Compiler Design

Prof. Y.N.Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

Module No. # 18

Lecture No. # 36

Just-In-time Compilation and Optimizations for .NET CLR

(Refer Slide Time: 00:18)

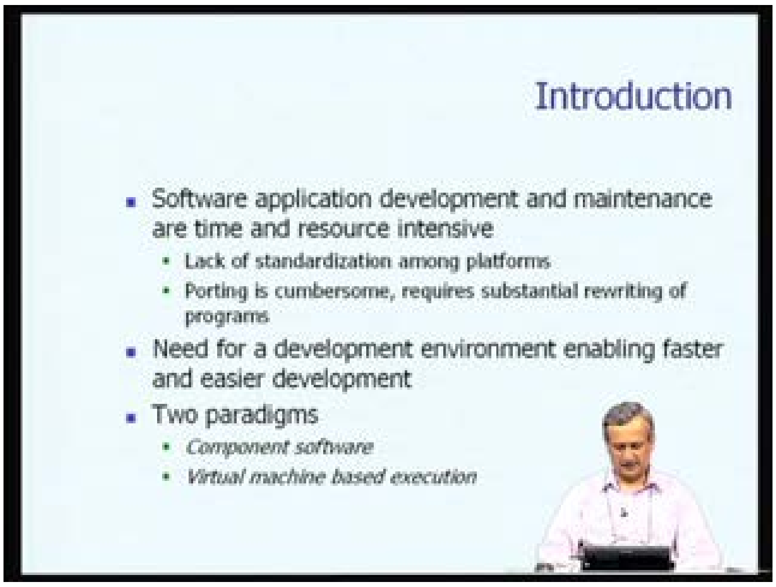
Just-In-Time Compilation and Optimizations for .NET CLR

Y.N. Srikant
Department Of Computer Science & Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Welcome to the lecture on Just-In-Time Compilation and Optimizations for the .NET Common Language Run time.

(Refer Slide Time: 00:23)



Introduction

- Software application development and maintenance are time and resource intensive
  - Lack of standardization among platforms
  - Porting is cumbersome, requires substantial rewriting of programs
- Need for a development environment enabling faster and easier development
- Two paradigms
  - *Component software*
  - *Virtual machine based execution*

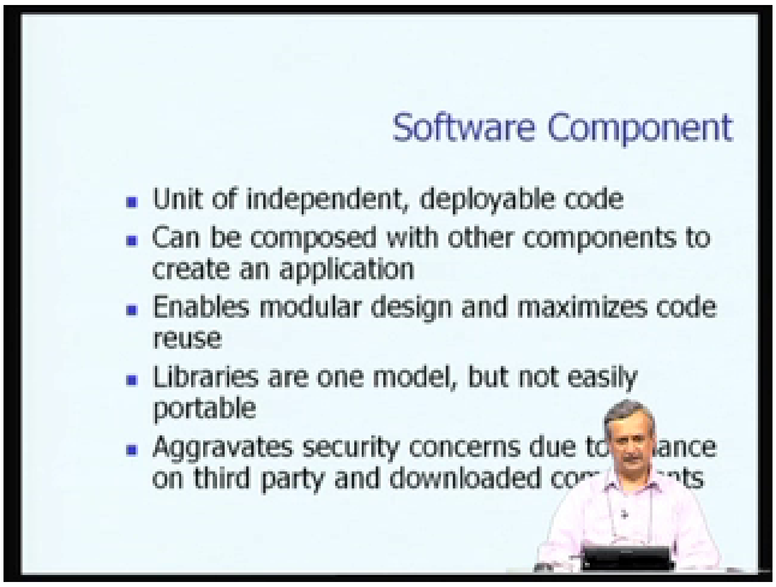
This is really a specialized topic slightly advanced compared to what we have been discussing over the past few weeks. So, here in this lecture, we will learn about a new compilation strategy called Just-In-Time compilation, why is it needed, where it is useful and so on.

Let us consider software application development and maintenance. So, these are really time and resource intensive tasks, and there is a lack of standardization among platforms. So if you develop something, if you have an application development environment for windows it would not run on Linux and vice versa and so on.

And then porting from one platform to another is extremely cumbersome. It requires substantial rewriting of programs because the libraries and so on, everything else is different. So, there is a need for a development environment enabling faster and easier development.

So two paradigms have been proposed in literature and they have also been in use for a very long time: one of them is the component software paradigm and the other is virtual machine based execution paradigm.

(Refer Slide Time: 02:09)



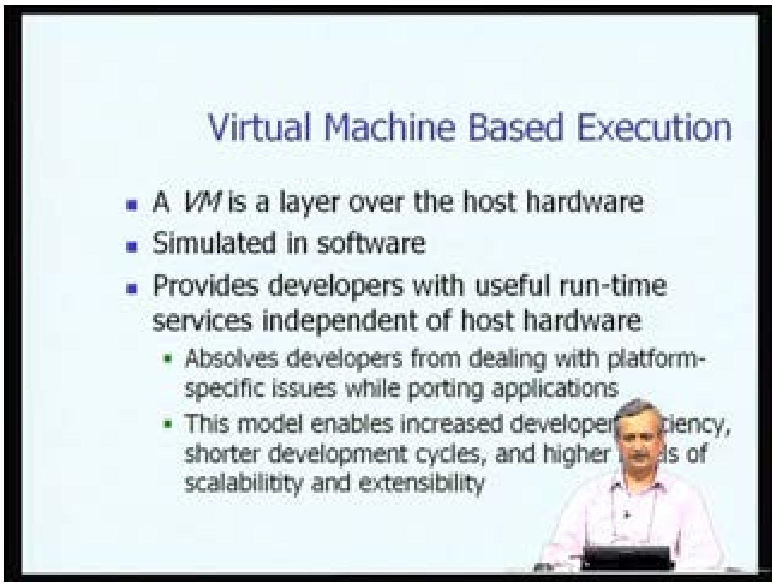
Software Component

- Unit of independent, deployable code
- Can be composed with other components to create an application
- Enables modular design and maximizes code reuse
- Libraries are one model, but not easily portable
- Aggravates security concerns due to ance on third party and downloaded co nts

In the software component paradigm a software component is nothing but a unit of independent deployable code. If you can take one component, compose it with other component - one or more other components and then create an application. Such a methodology enables modular design and maximizes code reuse; these are all well-known. Libraries are one such model; there are lots of rough libraries are available, but again portability is always an issue; you cannot port one library to another environment very easily without rewriting code.

Then, this software component methodology also aggravates security concerns due to reliance on third party and downloaded components because not every component is developed by the programmer. He or she would use third party components or free components downloaded from the internet and so on. So, he or she may not have any idea of how security is handled within that component and therefore there may be some compromises here.

(Refer Slide Time: 03:23)


Virtual Machine Based Execution

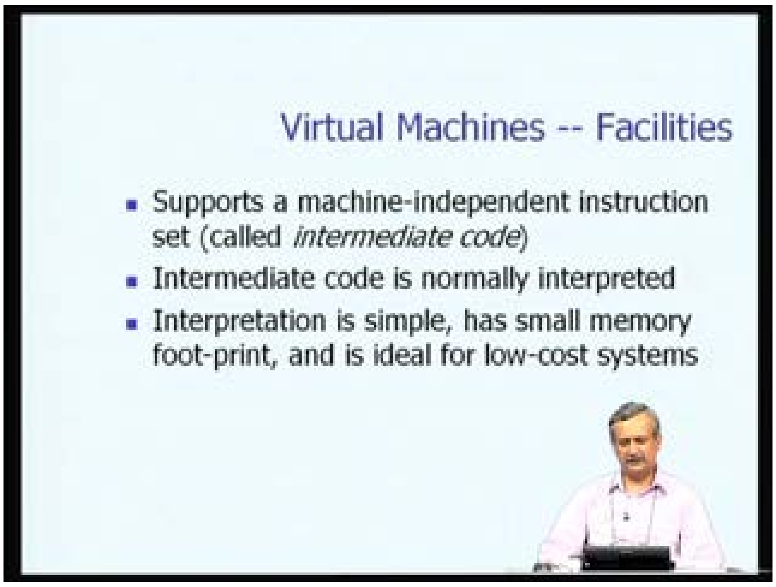
- A *VM* is a layer over the host hardware
- Simulated in software
- Provides developers with useful run-time services independent of host hardware
  - Absolves developers from dealing with platform-specific issues while porting applications
  - This model enables increased developer efficiency, shorter development cycles, and higher levels of scalabilitity and extensibility

What is Virtual Machine Based Execution? A virtual machine is a layer over the host hardware. In other words it covers just the hardware and makes sure that the programmer need not dabble with hardware features, but actually worries only about n a p i. This virtual machine is simulated in hardware; it is more like an extended instruction set - really lots of extensions - extended instruction set which provides a large number of services to the programmer. It provides developers with useful run time services independent of host hardware. So, the advantage is the virtual machine is an abstraction, it can be implemented on many hardware platforms, but the programs which use the virtual machine abstraction will run unchanged on any of these platforms; only the implementation of the virtual machine is going to be different for each platform.

Applications which are larger in number in they will be developed every day whereas, the virtual machine is going to be implemented once on each platform. So, the effort in implementing a virtual machine very well on a particular platform is well spent.

What is the advantage of providing such an extended instruction set or application interface? It absolves the developers from dealing with platform-specific issues while porting applications; they just have to worry about the virtual machine. This model also enables increased developer efficiency because now hardware specific issues which are hard to learn need not be addressed by the developer; it enables shorter development cycles - again, the same reason hardware is not necessarily learnt - the specialties of hardware are not necessarily learnt by the programmer and it also enables higher levels of scalability and extensibility again due to independence of the platform.

(Refer Slide Time: 05:46)



Virtual Machines -- Facilities

- Supports a machine-independent instruction set (called *intermediate code*)
- Intermediate code is normally interpreted
- Interpretation is simple, has small memory foot-print, and is ideal for low-cost systems

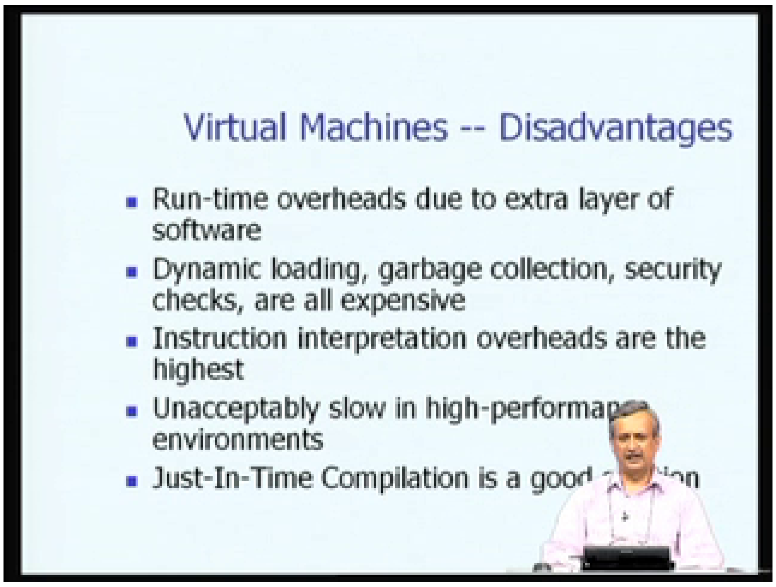
What are the facilities available in virtual machines? Typically, these are only samples; there will be many more such facilities available. For example, it checks security violations by the software components, and components can be dynamically loaded and linked. So, it is not necessary to have every component available at the time of running the program, it could be possibly downloaded from the internet and then used. So, java virtual machine and the applets etcetera, are examples of such dynamically loaded and linked software.

Virtual Machines also provide automatic memory management and garbage collection. Again, java is an example and c sharp in windows is another example. Both of them have automatic memory management. The c sharp has been implemented on many platforms and specifically on windows, it runs under the common language run time and common language interface and so on and so forth. We will see that very soon.

So the java virtual machine is very famous; it is been there for more than 15 years now. Everybody knows that applets which can be downloaded on any architecture and on any environment will run without any difficulty. So, these virtual machines also provide architecture independent interface for exception handling. So, java virtual machine and its implementation is one virtual machine interface that is available to programmers and similarly, in windows we are going to look at the common language system.

Other facilities available in virtual machines - it supports a machine independent instruction set called intermediate code. Specifically, again for java it would be java byte code. Intermediate code is normally interpreted. So, for example, there is an interpreter for the java byte code. This interpreter is very small, interpretation is very simple, so, it is a very small foot print program and this is ideal for low cost systems. So, the problem is it could be slow, interpretation of every java byte code instruction is slow compared to running AC program on a native machine, but it gives you portability etcetera as is available on other for virtual machines in general.

(Refer Slide Time: 08:54)



Virtual Machines -- Disadvantages

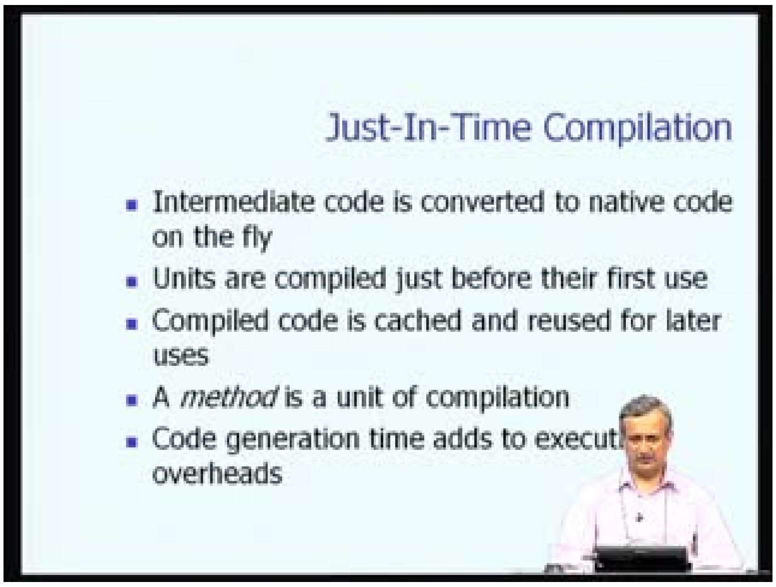
- Run-time overheads due to extra layer of software
- Dynamic loading, garbage collection, security checks, are all expensive
- Instruction interpretation overheads are the highest
- Unacceptably slow in high-performa[illegible] environments
- Just-In-Time Compilation is a good [illegible]

What are the disadvantages of Virtual Machines? So, run time overheads due to extra layer of software, this is very obvious every instruction in the java virtual machine for example, goes through the virtual machine layer of software - it is either interpreted or compiled and executed, but it is not like executing a compiled binary of a c program directly on a machine. So, there is a layer extra and there are run times over rates because of this layer.

Dynamic loading, garbage collection, security checks etcetera are all expensive. So, these are all provided by the virtual machine and there is a price to pay for all the convenience that we get. The question is - how much is this overhead, how much is this expenditure. So, instruction interpretation overheads are the highest. So, java byte code interpretation is very expensive and probably the major factor in the overheads corresponding to virtual machine.

Unacceptably, slow in high performance environments. Let us say, you want to write a piece of scientific software in java and then without a Just-In-Time compiler, you want to run it in interpreter mode. The java byte code which is available will be run in interpreter mode. So, the performance is going to be terrible and similarly, if you have a real time environment and you are thinking of executing java byte code in interpretation mode, you may not be able to achieve the deadlines given to you. So, these are all difficulties is with virtual machines. So, a solution to such problems of virtual machines is Just-In-Time Compilation.

(Refer Slide Time: 11:00)



Just-In-Time Compilation

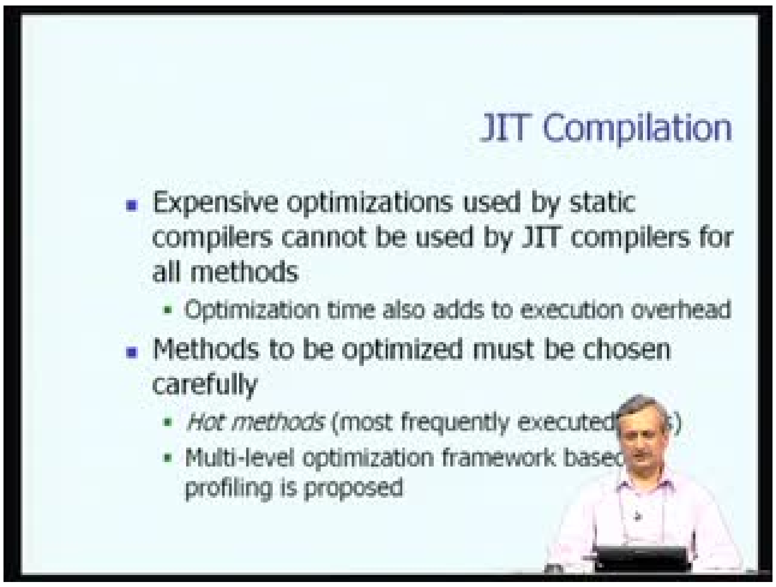
- Intermediate code is converted to native code on the fly
- Units are compiled just before their first use
- Compiled code is cached and reused for later uses
- A *method* is a unit of compilation
- Code generation time adds to execut overheads

What is Just-In-Time Compilation? I have been dropping this word already. Basically, the intermediate code - such as java byte code instead of being interpreted is converted to native code on the fly. So, in other words, the program, which is a unit of java byte code, is going to start in the interpretation mode. Then, by some mechanism we determine that some methods or some units of this java byte code program will be converted to machine code, native machine code. Let us assume that this is possible.

Once, it is converted to native machine code, instead of interpreting the instructions of java byte code for that particular program segment. The native machine code is going to be run. So, that is precisely, the way the Just-In-Time Compilation works. Why is it called Just-In-Time? The reason is simple, we did not convert java byte code into machine code right in the beginning, like we convert c program into machine code. It was converted to java byte code and then while running the java byte code. Whenever, there was a requirement, Just-In-Time, we were converting that java piece of java byte code to machine code that is the reason why it is called Just-In-Time.

Units are compiled just before their first use. Compiled code is cached and reused for later uses. So, this is what I explained. Normally, a method in a class is a unit of compilation, not a small loop or something like that, a complete method. Code generation time adds to execution overheads. If you want to convert to this machine code - this is also a factor in the execution overhead.

(Refer Slide Time: 13:14)



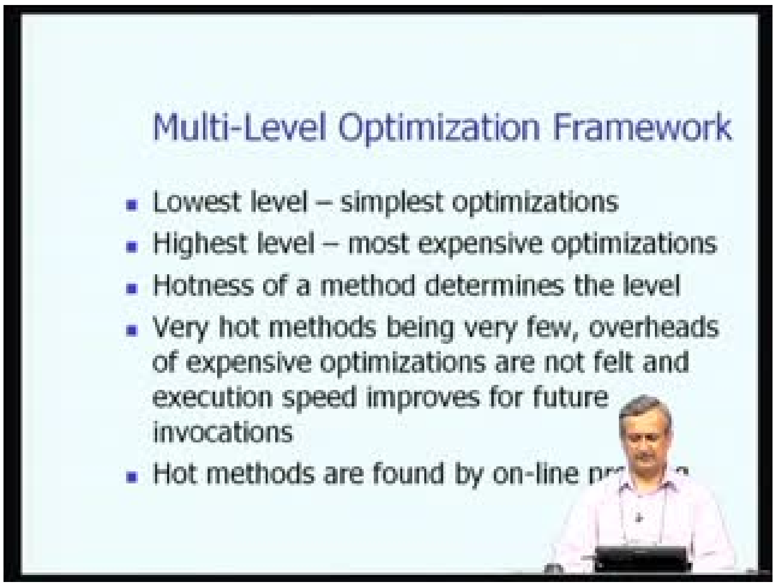
JIT Compilation

- Expensive optimizations used by static compilers cannot be used by JIT compilers for all methods
  - Optimization time also adds to execution overhead
- Methods to be optimized must be chosen carefully
  - *Hot methods* (most frequently executed s)
  - Multi-level optimization framework basec profiling is proposed

Once, the code generation Just-In-Time Compilation itself is a part of the run time or running time, expensive optimizations used by static compilers cannot be used by JIT compilers for all methods. So, this is very clear because, optimization time also adds to the execution overhead. The heavier the optimization the more the time and the slower the program becomes in a Just-In-Time or virtual machine environment.

So methods to be therefore, you cannot really generate code in a Just-In-Time machine, Just-In-Time compiler for all methods, So, you have to find which methods need to be recompiled. Methods to be optimized must be chosen very carefully. Hot methods which are nothing, but most frequently executed methods will be chosen for this purpose. We propose a multilevel optimization framework based on profiling for the Just-In-Time compilation

(Refer Slide Time: 14:26)



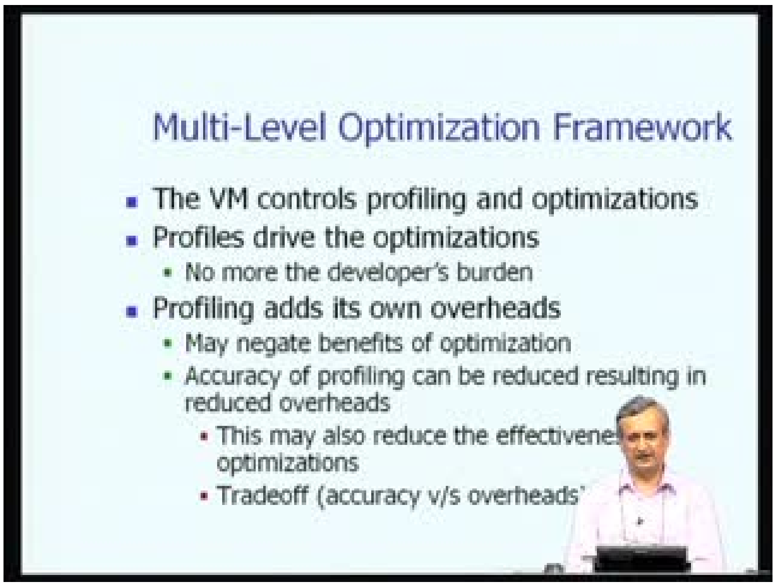
Multi-Level Optimization Framework

- Lowest level – simplest optimizations
- Highest level – most expensive optimizations
- Hotness of a method determines the level
- Very hot methods being very few, overheads of expensive optimizations are not felt and execution speed improves for future invocations
- Hot methods are found by on-line pr

Here is an overview of the multi-level optimization framework. So, there is a lowest level of optimization in which we perform very simple optimizations and then there is a highest level of optimizations - may be 1 2 3 something like that, where we perform the most expensive optimizations. We have 2 levels so, there is only low level and high level in our compiler, but it is in principle possible to have more than 2 levels.

Hotness of a method determines the level at which the optimizations should be performed. So, very hot methods being very few overheads of expensive optimizations are not felt. Execution speed improves for further invocations. So, since very few methods are going to be optimized at level 2 are higher the overheads are very few, very little. The execution speed - they are very hot so, these methods are going to be executed very frequently. So, the execution speed improves for future invocations. How do we find hot methods? Hot methods are found by online profiling methods.

(Refer Slide Time: 15:51)



Multi-Level Optimization Framework

- The VM controls profiling and optimizations
- Profiles drive the optimizations
  - No more the developer's burden
- Profiling adds its own overheads
  - May negate benefits of optimization
  - Accuracy of profiling can be reduced resulting in reduced overheads
    - This may also reduce the effectivene optimizations
    - Tradeoff (accuracy v/s overheads

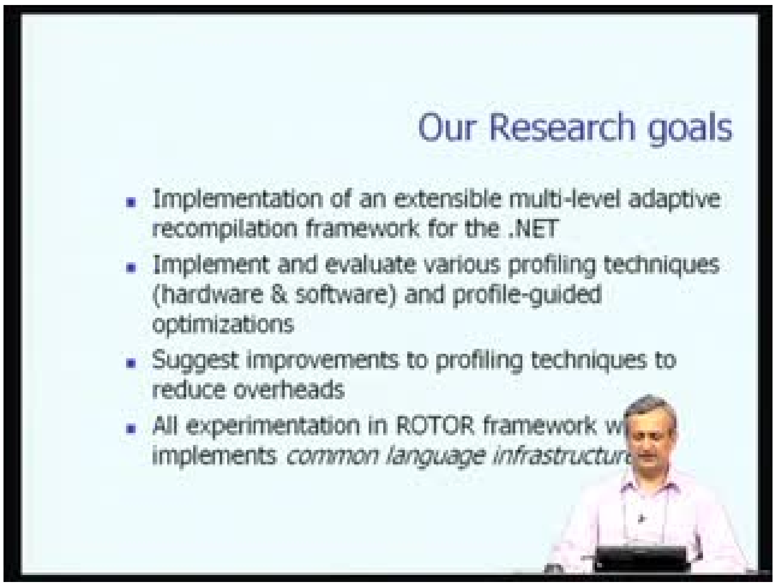
More details of the Multi-Level Optimization Framework: The Virtual Machine controls profiling and also the optimizations, so our frame work is built into the virtual machine itself. So, the profiles actually drive the optimizations. The Virtual Machine controls profiling and optimization process, the profiles that we accumulate drive the optimization process, so no more the developer's burden. In other words, the developer need not say do these optimizations. For example, when you compile GCC - you have to say, which level of optimizations you really want. There are several levels so those switches have to be enabled. Whereas, in the case of this profile driven optimization - the developer does not say anything about which optimizations have to be performed, it is the profiler which says this method is very hot. Therefore, its worthy of being optimized at the highest level so, let us optimizes this. Whereas, for a different method - it may say it is not so hot, but it still a bit hot. Let us do level 1 optimization for this, which is sufficient. For a third method - it may say this is not at all hot, so, let us just generating simple unoptimized code. In fact, in some cases, for the fourth method - it may still say, let us not even generate machine code - let us actually run it in interpretation mode. That may also be possible in a different machine. In our case we do not do that. We always convert it into machine code and then run it.

But profiling adds its own overheads. For example, it may negate the benefits of optimization. The profiling also actually runs along with the program, so, there is a thread which runs as a profiler. It adds extra overheads - it slows down the program to some extent. It may negate the benefits of optimization, so, if the profiling overhead itself is very large then you lose out.

Accuracy of profiling can be reduced resulting in reduced overheads. So, we for example, if you really want to do basic block level profiling and count how many times each basic block is executed. This requires a lot of code instrumentation; is very heavy, so it there is a lot of overhead because of basic block level profiling. Whereas, if you are going to do method level profiling. In other words, you just want to find out - which methods are very hot; which methods are not very hot and so on. Then it is not going to be a very expensive process as we will see very soon.

If you actually reduce the grain level of the profiling profiler so then the effectiveness of the optimization will be slightly lower, but well it may not matter in certain applications and it may be the bottle neck in some other application. So, this is always a tradeoff - Is it necessary to become be very accurate in profiling or should we reduce the overheads and use a less accurate profiler.

(Refer Slide Time: 19:44)



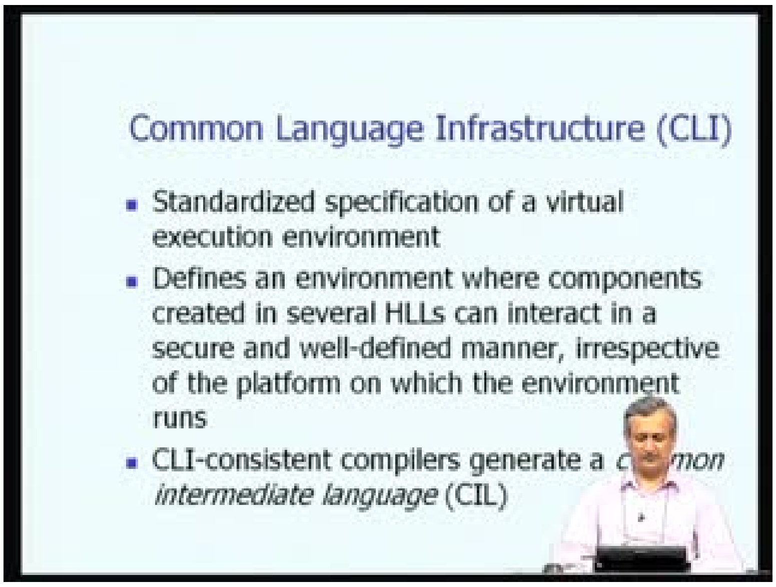
Our Research goals

- Implementation of an extensible multi-level adaptive recompilation framework for the .NET
- Implement and evaluate various profiling techniques (hardware & software) and profile-guided optimizations
- Suggest improvements to profiling techniques to reduce overheads
- All experimentation in ROTOR framework w implements *common language infrastructur*

What are our research goals? So, we started off with the following goals. We said implementation of an extensible multilevel adaptive recompilation framework for the .NET was one of our goals. So, this will become clear what is extensible, what is multilevel I already mentioned .So this will become clear as we go on.

Implement and evaluate various profiling techniques and profile guided optimizations. Suggest improvements to profiling techniques to suggest over, reduced over heads; all experimentation and rotor framework which implements the common language infrastructure so these are our goals. So let us see a couple of these now.

(Refer Slide Time: 20:36)

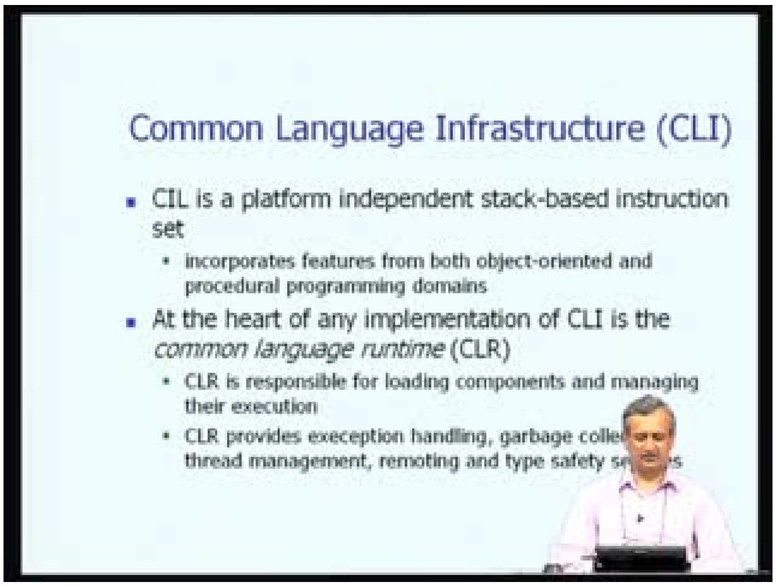


Common Language Infrastructure (CLI)

- Standardized specification of a virtual execution environment
- Defines an environment where components created in several HLLs can interact in a secure and well-defined manner, irrespective of the platform on which the environment runs
- CLI-consistent compilers generate a *common intermediate language* (CIL)

What exactly is common language infrastructure CLI? This is a standardized specification of a virtual execution environment. So, it defines an environment where, components created in several high level languages can interact in a secure and well defined manner. So, irrespective of the platform on which the environment runs. In other words, we actually can take components written in different high level languages and make them work together. So, this has not been you know possible before the common language infrastructure was proposed. It was not so easy to for example, even run c and Fortran programs in a general way. It is always possible to run them in a specific environment, but in general it was not possible, but now it is possible with the common language infrastructure. That too the platform is not important the environment runs on a particular platform. We compile everything c java and all that c sharp on the same platform, but the deployment of these components in a composition composed manner will be on a different platform. All this is possible.

(Refer Slide Time: 22:17)



Common Language Infrastructure (CLI)

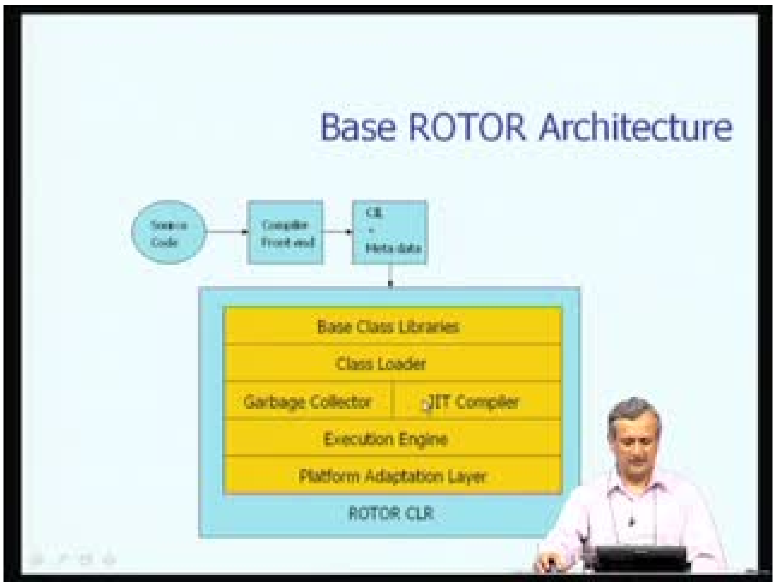
- CIL is a platform independent stack-based instruction set
  - incorporates features from both object-oriented and procedural programming domains
- At the heart of any implementation of CLI is the *common language runtime* (CLR)
  - CLR is responsible for loading components and managing their execution
  - CLR provides exeception handling, garbage colle thread management, remoting and type safety s s

The Common Language Infrastructure consistent compilers generate a common intermediate language. This is the key. Since, we do not generate machine code directly, but we generate a common language intermediate code or infrastructure. So, this is a platform independent stack based instruction set and therefore, it is like the java virtual machine. It incorporates features from both object oriented and procedural programming domains. So, that is the reason why, we can mix both c plus plus, java, c and other programming languages.

At the heart of any implementation of the Common Language Infrastructure is the common language run time. So c i l is common intermediate language. That is one part of the Common Language Infrastructure when we want to implement it, we need a common language run time system C L R. C L R is responsible for loading components and managing their execution.

So C L R also provides exception handling, garbage collection, thread management, remoting and type safety services. So, this is the virtual machine, these are all the services that the common language run time provides.

(Refer Slide Time: 23:42)

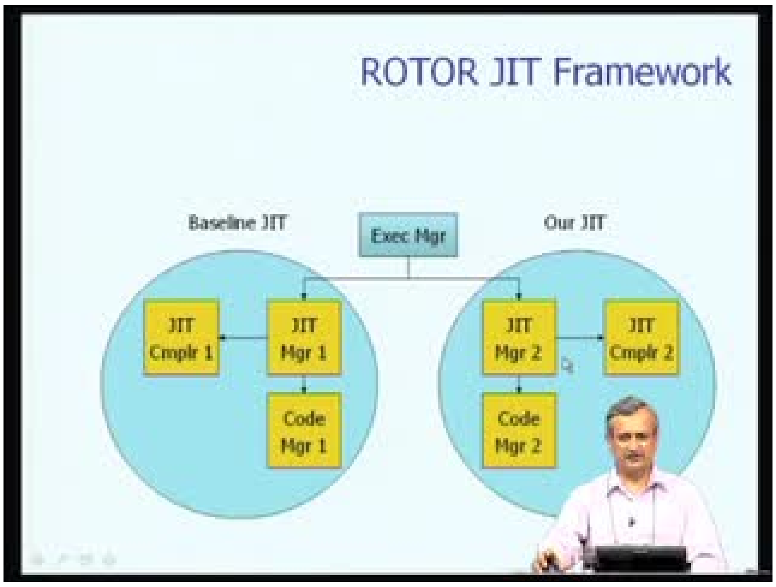


Base ROTOR Architecture

Rotor is one such the common language run time system. That was actually made available as a shared source. Microsoft made this available a couple of years ago; our experimentation and implementation has been on this rotor architecture. So, this provides c sharp compilers and also a common language run time implementation on the windows platform, on the Berkeley Unix platform. This can be used by many experimenters and it has been used that way.

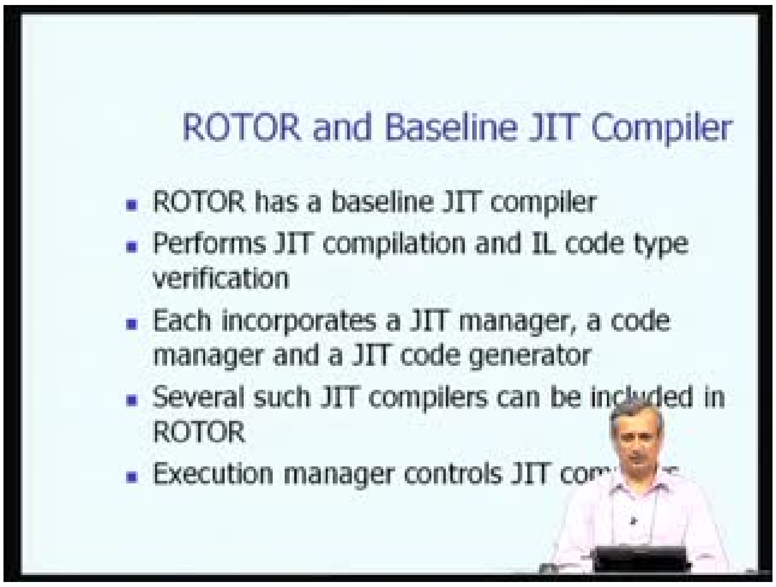
So for example, let us look at the block diagram of the base rotor architecture. If you are given a source program source code, the compiler converts it into the compiler front end, converts it into the common language common intermediate language c i l and then feeds it into the rotor C L R. The rotor C L R has an execution engine which is responsible for running the program in general. It can there is no question of interpretation here. It uses a Just-In-Time Compiler. In order to convert the common intermediate language to machine code; run it then there is a platform adaptation layer which is between the execution engine and the bare hardware, which actually provides the interface to the hardware. There is a garbage collector and then there are class loaders base class libraries etcetera available in the rotor architecture. This is the base rotor architecture.

(Refer Slide Time: 25:45)



Another specialty of the rotor framework is that you can really have many Just-In-Time Compilers. So, as it is the rotor architecture comes with 1 JIT compiler, which is not very good; we will see its characteristics very soon, but the nice thing about the rotor architecture is there is an execution manager, which can say which JIT compiler it wants to use. This is the base line JIT which is provided by rotor. Here is our JIT which is return by us; so we could either use the base line JIT or we could use our own JIT. This is controlled by the execution manager. So, we really did not have to touch and modify the base line JIT and the rest of the C L R. We could just add our own compiler JIT compiler to this particular frame work in an easy manner.

(Refer Slide Time: 26:48)



ROTOR and Baseline JIT Compiler

- ROTOR has a baseline JIT compiler
- Performs JIT compilation and IL code type verification
- Each incorporates a JIT manager, a code manager and a JIT code generator
- Several such JIT compilers can be included in ROTOR
- Execution manager controls JIT com

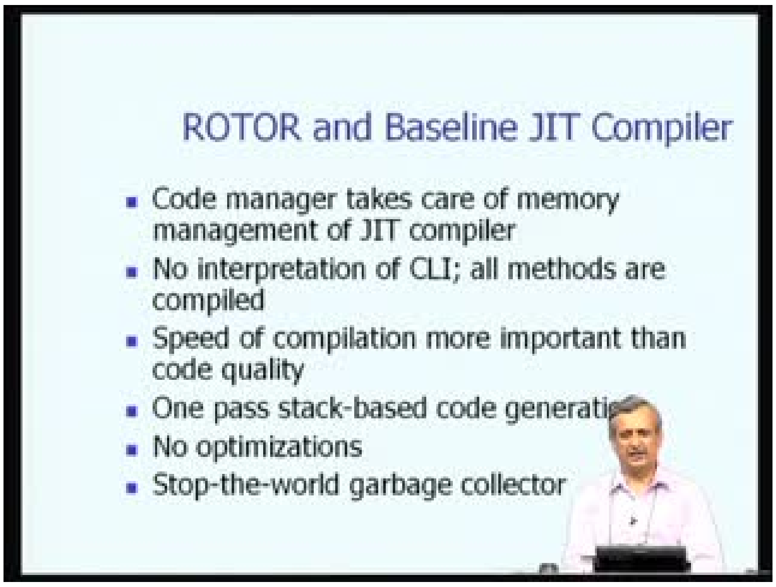
What does the Baseline JIT Compiler do? Rotor has a Baseline JIT Compiler; it performs Just-In-Time Compilation and Intermediate Language code type verification. The common intermediate language is actually type less. One has to run it through a type verifier, which actually infers types and make sure that all the types for all the instructions is correct. Once the types are inferred and types are all tagged, it is possible to compile it to machine code, before that it is not possible to compile it to machine code.

This type verification is provided on the intermediate code by the Baseline JIT Compiler.

We actually use the same IL code type verifier, available in the Baseline JIT Compiler. Each of these JIT Compilers must have a JIT manager, a code manager and a JIT code generator. These actually are the essential components. So, out of which the JIT code generator is important because, it generates machine code. The code manager is important because it controls how and where the code is placed in memory; how to jump to it and maintains the other data structures, which are essential to run the program correctly.

Several such JIT Compilers can be included in rotor. I already showed you the diagram which explains this. The execution manger controls the JIT Compilers.

(Refer Slide Time: 28:41)



ROTOR and Baseline JIT Compiler

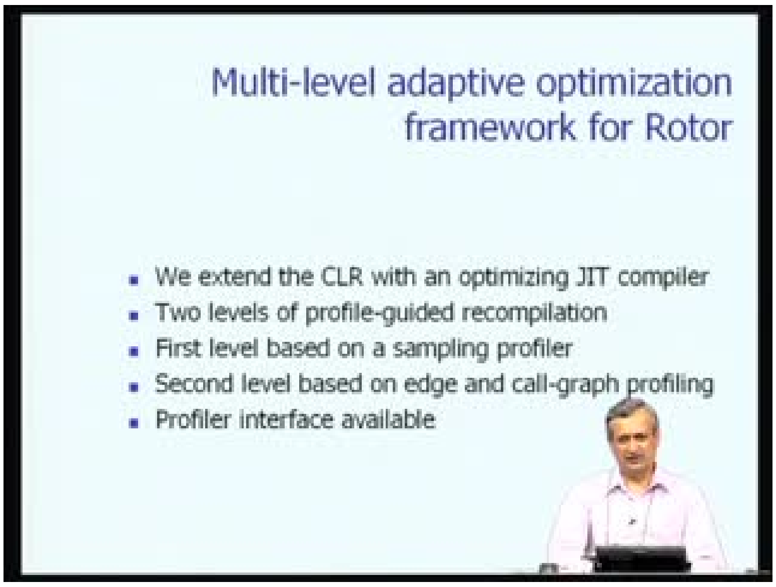
- Code manager takes care of memory management of JIT compiler
- No interpretation of CLI; all methods are compiled
- Speed of compilation more important than code quality
- One pass stack-based code generati
- No optimizations
- Stop-the-world garbage collector

What does the Code manager do? The code manager takes care of memory management of the JIT compiler. As i mentioned, there is no interpretation of the common intermediate language programs all methods are compiled. In the common language infrastructure there is no interpretation possible whereas, java byte code is interpreted the C I L is never interpreted. All methods are compiled at least once. But the speed of compilation is more important than the code quality in the case of Baseline JIT Compilation; it does a 1 pass stack based code generation, it goes through the program and generates code in a very simple manner.

Therefore it does not perform any optimizations. The code generated is extremely naïve and therefore, very inefficient. Remember that the Baseline JIT compiler even though provided by Microsoft, was more for experimental purposes and to prove that the .NET is useful. Now what exists in the .NET is a very professional JIT compiler, but a few years ago this was not so.

The garbage collector provided in rotor is also stopping the world garbage collector. In other words, it would stop the program. The garbage collector would stop the program, then do garbage collection based on any particular strategy; then restart the program. So, this is the stop the world garbage collector.

(Refer Slide Time: 30:31)



Multi-level adaptive optimization framework for Rotor

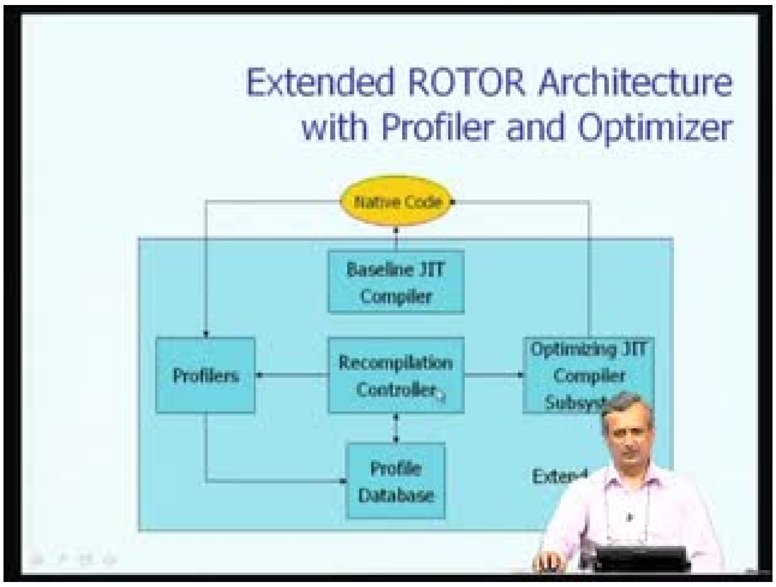
- We extend the CLR with an optimizing JIT compiler
- Two levels of profile-guided recompilation
- First level based on a sampling profiler
- Second level based on edge and call-graph profiling
- Profiler interface available

Let us start looking at the multi-level adaptive optimization framework for rotor. We extend the common language run time with an optimizing Just-In-Time Compiler. So as i said, the Baseline JIT Compiler does not perform any optimizations. In fact, it does not perform even very simple optimizations. Therefore, the level of the code or the quality of the code that is generated by the Baseline JIT Compiler is very bad. We provide 2 levels of profile guided recompilation: the first level is based on a sampling profiler; what is a sampling profiler.

The sampling profiler looks at the various methods which are currently active, counts how many times each method is being called. So, determines how hot each method is. So, this is the sampling profiler; it samples the calling stack. The first level of optimization is guided by the sampling profiler. We perform simple optimizations at the first level. So, a few more details will be provided very soon.

Second level of optimization: it is a little more advanced and is based on edge and call graph profiling. What is this edge profiling? Edge profiling is basically how many times is each edge traversed in the call graph or the control flow graph. Call graph profiling is trying to find out exactly how many times, with what type of parameters, is each procedure call being made. The parameters are the same, are they different etcetera. Profiler interface is also available. We have made a Profiler interface that available. So, that the programmer can write JIT Compilers using the Profiler interface and need not dig it in the code of the profiler itself.

(Refer Slide Time: 33:05)



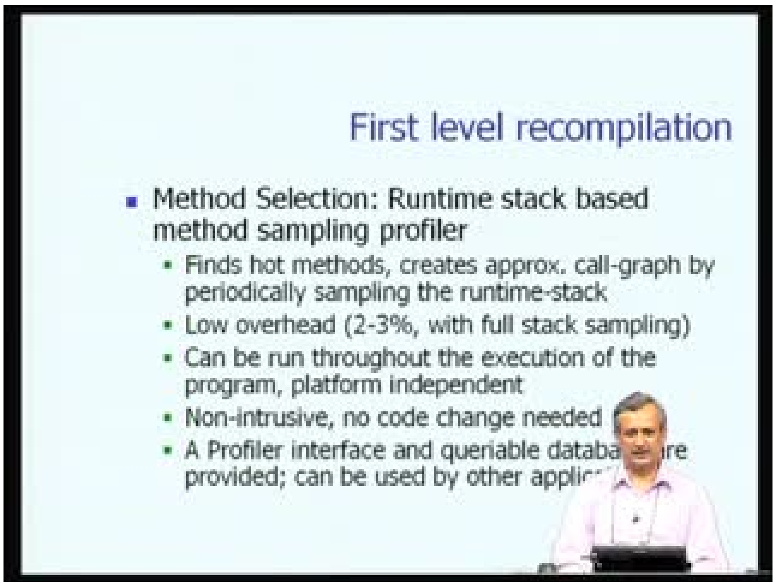
Extended ROTOR Architecture with Profiler and Optimizer

Here is the Extended Rotor Architecture with Profiler and Optimizer. We have the Baseline JIT Compiler available here as it is. We have our profilers of various kinds: edge profilers, call graph profilers, sampling profilers and so on.

We have a recompilation controller, which determines which methods have to be recompiled into efficient code. So, that drives the optimizing JIT compiler to make some of the optimizations; the code generation takes place on the optimized intermediate code. There is a profile database, which is created by the profiler. This profiler data base is not really a huge data base maintained on the disc. It is a small database; in fact maintained within the main memory.

This profile database can be queried by the JIT Compiler writer; using its API. This is also useful in determining which methods are very hot and then ask the recompilation controller to recompile that particular method.

(Refer Slide Time: 34:29)



First level recompilation

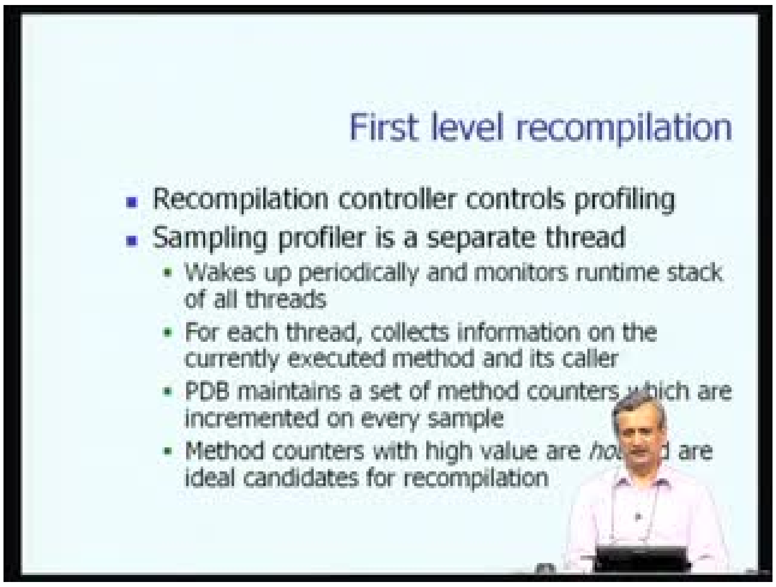
- Method Selection: Runtime stack based method sampling profiler
  - Finds hot methods, creates approx. call-graph by periodically sampling the runtime-stack
  - Low overhead (2-3%, with full stack sampling)
  - Can be run throughout the execution of the program, platform independent
  - Non-intrusive, no code change needed
  - A Profiler interface and queriable databa re provided; can be used by other applic

What are some of the First level recompilation strategies, you know what exact tasks. So, method selection, which method should we choose to recompile and optimize; that is the question. If we do not want to optimize; we could just use the Baseline JIT Compiler and be done with it. But using the First level recompilation; we could do slightly better.

Method selection is based on run time stack based method sampling profiler. It finds hot methods and creates approximate call-graph, periodically, sampling the runtime-stack. How does it do it; we will see very soon. How exactly this profiler works. It finds hot methods and creates only the approximate call-graph. Why is it approximate? The reason is, it does not actually analyze the call stack continuously after every call. It only samples the runtime-stack periodically and therefore, you may end up missing a call for all; that is why it is approximate. But a sampling profiler has very low overhead so a 2 to 3 percent even with full stack sampling. It is not necessary to look at the top of the stack, you can look at the whole stack; but periodic.

It can be run throughout the execution of the program and it is platform independent. So, it is Non-intrusive, no code change is needed. A Profiler interface and a queriable database are also provided. They can be used by other applications, which require this kind of a sampling profile.

(Refer Slide Time: 36:40)



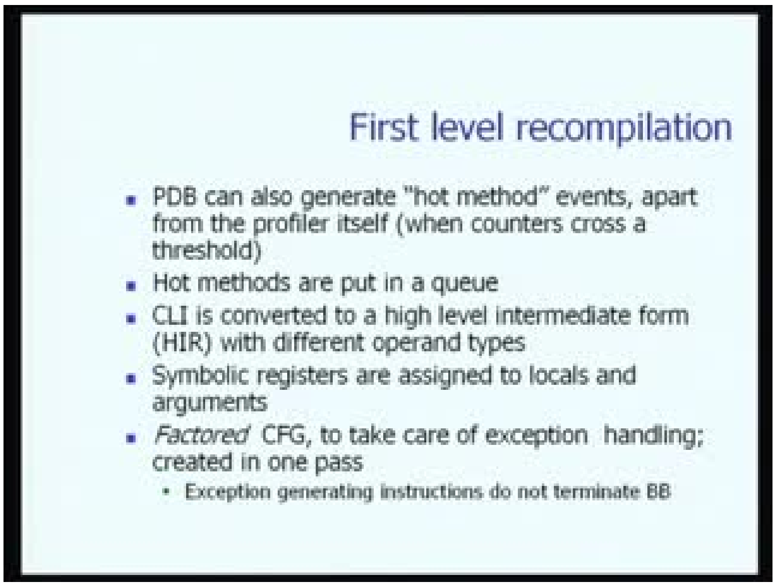
First level recompilation

- Recompilation controller controls profiling
- Sampling profiler is a separate thread
  - Wakes up periodically and monitors runtime stack of all threads
  - For each thread, collects information on the currently executed method and its caller
  - PDB maintains a set of method counters which are incremented on every sample
  - Method counters with high value are *ho*[illegible] are ideal candidates for recompilation

Recompilation controller controls the profiling. So, which type of profiling is to be done etcetera are all controlled by the recompilation controller. Because at second level, different profiling strategies have to be used with the same method whereas, in the first level, we just use a simple sampling strategy. What is a sampling profiler? It is a separate thread. It wakes up periodically, so the thread is sleeping. It wakes up periodically, it monitors run time stack of all the threads. So, there is a separate stack for each one of the threads, because within each thread there could be method calls. So, there is a run time stack.

This stack is monitored by the thread that is the profiler thread. It collects for each of the thread informations on the currently executed thread and its caller. That is it just looks at the top two entries, so the currently active method and its caller. If it had seen the entire stack; it would have got information on all the procedures and methods which are active, but to reduce the overhead looks at the top 2, because we are more keen on the hot methods. If a method is hot; it will be called again and again. Then the profiler database maintains a set of method counters, which are incremented on every sample. So, it looks at the top 2, goes to the profiler database, chooses the method counter corresponding to the method which is active and the caller method increments their counters.

(Refer Slide Time: 38:45)



First level recompilation

- PDB can also generate "hot method" events, apart from the profiler itself (when counters cross a threshold)
- Hot methods are put in a queue
- CLI is converted to a high level intermediate form (HIR) with different operand types
- Symbolic registers are assigned to locals and arguments
- *Factored* CFG, to take care of exception handling; created in one pass
  - Exception generating instructions do not terminate BB

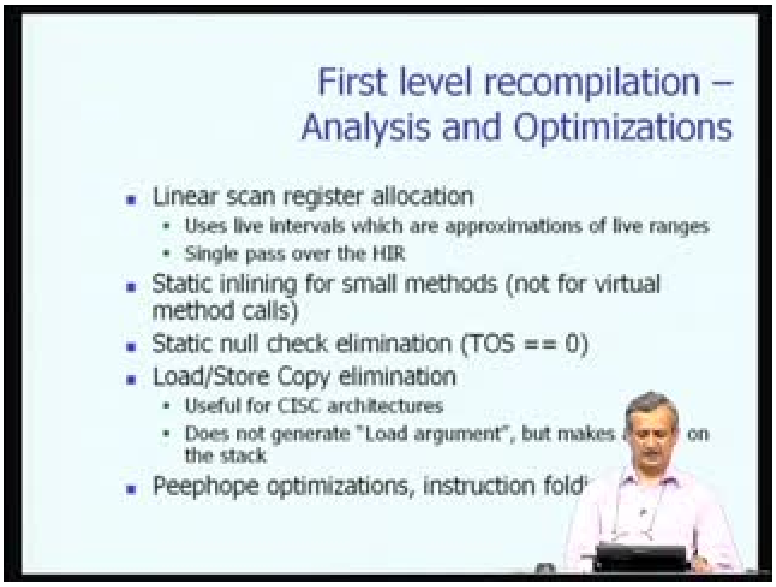
Method counters with high values are very hot and are ideal candidates for recompilation. This is how it determines hot methods. The profile data base can also generate hot method events apart from the profiler itself. If the profiler has not yet determined something to be very hot. But the profiler database has a threshold. It can say if anybody crosses this threshold, then that particular method is hot; it can say that. So, there by, it can instruct the recompilation controller to recompile a particular method which is very hot.

This is to make sure that hot methods are not missed. So, either the profiler or the profiler database will take care of informing the recompilation controller about hot methods. The hot methods are put in a queue taken by one by one and then recompiled.

The CLI is converted to a high level intermediate form. The common intermediate language is converted to a high level intermediate form with different operand type. So, remember the type verification, type inferencing of the intermediate code is already over. The types are now tagged to each one of the instructions of the intermediate code. Now, this can be converted to a symbolic register intermediate language; which is at a slightly higher level. It is not yet machine code level; its slightly higher level.

Symbolic registers are assigned to locals and the arguments of methods. What happens to exceptions? Exception actually breaks control flow, jumps to the exception code. But this creates difficulties in processing in data flow analysis. So, factored control flow graph to take care of exception handling is created in one pass. So, exception generating instructions do not terminate the basic block. So, that is the way it is handled. So, more details are available in a particular paper, based on Factored CFG. But at this point, it suffices to know that a factor CFG is meant to take care of exceptions and exceptions create difficulties, otherwise in data flow analysis.

(Refer Slide Time: 41:18)



First level recompilation – Analysis and Optimizations

- Linear scan register allocation
  - Uses live intervals which are approximations of live ranges
  - Single pass over the HIR
- Static inlining for small methods (not for virtual method calls)
- Static null check elimination (TOS == 0)
- Load/Store Copy elimination
  - Useful for CISC architectures
  - Does not generate "Load argument", but makes on the stack
- Peephope optimizations, instruction fold

What are the various analysis and optimizations at the first level that we perform? We have already studied linear scan register location. Basically, this method scans the entire set of instructions from top to bottom; then creates the intervals. So, then it looks at the intervals one by one, creates a data structure which contains intervals in some particular ascending, descending order of their start time, sent times etcetera, allocates registers using a simple strategy. So, we use this linear scan register allocation because it is very fast; it uses live intervals which are approximations of live ranges, it is a single pass over the high level intermediate code.

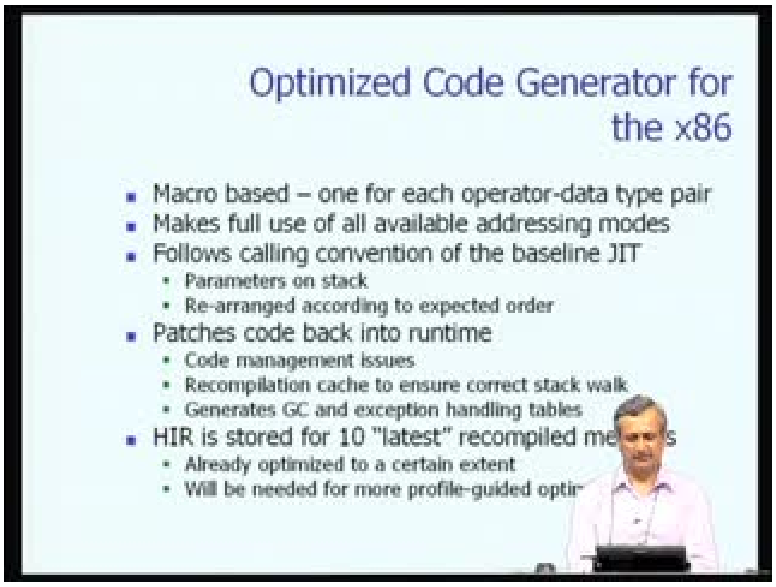
Remember, if we try using graph coloring or web based register allocation, it is extraordinarily slow. All these adds to the overhead of the virtual machine. It is not feasible to use heavy weight register allocators, such as, graph coloring based register allocators. That is the reason why we use a simple linear scan register allocation.

Static inlining is used for small methods, obviously, not virtual calls. Inline the method call, so that it becomes more efficient. But then code size will increase a little bit; so that is the penalty anyway.

Static null check elimination, top of stack is; is it zero or not is also performed. These are all very simple optimizations. But remember, you are performing all this at run time when the program is running. So, overheads have to be kept at a low level. We cannot really perform very heavy optimizations; at least at the first level. Load store copy elimination is another optimization, so this is useful for CISC architectures. Basically, it does not generate a load instruction, but makes a copy on the stack and uses that instead of loading from memory.

Several Peephope optimizations, such as instruction folding, that is, combining 2 instructions into 1 are performed.

(Refer Slide Time: 44:56)



Optimized Code Generator for the x86

- Macro based – one for each operator-data type pair
- Makes full use of all available addressing modes
- Follows calling convention of the baseline JIT
  - Parameters on stack
  - Re-arranged according to expected order
- Patches code back into runtime
  - Code management issues
  - Recompilation cache to ensure correct stack walk
  - Generates GC and exception handling tables
- HIR is stored for 10 "latest" recompiled me s
  - Already optimized to a certain extent
  - Will be needed for more profile-guided optir

We also wrote an optimized code generator for the x86. This is a macro based code generator, so there is 1 Macro for each operator-data type pair. We start looking at the common intermediate language instructions one by one. Then use the macro corresponding to that CIL instruction, expand the macro and generate the code for it. It makes full use of all available addressing modes; it follows calling convention of the Baseline JIT Compiler that is already available in rotor. So, parameters are always placed on the stack, but they are rearranged according to the expected order of usage of the arguments.

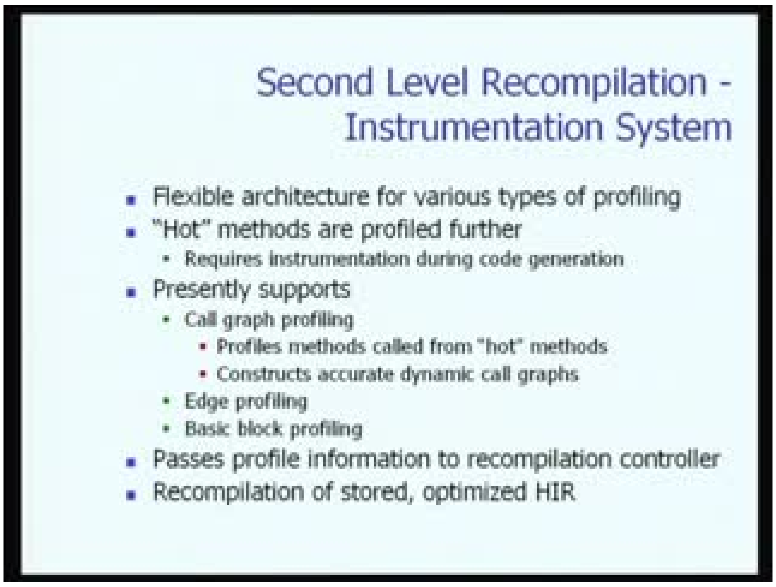
The code that is generated by this code generator has to be patched back into the run time system. So, there are code management issues here. Let us say, there is one version of the code which is already running and we generate another version of the code. Now, when exactly do we start running the second version? That is a question, which has to be answered properly without creating inconsistencies.

There is a Recompilation cache, which ensures correct start walk and answers this question. It keeps track of the old version and also the new version. Whenever, there is a request to use a particular piece of machine code, the old version JIT Compiler, the code manager is checked to see whether it is still being used. If there is no usage at that point; then the new one can be activated. But if the old one is still active; is still on the caller stack, then we really cannot use the new one right away, we have to wait until the activation of the old call is completed and for other new calls new version can be used.

We also generate simple garbage collector and exception handling tables. So, that is part of our code generation strategy.

The high level intermediate code is stored for 10 latest recompiled methods. We do not have to go back to the CIL and generate the HIR. We actually keep the HIR for the 10 latest methods, recompiled methods; these can be reused for advanced optimizations at level 2. They are already optimized to certain extent, because of first level optimizations; these will be needed for more profile guided optimizations at level 2.

(Refer Slide Time: 47:08)

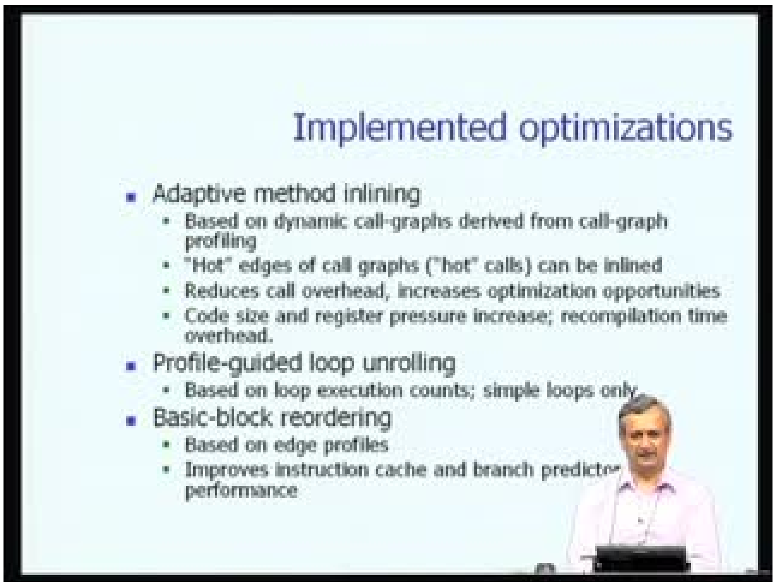


Second Level Recompilation - Instrumentation System

- Flexible architecture for various types of profiling
- "Hot" methods are profiled further
  - Requires instrumentation during code generation
- Presently supports
  - Call graph profiling
    - Profiles methods called from "hot" methods
    - Constructs accurate dynamic call graphs
  - Edge profiling
  - Basic block profiling
- Passes profile information to recompilation controller
- Recompilation of stored, optimized HIR

The Second Level Recompilation System, there is an Instrumentation System here. This is a very flexible architecture for various types of profiling. So, hot methods are profiled further. This requires instrumenting the code during the code generation. The profiler code instrumentation has to be inserted. We presently support Call graph profiling; that is profiling methods called from hot methods. Then we construct accurate dynamic call graphs here. So, that is the profiling that we do. We also support edge profiling in the control flow graph and we also finally, support basic block profiling. So these are all now going to be used for advanced level optimizations. We pass the profile information to the recompilation controller, which in turn chooses which optimizations to perform. So, recompilation of stored optimized HIR is performed; we already stored the HIR, so that is reused.

(Refer Slide Time: 48:21)



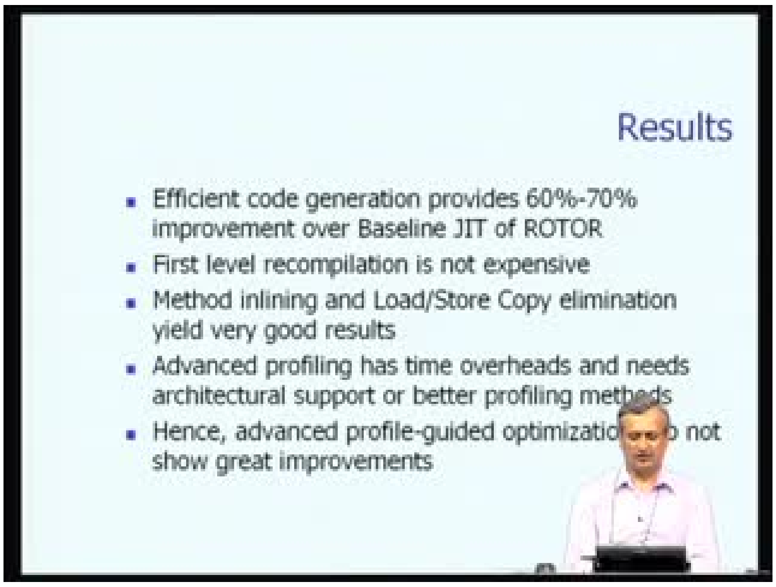
Implemented optimizations

- Adaptive method inlining
  - Based on dynamic call-graphs derived from call-graph profiling
  - "Hot" edges of call graphs ("hot" calls) can be inlined
  - Reduces call overhead, increases optimization opportunities
  - Code size and register pressure increase; recompilation time overhead.
- Profile-guided loop unrolling
  - Based on loop execution counts; simple loops only
- Basic-block reordering
  - Based on edge profiles
  - Improves instruction cache and branch predictor performance

What are the optimizations at the second level? Adaptive method inlining is one of them. We know what is method inlining; it is just inline expansion of the procedure corresponding to the method. This is based on the dynamic call graph, derived from call graph profiler. Hot edges of call graphs that is the hot calls can be inline. It is not necessary to inline every occurrence of a method call, but only the hot occurrences of method calls need to be inlined. So, then code explosion will be much lesser. It reduces call overhead, because there is no call, it is inline; increases optimization opportunities because this is very hot. We can do much better optimizations on this piece of code. Code size and register pressure increase and recompilation time also is extra, so these are the overheads.

We also perform profile guided loop unrolling, based on loop execution counts by and for simple loops only. So, basic block reordering is performed based on edge profiles. Basically, this improves instruction cache and branch predictor performance, because the code corresponding to 2 basic blocks. If suppose we know that one basic block follows the other, more frequently than the other basic block in the branch statement. Then we can put the codes of generate the codes of the 2 basic blocks of which occur more frequently one after another.

(Refer Slide Time: 50:10)



Results

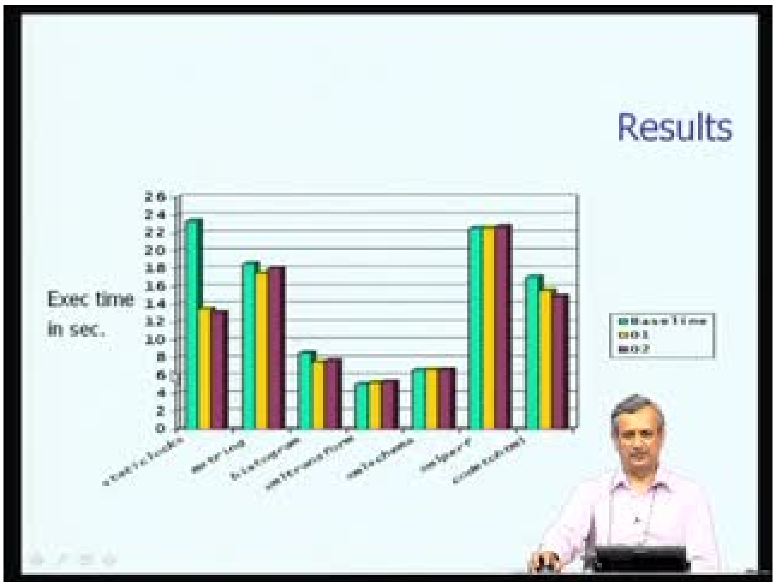
- Efficient code generation provides 60%-70% improvement over Baseline JIT of ROTOR
- First level recompilation is not expensive
- Method inlining and Load/Store Copy elimination yield very good results
- Advanced profiling has time overheads and needs architectural support or better profiling methods
- Hence, advanced profile-guided optimizatio[illegible] not show great improvements

So here are some Results that we got. Efficient code generation provides 60 to 70 percent improvement over the Baseline JIT of Rotor. I already mentioned that the code generator of the Baseline JIT are provided in rotor is very inefficient. It generates simple, naïve code. So, our code generator was far more efficient. I already mentioned that uses all the addressing modes. So, it was very efficient. So, just better code generation gave us 60 to 70 percent improvement over the existing JIT of rotor.

The first level recompilation is not expensive. As i said only 2 to 3 percent extra over head is involved. Method inlining load store copy elimination yield excellent results. Whereas, advanced profiling that is take the methods, which are very very hot and perform advanced optimizations on them. The advanced profiling, which is needed for this has time overheads and needs architectural support for better profiling of methods.

It is very expensive to actually build the dynamic call graph or do edge profiling or basic block profiling etcetera. The overheads really shoot up and even the optimizations which are performed, do not actually give enough benefits. Hence, advanced profile-guided optimizations do not show too much improvement.

(Refer Slide Time: 52:02)



Here is a graph. So, here the baseline is not the Baseline of the JIT Compiler provided by rotor. But the Baseline is our native code generator based JIT Compiler without any optimizations. So, as you can see, just the very good code generation itself has done wonders. So, here this is execution time in seconds. O 1 is level one optimization with simple sample profiling; O 2 is advanced optimization with profile guided optimization, with advanced profiling. So, as you can see it is enough to do good code generation, for these bench marks and O 1 and O 2 hardly yield extra benefits.

It is possible that different bench marks give you different types of behavior, but in general, unless the profiling overheads are reduced; either by providing hardware support for profiling or by inventing very efficient methods of profiling. Very high level advanced profile guided optimizations may not help in a JIT Compiler, but very good code generation and simple profiling profile guided optimizations will go a long way in improving the performance of a JIT Compiler. So, this is our inference of the project on Just-In-Time Compilation for the .NET Common Language Run Time.

This is the end of lecture. Thank you very much.