McGill University
School of Computer Science
Sable Research Group

Measuring ILP in Java Programs

Sable Technical Report No. 3
CS605 Course Project

Raja Vallée-Rai
kor@sable.mcgill.ca

June 8, 1998

Abstract

Java programs can greatly differ in terms of required runtime support. C-like Java programs which are usually computation intensive tend to require little, whereas heavily object oriented programs such as compilers, can spend most of their execution time in runtime handlers. This report begins to explore the differences in instruction level parallelism between these two extreme types of Java programs. No solid conclusions can be drawn, however, due to an oversimplified execution model and the scarcity of appropriate benchmarks.

1 Introduction

In our previous studies of Java [1], we discovered that Java programs can differ greatly in terms of required runtime support. At one end, we have heavily object oriented programs, such as object oriented compilers, which make heavy use of runtime features such as dynamic binding and garbage collection. At the other end, we have C-like programs, which avoid these features and behave like regular C programs, minus complicated pointer manipulations. If compiled properly, C-like Java programs should perform as well as their pureC counterparts. Heavily object oriented programs, however, are more difficult to compile and execute efficiently due to their special runtime requirements.

With this project, we begin to explore the fundamental differences between these two extremes of Java programs. We concentrated our efforts at measuring the amount of instruction level parallelism (ILP) available to each type of program. It is reasonable to think that there might be a difference since heavily object oriented programs may have smaller block sizes due to their short methods. This was one of the hunches we hoped to verify.

This report presents the approach used to measure the instruction level parallelism of different Java programs and the results obtained.

2 The Problem

The design of an appropriate measure of ILP was the first problem to solve. Although we have studied this concept in several courses, there is no general formula which maps programs to values of ILP. But this is what we need however, if we are to compare quantitively the amount of parallelism available in different Java programs.

Using [2] as a guide, we decided to define the amount of ILP in a program P as follows: ILP(P) = x where x is the average number of instructions executed per second for the Java program P on the Intel x86 architecture running the Kaffe virtual machine. Calculating this value for a given Java program can be achieved in two steps:

1. Record the instruction trace generated by executing the program.
2. Simulate the scheduling process used by the Intel x86 architecture.

Recording the instruction trace is more complicated than it seems. With C programs, one can simply annotate the assembly code to record each instruction as it is executed. This is not possible with Java programs, because they require a Java virtual machine to execute, and the corresponding assembly code is generated on the fly[1]. In order to record the native instruction trace generated by the execution of a Java program, one must be able to modify the Java virtual vachine so that it generates the trace recording instructions as well as the converted Java bytecodes. Luckily, this was possible due to the freely available Java virtual machine called Kaffe written by Tim Wilkinson. Complete with sources, it is a multi-platform virtual machine. We only modified the Linux/x86 version, since that is the platform for which we wanted to simulate the scheduling process.

Simulating the scheduler is an arbitrarily complex problem, depending on the accuracy desired. Our goal was to produce a reasonable simulator which produces numbers which are directly related to the amount of parallelism available in a program. This way we are able to compare the relative parallelism of different Java programs, as desired.

3 Trace Generation

Recording the sequence of instructions generated by executing a Java program means that we need to determine the exact procedure Kaffe uses to execute Java programs. Kaffe follows the model of Just-In-Time (JIT) compilers, and translates each Java method to native code just before it executes. Our first goal was to locate this translation process so that we can tweak it to produce trace recording instructions as well.

The recorded trace does not need to contain the exact details of the instructions executed. Since we will only simulate the scheduler on the trace, and not actually execute any instructions, it suffices to simply store the dependencies and control flow information of the instruction. Namely:

1. The registers that it reads/writes.
2. Whether it reads from memory/writes to memory.
3. If this instruction is a conditional branch.

This information can be stored in three `int`s: `readSet`, `writeSet`, and `isBranch`. The registers considered are: `eax`, `ebx`, `ecx`, `edx`, `ebp`, `esp`, `edi`, `esi`, `af`, `cf`, `pf`, `tf`, `if`, `zf`, `of`, `df` and `sf`. Note that the last 9 registers are not real x86 registers, but status fields. These are considered registers in order to make the status field dependencies explicit. For example, instructions such as `add` modify these status fields, and instructions such as `jne` read them. If we are to schedule such a pair of instructions, obviously we must maintain their order and this can be done easily by treating the status fields as regular registers.

This remainder of this section presents the Kaffe JIT compilation process, and how we modified it so that the desired dependency traces are recorded.

3.1 Kaffe JIT Compilation

Java programs are compiled into units called *classfiles* and these contain Java assembly instructions called *bytecodes*. The Kaffe JIT compilation process converts each bytecode one at a time. Kaffe splits the bytecodes into two classes and compiles each differently:

[1] At least for the JIT compiler case which is the one we're considering.

Simple: These bytecodes, such as `iload` and `iadd`, perform simple tasks and can be converted to just a few native instructions (typically 2-6.) These produced instructions will be referred to as *hard instructions.*

Complex: These bytecodes get translated to calls to C functions present in Kaffe and may result in the execution of hundrends of native instructions. For example, `invokevirtual` or `new` behave in this manner. The instructions within these functions will be to referred to as *soft instructions.*

These two types of bytecodes must be handled differently. In one case, code is dynamically being generated, and so profiling instructions can be easily inserted. Wheras with the complex bytecodes, we must modify the functions which are called so that their native instructions are recorded as well.

3.2 Dealing with Simple Bytecodes

The key to recording an instruction's effect is to insert a call to `recordInstructionEffect` before it. The exact code to insert is of the form:

```
pushf
pusha
pushl $isBranch
pushl $writeSet
pushl $readSet
call recordInstructionEffect
addl $12, %esp
popa
popf
```

where the constants `isBranch`, `writeSet` and `readSet` are replaced with immediates depending on the following instruction. Note the use of the two pairs of state saving instructions `pusha` and `popa`, and `pushf` and `popf`. It is important to use these in order to avoid changing the registers being used in the `recordInstructionEffect` function or the status fields. Without these, the program would behave differently and we would no longer be recording the correct trace.

Since simple bytecodes are translated directly into assembly code in the translation process, we simply had to insert a macro to emit the above trace instructions before each regular instruction. This unfortunately increases the code size by roughly a factor of 10, and the program runs much slower. But it works.

3.3 Dealing with Complex Bytecodes

The complex bytecodes are a bit trickier to deal with, because they involve calls to C functions which have already been compiled. Ideally, all the instructions that are executed in these C functions must be recorded. We had a similar problem in [1], and we discovered that the way to solve this problem is to modify the assembly code for the functions. This works relatively well, except that it does not guarantee to capture all the instructions executed. For example, a C function called may contain a call to a function such as `sprintf` for which we do not have the code.

The problem is essentially the same: for each instruction in the assembly code, we must insert a call to `recordInstructionEffect` with the correct arguments. Due to the size of the assembly code, it would be extremely tedious to annotate it by hand. Instead we wrote a parser which reads a summary of the effects of the Intel x86 architecture and an assembly file, and then inserts the appropriate calls to `recordInstructionEffect`.

Note that this annotation process is not trivial, because the Intel x86 instruction set supports many different addressing modes. For example, when analyzing the instruction `movl -5(%ebx), %ebp`, it must determine that memory is being read as well as the `ebx` register since it is used as an index.

Here is a portion of the rule file which summarizes the effect of each instruction:

```
adc Rop1 Rop2 Wop2 Rcf Wof Wsf Wzf Waf Wpf Wcf
add Rop1 Rop2 Wop2 Wof Wsf Wzf Waf Wpf Wcf
and Rop1 Rop2 Wop2 Wof Wsf Wzf Wpf Wcf
bsf Rop1 Rzf Wop2 Wzf
bsr Rop1 Wop2 Wzf
bswap Rop1 Wop1
bt Rop1 Rop2 Wcf
btr Rop1 Rop2 Wop2 Wcf
bts Rop1 Rop2 Wop2 Wcf
cbw Reax Weax
cbq Reax Wedx
clc Wcf
cld Wdf
cli Wif
cmc Rcf Wcf
cmp Rop1 Rop2 Wof Wsf Wzf Waf Wpf Wcf
```

And a portion of an assembly file called `soft.s`. Before it is annotated:

```
soft_new:
pushl %ebp
movl %esp,%ebp
pushl %esi
pushl %ebx
call .L34
.L34:
popl %ebx
addl $_GLOBAL_OFFSET_TABLE_+[.-.L34],%ebx
```

And after:

```
soft_new:
pushf
pushal
incl softAssemblyInstructionCount
pushl $0
pushl $256
pushl $32
call recordInstructionEffect
addl $12,%esp
popal
popf
pushl %ebp
pushf
```

```
pushal
incl softAssemblyInstructionCount
pushl $0
pushl $32
pushl $16
call recordInstructionEffect
addl $12,%esp
popal
popf
movl %esp,%ebp
    ...
```

3.4 Overall Approach 1: Two Discrete Steps

Two different approaches for the overall organization of the trace generator and scheduler simulator were considered.

The first was to view these two tasks as discrete steps, such that one must be completed before the second could start. This could be implemented by making `recordInstructionTrace` output its arguments to a trace file, and then have the scheduler simulator simply read the contents of this file to determine which instructions to schedule.

There are two advantages to this approach:

No overhead to start the simulation: If you wish to perform several simulations with different options on the same trace, then the trace does not need to be regenerated for each simulation as is the case with approach 2. (see below)

Java simulator: By being two discrete steps, the second step can be implemented in any language, such as Java which is easier to use than C or C++.

Unfortunately, this approach has a big disadvantage: the complete trace file needs to be stored. For reasonable benchmarks, more than 400 million instructions are executed, and if we record all of these, then we will end up with trace files of several gigabytes in size. Only a percentage of the trace could be stored, and so we considered a different approach.

3.5 Overall Approach 2: Two Concurrent Tasks

In order to avoid the headaches of dealing with huge files, we decided to view the trace generation and simulation as two concurrent tasks, following a typical consumer-producer model. This was implemented by embedding the scheduler simulator into the Kaffe virtual machine, and invoking it whenever the trace buffer is full.

Obviously, this is appealing because the entire trace does not need to be stored, and one might think that this would allow us simulate the entire execution. Unfortunately, this is not true. Even though we wrote the simulator in C, it can only simulate roughly 50,000 instructions/sec. Hundreds of millions of instructions would take hours to simulate which is longer than we had available since we had to perform 36 test runs. So we had to simulate a percentage of the trace anyways. We chose between 3% and 10% depending on the program simulated.

We also discovered another disadvantage when we started gathering results. Every simulation on the same trace has a significant overhead because the trace has to be re-generated. This overhead is significant enough (more than 75% of the execution time), that it might have been more efficient to actually use the first approach instead.

4 Scheduler Simulator

This section discusses the design and implementation of the model for the Intel x86 instruction scheduling process.

4.1 The Model

Our measure of the instruction level parallelism of a given Java Program is the number of instructions executed per cycle on average by the Intel x86 architecture running the Kaffe Virtual Machine. Clearly this can not be calculated directly, even with the complete instruction trace. We can approximate the answer with a model, however.

Our goal was not to design an extremely accurate model of x86 scheduling process, but to develop a model which can compare the relative inherent parallelism of different types of programs. Unfortunately, we discovered that our simplified and idealized model does not pipeline the instructions properly and so yields much lower than expected values. The resultant model is the is the equivalent of a superscalar unpipelined machine. The results obtained are still proportional to the amount of ILP present, but more work is needed to simulate the pipeline more accurately to ensure the model's reliability.

Nevertheless, here are some of the features of this virtual unpipelined superscalar machine:

1. Infinite number of functional units.
2. Uses a modified version of Tomasulo's reservation station algorithm to perform renaming and WAW hazard avoidance.
3. 4 stage instruction execution (setup, read, execute, write).
4. No cache misses.

The configurable options for a simulation are:

Number of reservation stations: 1 to 64. This number also controls the window size. (the number of instructions to consider to issue)

Type of memory dependences: In our instruction trace we do not store the addresses of the memory accesses. Instead, we assume one of three different memory models: no memory accesses conflict, all accesses conflict, all reads after writes and writes after reads conflict (no waw, however.)

Type of branch prediction: Two models of the branch predictor are available: none and perfect. With no branch predictor, issuing a conditional branch will halt further instruction issues until the conditional branch is executed.

4.2 Algorithm

The simulation algorithm is basically a state simulation on the following data structures:

Station: one reservation station per instruction. Contains integers such as the `readSet`, `writeSet`, `isBranch`, the list of dependent stations, the current state (`free, fetch, setup, write, execute, waitread or waitwrite`), and the number of stations on which it is dependent.

RegisterOutputFile: A register to reservation station map which indicates which station will output the latest value.

MemoryAccessQueue: A queue of stations requesting writes or memory reads.

Here is the algorithm pseudo-code, assuming perfect branch prediction and no memory dependences.

```
while(there are instructions to fetch or execute)
{
    while(there are reservation stations free)
        issue next instruction to free station

    for every station s
    {
        switch(s.state)
            case fetch:
                s.state <- read
                break;

            case read:
                for every register being read, check registerOutputFile
                   and record dependences

                for every register being written, fill out registerOutputFile

                if dependent on register or memory request
                    state <- wait_read
                else
                    state <- state


                break;

            case execute:
                s.state <- write;
                break;
```

```
            case write:
                update registerOutputFile
                free station
                break;
    }

    perform every pending memory request and change states of stations
        waiting on memory reqs
}
```

Here are some notes on how the basic algorithm was changed to support the various configuration options.

No branch prediction: While there is a conditional branch in an active reservation station, simply do not issue any more instructions.

All memory access conflict: When performing memory requests at the end of the loop, only consecutive reads or a single store are allowed in the current cycle.

No writes conflict: Only consecutive reads or consecutive writes are performed. For example, if the memory request contains a read followed by a write, then it will take 2 cycles to complete. Two writes or two reads however could be completed in the same cycle.

Sample Algorithm State

```
Cycle: 93  IssueCount: 226  ExecutionCount: 210  BranchCount: 24  NumFreeStations: 0

Station State      Action  ReadSet         WriteSet                 dependent on   flags
   1    write         94  ebx             ebx of sf zf af pf cf
  13    wait_read      -  ebx mem         of sf zf af pf cf        1               m
   4    wait_read      -  zf                                       13              b
   9    write         94  ebp             mem
   2    wait_read      -  ebp mem         mem of sf zf af pf                       m
  14    wait_read      -  ebp mem         eax                                      m
  10    wait_read      -  eax ebx mem     of sf zf af pf cf        1 14            m
  12    wait_read      -  cf                                       10              b
  11    setup         94  eax mem         edx
  15    setup         94  ebx mem         edi
   8    setup         94  eax ebp         mem
   3    setup         94  ebx mem         edx
   0    setup         94  edx esi mem     esi
   7    setup         94  ebp esi         mem
   6    setup         94  esi mem         ecx
   5    setup         94  ecx             of sf zf pf cf

[Memory Access Queue] W  W  R13  R2  R14  R10

[Output file] eax:14  ebx:-  ecx:-  edx:1  ebp:-  esp:-  edi:-  esi:-
               of:10   df:-   if:-   tf:-   sf:10  zf:10  af:10  pf:10   cf:10
```

Note that the reservation stations are presented in issue order.

Limitations

The limitations of our scheduler are:

Floating point instructions not handled: They are not generated in the trace and the scheduler does not handle them either. Support would require the recognition of the special floating point registers. This limitation was not really a problem since both benchmarks used contained no floating point instructions anyway.

Not properly pipelined: As mentioned before, our model of the Intel x86 is pretty different from reality because the instruction execution is not properly pipelined. The results still are proportional to the amount of ILP present in the program, but the model should be pipelined in the future to it provides a reasonable estimate of real world execution.

Only a percentage of trace simulated: This was required because simulating the entire trace would take too long. We did however compare the results between a few full trace and percentage trace simulations. They were similar (within a few percent of each other.)

Native methods not handled: Not really a limitation of the scheduler, but of the trace generator. Native methods in the Java API are not handled. This is not really a problem since the number of native method calls is low, but it is still something to keep in mind.

5 Results

We tested our simulator on two different benchmarks, which are supposed to represent the extremes of Java program behavior. Note that before establishing any concrete rules of behavior for these extremes, more benchmarks would need to be used.

Puzzle: This is an image processing benchmark written by Madeleine Mony. It attempts to solve a simple 4-piece jigsaw puzzle by matching the edges together. It is computation intensive and fits our notion of a 'C-like' Java program. 50 million native instructions are executed in a run, of which 88% are hard assembly instructions.

EarlyJimple: An early version of Jimple written by Raja Vallée-Rai, this program converts a Java classfile to a simple intermediate representation. It is a heavily object oriented program, and more than 75% of the instructions are soft. 200 million native instructions are executed in all.

For each benchmark we tried 18 different option selections. See figures 1, 2, 3 and 4 for the results. The numbers contained in the tables indicate the average number of instructions executed per cycle. The next subsection discusses their significance.

Results Discussion

1. The first thing one notes when looking at figures 1 and 3 are the unusually low execution rates. This is a result of the model not capturing the pipeline process properly. Effectively we are simulating a superscalar machine which is unpipelined, and so to get realistic numbers, we should probably multiply these figures by 3-4.

	memory dependence		
stations	none	no waw	all
16	0.99	0.98	0.97
32	1.00	1.00	1.00
64	1.00	1.00	1.00

Figure 1: Puzzle with no branch prediction (average # of instructions executed per cycle).

	memory dependence		
stations	none	no waw	all
16	2.28	2.19	2.12
32	3.82	3.65	3.33
64	6.97	6.25	4.67

Figure 2: Puzzle with perfect branch prediction.

	memory dependence		
stations	none	no waw	all
16	0.92	0.91	0.88
32	0.94	0.94	0.93
64	0.95	0.94	0.93

Figure 3: EarlyJimple with no branch prediction.

	memory dependence		
stations	none	no waw	all
16	2.23	2.14	1.95
32	3.79	3.57	2.92
64	6.54	5.83	3.64

Figure 4: EarlyJimple with branch prediction.

2. We see that without branch prediction, the amount of instruction level parallelism available to exploit is very small. This confirms the results obtained in [2], and clearly justifies the need for hardware branch prediction.

3. As expected, by increasing the number of stations available, the amount of ILP increases. And by increasing the restrictions on the memory accesses, it decreases.

4. Our goal was to compare the parallelism levels between EarlyJimple and Puzzle. By comparing figures 2 and 4 we see that the parallelism available is virtually identical except when there are a large number of reservation stations available. If we consider the case of 64 reservation stations, then there are two interesting comparisons to make. With no memory dependences, Puzzle has 7% more executions per cycle than EarlyJimple which is not that significant. If we assume that all memory accesses conflict, however, then the difference climbs to 28% which is significant, but these are unrealistic constraints on the memory accesses.

6 Conclusion

Our goal was to count the number of instructions executed per cycle for different types of Java programs. Because our model for the Intel instruction scheduler does not properly simulate instruction pipelining, the results obtained are unusually low. They still can be used to perform relative comparisons on the amount of ILP available, however. In particular, we discovered that the differences between Jimple and Puzzle only manifest themselves when there are a large number of reservation stations, and even then, the differences are small.

In order to come to a solid conclusion about instruction level parallelism in Java programs, we need to work on two areas. First, the model needs to be improved so that it simulates the Intel scheduling process more realistically. Second, more benchmarks need to be developed. They need to represent the extremes, as well as programs which contain roughly the same amount of soft and hard assembly instructions. Only with these two fixes will we be able to conclude anything significant.

References

[1] Raja Vallée-Rai. *Profiling the Kaffe JIT Compiler.* Sable Technical Report 2. School of Computer Science, McGill University, 1998.
`http://www.sable.mcgill.ca/reports/report2.ps.gz`

[2] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach 2nd Edition.* Section 4.7. Morgan Kaufmann.

A Usage

We have simply added some run-time flags to Kaffe:

```
-simsched
-noprediction
-prediction
-memdep_all
-memdep_none
-memdep_noww
-numstations <n>
-simperc <n>
```

They behave as expected.

B Obtaining this Modified Version of Kaffe

Kaffe, in its modified state, can be obtained from `~kor/public/KaffeCS605.tgz` on the ACAPS filesystem. Please note that this program will only run correctly under Linux/x86.