MANCHESTER 1824

The University of Manchester

An Automatic Runtime DOALL Loop Parallelisation Optimization for Java

Dr. Ian Rogers, Jisheng Zhao,

Dr. Chris Kirkham, Prof. Ian Watson

The Advanced Processor Technologies group

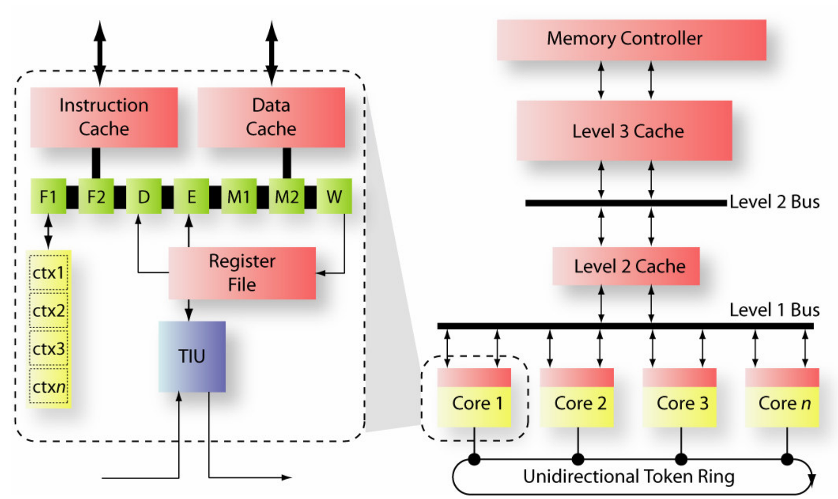
http://www.cs.manchester.ac.uk/apt

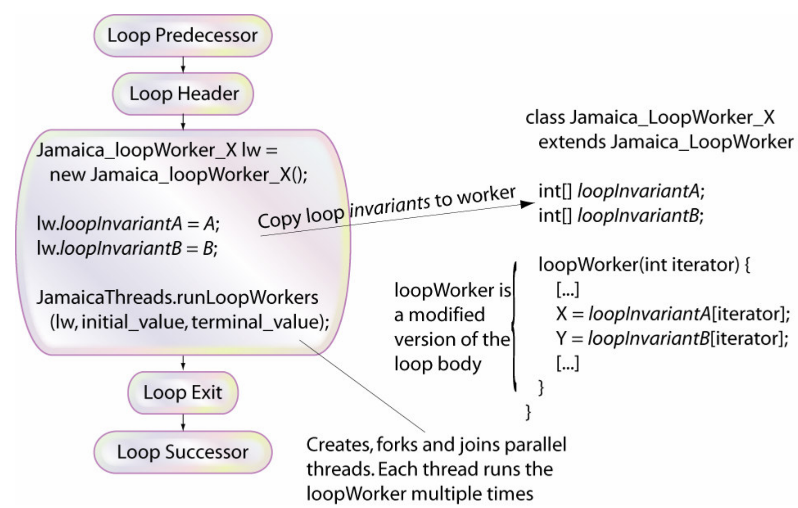
MANCHESTER
1824

The University
of Manchester

Presentation outline:

- Motivation
  - simultaneous multithreading, chip multiprocessor architectures
  - the JAMAICA architecture
  - work distribution
  - virtualization
- Annotated Loop Structure Trees
- Null and bound check elimination
- Parallelisation optimisation
- Performance analysis
  - SpecJVM 98
  - simple kernel
- Future work
- Summary





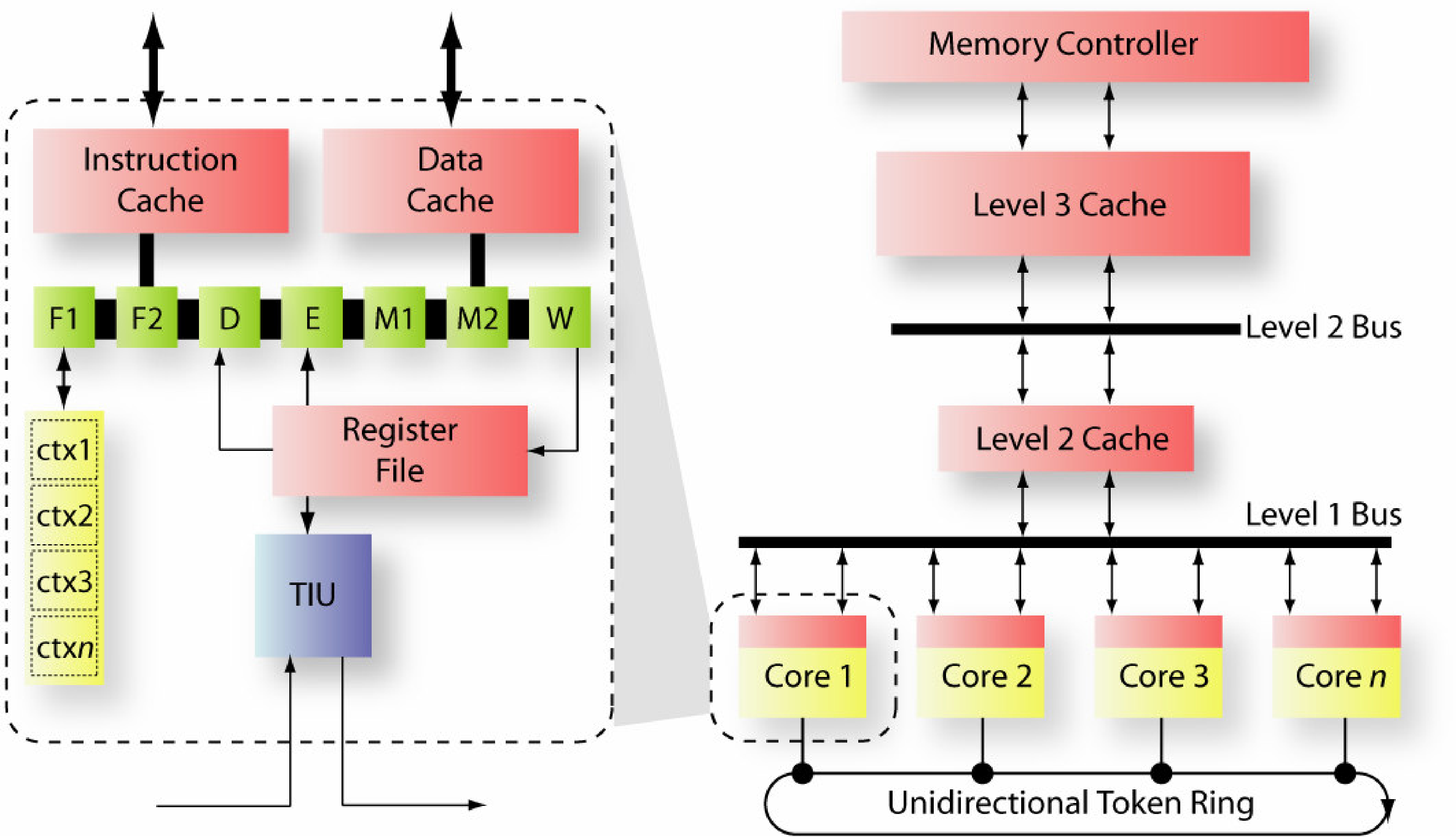
MANCHESTER
1824

The University of Manchester

SMT and CMP Architectures

- Simultaneous MultiThreading (SMT):
  - performance gap between processor and memory is growing
  - threads can be scheduled on cache misses to hide memory access time
- Chip MultiProcessors (CMP):
  - instruction level parallelism reaching limits
  - reduce design complexity
  - local clocks aid clock distribution
- Threaded code necessary to expose parallelism
- New mechanisms to help expose threaded parallelism
  - thread scheduling and work distribution
  - speculative threading (transactional commit mechanism)
- This work is a first step into a runtime support system

Overview of the JAMAICA architecture



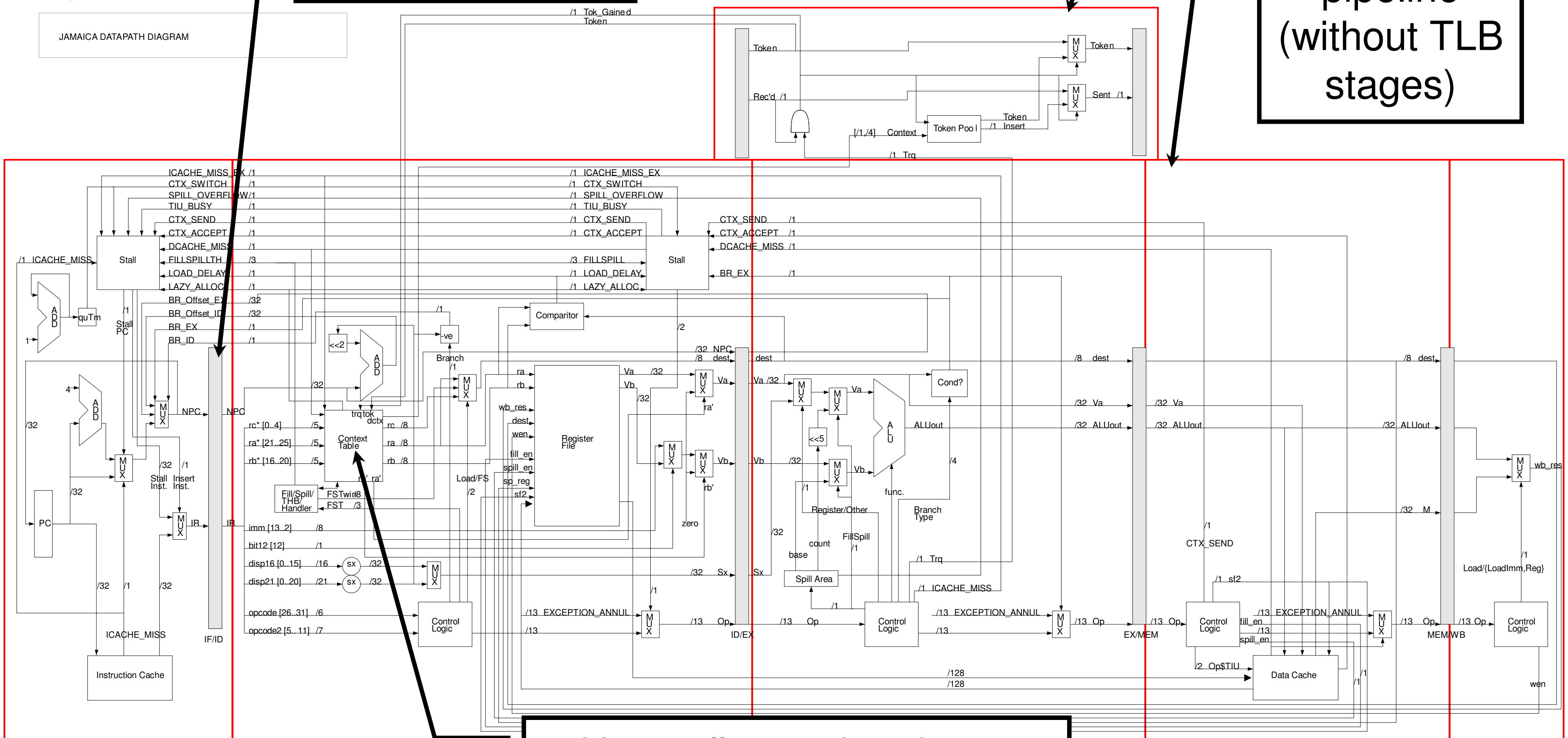
The University of Manchester

Some more detail

Token ring interface

5 stage MIPS based pipeline (without TLB stages)

Alpha based instruction set

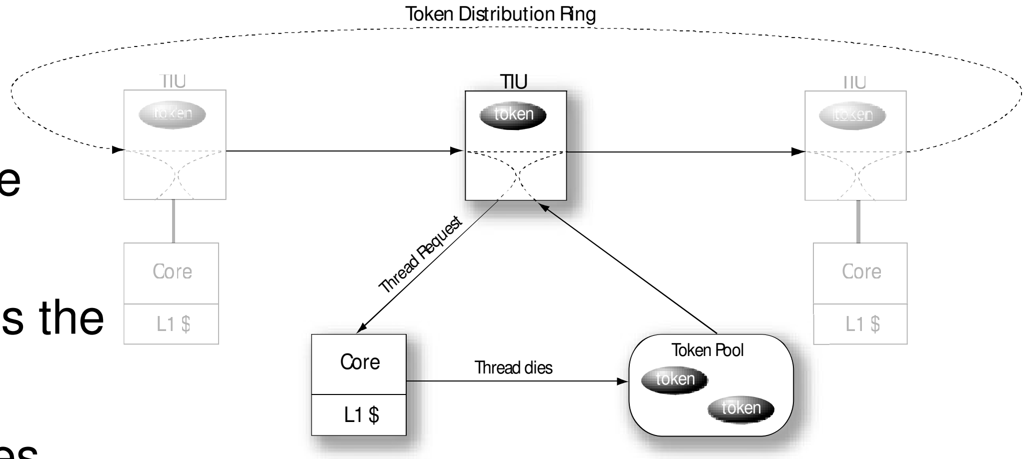


Heap allocated registers and context management

The University of Manchester

Work distribution

- Idle threads distribute tokens on a token ring bus
- Executing context on a core requests to ship work to an idle context or core and context
- Taking a token from ring grants the use of a particular context
- Shipping of work between cores occurs over data bus
- Gives lightweight thread creation
- When token is redistributed, work has been completed
- Thread unit monitors for completion of forked work



Virtualization

- Platform independence
  - Operating system virtualization
    - Run multiple operating systems simultaneously on virtualized hardware
  - Application virtualization
    - Standard application formats such as ELF can run on a multitude of operating systems as binary format and system call interface are standardized.
    - Wine allows windows applications to run on FreeBSD, Linux and Solaris
  - Instruction set virtualization
    - Dynamic binary translators (see presentation in tomorrows PLOS workshop)
- Hardware flexibility
  - Transmeta - 4-way VLIW TM3000 and TM5000 processors, 8-way VLIW TM8000 processor all run IA32 code
- New compiler optimizations …

MANCHESTER
1824

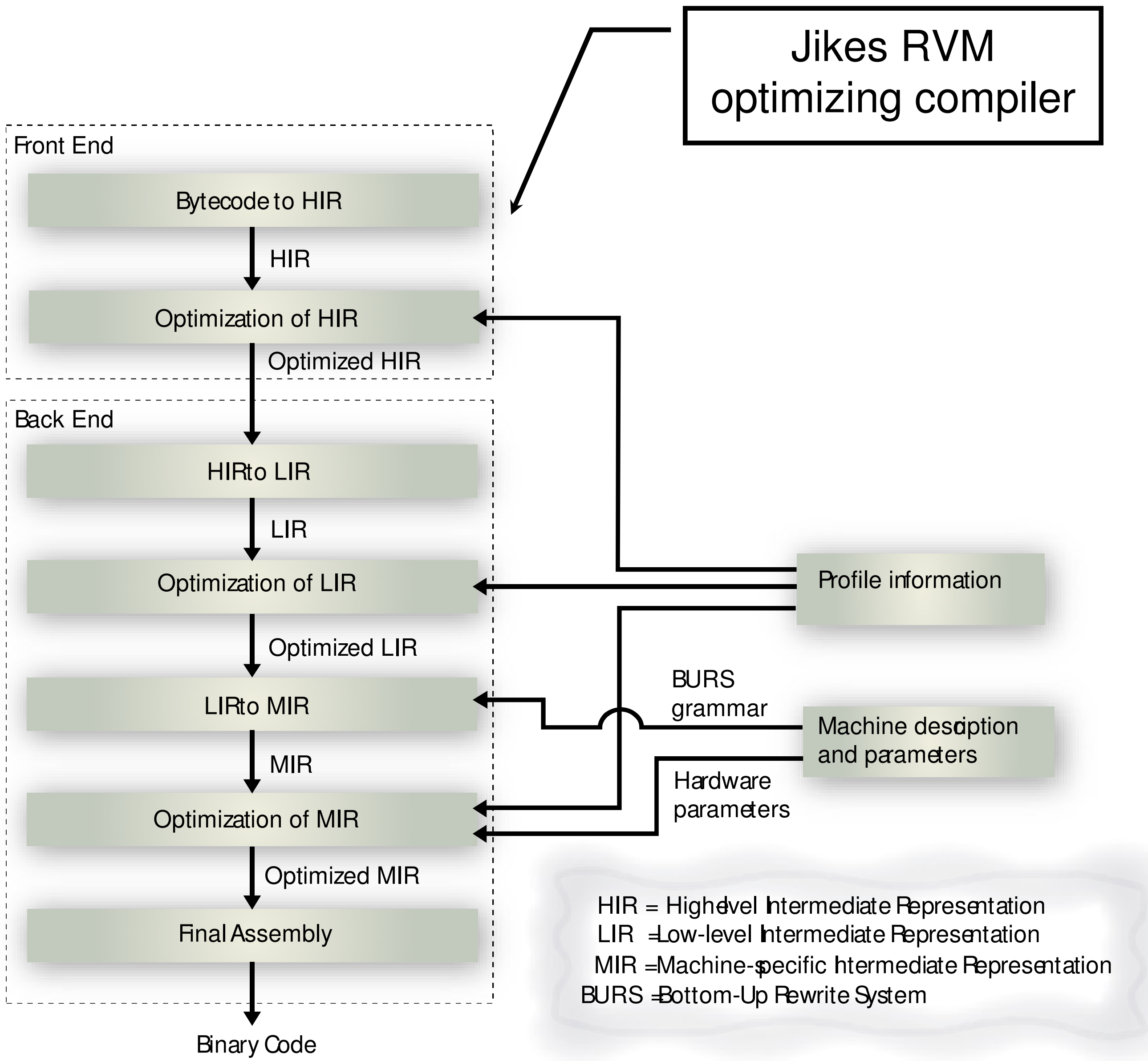
The University of Manchester

Software support for the JAMAICA architecture

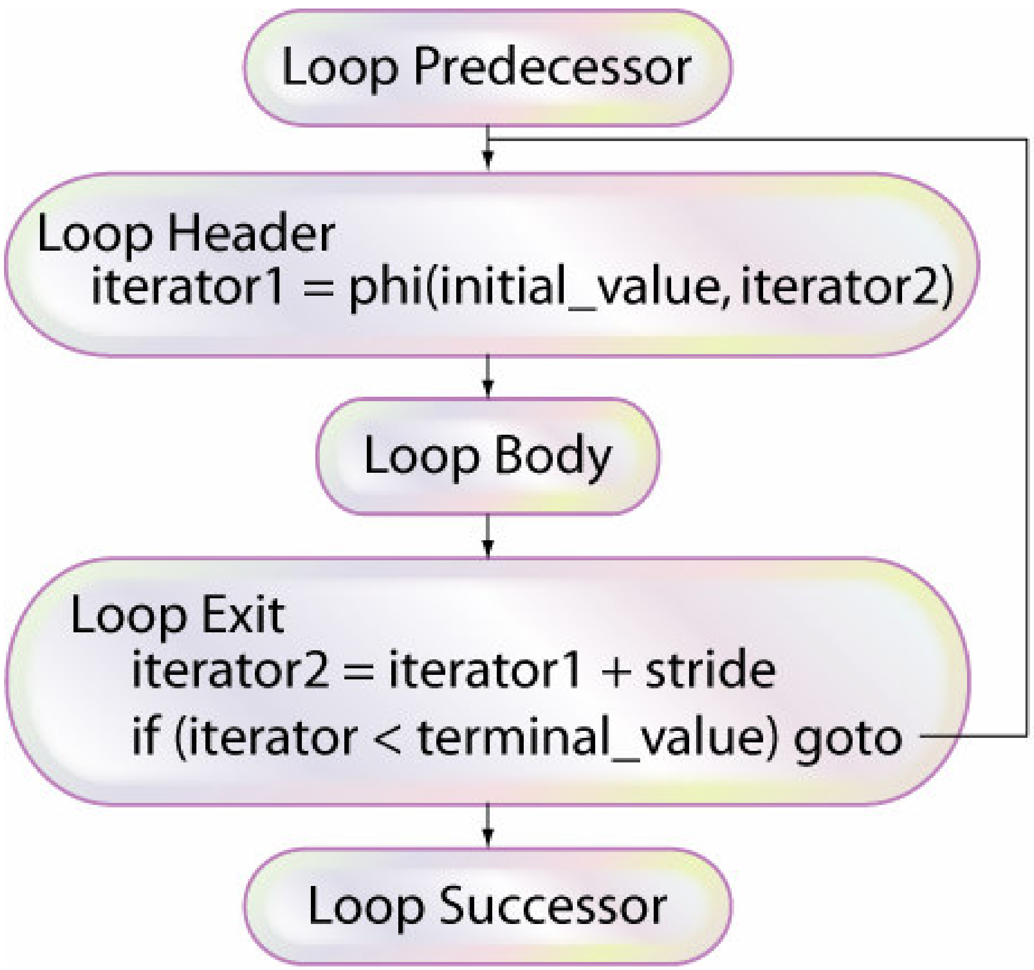
- Tools
  - C compiler – based on Princeton's LCC
  - jtrans – Java class file to assembler
  - javar – modified to generate jtrans parallel constructs
  - sim-idbg – interactive debugger and simulator in C
  - SIMPA – threaded, interactive, cycle accurate and fast simulator in Java
  - Jikes RVM – JAMAICA back-end and runtime

The Jikes RVM

- JVM written in Java
- Support for IA32, PowerPC and JAMAICA
- Baseline (quick) and optimizing compilers
- Adaptive optimization and feedback system
- Extended array SSA form sub-stages in HIR and LIR optimization



Annotated Loop Structure Trees



Loop blocks may be separate or combined

Stride and compare operators can vary from those shown, as can the placement of the iterator2 computation

Null and bound check elimination

- ABCD analysis eliminates checks when the values of the arraylength and non-nullness are known
- Length and non-nullness are known following a test or after an array is created
- Analysis of spec benchmarks showed ABCD wasn't enabling loops to be parallelisable
- Annotated LST used to duplicate loop body and create one without tests and one with, with explicit tests beforehand

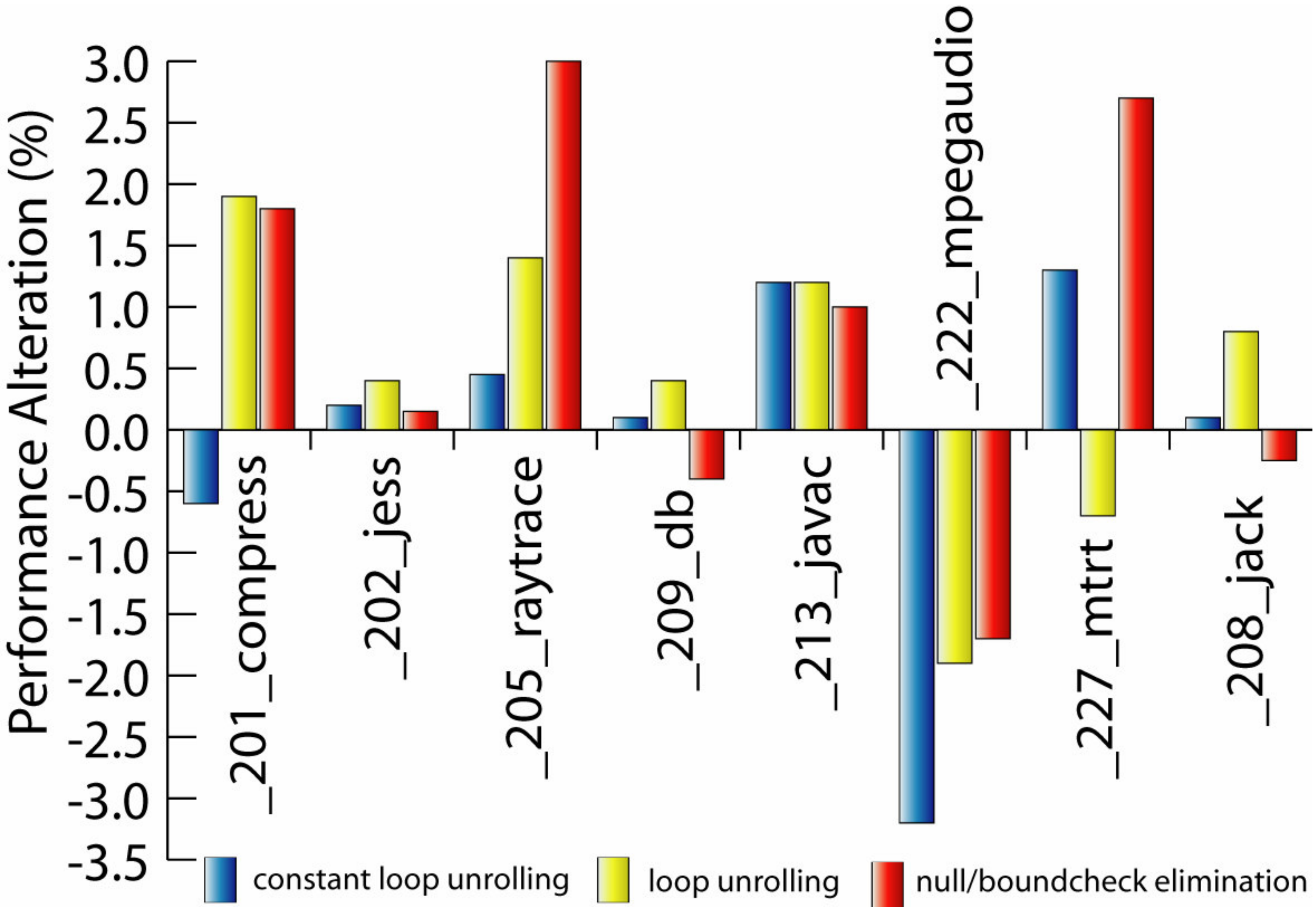
```
for (int i = fromIndex; i < toIndex; i++) {
    g1 = null_check a;
    g2 = bounds_check a, g1;
    g3 = guard_combine (g1, g2);
    a[i] = val, g3;
}
```

MANCHESTER
1824
The University of Manchester

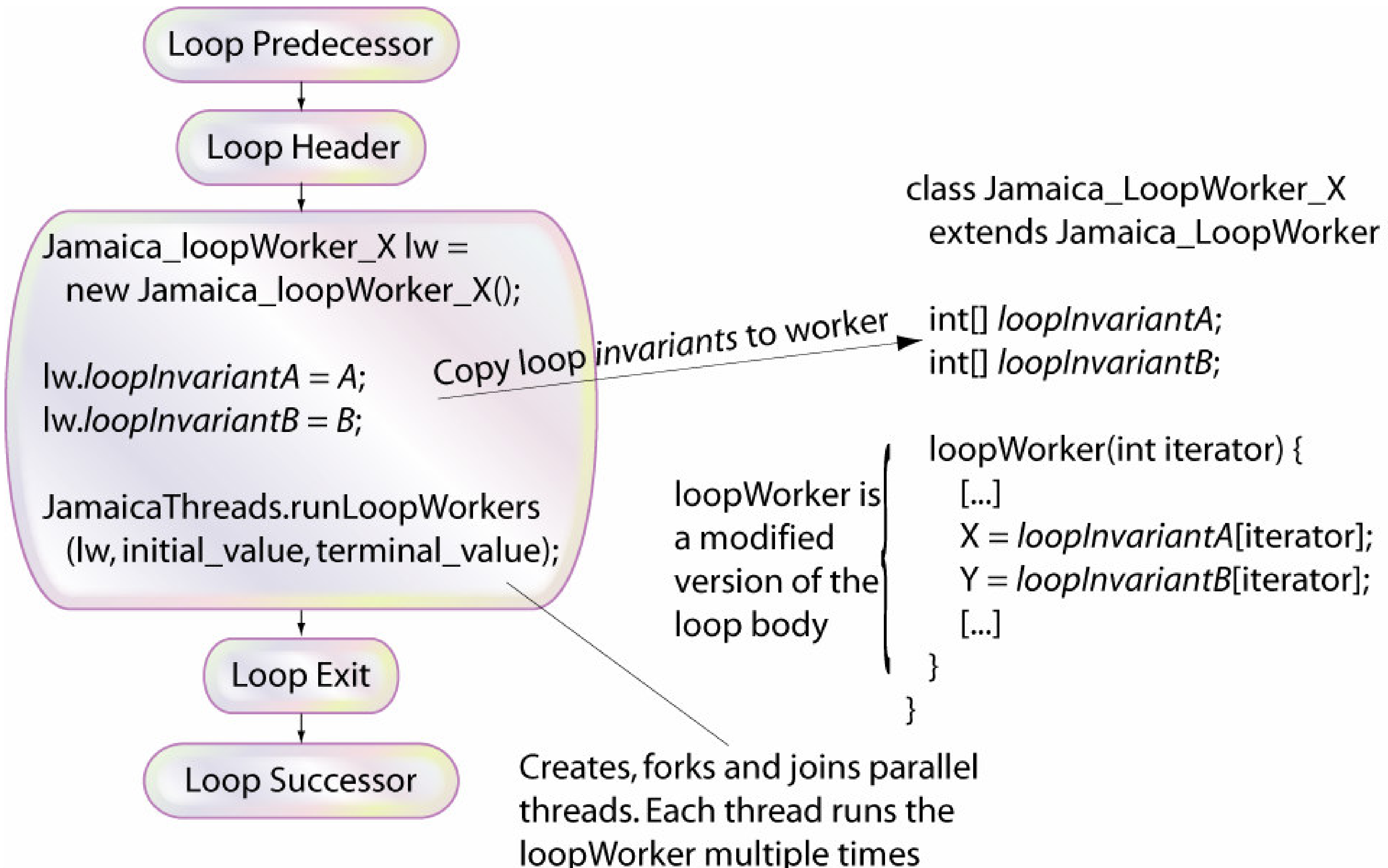
Duplicated loops, one without exceptions

```
[...]
-16          LABEL12 Frequency: 8.999998
-1           arraylength      t43i(I) = l0pa([Z,d), t2pv(GUARD)
-1           int_ifcmp        t36v(GUARD) = t3pi(I), t43i(I), >U,
                              LABEL8, Probability: 0.00999999
-1           bbend            BB12

-16          LABEL13 Frequency: 8.999998
-1           goto             LABEL10
-1           bbend            BB13

19           LABEL4 Frequency: 8.999998
-1           phi              t30pi(I) = t39i(I), BB9, t40i(I), BB11
-1           phi              t26v(GUARD) = t35v(GUARD), BB9,
                              t36v(GUARD), BB11
-1           phi              t27v(GUARD) = t37v(GUARD), BB9,
                              t38v(GUARD), BB11
-1           phi              t31pi(I) = t39i(I), BB9, t40i(I), BB11
-1           phi              t32v(GUARD) = t41v(GUARD), BB9,
                              t42v(GUARD), BB11
-1           goto             LABEL6
-1           bbend            BB4
[...]

-16          LABEL8 Frequency: 8.999998
-1           bbend            BB8

19           LABEL9 Frequency: 8.999998
-9           phi              t33i(I) = 0, BB8, t39i(I), BB9
19     G     yieldpoint_backedge
23     EG    bounds_check     t35v(GUARD) = l0pa([Z,d), t33i(I), t2pv(GUARD)
23           guard_combine    t37v(GUARD) = t2pv(GUARD), t35v(GUARD)
23           byte_astore      l1pi(Z,d), l0pa([Z,d), t33i(I),
                              <mem loc: array <BootstrapCL, Z >[]>,
                              t37v(GUARD)
24           int_add          t39i(I) = t33i(I), 1
30           int_ifcmp        t41v(GUARD) = t39i(I), t3pi(I), <, LABEL9,
                              Probability: 0.9
-1           goto             LABEL4
-1           bbend            BB9

-16          LABEL10 Frequency: 8.999998
-1           bbend            BB10

19           LABEL11 Frequency: 8.999998
-9           phi              t34i(I) = 0, BB10, t40i(I), BB11
19     G     yieldpoint_backedge
23           guard_combine    t38v(GUARD) = t2pv(GUARD), t36v(GUARD)
23           byte_astore      l1pi(Z,d), l0pa([Z,d), t34i(I),
                              <mem loc: array <BootstrapCL, Z >[]>,
                              t38v(GUARD)
24           int_add          t40i(I) = t34i(I), 1
30           int_ifcmp        t42v(GUARD) = t40i(I), t3pi(I), <, LABEL11,
                              Probability: 0.9
-1           goto             LABEL4
-1           bbend            BB11
```

Performance of Annotated LST Optimizations



Loop Parallelisation Optimisation



Parallelised SpecJVM Performance

Benchmark	Parallelisation speed-up[1]	Overhead	Normal benchmark execution time	Executed parallel loop bodies
_201_compress	1.6%	10.5s	7.1s	3500
_202_jess	0.7%	3.8s	3.5s	1500
_205_raytrace	3.7%	3.0s	4.3s	2400
_209_db	0.9%	8.4s	11.5s	4500
_213_javac	2.1%	0.2s	5.1s	1200
_222_mpegaudio	2.7%	11.9s	7.2s	12500
_227_mtrt	2.6%	1.7s	5.0s	1200
_228_jack	1.1%	1.3s	5.6s	1800

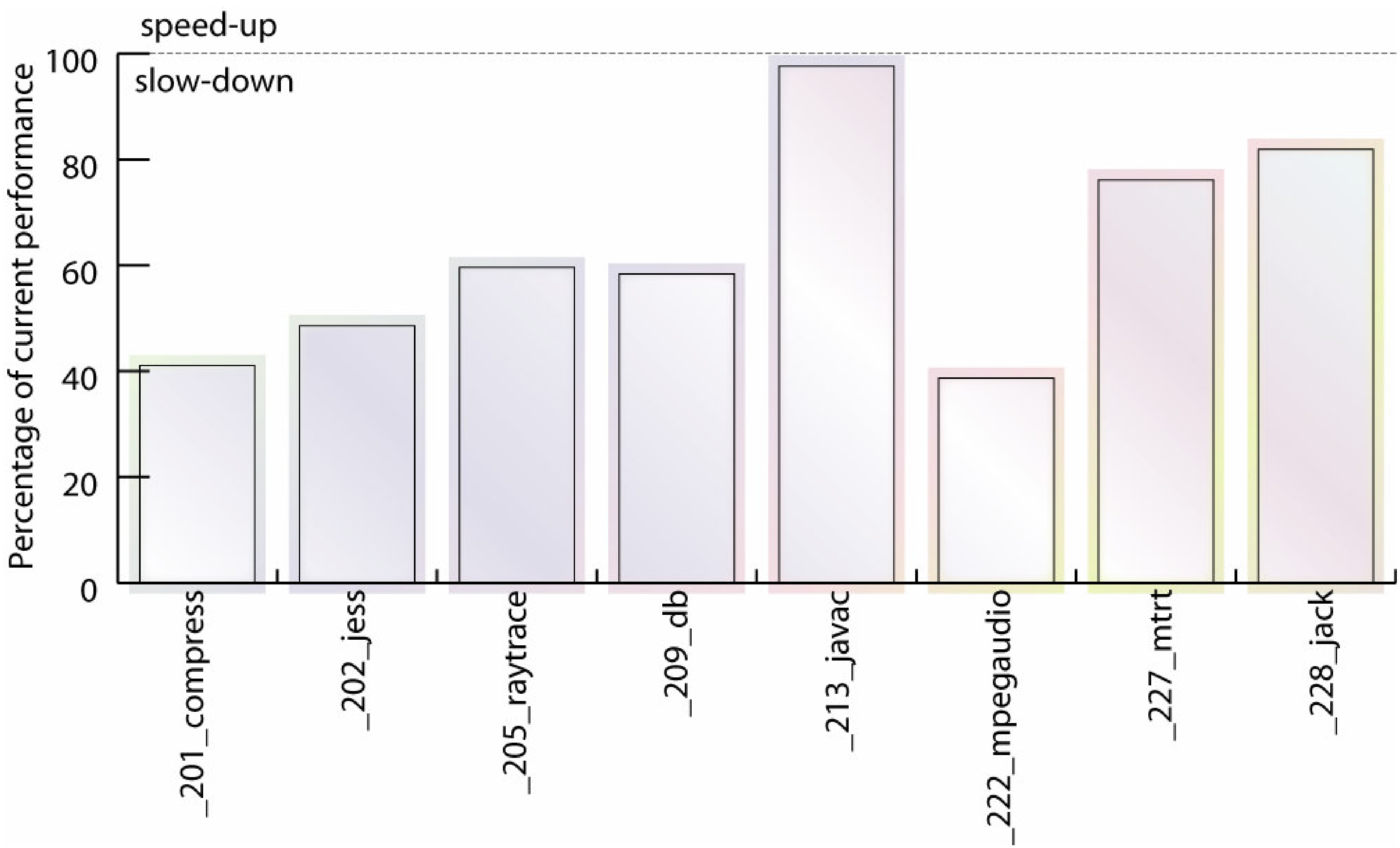
Average speed-up of 1.9%

[1] This is the performance speed-up excluding overheads introduced by creating threads and performing the optimisation.

MANCHESTER
1824

The University
of Manchester

Parallelised SpecJVM Performance



MANCHESTER
1824

The University of Manchester

Simple kernel performance

- Simple test to see if optimisation can parallelise and get performance from simple case
- Performs no useful work ☺
- Achieved 79% speed-up on dual CPU Intel

```
int size = 3000;
double[] matrix1 = new double[size];
double[] matrix2 = new double[size];
double[] result = new double[size];
for (int i = 1; i <= 500; i++) {
    for (int p = 0; p < size; p++) {
        matrix1[p] = p * p / i;
        matrix2[p] = p * (p + 1) / i;
        result[p] = (i * p + 1) / i;
    }
}
```

Future work

- Speculative execution
  - Range of speculative and non-speculative execution states
    - tree rooted at non-speculative state with branches for every spawned speculative context
    - speculative contexts may spawn more speculative contexts
  - If speculation goes wrong squash speculative state
    - throw away values in cache or a buffer
  - Detect speculation problems:
    - in software: when a value isn't that expected explicitly squash
    - in hardware: when an address is loaded by a speculative context, ensure that stores to the same address from a less speculative context cause a squash
  - Problems with creating speculative threads and avoiding excessive squashing
  - Mechanism may aid virtual machines, e.g. handling of unaligned memory accesses

Future work

- Loop parallelisation can recognize more loops if loops with break out paths are including in analysis
- Parallelisation can work for these loops with more speculative threads being squashed if a break-out path is taken

MANCHESTER 1824

The University of Manchester

Summary

- We have presented a series of runtime optimisations designed to increase the number of parallel threads for next generation CPUs
- Threads are light-weight and may comprise just 1000s of instructions
- Our optimisation doesn't work on current CPUs with the current threading model (upto 2.48 times slow-down)
- Performance improvements on a standard benchmark suite are modest (1.9% on SpecJVM ignoring threading costs)
- Future hardware support for light-weight and speculative threading should improve the situation
  - cheaper to create threads (e.g. JAMAICA)
  - possible to create more threads
- We have a portable infrastructure for virtualization of the CPU, this work includes work on a Java oriented operating system and legacy code execution environment

The University of Manchester

MANCHESTER 1824

Thanks!

- … and any questions?