MANCHESTER 1824

The University of Manchester

Dynamic Binary Translation: from Dynamite to Java

ICSA Colloquium, Edinburgh, University
April 10th

Dr. Ian Rogers,
Research Fellow,
The University of Manchester
ian.rogers@manchester.ac.uk

MANCHESTER 1824

The University of Manchester

Presentation Outline

- A brief history of binary translation
- The Dynamite project
  - A look inside the compiler
- A fresh approach (Java with everything)
- Some tales of sorrow

A Brief History of Binary Translation

- Late 1960s and 1970s assembler to assembler translators and microprogramming in the interest of not having to rewrite code and for fast simulation
- 1980s, financial incentive to run other architectures machine code recognized:
  - 1987 HP Object Code Translator – MPE V binaries to PA-RISC MPE XL
  - 1988 AT&T Flashport, static translation of many architectures (e.g. 680X0, IBM 360, PDP11) to many (e.g. PowerPC, SPARC, PA-RISC, IA32)

A Brief History of Binary Translation (continued)

- Early 1990s continued to recognize cost saving in translating rather than porting using static translation:
  - Accelerator - TNS CISC to TNS/R
  - VEST & TIE – VAX VMS to Alpha VMS
  - mx & mxr – MIPS to Alpha
- Runtime environments emerging to interpret in the cases where translations weren't present:
  - Mae – Mac emulator for
  - Executor & Syn68k

A Brief History of Binary Translation (continued 2)

- Mid-1990s saw dynamic binary translation emerging to speed up slow interpreters in runtime environments:
  - SoftWindows, RealPC – run Windows on Mac
  - FX!32 – run x86 Windows NT binaries on Alpha Windows NT
  - DAISY – PowerPC to VLIW PowerPC
  - Wabi – IA32 Windows to SPARC

A Brief History of Binary Translation (continued 3)

- The emergence of open source:
  - Bochs
  - Wine
  - QEMU
  - PearPC
- 2000s, the emergence of virtualization companies
  - VMWare
  - Transmeta
  - Transitive Technologies

A Brief History of Binary Translation (continued 4)

- Mid-2000s
  - Rosetta shipped with Apple Intel Macs, Lx86 shipped with IBM PowerVM
  - Virtualization a hot-topic
  - QEMU becomes integrated with the Linux kernel, QEMU drivers used to virtualize and migrate Linux
  - VMWare floats
  - Interest in binary translation as part of a bigger platform
    - PearColator, VEELS, JPC – leverage Java
    - CLR and LLVM related projects too

A Brief History of Binary Translation (continued 5)

- In parallel binary translators became an important tool in simulation:
  - atom/shade
- Security analysis of compiled software
  - Valgrind
- Instrumentation and profiling
  - Dynamo RIO

Dynamite

- Research project started in 1995
  - Inspired by the use of Shade for cache simulations
  - Cristina Cifuentes (UQBT and UQDBT) visited Manchester 1995
- Aim to create framework for dynamic binary translation
- Caught up in dot-com boom, Transitive launched in 2000

Dynamite - backend

- Originally used Dawson Engler's vcode
  - Emit statements like RISC code
  - Porting means implementing emit statements for new architecture
- Dynamite tcode backend designed to handle x86 as a host architecture

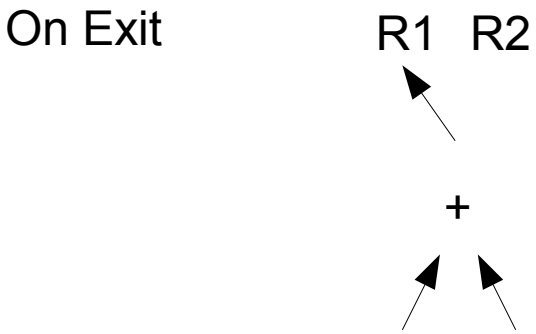
Dynamite – intermediate form

- Basic block based
- Basic blocks identified by an address and lazy evaluation state
- Translator can assume that values in emulated registers are those that triggered translation
  - Possibility of value specialization optimizations

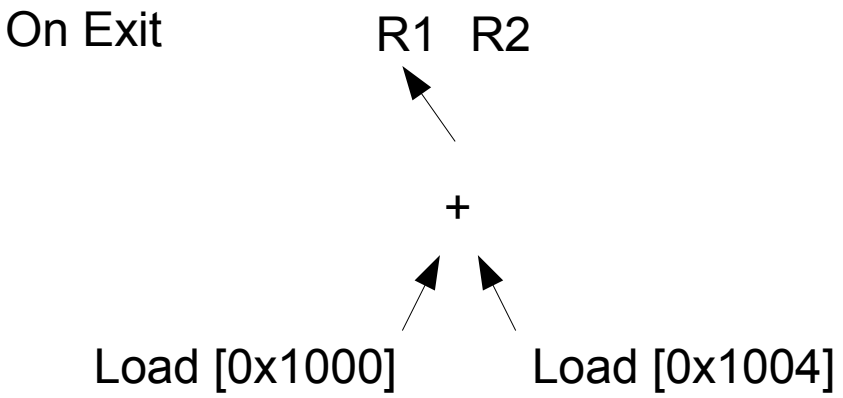
Dynamite – intermediate form (continued)

- Instructions translated to produce DAGs

Add r1, r1, r2



Load r1, [0x1000]
Load r2, [0x1004]
Add r1, r1, r2



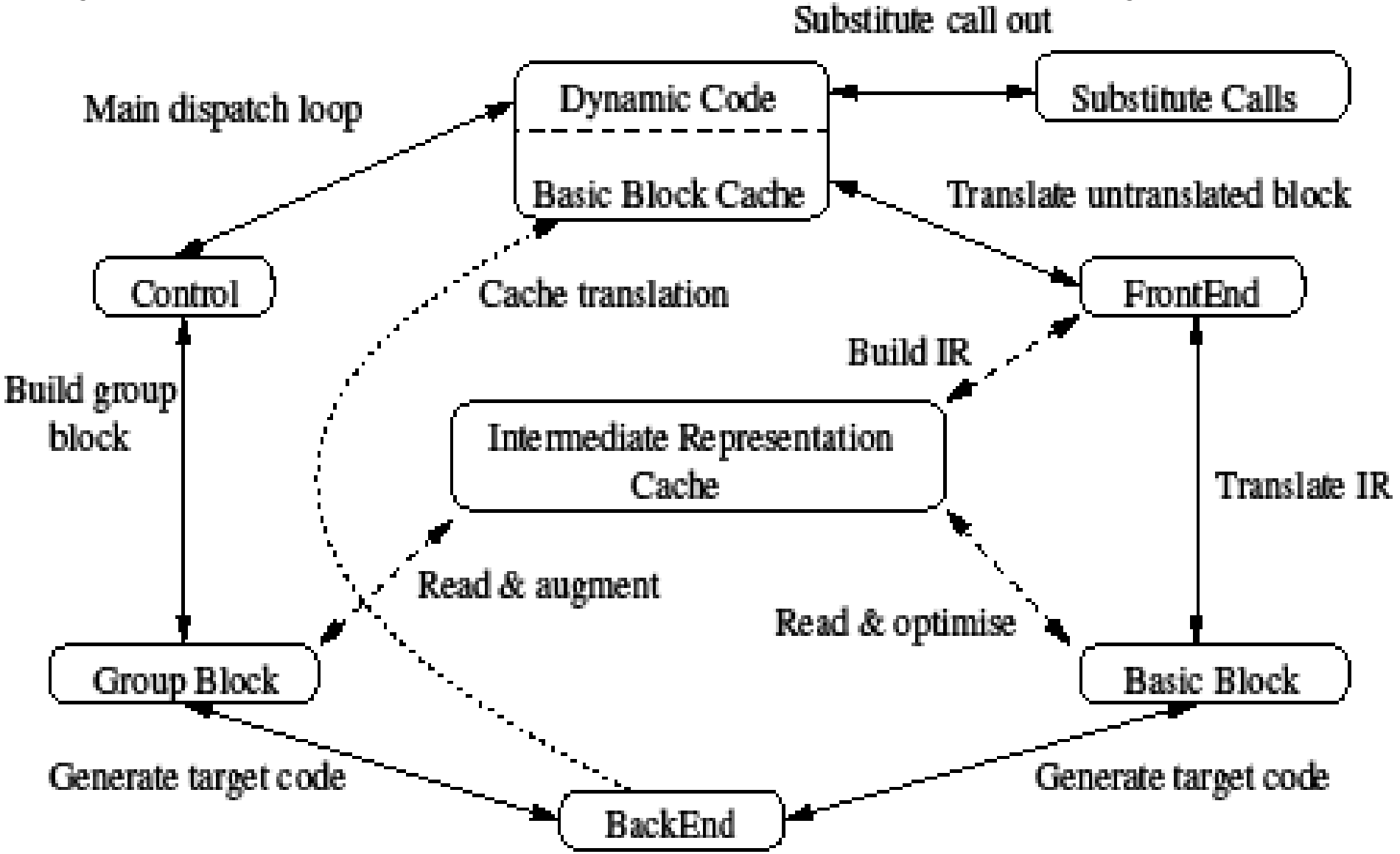
Dynamite – intermediate form (continued 2)

- Instructions emitted by traversing tree for each register
- When stores were present, their creation order was recorded and they were generated in order (could be relaxed in some cases)
- Register allocation was greedy, any spilt registers were placed in the memory version of the emulated register
- Howson found that breadth-first traversal of trees produced slightly better code for VLIW architectures

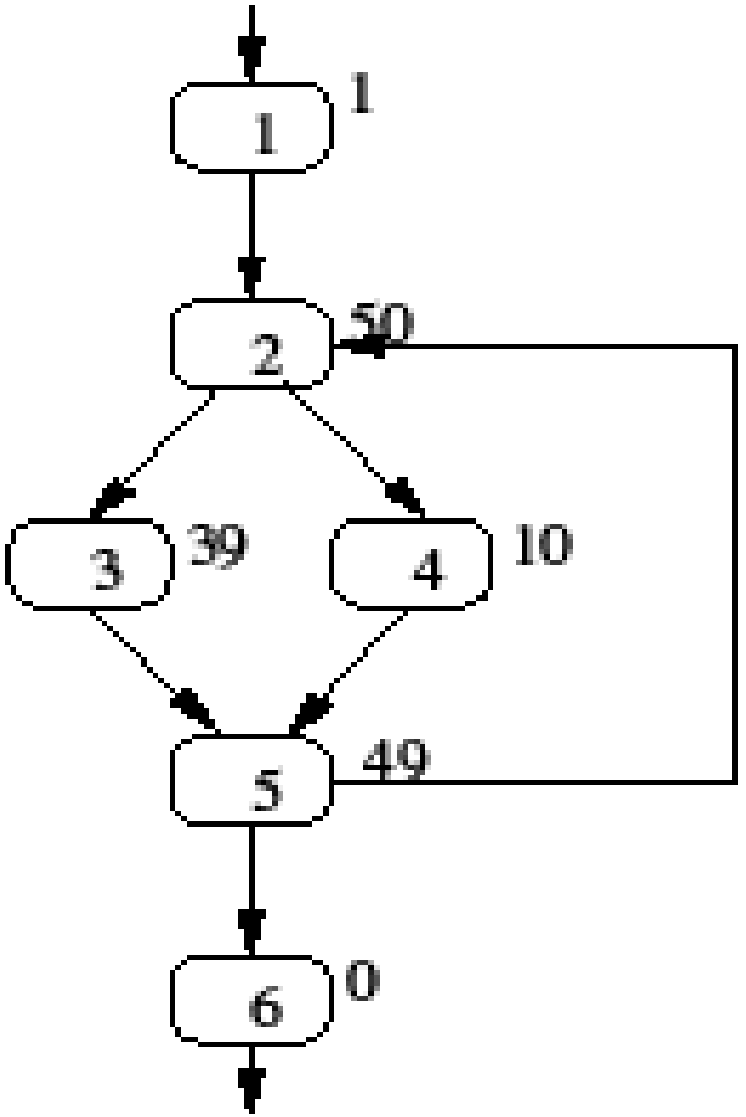
MANCHESTER 1824

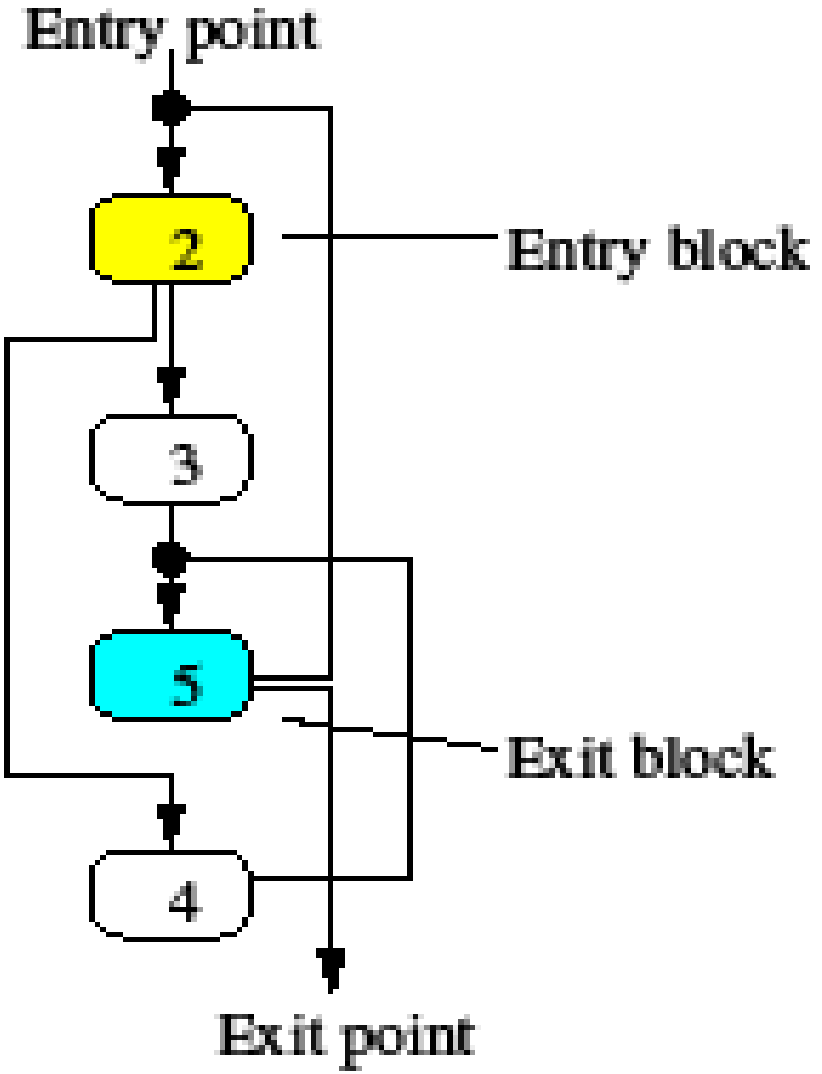
The University of Manchester

Dynamite – adaptive optimization system



Dynamite – group block creation





Control-flow graph	Group block

Dynamite – frontends

- In Dynamite a frontend performs the hard work of maintaining the memory layout, handling system calls, linking and loading, and translating instructions
- Other than code placement and constant propagation, no optimizations are performed on the IR
- The frontend also performs the main optimizations using lazy evaluation

Dynamite – lazy evaluation

- Lazy evaluation information is held per basic block
- Used to perform ahead of time liveness analysis
- Most architectures have flags that are set by instructions but most commonly destroyed by some subsequent instruction
- On x86 you can have 8, 16, 32 and 64bit views of the same register, but only 1 view will be active. For example, there are architectural penalties to writing an 8bit register and then reading it as a 32bit register.

Lazy evaluation

- On entry record assumptions
  - e.g. For x86 start by assuming only 32bit registers are defined
- Generate code recording changes to assumptions
  - e.g. Definition of an 8bit register will mean that both the 32 and 8bit register are live
- When generating code use recorded assumptions to guide which instructions to generate
  - e.g. If 8 and 32bit version of a register are live, combine them prior to use
- Make exit assumptions available for subsequent code

Lazy evaluation (continued)

- For flags, rather than set the flags make copies of the operands that define and record the instruction in the lazy state
- When generating a branch, or predicated instruction, use the lazy state to
  - If the flag isn't lazy - directly generate code to access the flag
  - If the flag is lazy then generate the setting instruction and branch together

Lazy evaluation (continued 2)

- Lazy evaluation wins because most flag values are never read or are killed by subsequent instructions
- Lazy evaluation loses because it increases register pressure, peels loops, causes code bloat

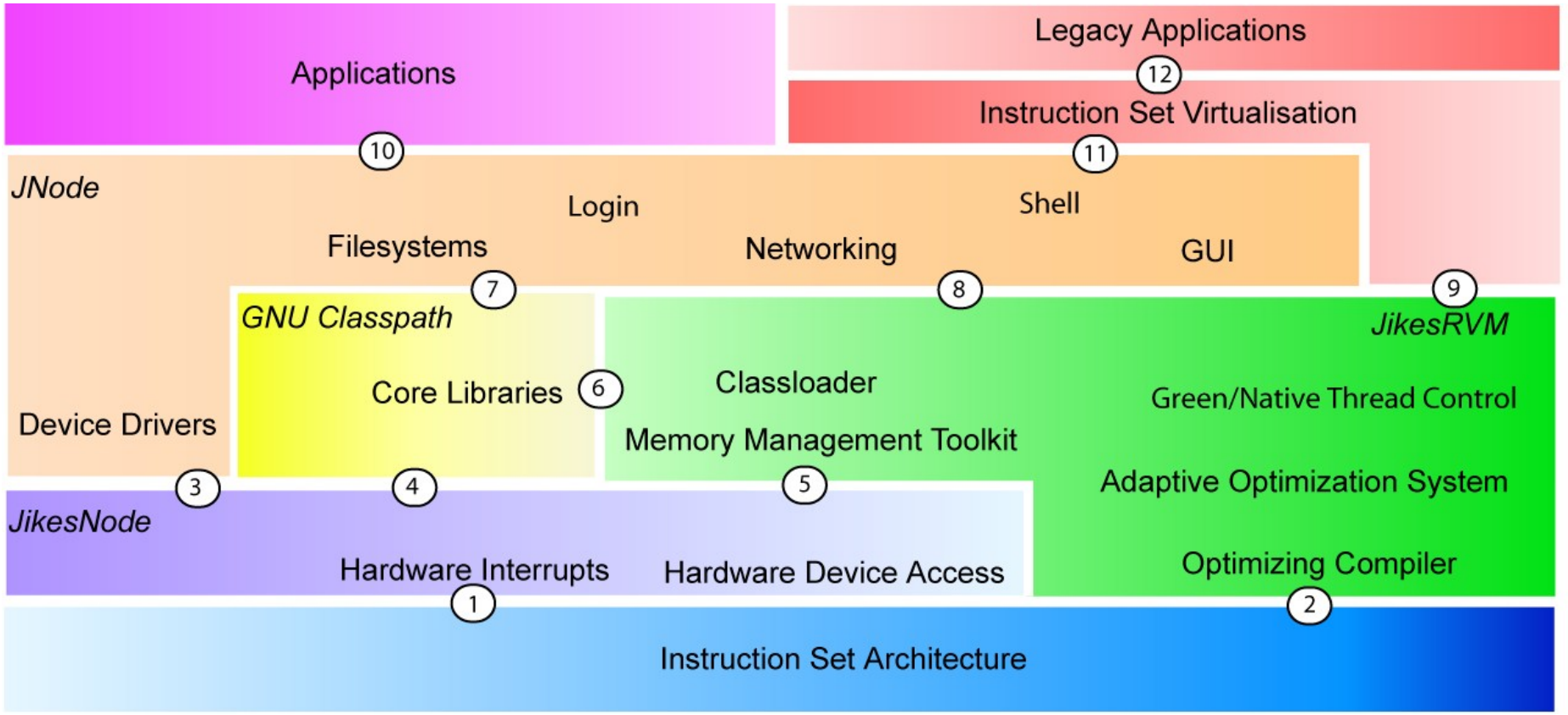
Jamaica Project

- Parallel hardware with lightweight threading
- Parallelizing compiler, small threads created that exploited lightweight infrastructure
- Cost effective to run 100s of instructions runnable in parallel

MANCHESTER 1824

The University of Manchester

Jamaica Systems Software Overview



Legacy Emulation Layer - PearColator

- Performance of code generated by JVM optimizing compiler is much better than Dynamite, at least:
  - Instruction selection
  - Register allocation
- Why not emit Java bytecodes?

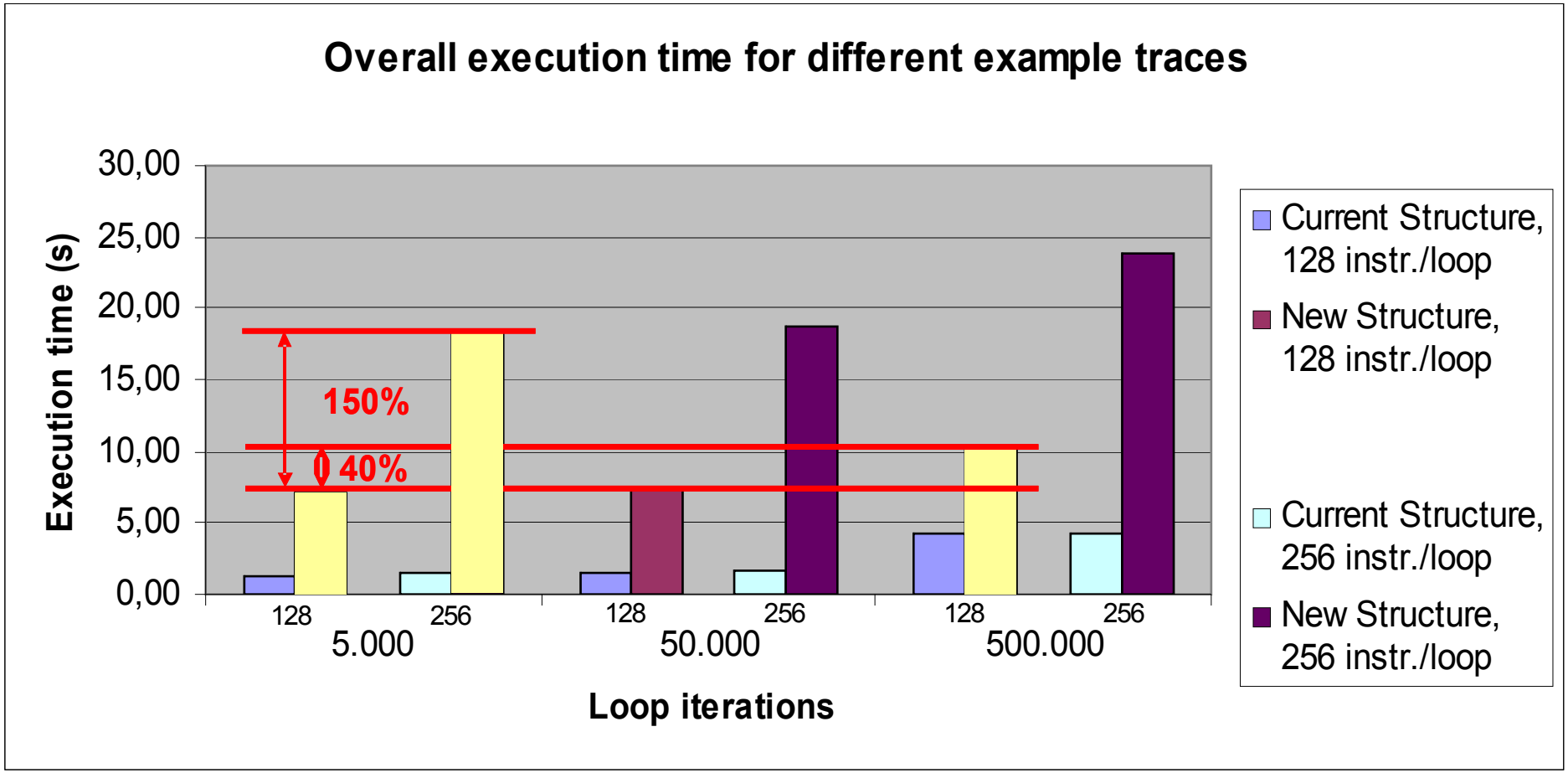
Why not emit Java bytecodes?

- QEMU translates code by memcpy-ing regions of GCC compiled code into consecutive memory addresses
- Performing the same for Java would be more portable and enable the creation of single code for both interpretation and compilation
- But...

MANCHESTER 1824

The University of Manchester

Why not emit Java bytecodes? (continued)



The overhead in executing more loop iterations is small

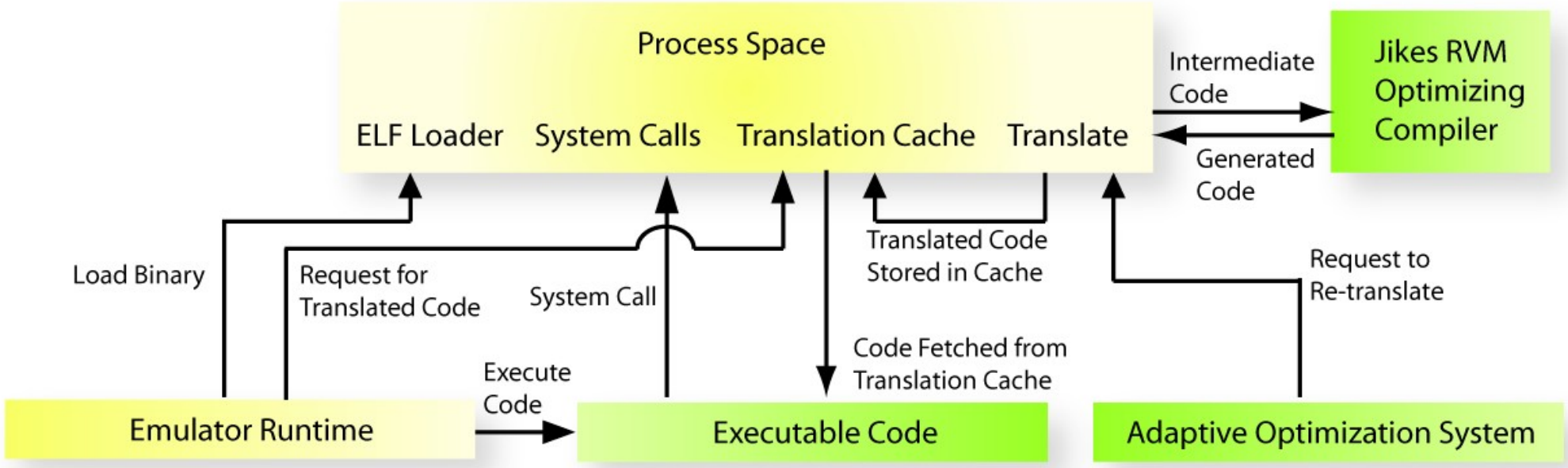
100 times more loops equals 40% slower

The overhead in translating twice as much code using bytecodes is 150% slower, directly generating IR incurs a very small cost

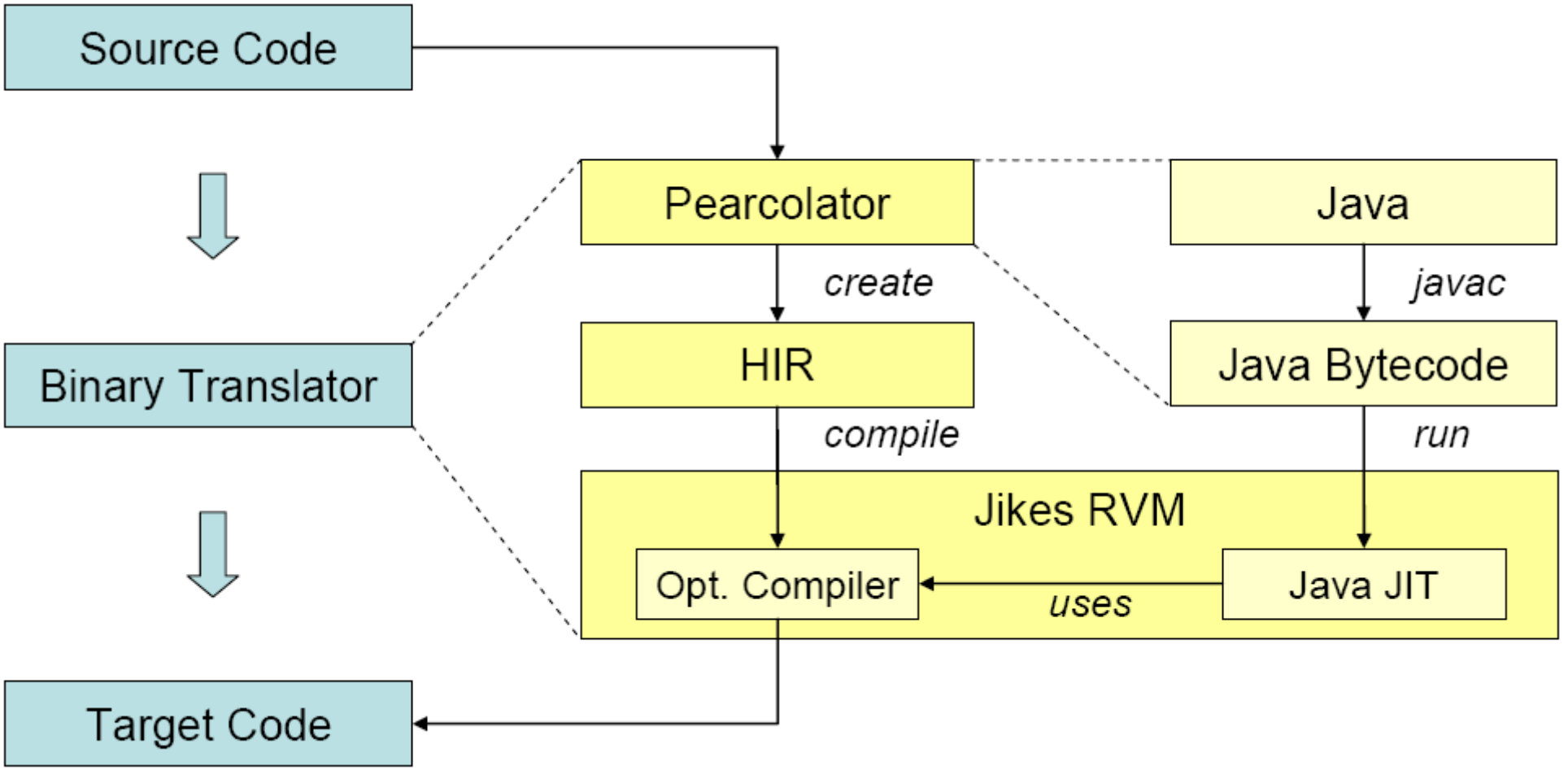
MANCHESTER 1824

The University of Manchester

PearColator

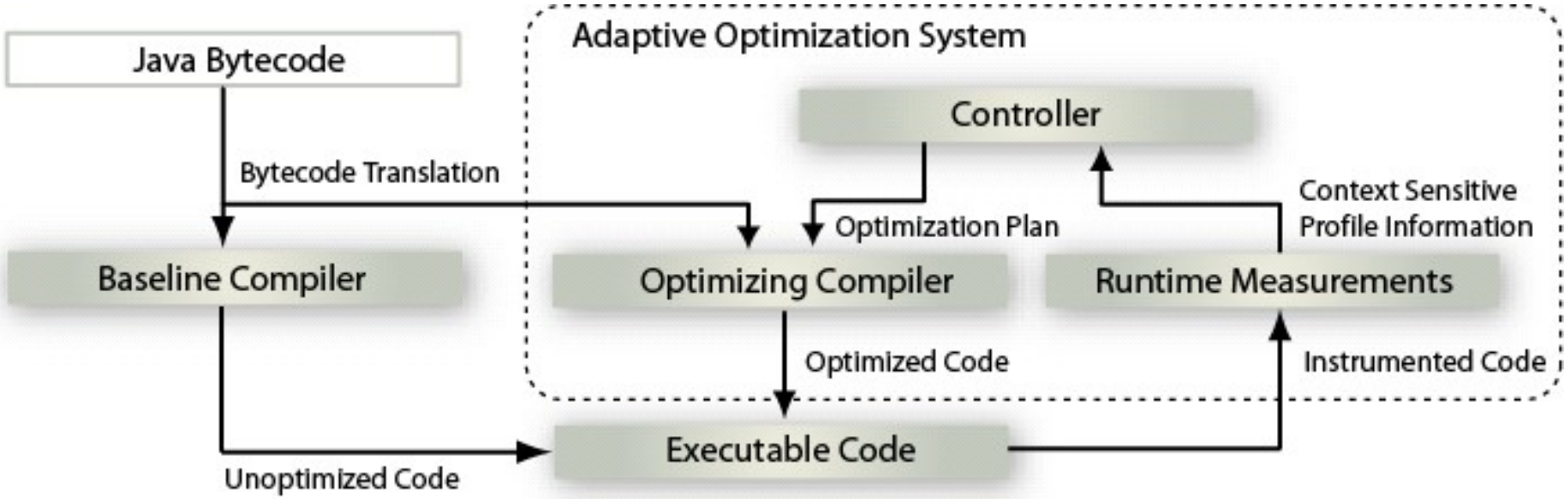


PearColator Overview (continued)



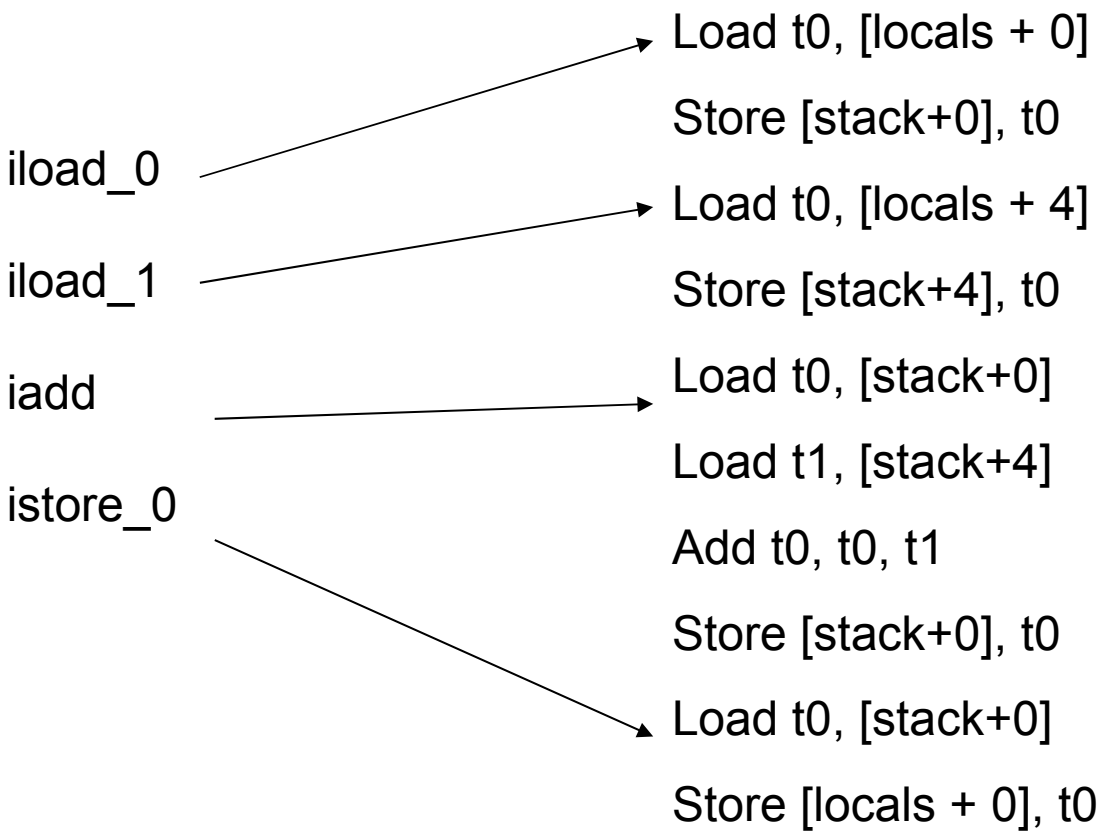
The Jikes RVM

- Overview of the adaptive compilation system:



The baseline compiler

- Used to compile code the first time it's invoked
- Very simple code generation:



The baseline compiler

- Pros:
  - Easy to port – just write emit code for each bytecode
  - Minimal work needed to port runtime and garbage collector
  - Fast compilation rate
- Cons:
  - Produces slow code

The boot image

- Hijack the view of memory (mapping of objects to addresses)
- Compile list of primordial classes
- Write view of memory to disk (the boot image)
- The boot image runner loads the disk image and branches into the code block for VM.boot

The boot image

- Problems:
  - Difference of views between:
    - Jikes RVM
    - Classpath
    - Bootstrap JVM
  - Fix by writing null to some fields
  - Fix other fields by using “Oracle” to lookup field values
  - Jikes RVM runtime needs to keep pace with Classpath

The runtime

- M-to-N threading
  - Thread yields are GC points
  - Native code can deadlock the VM
  - Refactored thread system means native threading implementation can be plugged in!
- JNI written in Java with knowledge of C layout
- Classpath interface written in Java

The optimizing compiler

- Structured from compiler phases based on HIR, LIR and MIR phases from Muchnick
- IR object holds instructions in linked lists in a control flow graph
- Instructions are an object with:
  - One operator
  - Variable number of use operands
  - Variable number of def operands
  - Support for def/use operands
- Some operands and operators are virtual

The optimizing compiler

- HIR:
  - Infinite registers
  - Operators correspond to bytecodes
  - SSA phase performed
- LIR:
  - Load/store operators
  - Java specific operators expanded
  - GC barrier operators
  - SSA phase performed
- MIR:
  - Fixed number of registers
  - Machine operators

The optimizing compiler

- Factored control graph:
  - Don’t terminate blocks on Potentially Exceptioning Instructions (PEIs)
    - Bound check
    - Null check
  - Checks define guards which are used by:
    - Putfield, getfield, array load/store, invokevirtual
  - Eliminating guards requires propagation of use

The optimizing compiler

- Java – can we capture and benefit from strong type information?
- Extended Array SSA:
  - Single assignment
  - Array – Fortran style - a float and an int array can't alias
  - Extended – different fields and different objects can't alias
- Phi operator – for registers, heaps and exceptions
- Pi operator – define points where knowledge of a variable is exposed. E.g. A = new int[100], later uses of A can know the array length is 100 (ABCD)

The optimizing compiler

- HIR: Simplification, tail recursion elimination, estimate execution frequencies, loop unrolling, branch optimizations, (simple) escape analysis, local copy and constant propagation, local common sub-expression elimination, local expression folding
- SSA in HIR: load/store elimination, redundant branch elimination, global constant propagation, loop versioning, expression folding
- AOS framework

The optimizing compiler

- LIR: Simplification, estimate execution frequencies, basic block reordering, branch optimizations, (simple) escape analysis, local copy and constant propagation, local common sub-expression elimination, local expression folding
- SSA in LIR: global code placement, live range splitting
- AOS framework

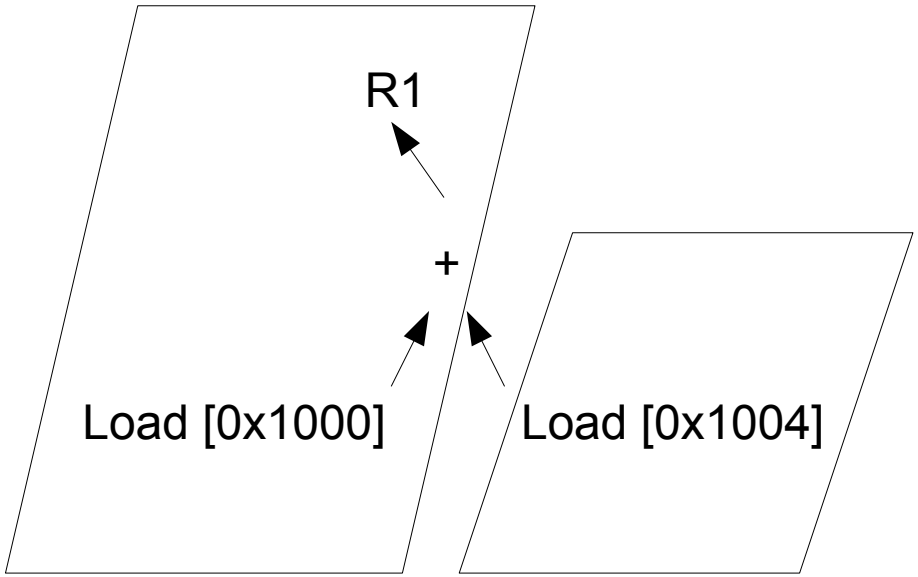
The optimizing compiler

- MIR: instruction selection, register allocation, scheduling, simplification, branch optimizations

- Fix-ups for runtime

Instruction Selection

- Bottom-Up Rewrite System
- Consider different coverings over DAG
- Choose least cost cover



Speculative Optimisations

- Often in a JVM there's potentially not a complete picture, in particular for dynamic class loading
- On-stack replacement allows optimisation to proceed with a get out clause
- On-stack replacement is a virtual Jikes RVM instruction

MANCHESTER 1824

The University of Manchester

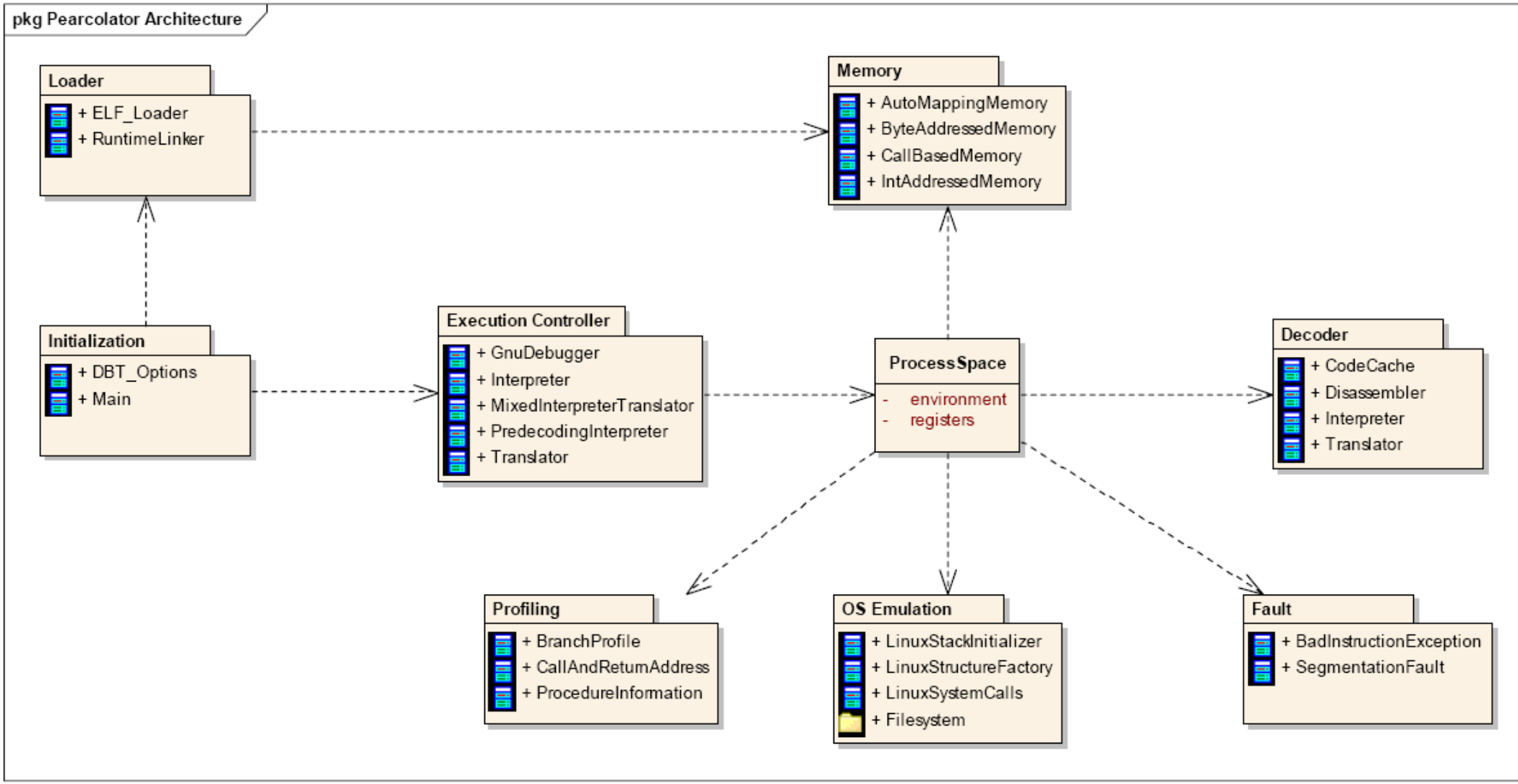
Applications of on-stack replacement

- Safe invalidation for speculative optimisation
  - Class hierarchy-based inlining
  - Deferred compilation
    - Don’t compile uncommon cases
    - Improve dataflow optimization and improve compile time
- Debug optimised code via dynamic deoptimisaton
  - At break-point, deoptimize activation to recover program state
- Runtime optimization of long-running activities
  - Promote long-running loops to higher optimisation levels

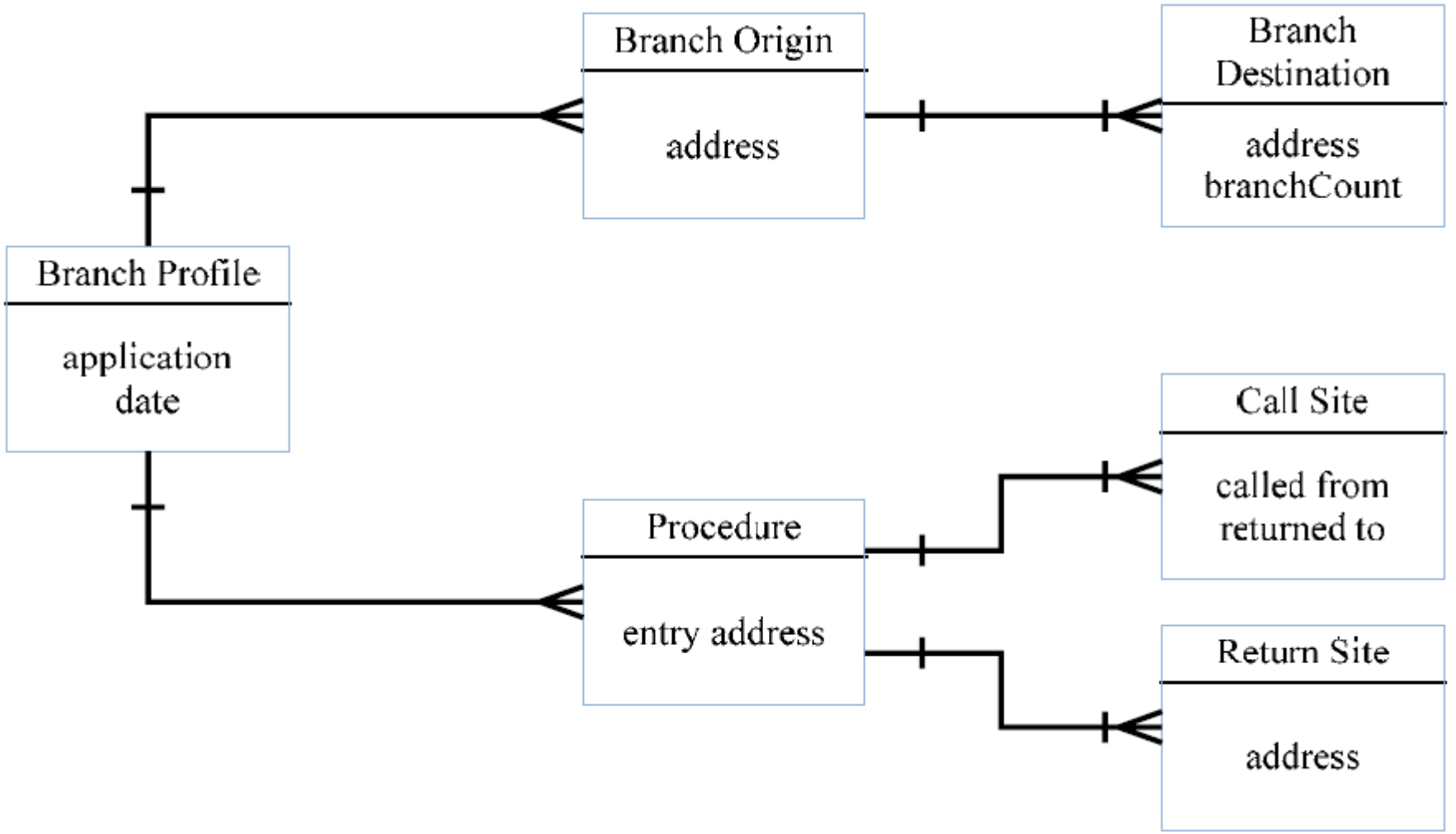
MANCHESTER 1824

University of Manchester

PearColator Overview (continued 2)



Profiling



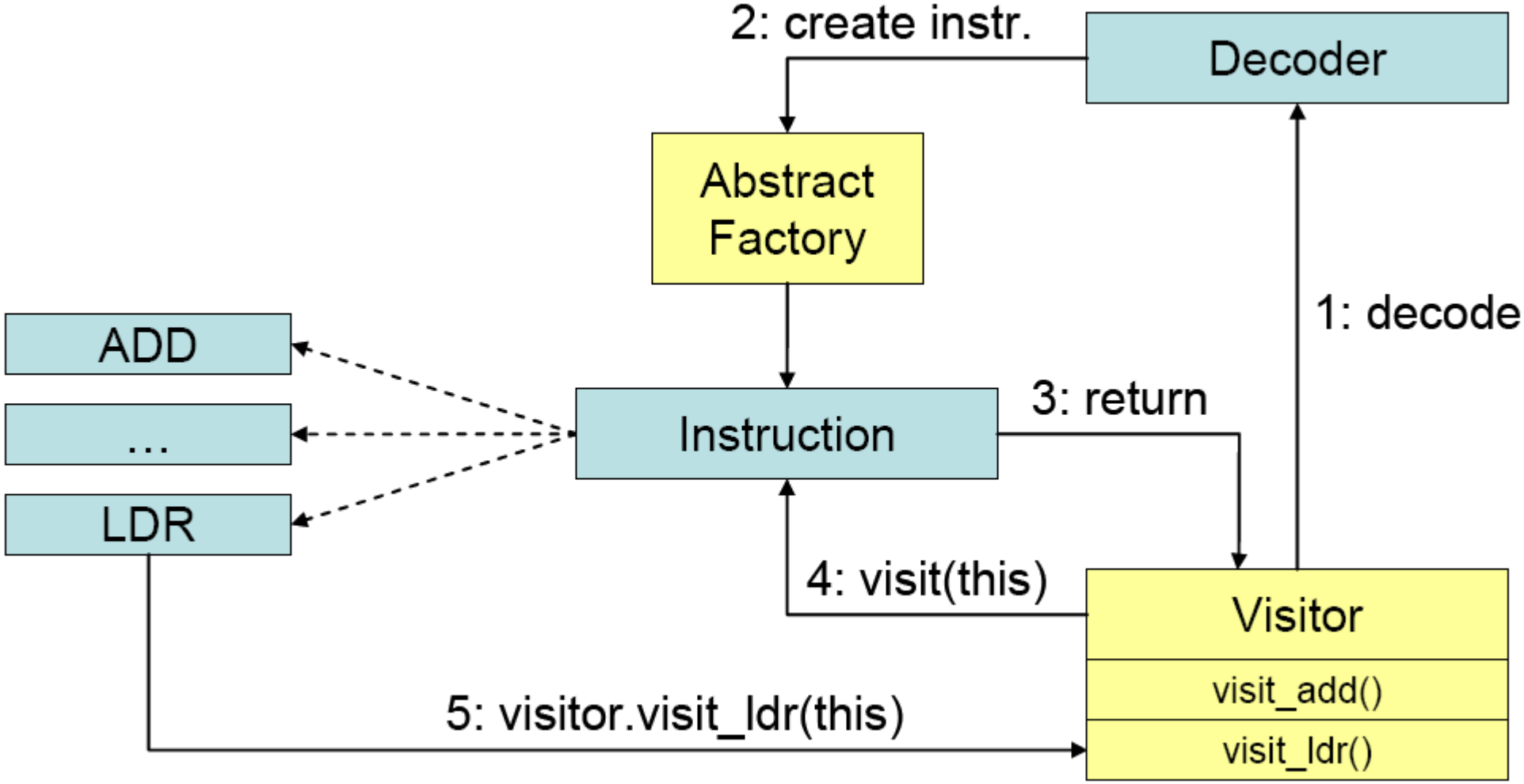
Profiling (continued)

- Generate information on indirect branch targets
- Use branch and link instructions to approximate function boundaries
- Use profile information to build up accurate trace for a function
- For hot functions consider inlining

Trace formation

- Traces start at beginning of a function
- Compilation is concurrent with interpretation of code
- Switch to trace execution on function calls
  - on-stack replacement not yet implemented
- Lazy information recorded per instruction in the trace, bigger basic blocks formed by merging instructions after initial translation

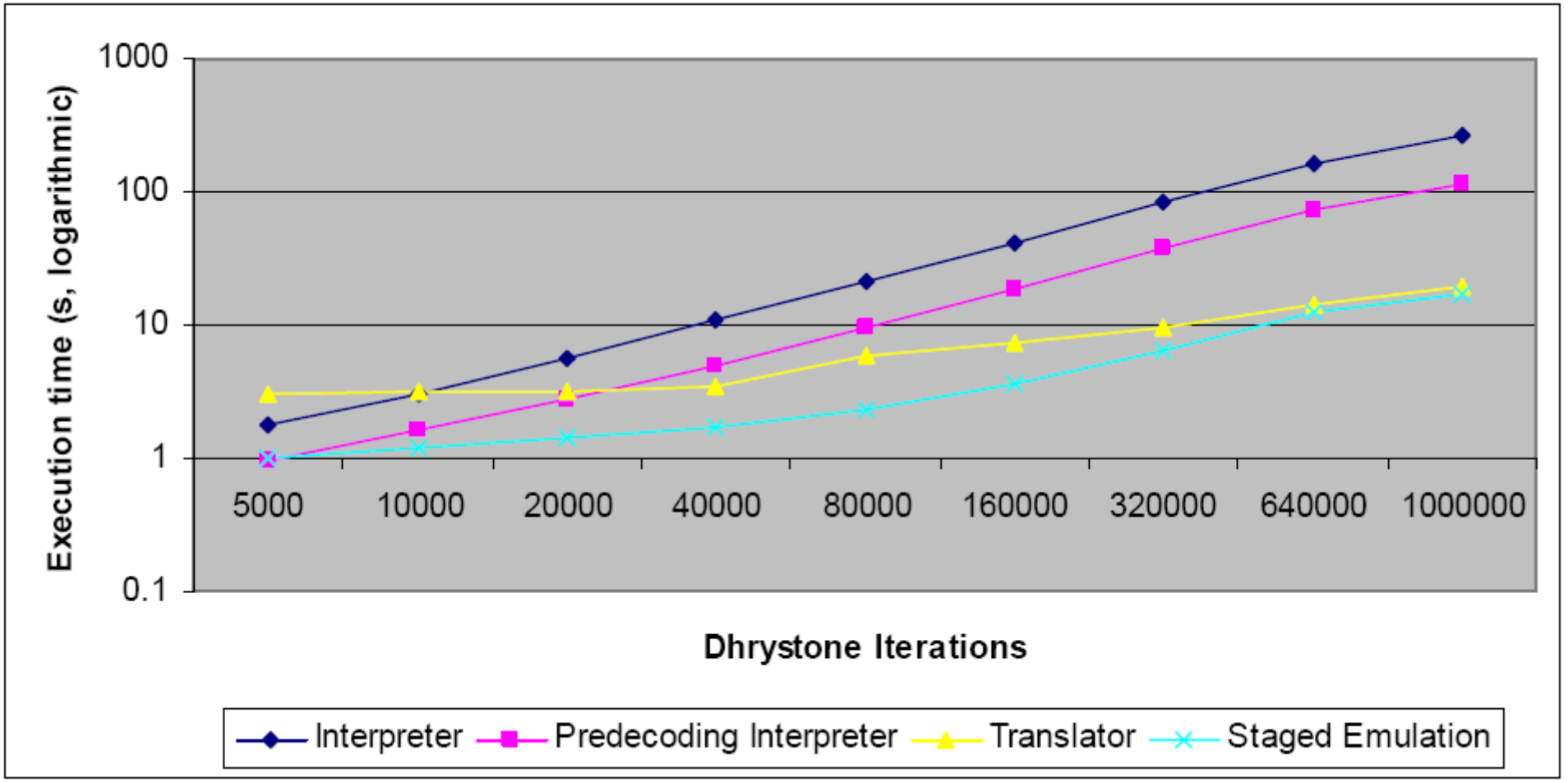
Decoders



MANCHESTER 1824

The University of Manchester

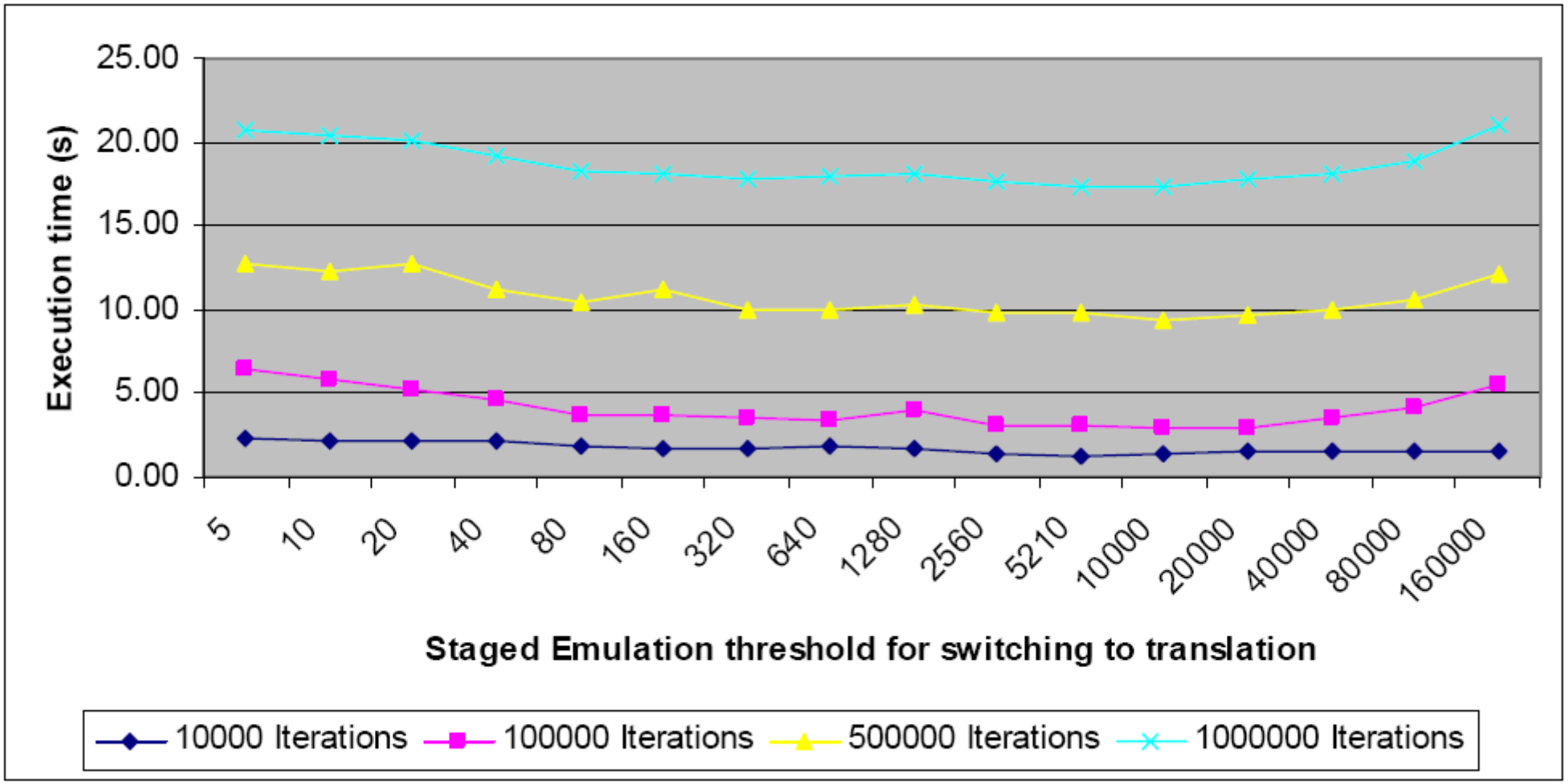
Execution Models



MANCHESTER 1824

The University of Manchester

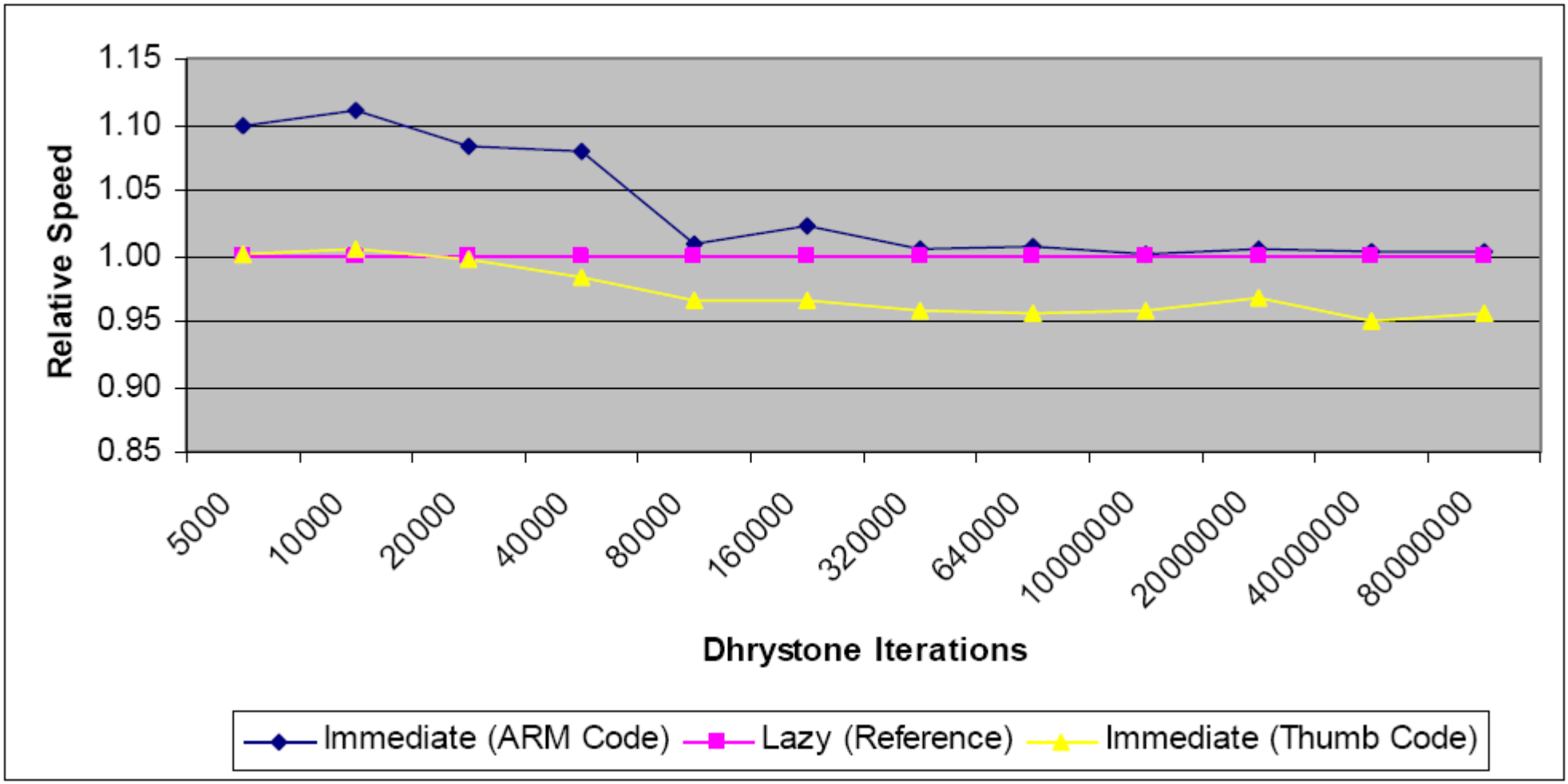
Staged Execution Threshold



MANCHESTER 1824

The University of Manchester

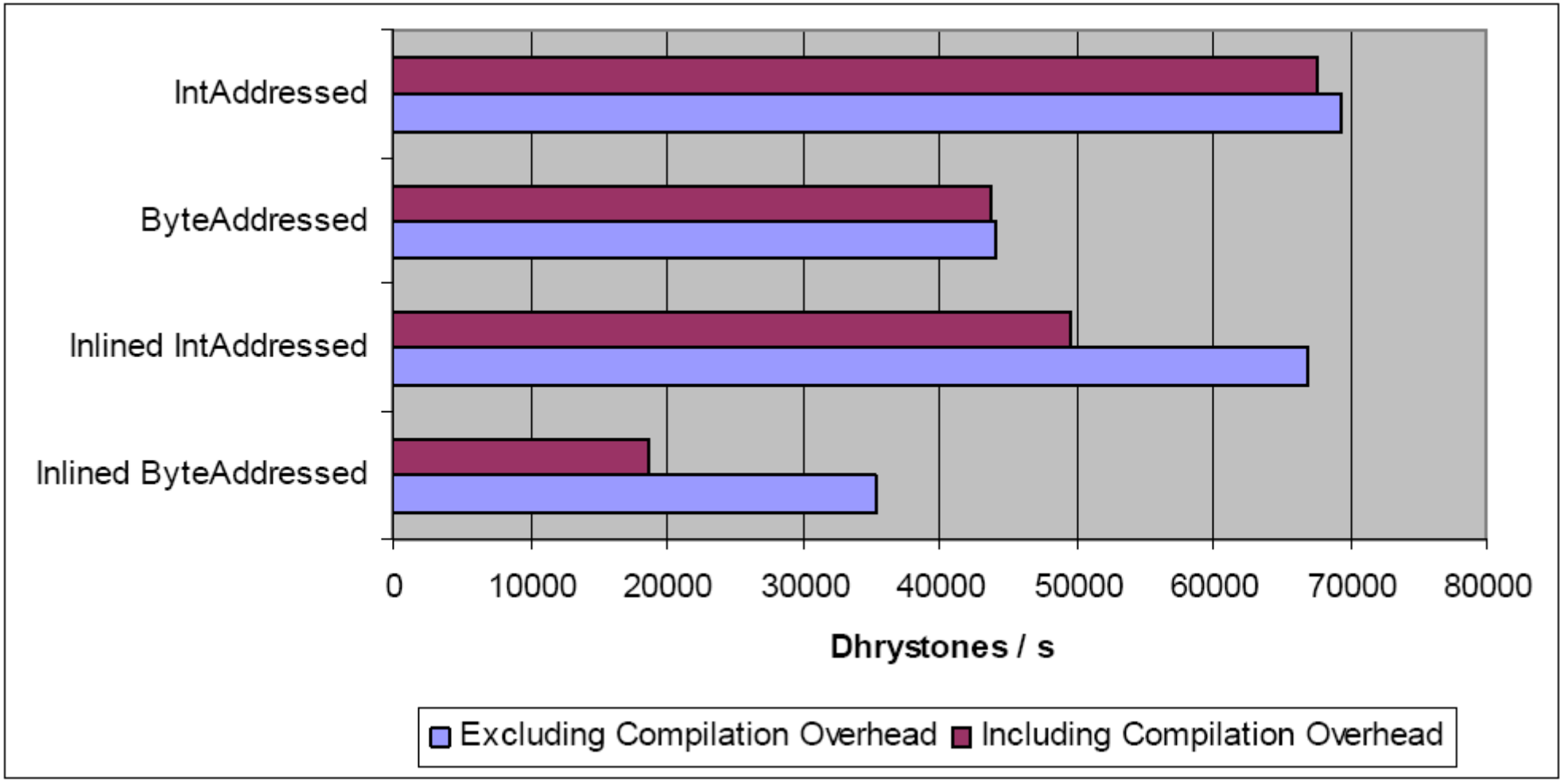
Lazy vs Immediate Evaluation



MANCHESTER 1824

The University of Manchester

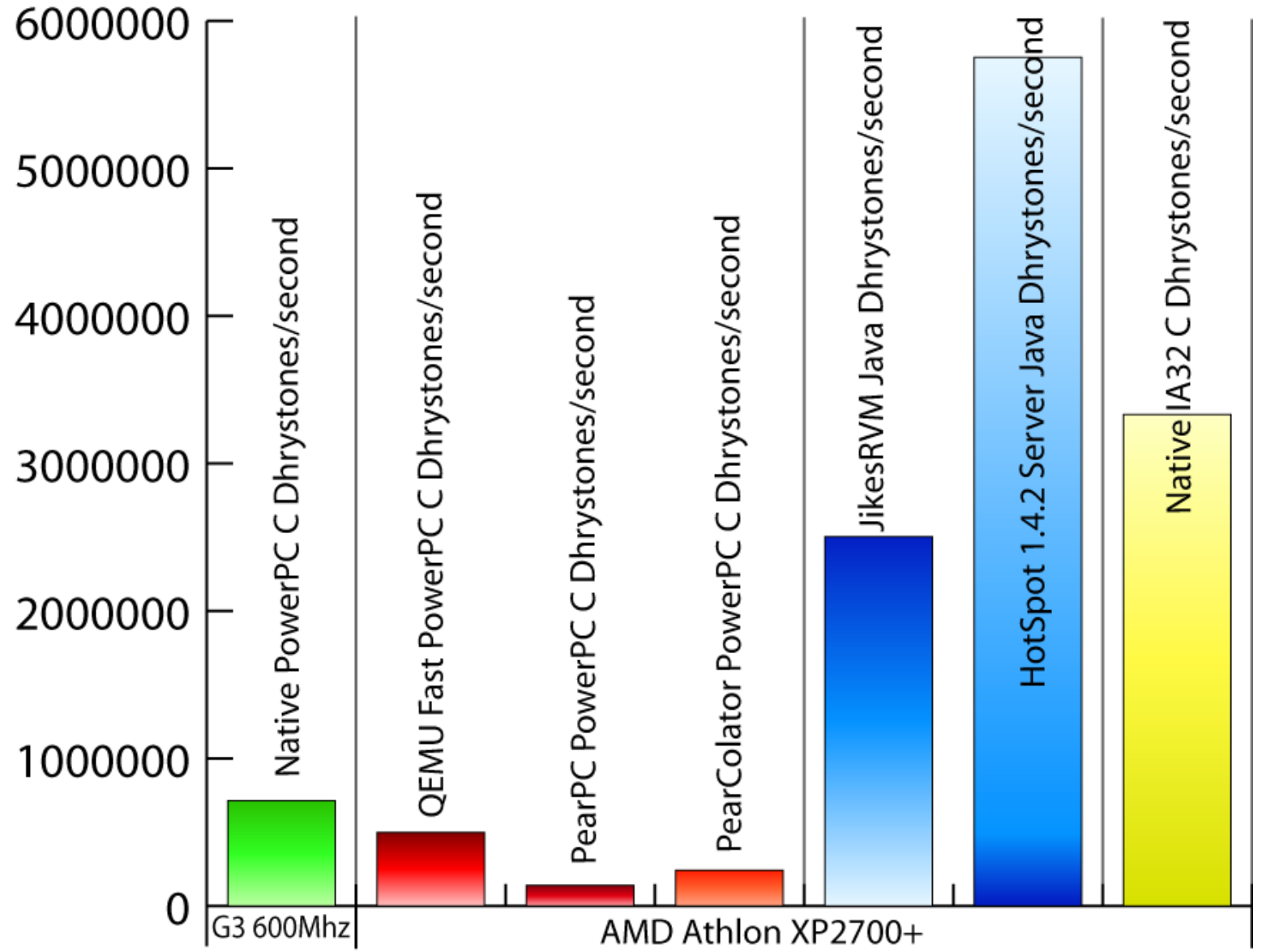
Memory Models



MANCHESTER 1824

The University of Manchester

PearColator



Where we are

- ARM, x86 and PowerPC decoders
- Generic and shared OS emulation library
  - Insufficient to run SpecCPU – needs support for exec, pipe...
- No hardware emulation, a la JPC or PearPC
- No direct to memory memory-model

Tales of sorrow

- Linux isn't the same across architectures
- On PowerPC brk on Linux 2.4 maps from /dev/zero, whilst on x86 it maps using mmap
- Calloc routine in glibc took advantage of the fact that on PowerPC new pages allocated with brk would be zeroed
- On x86 new pages weren't zeroed with disastrous effect

Tales of sorrow

- Mach is message passing and pointers can be passed in messages
- Different servers are needed for different pointer sizes, with binary translation also for different byte sex
- Possibility of having 4 concurrent font servers running on OS/X

Tales of sorrow

- Some libraries generate code on the stack to perform certain operations, this can create a lot of apparently self-modifying code

Tales of sorrow

- Most binary translators steal some address space from the emulated binary
- This causes a security problem as the subject application can modify dynamically generated code
- (PearColator doesn't suffer from this problem)

Tales of sorrow

- Linux isn't the same across architectures
- On PowerPC brk on Linux 2.4 maps from /dev/zero, whilst on x86 it maps using mmap
- Calloc routine in glibc took advantage of the fact that on PowerPC new pages allocated with brk would be zeroed
- On x86 new pages weren't zeroed with disastrous effect

Tales of sorrow

- Floating point precision can be the key to font rendering and good looking applications
- Many architectures have obscure floating point modes
  - Intel x87 80bit
  - PowerPC fmacs 66bit

Tales of sorrow

- Debugging can be hard due to “random” values in code
  - Genuine random numbers
  - Date and time values

MANCHESTER 1824

The University of Manchester

Thanks and…

any questions?