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

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*

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 reliance on third party and downloaded components

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

Virtual Machines -- Facilities

- Checks security violations by components
- Components can be dynamically loaded and linked
- Provides automatic memory management and garbage collection
- Provides architecture-independent interface for exception handling

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

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-performance environments
- Just-In-Time Compilation is a good solution

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 execution overheads

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 ones)
  - Multi-level optimization framework based on profiling is proposed

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 profiling

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 effectiveness of optimizations
    - Tradeoff (accuracy v/s overheads)

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 which implements *common language infrastructure*

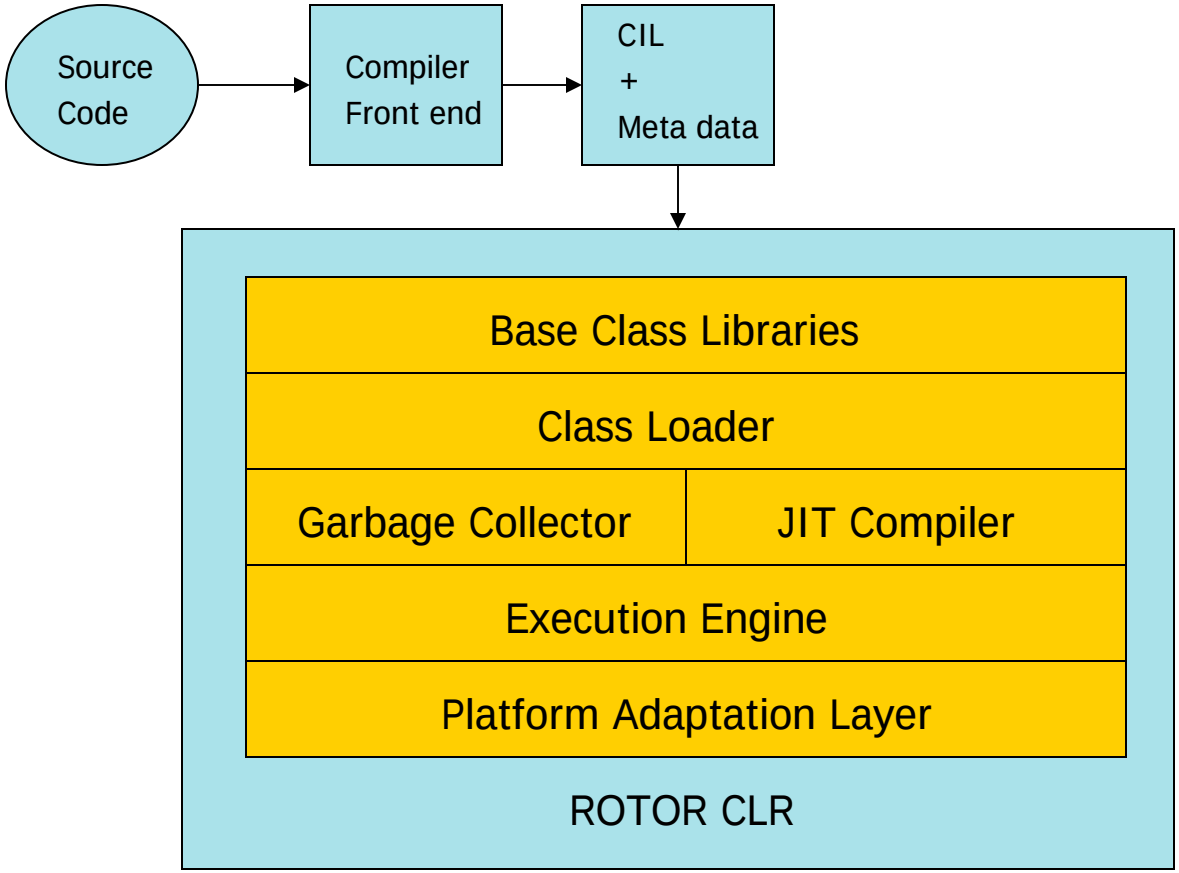
Common Language Infrastructure (CLI)

- 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)

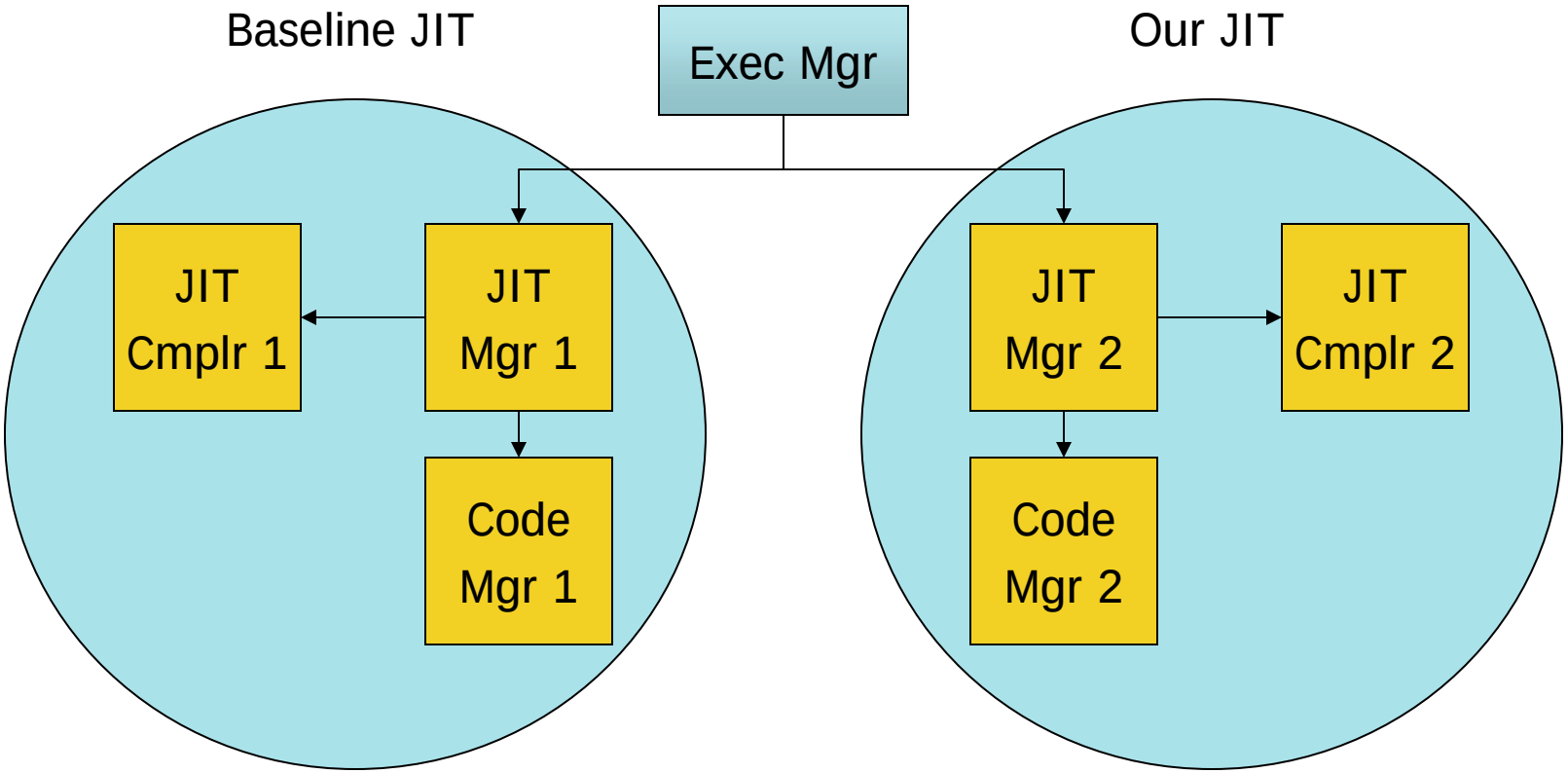
Common Language Infrastructure (CLI)

- 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 collection, thread management, remoting and type safety services

Base ROTOR Architecture



ROTOR JIT Framework



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 compilers

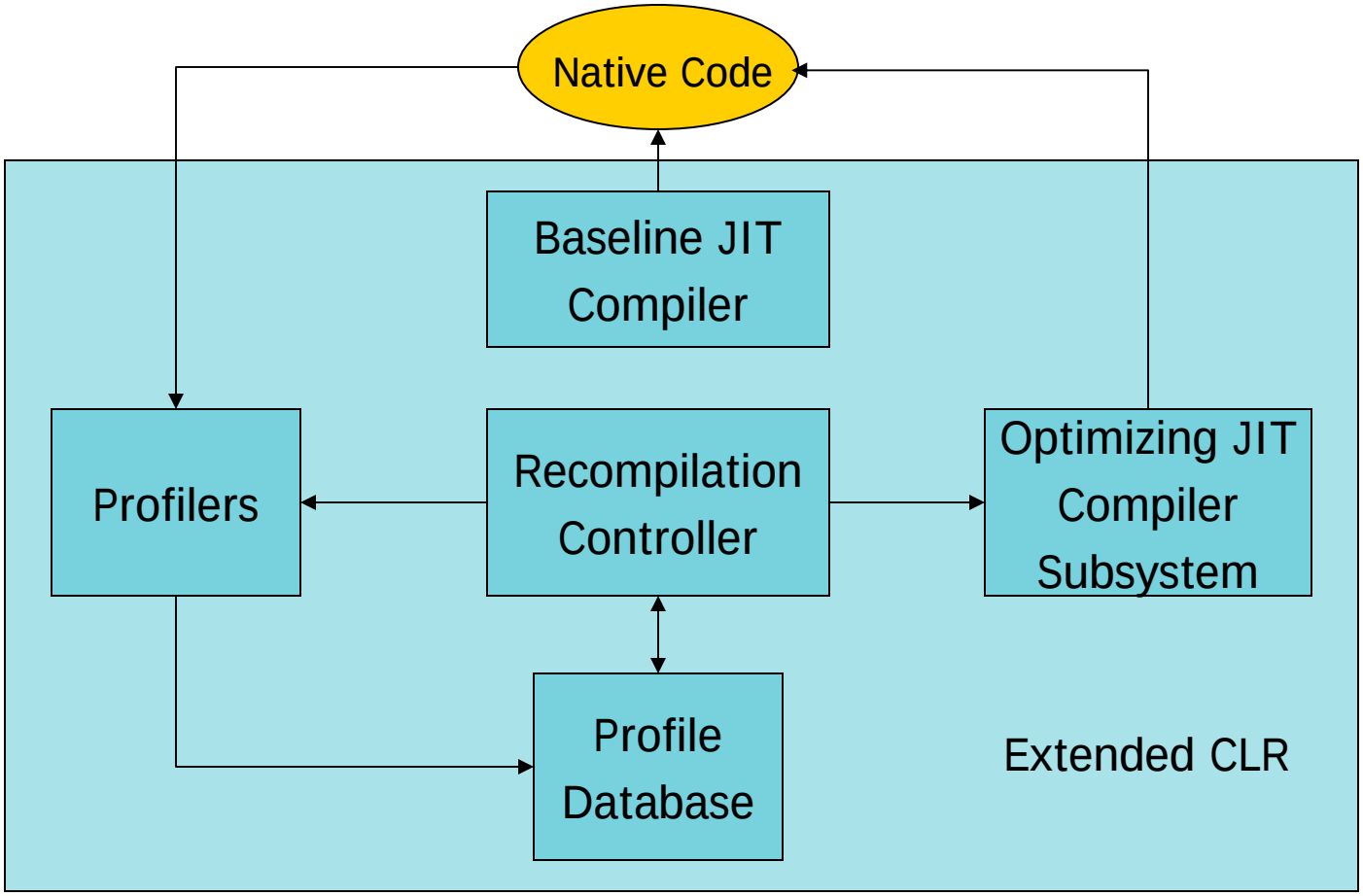
ROTOR and Baseline JIT Compiler

- 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 generation
- No optimizations
- Stop-the-world garbage collector

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

Extended ROTOR Architecture with Profiler and Optimizer



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 database are provided; can be used by other applications

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 *hot* and are ideal candidates for recompilation

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

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 a copy on the stack
- Peephope optimizations, instruction folding

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 methods
  - Already optimized to a certain extent
  - Will be needed for more profile-guided optimizations

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

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

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 optimizations do not show great improvements

Results

Exec time in sec.