this is moving on me. OK, OK, here we go.

So what you first have to say is every time you cache a value you have to know, I had to know old arguments and results, I had to store that, and then I'm keeping this many precomputed values in here. And so every time I call the function with

arguments first I have to see I had to get the get the has with these arguments because that might be the place, this buffer disk just might be the place I have stored a previously computed [UNINTELLIGIBLE]. Then you say, OK look, is my [UNINTELLIGIBLE] stored, has the same argument.

If that means I already have that precomputed, at that point I just return it, I have no issue in there. Otherwise you're to call the function because I haven't stored it, and then I will go about storing the value because I calculated I need a storage because the next time I call these arguments I want to get the same results, and then I return the result. OK. Is this kind of clear, how this caching works?

OK, what's a good place for caching versus precomputing? Any idea what can make a good scenario where actually caching can have a big impact?

You have heard a lot of things that have the word cache associated with it, and you can say, OK, what that means.

AUDIENCE: Web browsing.

PROFESSOR: Web browsing OK, you web browse actually that's a cache, because there's a good probability that if you look at a page and you might be refreshing that page or you might be looking at that page, so a lot of times what you do is you keep a copy locally, and the next time you ask for that and if you have a local copy you can give that local copy back.

And then of course there's some issues about how to make the local companies stay, the coherence issues and stuff like that it can get complicated but that's a situation where actually you can use caching.

So finally, in this category, we are going to talk about lazy evaluation. So lazy evaluation means basically a lot of times you might ask for value but you might not really need it at that point.

Just keep the computation to the side without doing it and then do the computations only when you really, really need the result.

So basically this is useful, a lot of times you might try to compute a lot of values but only a certain of that actually is used. And at that point you just basically further that calculation and only use it when it's actually done. The nice part is when can be done in a function call, and also results can be calculated kind of incrementally. And also, the arguments or what's needed to calculate the results can be nicely packaged up so it can be recalculated later.

So here what you do here is in the precomputing is something like precomputing but you don't precompute early, at the beginning nothing is precomputed. When you only ask to apply a function then if there's no precomputed value you'll recompute, otherwise you just return the value.

So if you compare this one Pascal's Triangle beforehand I just computed, there's a bunch of values in here and kept it. OK assuming OK up to PDMAX I will have the results in there. But you might not do that. PDMAX might be very large and you might be looking only for small Pascal Triangle.

So instead of doing that, what you can do is you can do this way, you can keep the array for results, assume everything is normally in [UNINTELLIGIBLE] global array. and then what happens is you look at the results if the results is greater than zero we are what you're looking at, that means, ah-ha. I have calculated it. I have a new value, I just iterated it. I'm happy. Otherwise I will call Pascal by recalculating values because I don't have the current results. And once I get it I will store it. Once I get the value I will basically, I will calculate the Pascal, I will store it and I will return the value.

So what that means is this value of the Pascal yx will be only calculated once fully, and afterward every other time you lose that you actually use the previous one. This works very nice, this works both very nice about precomputation but you don't pay that extra overhead. So you are doing the minimum amount of computation. You can never exceed the computation of the normal computation.

So if you only ask for Pascal very small one, that's exactly the work we

[UNINTELLIGIBLE] do. But it will also really use this kind of exponential blow up that happens in the normal computation. So this is kind of the you can have the cake and eat it too in this work. OK good.

So now let's switch from space for time, for time for space. OK? So here, one thing you can do is if you have a huge amount of data what you can do is you can store the data in some kind of reduced form and then what you can do is then you can get a big compression of data basically and then as you need the data you can do some more computation to get the data you want. So what you're doing is you are really reducing the space you need but now you are paying by time to get access to that data.

So this is why [UNINTELLIGIBLE] storage is at a premium. If you look at systems before 1980s everybody had to do it because storage was a premium and all the memory and disk space was so complicated everything got compacted in there. Now, most your laptop and desktops, you don't care, you have enough space.

However, if you got things like embedded devices and things like that, this still becomes a issue. Or if you have a very large data set it becomes an issue.

And what you can do is by compressing, drastically reduce the data size. And you can do it in different ways, sometimes you can do it in batch. That means you keep the data compressed, expand it all, and process, hopefully you have room for that. Or sometimes if you don't have room you do it by streams, that means you're looking at a small amount of data expanded, work at it and then either throw it or again compress it and put it back in.

And hardest thing is if you actually do random access. That means you'll randomly go to some data, expand and understand that sometimes there's some complications.

So there are a bunch of packing methods and some simplest things are use a small data size. So right now you're dealing with 64 bit words, you can say wait a minute, why does this data only need 8 bits? Why do you keep storing everything 64-bit if

my data maximum can be only 0 to 256. I can use 8 bits. Or I can look at that and I can use small data size, that's something. And a lot of people realize a lot of data we are storing are just zeros. There are many different data structures, you have huge data structures and most of it's zeroes.

And eliminating zeroes in many instances can you get a long way in many, many, many data storage. And something like LZ77 eliminates repetition. So a lot of times what you find in there you have repetitive patterns you can eliminate that and then, of course, you can go into a lot more complicated. heavy-weight compression.

So to just give you a feel for what compression is like I found this cute animation so I will show it to you. So here is the input stream, but it has this corrector LA, and then it has some notion about how much a repetition it has, corrector O and 0 3. So what happens if it's just a character? It will get taken in, so these two correctors go.

Now these two forces from the point I point in here go to inwards, and repeat it four times.

What that means is basically in here it will get repeated four times because that 2 4. So that means instead of having these four correctors in there, what it had was it had two things, and then all get repeated three times and then it'd get shifted in here.

So instead of having ten correctors before, we had 2, 3, 4, 5, 6, 7. Seven, out of seven correctors you compress you stored information off ten correctors because of repetition. OK, so this is a very simple scheme that people use to do this LZ77 compression.

More complicated thing you can do a lot of times is build an interpreter. What that means is you have some complex data, instead of storing the data you have a summary of the data stored. OK. And then what you can do is you can do this abstract representation.

I mean you were already tested. Like, for example, you [UNINTELLIGIBLE] exactly doing that. What you do is you start off storing all the beats of machine instructions.

You store it in abstraction called add register names or something like in smd. What that means is you're building a very simple abstraction which that would be more complex sometimes to store than the actual data.

So in here, things like if you look at a java byte cord, that's a nice interpreter. So instead of storing all this complex instructions you interpret, you stored some higher level set of instructions, that byte code, that is much more compressed and that you run you map it into the machine instructions. There's a lot of advantages other than just compressing storage, it's a nice abstraction, you can change it at the highly level easily.

OK, so those are something that you can do, you give up one thing for the other. And if you're lucky you can have the cake and eat it too. Which is you get both space and time at the same time.

So we looked at it in last lecture sometime. This is bithack. Bithacks kind of gave you that because instead of having 64 bit word to represent you got 64 of them into one big stream, if you're only representing zeroes and ones. And by doing that you get a lot more compressed storage. Also now you can operate that entire 64 in parallel.

We can generalize it a little bit, so instead of just doing 64 Boolean, sometimes you can do in a 64 bit word, you can store two 32 bit words, or four 16 bit words, so eight 8 bit words.

OK. In [UNINTELLIGIBLE] is called a Memex SSE extensions. Normally it's called SIMDs, a single instruction multiple data, because what you do to that, all the bits, are the same thing for everybody. So if you're adding you're doing same add for both of the data. And because of that you basically going to get both the compression. Because now for 64 bit you are keeping two thing, or four things, or eight things.

Also, now you're operating. In a single operation you can get the same operation happening to all the data. So you're getting both storage and fast operations in

here.

Of course if you only look at one you have to do a little bit more work, you take it out of that data word, so if you're just looking at [UNINTELLIGIBLE] thing can be expensive.

So when is it viable? You'll do the same operations all the data. OK, if each day you do something different it's not that great and items can be stored in contiguous memory, so when you load you can say I'm getting nice in one word, if I'm getting two 32 bit, so four 16 bit, so I can do that. And you don't end up picking each operations and doing individual things too much. So if you are to pick each individual elements a lot, then you have to actually now pick apart that word and that can be multiple operations and expensive.

So here's a simple example. This, I think, from after last bithacks class, this is a little bit, you probably know, can come up with enough examples here. What I'm doing is I'm doing a Battleship board game.

So you represent the board, and the interesting thing in this board is to know whether that location has something or not, it's empty or full, that's all I need to know.

And so normally if I do this board game I can have two boards in here. But what I'm doing is having this overlap calculations, saying of the two boards, how many things overlap. I can just have each location represented by integer, but then only the representing one value is not good enough.

One thing I can do is I can just put each row into one 64 bit word, and then instead of doing the two loops I can, in one loop, I can just do the ending off each row, and I get the result.

So after the the bithack lecture, this should be kind of trivial. OK, everybody get it? Anybody have questions so far? Are we all on the same page? OK, good.

There's a lot of blank faces so I want to help, I'll start asking more questions or

something. OK, so that's modifying data.

Now we're going to modifying code, where you will change the programs to actually get performance and that can be done in loops, logic rules, procedures, expression, and parallelism rules. When we get to parallelism rules we will actually do a lot more on them in future lectures, but I just want to address some basic stuff now.

Bunch of different loop rules. I will go each of them individually.

So first of all, why loops? Why do you think loops are so important? OK, somebody who hasn't answered the question. Why is loops so important? Why are we like really focused on loops all of the time? Anyone want to answer here, this side? OK.

AUDIENCE: [INAUDIBLE] in terms of maintainability, [INAUDIBLE]?

PROFESSOR: Yeah, so loops give you a nice, simple abstractions it's easier to maintain instead of having multiple computed times. Yes, instead of having same thing repeated millions of times for the same thing, if I write a loop, it's much more concise representation. That's why I think we do a lot of loops, because it makes programs easier.

In fact, assume this word, loops didn't exist, OK? If loops doesn't exist that means you're going at each instruction get executed only one time. If you have a 3 gigahertz machine that requires even if you do one instruction per cycle in 32 bit instruction you need 12 gigabytes of instruction per second, if you only execute one instruction once.

That means if you have 100 gigabyte disk full of a program you go through each instruction the entire 100 gigabytes in 8 seconds. OK. Of course this can never be done, that means you cannot feed 100 gigabytes into the processor in 8 seconds.

That won't work because of the disk access time, caches, and all those. Even if it is doable, what that means is the entire reason your Pentium is running at 3 gigahertz and actually showing like it's running as 3 gigahertz is because you are using instructions many, many times. You have loops that run millions of times. If you

don't have loops you can use whatever the data machine you build in '004 and that probably as fast you could get because of this instruction thing.

So loops are critical, and loops are what makes our programs run fast. Repeated execution of same instruction again and again and again. OK. So in the world, in that sense, if you think about this, unless there are some instruction that's run millions of times, you can never keep this ferocious beast's appetite full because there will always be new instructions.

So because of that looking at loops, if you want to get good performance just looking at the most important loops, the inner loops, and just doing only that we'll probably get you most of way in many, many cases.

And basically I could even say 99 of the program execution time is in 10% of the code, even, more than that in many cases. Because otherwise [UNINTELLIGIBLE] back of the envelope calcuation you can do, you realize how much code you need to feed the beast to keep it active otherwise.

So first loop optimization, loop and invariant code motion. So normally in a loop you run millions of times, and if you do the same thing millions of times and get the same results you are just doing useless computation. And so here the key thing is most of the time the compiler, a good compiler will come help you.

So in this class you're not trying to replace the compiler. We, as humans, we want to the least amount of work. The compilers [UNINTELLIGIBLE] dumb things sitting in there, it can do most of the hard work. The key thing is first trying to get the compiler to do the work. Only do things yourself if the compiler cannot do.

So most of the time compiler, if it can analyze the code, and prove the results are the same in each iteration, it's going to get rid of it, it will do that for you. But there are cases it's not possible because sometimes the computation might be too costly and also there might be cases the computation is too cheap if you hoist it out, you have to keep the results in some [UNINTELLIGIBLE] and if you keep too many things in too many registers, that cost might be too expensive. So there might be

case that you might not do, but most of the time what happens because the compiler can't do anything.

So here's a good example. I'm doing, I don't know why I did that, but square root and take exponential off that. Compiler look at that as functions. I have no idea what the function are most of the time. The compiler can't tell what each function does because the function call. It can do any arbitrary thing as far as the compiler knows. It says, OK, function call, I don't know what happened inside so I'm just giving up and I'm just leaving it, but you, on the other hand, knows that the square root function, exponential function doesn't have any what you call the side effects.

That means those functions doesn't go changing anything else. It just calculates [UNINTELLIGIBLE] the value, you'll know this basically calculate the same value again and again and you can just basically take it out, do it twice, happy.

So lot of times make sure the compiler can do it. If the compiler cannot do then you have to go in and interfere at that point.

So here's another interesting thing the compilers cannot do. A lot of times you're going through this array or data structure looking for some value. OK, once you've found the value, you will say OK, found it, I return.

OK. Normally what that means is when you're going to add it, there's two things. First of all you had to test whether you found the value that you're trying to exit. Second, you have to make sure that you don't overrun on the data structure, you're not overrunning that. you have to make sure you're not at the end of that because that value might not be in that area.

If that's the case, you run to the end of that. So pretend I don't find it. So you do two tests. One thing you can do is you start doing the two test at the end of that array, how about you add the value you are testing for?

So you know when you reach to the end you will always find the value because you added there, so you know it's there already. So by doing that you can reuse the test by just looking for the value because now when it comes to the end, you don't have

to do anymore tests because the value you're looking is already there and you found that, you go out. But you should be able to add that value to the end of that.

So what you do is this is your normal test that's what you're doing is you're, if I can get my mouse, you're going through this array, looking for a value. A value I return otherwise I return minus 1 in here. Now I do two tests. I do this test to make sure that I'm not at the end of the array. I'm [UNINTELLIGIBLE] this one.

So, in fact, [UNINTELLIGIBLE] loop I just can put the two tests next to each other so this is what you're doing. But instead of doing that one thing I can do is put that at the end of the array, after all the data that I care about, I can put the value I'm looking for and then I just only do one test. I check for that value.

So if you find the real value in the area, at that point I'm done. Or I ran out of the array, and after I ran out the real data I just found the value because I, myself, put it there. And so I have to find it, and the minute I find it, if I realize that's the case I say, OK, that's the value I put and I [UNINTELLIGIBLE].

Here I have done something which is not really good programming practice. Can anybody put your 005 hat and say, that's bad programming?

AUDIENCE: You modified [INAUDIBLE].

PROFESSOR: I modified my input array. I know. That's not that great. I mean if I did it really well, if I want to make sure what I should have done is kept the old value of the n and at the end, it back so it looks the pristine thing I got. I mean, if I was really careful I would have done that, and then my invariant that my input array haven't changed would hold nicely. because here I have mucked with my input array.

So even though you are constantly doing performance programming you should still have a good eye for doing good correct programming and keeping nice programming variants around than mucking it all over the place.

So if you wonder putting a new hat in here, you should still have the 005 hat, at least partially.

What else can we do? A lot of times what you find is we have these loops, and loops have a couple of features. First of all, you have to keep checking the loop condition, that's expensive. And if the loop body is very small, the compiler doesn't have that much opportunity to do anything useful in the body, just basically compile as is.

So one thing you want to do is if the loop bodies are too small and the number of iterations is small, you want to unroll the loop.

So in here, instead of doing this calculation again and again, I want to actually do a computation unrolled, and this gives you two things. First of all I get rid of this disk, so I actually [UNINTELLIGIBLE] in here. And second, this computation can be done much faster, the compiler can do this much faster than this one. Do you see why? Can anybody see why this computation would happen much faster than this doing in the loop iteration?

AUDIENCE: Because you don't have to write the final result of the variable until all the numbers [INAUDIBLE]?

PROFESSOR: So that's one thing. Even the final results in the register what happens in here? So that's a good assertion but that's not exactly what's back there.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK, he got 90% of the way there. First of all here I had increment i. And I have it check [UNINTELLIGIBLE] the loop condition. That's another point in here.

AUDIENCE: [INAUDIBLE].

PROFESSOR: The one on right can be run in parallel. In here I am doing sum equals-- some plus A I, so that means I have to calculate [UNINTELLIGIBLE]. Once that is done and we get the results only then I can go add the two, only then I can add day three. So the last [UNINTELLIGIBLE] using, you can show 16 instructions at every cycle.

OK what six things you can do in here? Nothing. Because you're just waiting. You're twiddling your thumb for the first one to finish, to do that. In here, you can say, look,

I don't have to add A1A to it. I can just do these things in parallel.

The first cycle I can add is 0 to A of 1. that's one instruction 2 3 2 4 5 3 6 7. I can add all those things together as a tree. I can do tree addition instead of one after another, that can happen in one go. And I can get a six editions done and then the next instruction I add the [UNINTELLIGIBLE] and at the later stages do not have too much to do but I could get a lot of [UNINTELLIGIBLE]. So the compiler can actually [UNINTELLIGIBLE] all of that.

So that works very nicely if I have small number of iterations. But if you have a large number of iterations, you can still do some stuff. So in here what you can do is it goes to one and you can say, OK, I don't know. I don't know how to unroll fully because I don't know what n is, but I can unroll partially.

So here what I can do is say, look, I have been running it n times, I will unroll my loop four times. OK. So I at least get enough of unrolling here. That way you can choose. That is, you unroll enough that the meshing can get busy. It's not just crawling. And then I haven't blown up my code like crazy in here. And of course you had to clean up and then you have to know how many full iterations.

So how do you choose how much to unroll? What impacts my number of unrolls? I unroll four times. You can get greedy, and say, I'm going to unroll 8 times, 16, 32, 64, 128. Because a lot of time these are not zero-sum some gains. Some stuff impact some other way.

So what should I use normally for my amount of iterations to unroll? Anybody want to take a guess what helps?

AUDIENCE: It's probably the number [UNINTELLIGIBLE].

PROFESSOR: OK, so in one sense you can look and say number of instructions I can execute in one cycle, that's a good thing to keep, unroll enough to keep the process of [UNINTELLIGIBLE] most of the time. But sometimes we can say oh that's hard to figure that one out, I don't know that because I'm at the compiler. I'm at high level. I don't know what happens at low level. I'm going to unroll 1,000 times. So why

shouldn't I unroll a huge amount? What happens if I unroll too much?

AUDIENCE: [INAUDIBLE].

PROFESSOR: You might--

AUDIENCE: You might to be [INAUDIBLE].

PROFESSOR: OK, define it more closely. What do you mean?

AUDIENCE: Perhaps there's some [UNINTELLIGIBLE].

PROFESSOR: That's the one interesting thing. Assume your number of iterations are not always millions. Sometimes you might [UNINTELLIGIBLE] hundred times, or 200 times. So if you unroll 500 times, you would never get to the unroll loop. You end up in this clean up loop. And so clean up loop, of course, is not unrolled and so you're back to your square one. And so a lot of times if you unroll too much and if your number of iterations doesn't fit so you end up in here, that's not good.

Another thing is when you unroll loop here there's a huge amount of instructions now. That instructions might be too much, it might not fit in the cache, it might just overwhelm the memory system. So you have already now created a loop that doesn't fit in the instruction cache and that's not good either.

So that will limit down roll, so don't get too greedy, just sometimes these are the things you kind of play with in the problems you're looking at and some of it is architecture dependent. Some of them are things like too much unroll where the number of iterations not that high is problem dependent.

Another thing you can do there are many places if you have two different very same looking loops we can put them together into one loop and that gives [UNINTELLIGIBLE] all the loop tests and stuff and also the array access. At least done once, so that can give you a benefit in here.

Here's interesting one. Here you have this weird loop that the j loop run from I to J in minus I, I to n minus I. How can I improve this one? Can anybody tell me what

might happen in here? So I have outer loop I goes from 0 to N, inner loop goes from I to N minus I. What happen? Somebody's getting it. OK.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yes, what happens is because what the loop is at the beginning you're going from basically 0 to N, and then the lower bound is going like this, upper bound is going like this. After N over 2, this is N, N over 2, there's no iterations left. OK, you're trading and trading and trading and after it iterates, so you're running outer loop, N over 2 to end with nothing running on the inner loop.

And so if you have this kind of funky loops you can just look at it a little bit careful and say, wait a minute, that's not useful. So therefore I can just get rid of this new situations. And then sort of free trading to nothing.

So that's loops rules. Loops are very important. also you can do a lot of interesting logic rules. So first of all, can anybody shout out what each of these things can be done? Square root of x greater than 0. Square of [UNINTELLIGIBLE] zero, sorry. x [UNINTELLIGIBLE]. Actually, not [UNINTELLIGIBLE] sorry. x not equal to zero. Because x squared of x squared non-zero. OK.

Here's something your compiler probably doesn't know about the transformation. We can do that. How about the next one? How can I get to input the next one? Square root of base climb, looking at the distance between x and Y and AB.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Eliminate the square root. It doesn't help anything you do except do too much computation.

OK, next one. Yeah, I can basically multiply and then do take a one [UNINTELLIGIBLE] final one.

AUDIENCE: [INAUDIBLE].

PROFESSOR: One. OK, and then you should go and find that program who wrote that, but believe

me you'll find these kind of code sitting somewhere because you didn't think through, you just start writing and suddenly you have this kind of nice axiom.

So when do that this is something compiler can do, of for these kind of patterns. That was a stupid mistake but a lot of other things are things that a compiler cannot do but you know more than the compiler does.

OK. Other interesting thing in here is if you're looking something like [UNINTELLIGIBLE] increasing function- OK, let me look at this carefully. OK, this is very similar to what we talked before with the [INAUDIBLE] ending.

So one thing you can do is I can put the cut off at the end and then I don't have to take check both the bounds and basically [UNINTELLIGIBLE] through something. Basically this is the [UNINTELLIGIBLE] check we did done a little bit differently.

OK. And here's another interesting thing. Reordering test. So what can you do here? What do I do here? I'm looking at is, I am looking at two different a circles, see whether they are [UNINTELLIGIBLE] in here. This is expensive test because I had to do something in a square root and check whether it's less than or equal to radius.

So here's what I had to do. How can I make this faster? What's the [UNINTELLIGIBLE] in here? So assume I have this bunch of balls running around in my graphics program. I want to make sure that the balls haven't collided. They are actually moving with no collision with other balls and I'm doing this test. What's a good [UNINTELLIGIBLE] in here?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So he's getting there because most of the time what happens is if you're looking at two balls collision most times these balls are farther apart and if you just found the square around the ball, testing these two squares are colliding it's very cheap, OK? And then if the squares are all happy, then and only then you might want to check if actually all the balls are all happening.

OK and by doing this square test you can do very expensive operations in here. You can basically do some very simple thing, in fact I have even reduced it more by just doing first doing the x side and then if x is [UNINTELLIGIBLE] I don't even have to check for Y. And only if those two checks fail that I realize, look, the square might be now there's a good chance it's actually the color id. And there's a really good chance most of the test, after doing the first two things, will be done. You'll never have to go through this very expensive operations.

So here's something you have a little bit of intuition on what's going on out there in the problem, and use that intuition to look like more work, but in reality really reduce the amount of work you had to do because the test is very expensive.

OK, here what I have done, OK this is a little bit of a contrived example, what I want to do is assume I have basically a 4 bit word in here, OK. And I am checking- this would be 8 I think, so I have 8 bit word in here. OK, I'm checking whether the 8 bit word is a palindrome. So the way I check the word is a palindrome is first of all I set up some bits in here, bit masks, OK? And I have a bit mask for the two ends.

I said, OK, these two are same, then these two same, these two same, these two same, I just kind of go down the bit mask. Did everybody see how this work? And then in these two I kind of shift my bit mask to the side, these two, basically shifting. So I'm checking the most two bits are the same, the next two bits are the same, next two bits are the same, going on [UNINTELLIGIBLE].

OK, so every time I do that I have [UNINTELLIGIBLE] this bit mask do this operation and stuff like that. Assume I am checking whether my [UNINTELLIGIBLE] correctors. If I call this a lot of times what can I do? Kind of out of the box thinking.

So the one thing is if I am only looking for correctors, how many different correctors are there?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, because all the correctors in everything I look up, I have 8 bits, two to date. What's two to date? [UNINTELLIGIBLE].

OK, so you have 256 possibilities in here, OK. So I'm doing this to 256 why don't I precompute and store? So if I just calculate for 256 different possibilities, we see the palindrome or not, I just store a bit for each of them, saying it's a palindrome or not, and then my palindrome test is very simple. I just look up my look up table and say, hey, is this a palindrome, no? Yes? No? Done.

So if I'm doing this computation a lot of times I suddenly realize I can precompute the kind of things we did before. I am very safe, I can do that.

So sometimes you have this programs that has very complicated control structures but when you look at this here you realize [UNINTELLIGIBLE] use multiple times in here. There's S1, if v S2, else S3. S4 again, if v S5. And so there re a lot of conditions going on in here. And something like that you can sometimes look at that and say, like OK, look, because my conditions are simple. I mean in here, I only do one, I just do that one and I will create the path depending on what the Boolean expressions are. And even if you have two or three conditions you can create a couple of parts depending on that.

So I'm replicating some stuff, like S1 is in both, S4 is in both. So I have expanded my code a lot, but now instead of having this spaghetti code that my compiler and my architecture is going to stop all the time because [UNINTELLIGIBLE] prediction and stuff, as Professor Leiserson pointed out, can have issues. I created this nice single branch, lot of code to execute, lot of chances for my compiler to do a lot of optimization. I can say things like that. And compiler can do it most of the time and sometimes you might want to help the compiler.

Procedure rules. So one interesting thing is a lot of times when you have a lot of small procedure calls. A couple of things. OK, let me ask you, what's the problem with having a lot of small procedure calls? You cause a lot of small methods all over the place. What's the problem with small methods?

AUDIENCE: [INAUDIBLE].

PROFESSOR: [UNINTELLIGIBLE] yes, because every time you do a method call you have to

basically go change everything [UNINTELLIGIBLE], put it back in memory, stack, and then go do that. I have a basically calling [UNINTELLIGIBLE] expensive part. OK, so if I'm doing a very small thing your calling context might have a lot more than overhead than what the procedure does, method does. What else? What's a more subtle thing that can happen? So if you're thinking more, I'm a compiler guy so I think like compilers, what would the compiler do at this point?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Compiler [UNINTELLIGIBLE] oops, barrier, I have no idea what's happening in here. I had retrieved this as just bit stop. I will do a bunch of things above that, I will do a bunch things below that. But I can't go through that kind of thing, so the compiler basically treats that as, I can't touch that barrier. And that [UNINTELLIGIBLE] means the amount of stuff that the compiler can do is [UNINTELLIGIBLE].

So one way to do that is basically inline. On a max function, it's pretty simple to inline. Basically you can define a macro that the CE front end will go inline for you. So this is simple. Or you can define inline method in here and put it in the same file, and then we will tell the compiler, look, go and try doing inline listening, and by doing that we are forcing the compiler, or telling it the ability to inline, and they're not only getting rid of the overhead of the method calls, but also you're letting the compiler a lot more opportunity to go and do something interesting.

So here's something you'll find in many streaming type programs. What you do is you run through a loop, reading through a bunch of data, do some processing and writing it back. And then you want to get the data you wrote back, read it again, do some processing and write it back. The problem in that this one is the first loop which stretches through all data.

So by the time I'm done with the first loop I have written all the data and the thing I wrote early is probably all the way down in my cache hierarchy. Because I wrote a huge amount and it doesn't count it in my cache so it goes down, down, down, and the next guy has to read it all the way from bottom up.

So [UNINTELLIGIBLE] means instead of doing that what you can do is run for a small amount of data. So don't run the entire data set. Run a small amount of data, write it into a buffer in the middle that we know sit in the cache, and the disk guys picks up the buffer and finish it up and write it back. So by keeping the data in the buffers what they have done is between these two I actually had some nice locality. I can keep the data up. So as we go into memory systems and stuff like that these kind of patterns come up a lot.

Ah ha. Tail recursion elimination. So here there are many functions that the last thing you do is call itself. OK? This is pretty expensive because what normally happens is when you call a function you set up a calling contest, you do all this context [UNINTELLIGIBLE] put things into the stack and go, but you don't need to keep instead because you're basically done with the previous guy. You're just calling just to go create another stack. In a lot of these cases you can reformulate the problem into nice loop that has no function calls.

So, in here, when you're doing factorial, you can actually formulate the factorial in a way that you don't have to do a function call. So a lot of times if you end up in a situation that the last thing you do in some part is the call itself and if you're not doing anything afterwards, you can always end up in a situation where you don't know how do that, you can actually convert it into [UNINTELLIGIBLE] loop.

And by doing this I got rid of all my calling overhead and stuff that I'm just starting a nice tight loop. OK? So if you have patterns that look like that you can think, OK, look, this is nice, I thought through [UNINTELLIGIBLE]. I got this there. But now I can [UNINTELLIGIBLE] transform it to more iterative way of executing. In many cases you will find a way of doing that.

OK, then there are a bunch of expression rules. So a lot of times you have this computation that if you think carefully you can let the preprocessor do in something like calculate something like that, if I know what [UNINTELLIGIBLE] are, stuff like that, I can just put it in as a constant and voila I don't even have to pay any cost to compute it because by the time it's compiled all the computing is done. And there

are a lot of good cases you can just get rid of work by giving it to the preprocessor.

I'm just calling sin(a) and sin(a) twice in here, and the sin(a) squared I just basically, the compiler probably won't know that the sine every time you give it the same number, you give it the same same input, it's a function, I think the compiler doesn't know that, and this is something you probably have to do for the compiler. Especially when you know the function has no side effects.

This is a little bit more funky, so assume I do two similar computations, you might want to do them together as a single computation that gives the compiler more opportunity, but that means you relay the function like a sine, cosine, s of 1 function. It's a little bit of funky thing that in this list of interesting things to do. But I put it there because there might be cases you might find that's interesting, like if you're doing taking min and max, it might be calling min and max separated. You might be able to call it together.

So the last set of rules is parallelism rules. Of course, we are going to visit parallelism like crazy in this class, but today we will just go through some very high level stuff to give you a little bit of feel for what can be done.

So I have this rule, this program. So what's wrong with this program? Why would this program run very slow? Can anybody tell me? Why can't, in this program, I can't do the full utilization of my machine. The problem in this program is every iteration until this condition is calculated I can't start calculating this. And so every iteration I had of it, this is finished, condition calculated, and [UNINTELLIGIBLE] and new calculate that. And once that's done, you go to the next iteration. We go to next iteration.

So by doing this I can't do too much work. On the other hand, if I can unroll the loop like two times and have two different conditions, now I can do this condition and these conditions, and these two can be done in parallel and when these two get resolved I can do these two operations. So instead of doing xmax, I'm operating two different variables and at the end I can resolve that.

So what I'm doing is I'm calculating two things in there. And they go to four things, whatever, to keep the machine busy. And each of them can be calculated independently. OK, so I can get parallelism and at the end of the day I can do that to help when I'm summing something or whatever. Do you see this? So to get the six instruction issued at a time, something like that will give you more abilities because there's more things that can run in parallel.

So most of the time it will be good to see whether you can eliminate these kind of chains like here xmax is a chain because xmax-- until the previous is calculated I can't calculate the next one, until that so I have this chain that until this is calculated I can't do this, until this is done I can't do the next iteration, so xmax is kind of keeping me very sequential. By doing that I have two things I can calculate.

So other interesting things is if you are doing through this kind of data structure link list. Sometimes if you have I believe the bypass have another point that points couple of things ahead then I can keep multiple of them going on. So I can do sort of waiting for the next one I can basically switch two elements, do that so the next one I can get some parallelism in here so I can do things like that. So to get some basic parallel performance going on. And this is basic kind of a little bit of a data structure augmentation. You add additional data make it run faster. And you want to get all your loops vectorized. That means you want to run things like assembly instruction. If you're using 32 bits, you want to put a couple of them together, run in one SIMD unit.

But I don't want any of you guys to go write assembly to do this. It's going to be a big pain to write all the SIMD assembly. We are not talking about assembly but most the compiler, [UNINTELLIGIBLE] compiler is somewhat good at doing it. So what this means is you're kind of really nudge the compiler, look at bunch of compiler frags to get the compiler to do it.

Sometimes it's-- when we go more into it, we'll figure out how we can do it looking at given the right flags to compiler, sometimes chaining the program a little bit makes it easier for the compiler. So sometimes just wrestling with the compiler, that's much

more easier, believe me, than trying to do it by yourself. By doing that you can get some interesting performance.

And we will talk a lot more about this coarse grain parallelism and we have a bunch of lectures on that. Trying to figure out how to get multicores, each core do something parallel, and get all your [UNINTELLIGIBLE] in the machine working for at a given time then only one guy.

And so here's an interesting situation. So I have this problem here. Can I run this parallel? So what I'm doing is I'm doing these two loops and I'm adding all the elements of an array to total. Is this parallel? Can I run anything or any of these things parallel?

Somebody says, yes, no, whatever, what do you think? Take a stand. OK, what do you think?

AUDIENCE: I think yes.

PROFESSOR: Somebody says yes, good. Yes, takers, no takers. Yes, everybody says yes. [UNINTELLIGIBLE] is parallel. But look at here, what happens is previous total is needed to calculate the next total, so how can I run this parallel?

AUDIENCE: [INAUDIBLE] for loop jumps every two.

PROFESSOR: OK, so she's somewhat onto it, so let me show you what we can do. One thing you can do is instead of calculating one total for each basically column, you can calculate its own total. OK?

So what happens is so for this i's row, j's column. So for each row, so each row gets a different total, a temp total, and so what that means is you can calculate all the columns total separately. You can total all the columns, and then total up those values to get the full total. So totalling up columns can be done parallel because now all of these things can be done parallel because it's totalling up two different, multiple different values. So it's kind of what she said but you do it for each row separately.

And so here's something, you made the program a little bit more complicated. Actually for a sequence it run a little bit slower but at the end of the day you actually get the program parallelizable. So a lot of times when you're parallelizing is to also looking at that algorithm little bit changes to the program that at the beginning might look like you're slowing down the program but ending up getting good parallel performance. So a lot of parallelism looks like this.

So here's the entire list of things we talked about. It's out there so you can go look at them and see if you understand them better.

And I'm leaving you with this example that I'm not going to have time to go through. which is my program of a traveling salesman problem. So what you're doing is you're looking at a way for a tradesman to travel to all of the city's with a minimum cost. Of course the minimum shortest thing is exponential algorithm so you don't do that. There's a good heuristic, that the greedy heuristic every time you go to the city you go to the cheapest city that's not visited and you just jump around and that's seem to be pretty good heuristic.

And so I call this up, and I will go to the end and then just finish it up. Oops, I went too far.

So what I did was I have eight different stages basically, each stage I did some changes and then I got this program to run about five times faster than from where I started. So you will see at each stage what I did. So this is basically out of the menu of things I showed you I found an instance I can do and in this program I got five times faster.

This is more normal than the 300 [UNINTELLIGIBLE] I got [UNINTELLIGIBLE] and that's a real extreme case, but enjoy looking at this code.