Jikesrvm is an open-source research virtual machine for Java bytecodes. It runs on IA-32 Linux and PowerPC64 Linux platforms. This manual describes how to build, run and collect the Java program traces by Jikesrvm on IA-32 and x86-64 Linux. x86-64 is not currently supported but Jikesrvm can be built for x86-64 by using 32-bit addressing. This manual also describes how to run these traces on Tejas simulator and find the heap memory usage of only some classes of Java programs.

1 Building Jikesrvm

1.1 Dependencies

To build Jikesrvm the system should have following packages.

JDK 1.6	JRE 1.6	Ant	Mercurial	gcc
gcc-multilib	g++-multilib	Bison	Perl	Awk

Figure 1: Dependencies for building Jikesrvm

gcc-multilib and *g++-multilib* are required only if you are building on 64-bit system as it enables the user to run and build 32-bit application on 64-bit installation.

1.2 Download Jikesrvm-3.1.2

Download Jikesrvm-3.1.2 from `http://sourceforge.net/projects/jikesrvm/files/jikesrvm/3.1.2/`, untar it in a jikesrvm directory.

1.3 Setting Proxy

This section describes how to set proxy for building Jikesrvm inside IIT Delhi.

- *Ant Proxy*
  To go beyond the proxy add these lines to `$HOME/.bashrc` file

```
export ANT\_OPTS="-Dhttp.proxyHost=10.10.78.62
-Dhttps.proxyHost=10.10.78.62 -Dhttps.proxyPort=3128
-Dhttp.proxyPort=3128 -Dftp.proxyHost=10.10.78.62
-Dftp.proxyPort=3128"
```

- *Command Line Proxy*
  To download packages from terminal we need to set the proxy. This can be done by appending these lines at the end of /etc/bash.bashrc.

```
http_proxy=http://10.10.78.62:3128
https_proxy=https://10.10.78.62:3128
ftp_proxy=ftp://10.10.78.62:3128
```

```
no_proxy="localhost,*.iitd.ernet.in,*.iitd.ac.in"
export http_proxy
export https_proxy
export ftp_proxy
export no_proxy
```

Restart your terminal and login to IITD proxy through the browser.

1.4 Modify bin/buildit.base_config file

You may get an error java not found if you start building Jikesrvm directly, therefore modify bin/buildit.base_config file and search for global.javahome.ia32-linux.
For 32-bit systems set `global.javahome.ia32-linux = /usr`
For 64-bit systems set `global.javahome.x86_64-linux = /usr`

1.5 Replace the rvm

The rvm directory in Jikesrvm directory has most of the code for virtual machine replace it with the modified rvm to get the traces of the Java program.

1.6 Setting the configurations

Jikesrvm can be build with different configurations. Two useful configurations used in this project are:

- *Development*
  This configurations defines the fully optimized versions of Jikesrvm with debug mode. It builds and executes slowly. We can use this configuration to collect the traces for normal execution of Java programs with all the optimizations turned on. This can be build by:
  - `$cd jikesrvm`
  - `$sudo bin/buildit localhost development`
- *FullAdaptive NoGC*
  This configurations defines no garbage collector versions of Jikesrvm. Since we need to implement hardware garbage collector. Therefore we require there is no memory reuse done by software garbage collector. This can be build by:
  - `$cd jikesrvm`
  - `$sudo bin/buildit localhost FullAdaptiveNoGC`

Other configurations details can be found in jikesrvm/build/configs directory. One or more configurations can coexist in the same directory of jikesrvm.

2 Editing Jikesrvm in Eclipse

Jikesrvm can be edited with eclipse.

- `$cd jikesrvm`
- `$sudo bin/buildit --eclipse localhost`
  - Import the project in eclipse workspace
  - From eclipse, select File –⟩ import
  - Select Existing Project into Workspace
  - Browse and select `jikesrvm-xxxx`
  - Finish

3 Running Jikesrvm

After Jikesrvm build is successful a directory jikesrvm/dist/config_name is created. We can check build is successful or not by running a sample Java program. To invoke Jikes virtual machine use rvm script in jikesrvm/dist/config_name. The command for running Java program by Jikes virtual machine:
`jikesrvm/dist/config_name/rvm <java_class_name>`
The program should run without crashing Jikesrvm.

4 Command Line options of Jikesrvm

- `-X:aos:enable_recompilation=false -X:aos:initial_compiler=base`
  This option will turn off all the runtime compiler optimization on the program.
- `-X:base:mc=true`
  This option can be use for printing the assembly instruction generated by the baseline compiler. It can be useful to compare the traces generated by jikes_trace.
- `-X:opt:mc=true`
  This option can be used for printing the assembly instructions generated by the optimizing compiler. It prints the assembly instructions only for the classes which could be optimized by the virtual machine.
- `-X:gc:ignoreSystemGC=true`
  This option can be used to ignore the system calls for garbage collection. It can be useful while running the program with no garbage collection mode.
- `-X:gc:harnessAll=true`
  This option can be used to print the stats related to garbage collectors like time spent in running mutator program and garbage collector.
- `-help`
  This option can be used to find the various option.

5 Getting traces of the Java program

The instruction and memory traces of the java program can be obtained as follows:

- Write a java program `hello.java`
- Compile it with java compiler `$javac hello.java`, which will create hello.class file
- `cd jikesrvm/dist/development-xxxx/`
- `./rvm hello 2>trace_file`

Note: The trace_file generated will have the traces of hello program. By default optimizations are enabled, therefore if you don't use any options the traces generated will be of optimized code.

Baseline Compiler: The execution traces of the program without any optimizations can be collected by:
`./rvm -X:aos:enable_recompilation=false -X:aos:initial_compiler=base hello 2>trace_file`

No Garbage Collector: The execution traces of the program without garbage collector activity can be collected by:

- `cd jikesrvm/dist/FullAdaptiveNoGC_xxxx/`
- `./rvm -X:gc:ignoreSystemGC=true -X:aos:enable_recompilation=false -X:aos:initial_compiler=base hello 2>trace_file`

Multithreaded programs: For multithreaded programs use rvm_multithread (rvm modified for multithreaded programs) and build jikesrvm again as described above. The traces generated by the Jikesrvm for multithreaded programs are interleaved. Corresponding to each instruction and memory traces a threadId is also printed, we split these interleaved traces into multiple trace file one for each thread:
`python splitter.py trace_file`
`trace_file` is the raw trace file generated by jikesrvm_multithread. This generates multiple trace files. Each thread will have one trace file named as its threadId. These trace files should be post processed individually.

Partial Traces: In case you don't want the complete trace of a program, you can specify the number of instructions to be traced. For this modify `jikesrvm/rvm/src/org/jikesrvm/runtime/RuntimeEntrypoints.java`. Search for `my_check_function` and add these statements at the beginning of function definition.

```
VM.inst_count++;
if(VM.inst_count == <noOfInst>)
    VM.sysFail("instruction completed");
```

`noOfInst` is the integer value of number of instructions to be traced.
At the end of trace files there will be virtual machine crash messages, those statements should be removed before disassembling the traces.
`cat trace_file | head - n -30 >trace_file1`
For multithreaded programs noOfInst should be sum of all instructions to be traced of each thread.

6 Disassemble to x86 Instructions

The traces generated by Jikesrvm are in a binary format. To run these traces on the architectural simulator Tejas, we require traces in VISA format(instruction set of Tejas). The x86 to VISA translator is already implemented in Tejas. Therefore we need to disassemble these raw traces into x86 instructions. We use the udis86 library for this purpose. libudis86 is a disassembler library for the x86 architecture which decodes a stream of bytes as x86 instructions.

- Separate the instruction and memory traces
  `python conv.py <tracefile>`
  It generates two files temp_file.txt and out_trace.txt.
  temp_file.txt has 8 bytes of instructions in hexadecimal format per line which is the input for the udis86 disassembler.
  out_trace.txt has instruction pointer and the memory address accessed by the program which we use later to merge the instruction and memory trace.

- Disassemble into x86 instructions

  - Download the libudis86 and untar
  - `$./configure`
  - `$make`
  - `$sudo make install`
  - compile disas.cpp using `$g++ -std=c++11 disas.cpp -o disas.o -ludis86`
  - `$./disas.o`

  disas.o reads the instructions bytes from **temp_file.txt** and disassembles it. The x86 instructions are of variable length. Therefore to disassemble it **disas.o** reads as many bytes required to form a instruction and ignores the remaining bytes in the line. The next instruction to be disassembled is read from the next line. The disassembled instructions are written to **temp_file1.txt**.

- Merge disassembled instruction traces and memory traces

  - `$javac conv1.java`
  - `$java conv1`

  It generates **fin_trace.txt_0** which will be in right format to run on Tejas simulator.

7 Running traces on Tejas

- Compress trace files using
  `gzip -c trace_file >trace_file_0.gz`
  Compressed file name should end with _x.gz. For single threaded program x should be 0. And for the multithreaded programs x is the thread number.

- Modify Tejas config file
  Set the following parameters in Tejas config file:

  - *EmulatorType: none*
  - *CommunicationInterface: file*

- Run Tejas
  `java -jar <tejas-jar><config-file><output-file><trace_file>`

8 Heap memory usage of program

The dynamic memory used by the program with and without garbage collector can be found by annotating benchmarks, collecting the traces using Jikesrvm, post processing the traces and running on Tejas.
The address of the dynamically created objects are visible in the traces when we access the fields of the objects. Therefore to find the memory address we insert the markers in the source code and using this markers we extract the memory addresses of dynamically created objects.

8.1 Annotating benchmarks

- Object creation
  In order to keep track of the heap memory used by the program, annotate the source code at places where memory is allocated to the object dynamically. Therefore add markers whenever the *new* keyword occurs in the program. The *new* operation in Java invokes the constructor of the class, thus add the *0xbee* marker in the constructor of the class whose memory usage need to be calculated. Insert this marker after every field of the object is accessed in the constructor. In case the class uses default constructor, write your own constructor which accesses all the fields of the class. Suppose there is a class *Employee* and its constructor is:

  ```
  public Employee(int id, String name, int age) {
      this.id = id;
      this.name = name;
      this.age = age;
      this.next = null;
  }
  ```

  Annotate this constructor as follows, *marker* is a global volatile static integer variable.

```
public Employee{
    this.id = id;
    marker = 0xbee;
    this.name = name;
    marker = 0xbee;
    this.age = age;
    marker = 0xbee;
    this.next = null
    marker = 0xbee;
}
```

- Reference Updation
  During the lifespan of the program, a reference may point to different objects. At some-point of time during the execution of the program, there may be an object which is not referred to by any reference. Then we can reclaim the memory allocated to that object, thus we keep track of updates to references. In order to do this add the function call `dummy_gc(Employee emp1, Employee emp2)` whenever there is a reference update i.e `Employee emp1` starts pointing to `Employee emp2` object. We decrement the reference count of the object to which `emp1` was previously pointing to and increment the reference count of `emp2`, to which `emp1` now points to.
  The `0xdea` marker is used for the object whose reference count has decremented and `0xdee` marker is used for the object whose reference count has incremented. We use this information in the hardware for maintaining the reference count and when the reference count of the object becomes zero that memory is reclaimed. Suppose there is a statement `emp1.next = emp2` in the program, insert the function call `dummy_gc()` before this statement. The annotated source code will be:

  ```
  dummy_gc(Employee emp1, Employee emp2)
  emp1.next = emp2
  ```

  Add `dummy_gc()` as the `Employee` class member function with following definition:

  ```
  dummy_gc(Employee emp1, Employee emp2){
      if(a != null){
          a.next.id = a.next.id;
          marker = 0xdea;
          a.next.name = a.next.name;
          marker = 0xdea;
          a.next.age = a.next.age;
          marker = 0xdea;
      }
      if(b != null){
          b.id = b.id;
          marker= 0xdee;
          b.name = b.name;
          marker = 0xdee;
  ```

```
            b.age = b.age;
            marker = 0xdee;
        }
    }
```

8.2 Collecting Traces

Use Jikesrvm to get the traces of annotated benchmarks and disassemble the traces as described in above sections.

8.3 Process the Traces

`$python postprocess` $\langle fin\_trace.txt\_0 \rangle$
`fin_trace.txt_0` is the disassembled trace file.
The traces are post processed to mark the addresses of the objects which will be managed by the hardware garbage collector. The hardware garbage collector maintains the reference count of the objects. When the object is created its reference count is one. When a new reference to the object is created the reference count is incremented by one. And when the reference is deleted the reference count is decremented by one. The memory space is dead when its reference count becomes zero. The markers in the annotated benchmarks are captured in the trace file and special instructions for allocation, incrementing and decrementing reference count are inserted in the trace files. Three type of instructions **allocate**, **ref_inc** and **ref_dec** are added to the trace file after extracting the useful information from the disassembled trace file.

Marker in source code	x86 instruction in trace file	Instruction inserted in trace file
`0xbee` marker	`Push dword 0xbee` or `Push 0xbee`	allocate `address`
`0xdee` marker	`Push dword 0xdee` or `Push 0xdee`	ref_inc `address`
`0xdea` marker	`Push dword 0xdea` or `Push 0xdea`	ref_dec `address`

Figure 2: Instructions inserted in trace files for memory managment by hardware

The object's memory address appears in the trace file just few lines above the assembly instruction of the marker statement . We use this address to insert the respective instruction operands in the trace file.

8.4 Run on Tejas

After postprocessing the traces, there are we inserted certain instructions in the trace file which are used by the hardware garbage collector in Tejas for memory management.
To calculate the memory usage with garbage collector set the Tejas config file parameters as follows:

- *memory_usage = true*
- *garbage_collection = true*

To calculate the memory usage without garbage collector set the Tejas config file parameters as follows:

- *memory_usage = true*
- *garbage_collection = false*