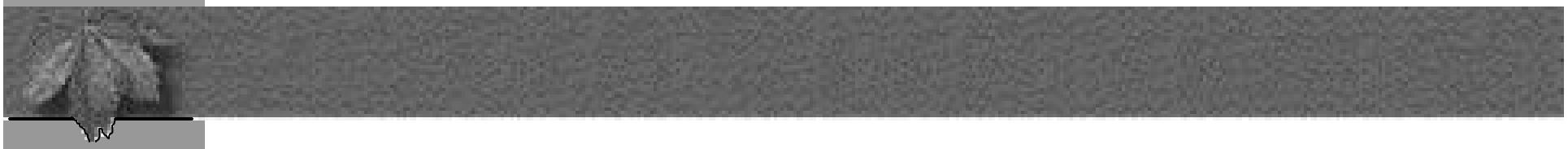
Dynamic Compilation - I



- **Dynamo**: Transparent Optimization System
- **Jrpm**: Dynamically Parallelizing Java Programs

Navneet Aron
Sorav Bansal

Potential Benefits of Dynamic Compilation

- Optimize legacy binaries & DLLs
- Use run-time info
- Optimize programs compiled for debugging
- Optimize for specific memory system
- Architectures do not need to worry about binary compatibility
- Reduced Hardware Complexity
- Virtual IT shop.

Hard Problems

- Self Modifying Code
- Precise Exceptions
- Address Translation
- Self Referential code
- Management of Translation
- Real-Time Behavior
- Boot and BIOS Code
- Reliability and Correctness

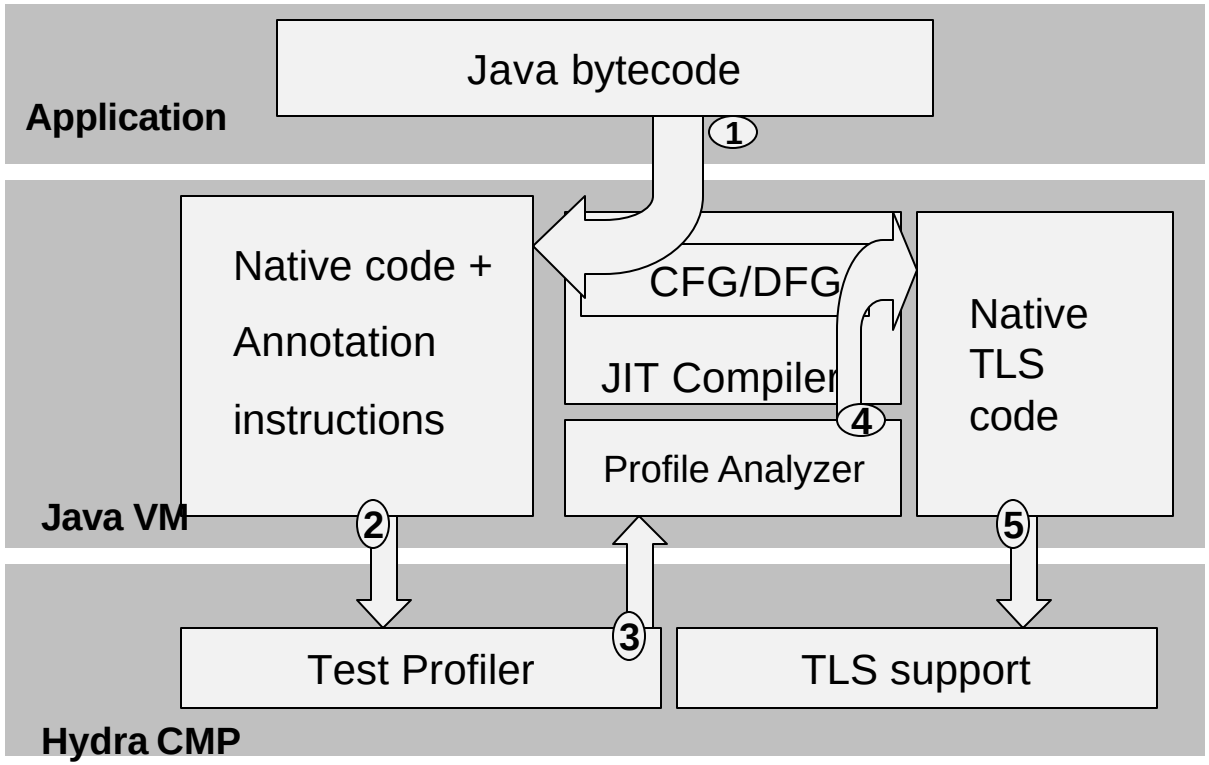
Different types of Dynamic Compilation

- Interprating vs translating/optimizing
- Emulating a real machine vs virtual machine
- Full system vs user mode only
- OS dependent vs OS independent
- Translating to a different architecture vs same architecture
- Emulating a single source architecture vs multiple source architectures

Negative effects of Dynamic Compilation

- Take away cycles
- Take away memory & other resources
- It can be slow especially during startup
- Debugging can be difficult

JRPM: Block Diagram



Components of JRPM System

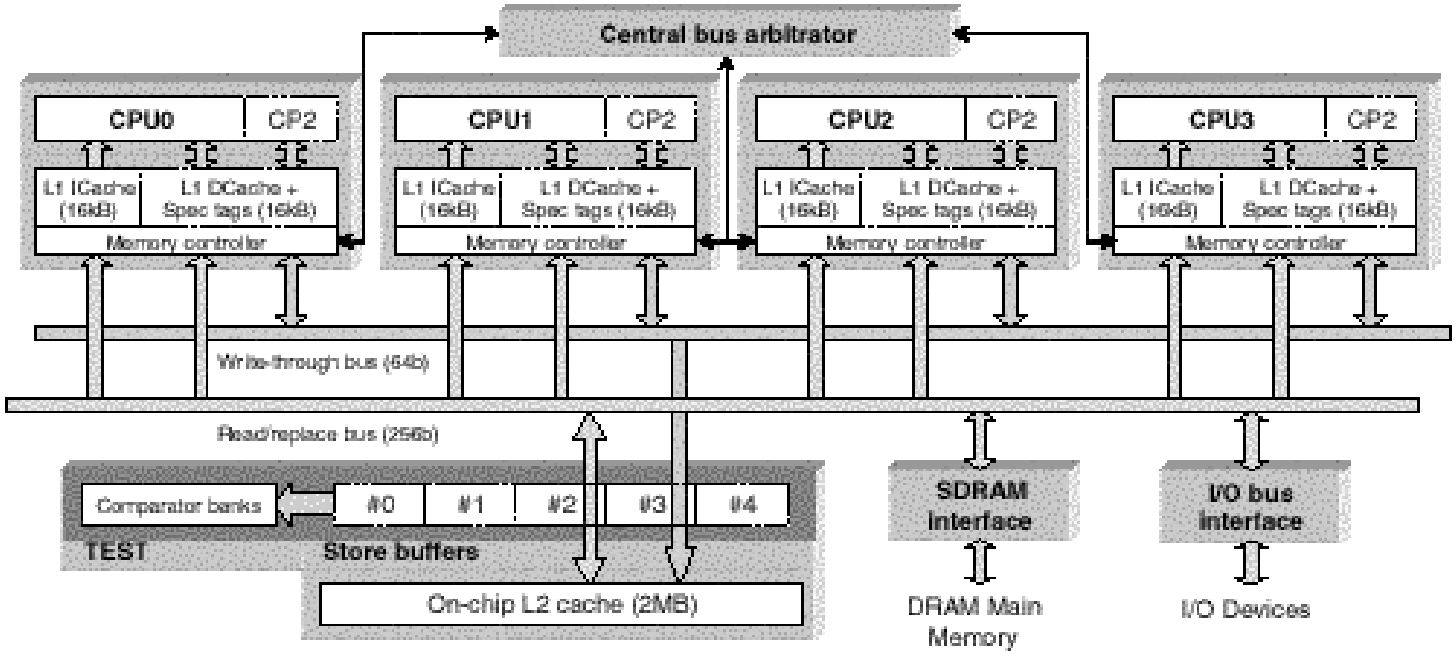
Components of JRPM

- Hydra:
  - Smaller inter process communication overheads than traditional multiprocessors
- TLS:
  - Allows division of sequential program into threads which can be executed in parallel
  - Hardware guarantees memory order of sequential program.

Components (contd..)

- TEST (profiler):
  - Hardware for analyzing sequential program execution in real time find regions to parallelize
  - Provides dynamic dependency, thread size, buffering requirement estimates
- Virtual Machine:
  - Hides dynamic analysis and thread level speculation

Hydra CMP



Buffers	Per-thread limit	Associativity
Load buffer	16kB (512 lines x 32B)	4-way
Store buffer	2kB (64 lines x 32B)	Fully

Access	Latency (cycles)
L2	5
Interprocessor	10
Main memory	50

Figure 2 – Block diagram of our CMP. TLS support blocks shown in dotted lines. TEST profile hardware blocks shown in dark blocks.

Implementing TLS

- Dependencies
  - RAW
    - Forwarding (data to sequentially later threads)
    - Restarting in case of violation
  - WAR
    - Buffering & committing speculative writes in program order
  - WAW
    - Buffered writes are available only to sequentially later threads
- Hardware support : Coprocessor, speculative tag bits, secondary cache store buffers

Selecting Threads

- Each loop iteration is one thread
- Loop Size
  - Large loops:
    - Better speculation
    - Lesser overhead
  - Small loops:
    - Lower penalties for RAW violation
    - Lower probability of speculation buffer overflow
- Load Dependency analysis
- State Overflow Analysis
- Hardware Profiling support (TEST)

Compiling Issues

- Developed MicroJIT (30% faster code generation) as compared to Sun-client Dynamic Compiler
  - Interleave compilation stages
  - Minimizing compiler passes
- Control routines inserted into STL.
  - STL_STARTUP
  - STL_SHUTDOWN
  - STL_EOI
  - STL_RESTART

Optimizations for Speculation

- Loop invariant register allocation
- Non communicating loop inductors
- Reset able non-communicating loop inductors
- Thread Synchronizing lock
- Reduction operator optimization
- Multilevel STL decomposition
- Hoisted startup and shutdown routines.

Virtual Machine Considerations

- Exception Handling
  - Only real exceptions are handled-So speculative threads must wait till they become head thread.
- Garbage Collection
  - Mark &Sweep, Free list of unallocated objects
  - parallelized access to allocator- several free list
- Synchronized Objects
  - Avoid serialization during speculative execution

Results

- Significant speedups for integer (1.5-2.5), fp(3-4), multimedia applications(2-3).
- Diversity in coverage of selected STL.
- Cannot parallelize all programs(System calls in critical code)
- STL selection based on input Data Sets
- Good speculative performance for selected STLs
- Low overhead of profiling & dynamic recompilation

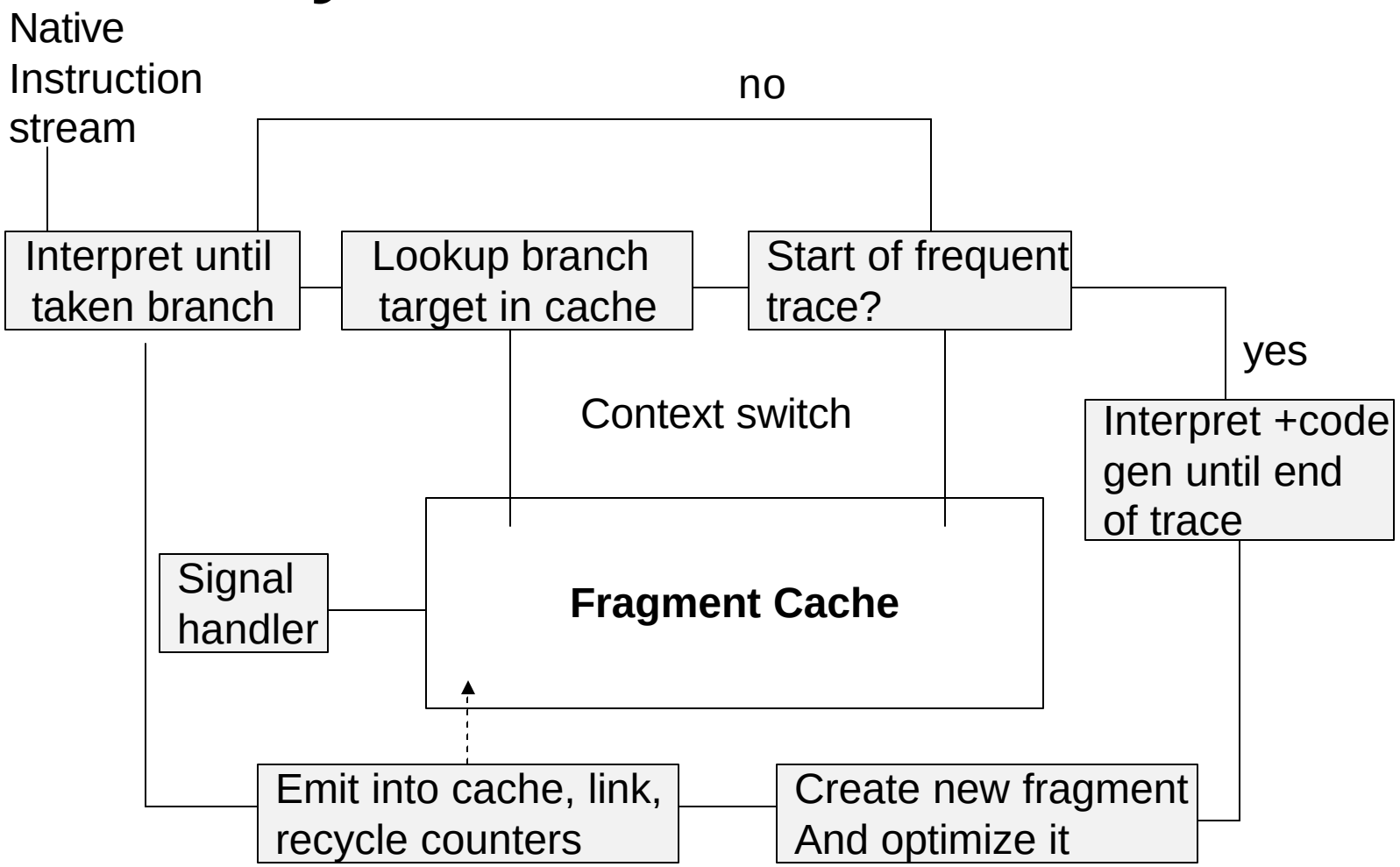
Critique

- Substantial performance benefits for programs that have loops or can be transformed into loops.
- No discussion of performance of applications that cannot be transformed into loops.
- Manual transformations (some times significant)-require high level understanding of program.
- No results where selected STLs were dependent on the input data set

Dynamo

- Transparent
  - Can even work on legacy binaries (requires no user/compiler support)
  - Requires no special OS/hardware support
- Dynamic
  - Uses runtime information leading to a more effective solution
  - -O2 + Dynamo ? -O4

How Dynamo Works



Targets

- Runtime opportunities
  - Redundancies across cross static program boundaries. Eg. Procedure calls
  - Instruction cache utilization
- Trace Optimization Opportunities
  - Conventional Optimizations (copy prop, …)
  - Partial Redundancies in native code become full redundancies in join-free code
  - Compensation blocks

Operations

- Trace Selection
  - Just profile *tails* to reduce overhead
  - No profiling for cached code
- Trace Optimization
  - Redundant/Predictable Direct/Indirect branches. Improve I-cache utilization
  - “Join-Free” Trace allows aggressive optimizations
- Fragment code generation and linking
- Fragment Cache Management
  - Need the fragment cache to be small. Hence, need replacement algorithms

Signal Handling

- Asynchronous signals
  - Queued till next fragment exit
  - Exit stubs are generated for loop exits to allow quick exit
- Synchronous signals
  - Either, suppress code-removing and reordering transformations *(too drastic)*
  - On receiving signal, De-optimize fragment code by attempting to reconstruct signal context
  - Split optimizations into categories: *conservative* and *aggressive*

Overheads

- Startup cost due to interpretation
  - Higher interpretive overhead ? need quicker trace prediction ? more number of traces ? larger fragment cache
- Counter updates and storage. Leads to increased memory footprint
- Linking fragments expensive
- Redundant copies of code
- Cache Replacement

Critique

- Demonstrate the limits of the system
- Don’t test it on different machines. Performance heavily depends on:
  - Branch misprediction penalty
  - I-cache size, TLB size
- Cache Replacement not scan-resistant (*go, vortex*). Also not good for small programs.
- Show wall-clock times. Do not give information about times spent in each module
- Take an approach and stick with it. Do not compare it with other possible approaches (e.g. Cache Replacement)

Discussion Topics

- Example Applications?
  - Interpretation vs Translation/optimizing
  - Emulating a real machine vs emulating a virtual machine
  - Full system vs user mode only
  - Translating to a different architecture vs same architecture

Discussion Questions

- Profiling
  - **What to profile?**
  - **How to profile?**
  - **How much data to collect before choosing Traces/STLs?**
- What hardware support can you provide to make Dynamo work better ?
- What information can you obtain from the compiler, user in case of Dynamo ?

Discussion Questions

- What other profiling information would you need (say in context of chip multiprocessors)
- What opportunities exist in the context of polymorphic architectures ?
- When can we do away with the overhead of interpretation?

Discussion Questions (JRPM)

- When do we need TLS? What characteristics should an application have?
- How do we extend JRPM to programs with TLP other than loops?
- Would superscalar/vliw cores instead of single issue have better performance ?
- How well does the JRPM system scale?