# Bytecode translation via compilation

- Bytecode → HIR (abstract interp) + basic optimizations
- LIR → LIR (expand calls) + CSE + data dependencies
- LIR → MIR (instr. scheduling) + Register Allocation
- Prologue/epilogue added to method
  - Prologue
  - Epilogue

# Bytecode translation via compilation

- Bytecode → HIR (abstract interp) + basic optimizations
- LIR → LIR (expand calls) + CSE + data dependencies
- LIR → MIR (instr. scheduling) + Register Allocation
- Prologue/epilogue added to method
  - Prologue
    - Allocate runtime stack frame
    - Save any nonvolatile registers
    - Check whether a thread yield has been requested
    - Lock if the method is synchronized
  - Epilogue
    - Restore any nonvolatile registers
    - Store return value
    - Unlock if the method is synchronized
    - Deallocate the runtime stack frame
    - Branch to return address

# Bytecode translation via compilation

- Bytecode → HIR (abstract interp) + basic optimizations
- LIR → LIR (expand calls) + CSE + data dependencies
- LIR → MIR (instr. scheduling) + Register Allocation
- Prologue/epilogue added to method
  - Prologue
  - Epilogue

- **Store in memory at address**
  - Convert intermediate-instruction offsets to machine code offsets
    - For exception handling
    - For garbage collection (reference maps)
  - Update VM tables (statics or VMTs) with address
  - Jump and execute (JIT-compiled methods), fixing up the stack to return to the caller of the JIT'd method

# The JikesRVM Adaptive Optimization System

# Adaptive Compilation (aka Adaptive Optimization)

- Compiling at the method level is
  - Slow – much slower than cost of interpreting one instruction
  - Optimizing compiler (as efficient as it is) is very high ohead
- If we compile everything
  - Big startup delay
  - Big delay the first time we execute a method
- Goal: combine interpretation and compilation to get the best of both "**mixed mode**"
  - Interpret first: fast startup, no long pauses
  - Identify the frequently executing methods (**hot** methods)
  - Compile them (with some optimization) in the background
    - Execute them the next time around

# Multi-compiler (Mixed Mode) System

- **Compile-only vs Compile+Interpret strategy**

- Baseline – (could be replaced with interpretation) …
  - Simulates execution using the bytecode and operand stack
  - Translates bytecodes to native code directly
  - No optimization, no register allocation
  - Performance much like an interpreter
  - **Fast compilation/interpretation, SLOW code**
- Optimizing
  - Translates bytecodes to HIR->LIR->MIR
  - Optimization is performed on each level
  - Linear scan register allocation
  - **Slow compilation/fast code**

# JikesRVM Compiler Differences

- Compile Time/Speed comparison
- 500MHz RS6000, 4GB Mem, 1-processor

- **Compile time**: Bytecode bytes per millisecond
  - Baseline: 378,    L0: 9.3,    L1: 5.7,    L2: 1.8
- **Code speed** normalized to baseline
  - L0: 4.3,    L1: 6.1,      L2: 6.6

- EX: L2 is 209 times slower to compile & produces code that is 6.6 times faster

# JikesRVM Threading

- Two alternatives
  - Native threads: Map each Java thread to an OS pthread; OS-managed
    - Less work for the runtime (simpler) for scheduling
    - More work for the runtime to facilitate GC (since thread switching can now can happen on any instruction
      - Compiler generates GC maps (list of roots) at every instruction
  - Green threads: Java threads are multiplexed on virtual processors; JVM/runtime managed in coordination with OS
    - A virtual processor is an OS pthread
    - Require software support for switching (yielding the processor so that other threads can take a turn) – **yield points**
    - Compiler generates this support
      - Generates GC maps (list of roots) at every yield point

# JikesRVM Threading

- Java threads are multiplexed on virtual processors
  - A virtual processor is an OS pthread
- Yield points
  - Compiler generated
  - Points in a method where a thread checks to see if it should give up the processor (& give another thread a turn)
    - Check a bit in a register, if set then call scheduler
    - Set is caused by timer interrupt
    - Method prologues
    - Back edges of loops

```
    x = 20
L1: if x>=10 goto L3
    . . .
    goto L1
L3: y = x + 5
```

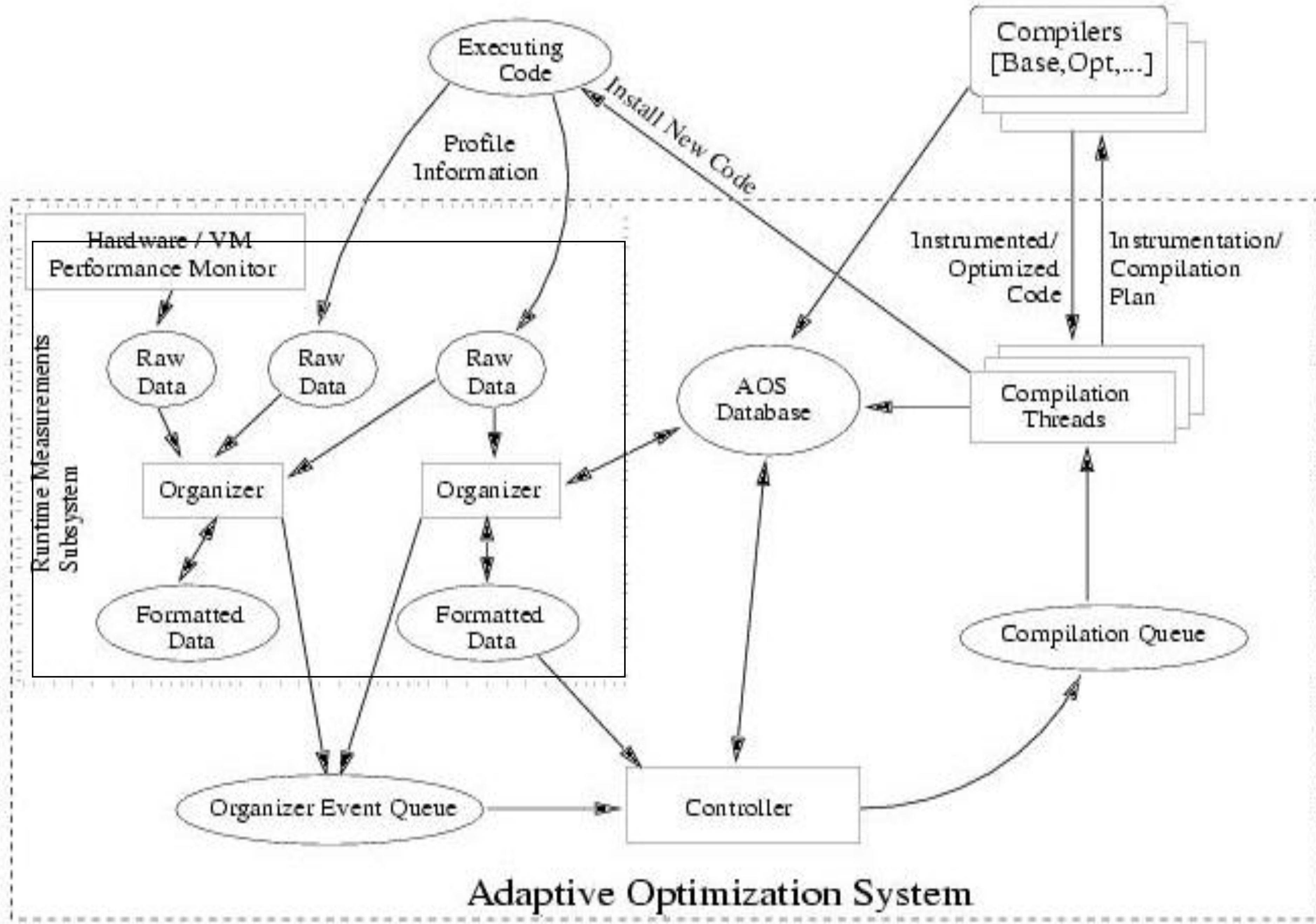
→

```
    x = 20
    goto L1
L0: yeild
L1: if x>=10 goto L3
    . . .
    goto L0
L3: y = x + 5
```

# Adaptive Optimization System Architecture

- Runtime measurements subsystem
- Controller
- Recompilation System

# Adaptive Optimization System Architecture



# Runtime Measurements Subsystem

- Gathers information about executing methods
- Summarizes the information
- Passes the summary to the event system
- Records the summary in a database

# Runtime Measurements Subsystem

- Gathers information about executing methods
- Summarizes the information
- Passes the summary to the event system
- Records the summary in a database

- Information
  - From the VM
    - When it performs services for the program (thread switch, memory allocation, compilation, etc.)
  - From instrumentation
    - Code added to the executing methods
    - Methods in application and VM
    - Invocation counters, edge, path, value profiling
  - Hardware performance counters
    - Cache misses (instruction/data)

# Runtime Measurements Subsystem

- Information is stored in raw format
- **Organizers**
  - Threads that periodically process the information, analyze it, and format it appropriately for use by the controller
  - Separates data generation from analysis
    - Why?

# Runtime Measurements Subsystem

- Information is stored in raw format
- **Organizers**
  - Threads that periodically process the information, analyze it, and format it appropriately for use by the controller
  - Separates data generation from analysis
    - Multiple organizers can process the same data (in different ways)
    - Profiling code can then operate under rigid resource constraints
      - Example: VM memory allocator profiler
      - Can't allocate memory
      - Should complete quickly so as not to interrupt execution
    - Overlap analysis with application execution

# Controller

- Manages the adaptive optimization system
- Coordinates activities of runtime measurement subsystem and the recompilation system
- Initiates all profiling activity by determining what profiling
  - Should occur
  - Under what conditions
  - For how long

- Gets its information from the runtime measurement subsystem and the AOS database
- Passes compilation decisions to the recompilation subsystem (continue or change)

# Sampling to Identify Hot Methods

- To estimate the time spent in a method
- Sample on yield points only (ie ***when*** a thread yields)

  - Before **switching** threads, a counter associated with the method that is executing (current) is incremented
    - When a loop backedge is traversed a counter is incr'd.
    - When a method prologue is entered
      - A counter for the invoked method is incremented
      - A counter for the calling method is incremented

- This information (and HW counter information) is stored as raw data

# Sampling

- Three threads access the raw data
  - Method listener object (created by the hot method organzr)
    - On each thread switch, records the currently active method in the raw data buffer – **runs on the application thread**
    - Wakes hot method organizer after sample size has been reached
  - Hot method organizer
    - Scans the method counter raw data to identify methods in which the most time is spent – **in the background**
    - "hot" if the percentage of samples attributed to that method exceeds a controller-directed threshold
      - And the method is not already compiled to maximum degree
    - Enqueues an event in the event Q for each hot method (and %age)
  - Decay organizer – **decrements method counters** (in bg)
    - Gives more weight to recent samples (for hotness identification)

# Recompilation

- Given a hot method, the controller decides if it is profitable to recompile a method
  - Cost model
    - Expected time the method will execute if not recompiled
    - Cost for recompiling the method at a certain optimization level
    - Expected time the method will execute if recompiled
  - Goal: minimize the expected future running time of the method in the future

# Recompilation

- Assumptions are made for all expected values
  - Program will execute for twice the duration that it has
    - Uses samples to estimate percentage of program time spent in the method in question
  - Offline measurements indicate the effectiveness of each optimization level
    - How much faster the method will run
  - Cost of recompilation
    - Linear model of compilation speed for each optimization level as a function of method size.
    - Calibrated offline

# AOS Optimization: Feedback Directed Inlining

- Statistical sample of the method calls in a running application
  - Maintains an approximation of the dynamic call graph
  - Identifies hot edges to inline
  - Optimizing compiler uses this information for inline decisions

- **On thread switch**, an edge listener thread **in background** walks the thread's runtime stack (frames) to identify the caller call site that init'd the call
  - <caller, call site, callee> is inserted into a buffer
  - When buffer is full, it wakes an organizer

# Feedback Directed Inlining

- Dynamic call graph organizer
  - Maintains the dynamic call graph
  - Updates the weights on the graph edges
  - Clears the buffer
  - Restarts the listener
  - Decay organizer periodically decays the edge weights

# Feedback Directed Inlining

- Dynamic call graph organizer
  - …
- Periodically invokes an adaptive inlining organizer
  - Recomputes inlining decisions
  - Identifies edges in the DCG whose percentage of samples exceed an edge hotness threshold
    - Added to an **inlining rules data structure**
    - Consulted by the controller (to formulate compilation plans)
    - All edges cause inlining to happen (subject to size constraints)
      - Edges that go to 0 are removed and **not inlined again**
  - Fewer at program start than later; past inlines are not lost

# Performance of On-line Profiling (Adaptive Opt.) JikesRVM