Improving locals stack placement in Java with Soot

Report No. 2005-13

Kacper Wysocki

April 17, 2005

Contents

List of Figures

List of Tables

Abstract

Efficient allocation of locals has traditionally been one of the most important parts of an optimizing compiler, typically giving gains in runtime of 25% compared to code compiled with simplistic allocators. However, finding the optimal allocation is an NP-complete problem, and it is therefore crucial to find approximation algorithms that give economic packings on average. It is often not practical to spend a lot of time compiling a program, such as during development where more time spent compiling means less time spent testing, or in a JIT where compilation time takes its toll on runtime. Therefore, it is important that the allocation strategy can complete in a reasonable timeframe. I look at how such allocation strategies can apply to the Java Virtual Machine. The Java Virtual Machine is a stack-based machine that lacks registers, keeping locals on a per-method locals stack. Bytecode instructions are used to store and load locals to and from the locals stack onto the operand stack. There are five load instructions and five corresponding store instructions for each basic type in the Java Virtual Machine. The first four are one-byte nullary instructions which address the top four positions on the locals stack, while the remaining load and store are unary instructions relying on an immediate operand to supply the position of the local. The nullary loads and stores are therefore smaller and should be faster to execute than the unary loads and stores. It would therefore be beneficial to store the most frequently accessed locals on top of the stack, hopefully leading to faster execution of the program. I show that for certain basic types, namely integers, addresses and doubles, there is a measurable decrease in execution time when using only the nullary bytecodes in comparison to using only unary bytecodes to load and store locals. I show how two existing weight-based heuristics for register allocation can be applied to placing locals on the JVM locals stack. The locals placement algorithms are then implemented in Soot, the Java Optimization Framework, where we show that although there are improvements in runtime, more work is needed to find a better heuristic for locals placement.

1 Introduction and Motivation

Effective global register allocation has long been viewed as one of the single most beneficial phases of an optimizing compiler. Improvements in runtime of up to an order of magnitude can be seen when employing a good register allocation strategy instead of a simplistic one. The register allocation problem can be cast as a graph colouring problem, leading to conceptually simple algorithms for register allocation. However, finding the optimal colouring and therefore also the best register allocation is an NP-complete problem, so no algorithms are known that can find the solution in time better than exponential in the size of the input. Compilation time is at a premium in many applications, for example during development and in a JIT compiler. It is therefore crucial to employ approximating algorithms that find a good register allocation on average, and do not blow up on boundary cases.

Soot [1, 2] is a Java optimizing framework that allows researchers, compiler developers and packagers to more easily explore program analyses and transformations in a common, open-source framework. It provides several intermediate representations to facilitate reasoning about and implementing program transformations. It includes an API and several local, global and inter-procedural analyses as well as a toolkit of useful classes for program analysis.

The Java Virtual Machine (JVM) divides the memory allocated to a program into heap and stack. Dynamic memory is kept on the heap, while frames are kept on the stack. Each frame comprises the execution of a method body, and is divided into the locals stack and the operand stack. JVM instructions are a byte long, and take as operands either immediate values or values popped off the operand stack, pushing the result back on the operand stack. Since each instruction is a byte long, there are a maximum of 256 representable instructions.

There are seven basic types in the JVM: byte, short, int, long, float, double and reference. Each basic type has its own class of load and store instructions, except for bytes and shorts, which are loaded and stored using the integer load and store instructions. Each class of loads and stores consists of five loads and five stores. The first four are nullary load and store instructions: the first six bits specify the opcode, and the last two specify the location on the stack. These four instructions can thus access positions zero through three on the locals stack. The remaining load and store are unary instructions that take a word-long immediate operand. The byte-long nullary loads and stores are therefore smaller and should be cheaper to execute.

Longs and doubles do however take up twice as much space on the locals stack.

I show through empirical results on several architectures that the nullary loads and stores do indeed execute faster in general. Building on these observations, I show how to adapt two heuristics-based register allocation algorithms to the Soot Framework and the Java Virtual machine. I use a method of prioritizing locals by the number of appearances of each local, weighted by the loop nesting level the local appears in. These two allocation strategies are then implemented in soot, and used to transform several test benchmarks, to examine how the allocation strategies affected performance.

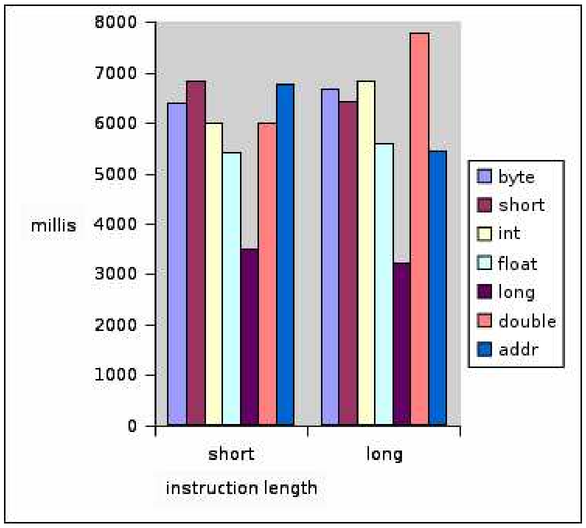

Figure 1: Instruction execution times for Intel Pentium 4 2.66MHz on Gentoo GNU/Linux 2.6. We see a 14% and a 30% improvement for nullary integer and double load/stores, respectively.

2 Background and Related Work

Much work has been devoted to developing efficient approximately good algorithms for register allocation. Appel [3] provides an overview of different allocation schemes, including interference graphs. Liberatore et al. [4] evaluated several different allocation schemes, including the graph colouring and interference graph algorithms. Chaitin et al. [5, 6] describe the graph colouring algorithm in detail. Chow and Hennessy [7, 8] confirm that priority-based colouring is both practical and effective. Briggs et al. [9] developed colouring heuristics that reduced the amount of register spill code. Poletto and Sarkar [10] developed the Linear Scan register allocation algorithm.

Derryberry and Lau [11] incorporated better register allocation into the Jikes RVM. Hummel et al. [12] annotated class-files to provide optimization hints to the VM. The annotations included information about register allocation.

Vallée-Rai[13] instrumented the kaffe VM to examine real costs incurred by JIT translation of bytecode into native code, showing that certain instructions, like "invokevirtual", are far more expensive than others. Alpern et al[14] developed methods to reduce the costs incurred with such expensive instructions.

Finally, Hendren et al. [15] describe an approach to register allocation based on hierarchical cyclic interval graphs, and also provide a survey of the existing work on the subject.

I will mainly concern myself with Chow and Hennessy's priority-based colouring strategy for register allocation [7, 8] and the colouring heuristics for register allocation developed by Briggs et al. [9]. There is little in terms of published work on applying register allocation techniques to the Java VM locals stack, and therefore the implementation detailed in this paper can be regarded as new approaches derived from these previous register allocation techniques.

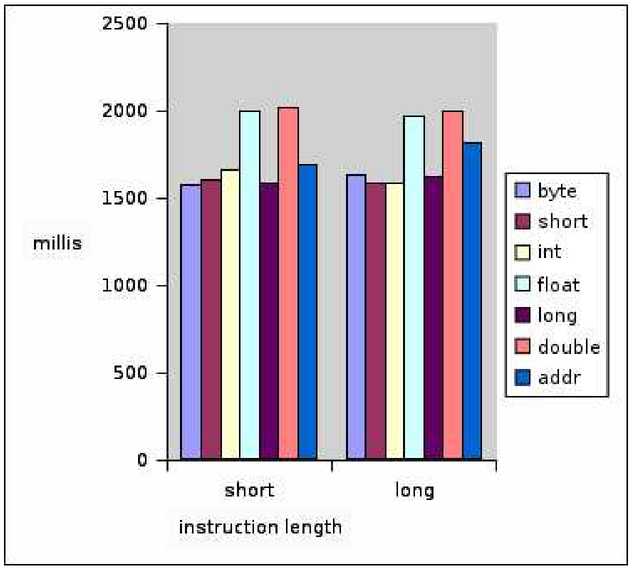

Figure 2: Instruction execution times on the IBM JVM.

Running Intel Pentium 4 2.66MHz on Gentoo GNU/Linux 2.6. This IBM JVM test shows that there are measurable differences in the load/store instruction execution times, but the largest difference here is a 7% speedup in the execution of nullary address load/stores.

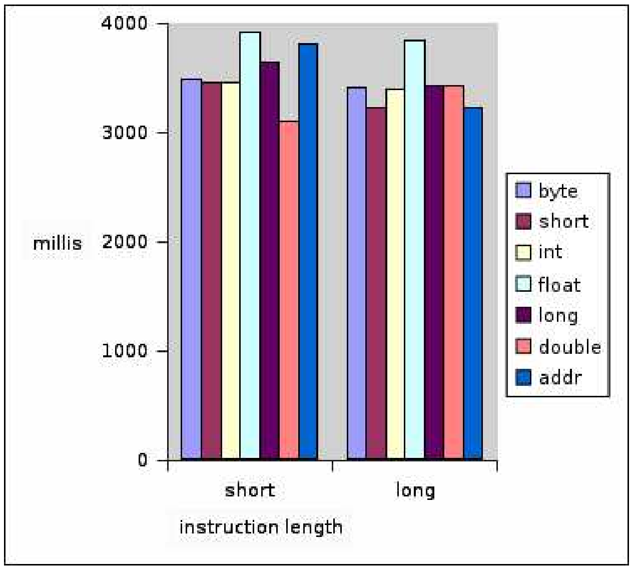

Figure 4: Instruction execution times for AMD Athlon 2400+ MP Dual running DebianGNU/Linux 2.6.

The nullary *double load/stores are faster than unary loads and stores on this platform, with a 10% improvement over the unary instructions.*

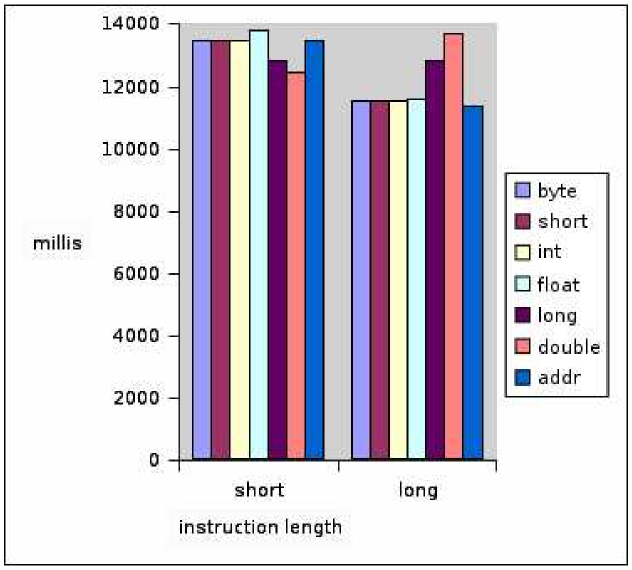

Figure 3: Instruction execution times for Intel Pentium 4 2.66MHz running FreeBSD R5.2.

The nullary double load/stores are the only load/stores faster than unary loads and stores on this platform, with a 10% difference.

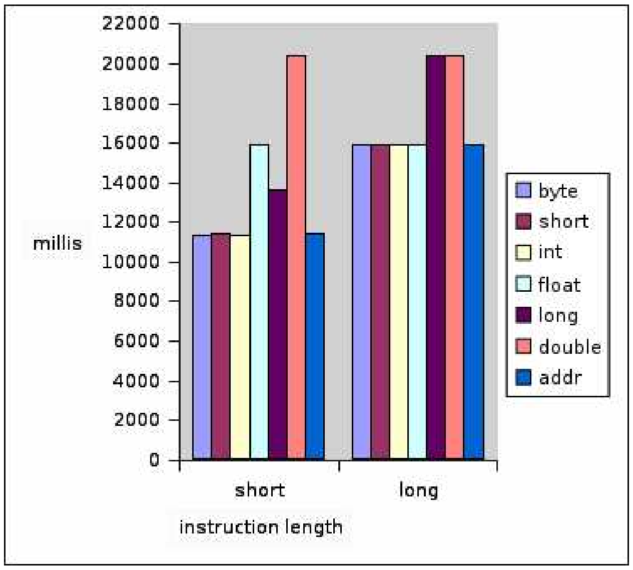

Figure 5: Instruction execution times for Sun Sparc Ultra-60 running Sun Solaris 5.8.

All nullary loads and stores are faster to execute than the unary loads and stores, with the long instructions seeing up to 50% improvement.

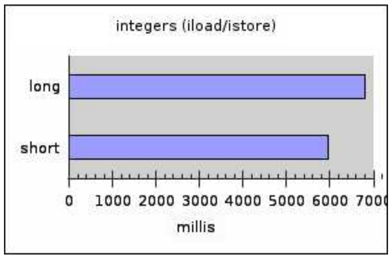

Figure 6: Detail of the int load and store times on the Pentium 4 running Gentoo GNU/Linux 2.6. There is a 14% increase in runtime when employing the unary loads and stores as opposed to the nullary loads and stores.

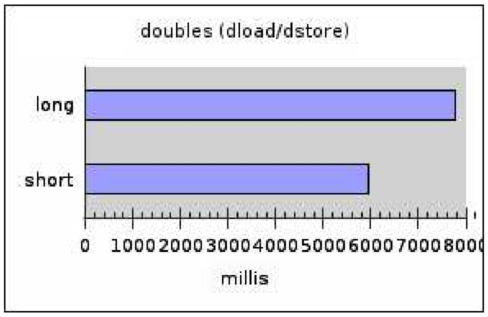

Figure 7: Detail of the double load and store times on the Pentium 4 running Gentoo GNU/Linux 2.6. There is a 30% increase in runtime when employing the unary loads and stores as opposed to the nullary loads and stores.

3 Problem Statement

My aim is to improve locals stack placement in the Java Virtual Machine by reordering locals by priority. I investigate whether one can expect gains in performance by more often using shorter load and store instructions. A quick investigation is made into the locals allocation strategies of the Sun Java compiler and the Soot compiler.

To this end, I adapt Chow and Hennessey's Priority-based colourer [7, 8] and the colouring heuristics of Briggs et al [9] to the Java Virtual Machine. I look at some potentially useful heuristics for weighting locals by priority.

The two locals placement allocators are implemented in the Soot Framework. The allocators are then used to compile several benchmark programs, which are subsequently timed to investigate what performance gains can be achieved with this approach.

4 Solution Strategy and Implementation

I measured the relative performance of the nullary and unary load/store. The torture test I devised involves repeatedly loading and storing a local in a tight loop, where the local is either one of the topmost four on the locals stack, or it is placed in position 37. The test was run on the short and long versions of the load/stores of each basic type, and the loop was executed 100 million times. The test was run twelve times, the highest and lowest results were discarded and the times were averaged out of ten. This was repeated on four different architecture/operating system combinations. With the exception of one test, which was run on the IBM JVM, the tests were run on the Sun VM as this was the only VM implementation readily available for all four platforms.

In Figures 1, 2, 3, 4 and 5 we see that although the timings are very platform-specific, the byte-long

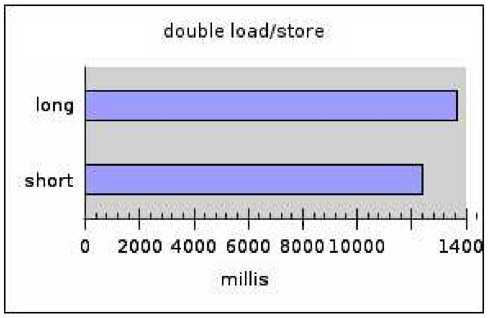

Figure 8: Detail of double load and store times on the Pentium 4 running FreeBSD R5.2. There is a 9% increase in runtime when employing the unary loads and stores as opposed to the nullary loads and stores.

load/stores in general execute faster than the unary load/stores, with the biggest differences being apparent in the int and double types. I therefore posit that reordering locals such that the most often accessed locals have the premium top four locations on the locals stack should give a measurable improvement in runtime performance for the general program.

Looking more closely at the figures, we see particularly large gains in execution speed with the shorter double operations across all architectures. On the p4 in particular, the difference is a whopping 30%! The shorter integer operations also see an improvement on the p4, while we can see that in general the longer operations are faster on the FreeBSD. The Sparc tests show that this platform could potentially benefit the most, as nullary operations are as good as if not far faster than the corresponding unary operations for all basic types, while the IBM JVM tests show that there is not much to gain with this particular optimization on this virtual machine.

I investigated the locals packing strategy present in the Sun Java compiler. The strategy is very simplistic, allocating the first locals position to the first declared local, and incrementally assigning locals as they appear in the Java source. The strategy properly accounts for scope, in that when a local has fallen out of scope, its position is freed for use by the next appearing local, but the allocation does otherwise not do anything clever. The Sun Java compiler does not optimize the bytecode at all, and this simplistic locals placement strategy seems to be what the IBM and Blackdown Java compilers have chosen to do as well.

The Soot optimizing framework does have a LocalsPacker module that explicitly packs locals; were this phase absent, some of the optimizing phases would cause an unnecessary blowup in the number of locals in a method. If the packing phase is enabled, the LocalsPacker reduces the amount of locals in jimple, which is a 3-address intermediate representation. The LocalsPacker interfaces with a FastColorer class that does the bulk of the work; this class employs a fixed-point liveness analysis to construct an interference graph, and attempts to find a good colouring by repeatedly assigning an available colour to each node in the graph. Furthermore, if the jimple intermediate representation is later taken directly to bytecode via JasminClass, which is the assembling module, the FastColorer is again called to produce an economical packing of the locals. However, if the program is taken to baf, an aggregated expression intermediate representation, locals are assigned in the order they appear in the baf method bodies. The FastColorer algorithm is detailed in Figure 9.

4.1 Locals Packing by Priority-based Colouring

To apply existing allocation algorithms to the JVM, we must keep in mind that we do not have any restriction on the number of positions to place our locals. Instead, we initially aim for a 4-colouring of the locals, to see if all the locals fit in our coveted top four stack locations. When the algorithm does not find a 4-colouring, it merely increases the number of colours available and restarts the algorithm, instead of spilling to memory as would be done in a k-register machine.

Figure 10 describes the Chow and Hennessey [8] priority-based colouring heuristic. The algorithm relies

```
for each method body
  do liveness analysis
  build interference graph

  for each local not already assigned a colour
    for each interference
      remove colour of interfering local from available colours
    colour this local with the highest-valued available colour
    if no colours are available,
      create a new colour with value one higher than highest colour
```

Figure 9: The Soot FastColourer algorithm.

on a heuristic for computing the relative weight of each local. At each iteration of the main loop of the algorithm, the colourer sets aside all nodes with neighbours fewer than the current number of colours as unconstrained nodes. This is because we are guaranteed to have colours available for such nodes. The nodes with more neighbours than available colours are put in a constrained nodes list.

Then, for each constrained node, if the number of coloured neighbours for that node is greater than or equal to the number of available colours, we simply increase the number of available colours. Otherwise, if the number of available colours is greater than the number of neighbours of this node, we can assign a priority to the node.

4.1.1 Assigning priorities to locals

To be able to assign priorities to locals, a good heuristic is needed to approximate which locals are going to be accessed most often. Ideally, we would assign a score based on how often a local is going to be accessed, but this information is never completely available before runtime, when the program has all its inputs. It is possible to use simple metrics, like the maximum abstract syntax tree depth each local appears in, but we would like to be able to transform programs for which we do not have the source. My approach is based on the weighted sum of all definitions and uses of a local, where higher weights are assigned to uses and definitions that appear in nested loops. Thus, the weight w_i of a local l_i is computed as:

$$w_i = \sum_{\forall defs(l_i)} 10^{depth(def(l_i)} + \sum_{\forall uses(l_i)} 10^{depth(use(l_i))}.$$

The loop depth of each definition is therefore required, and can not be simply looked up in the source code, because the intermediate representation is all that is available at optimization time. To find the depth of each use and definition of each local in the control flow graph, we use the algorithm detailed in Figure 12.

At each iteration, the NestDepth algorithm finds all loops of depth one in the control flow graph, and removes the heads of these loops from the graph. Any loops remaining in the graph will necessarily have been previously nested inside the detected loops, and will be found in the next iteration. This is illustrated in Figure 11.

4.2 Briggs et al alternate implementation of colouring heuristics

For comparison, I also implemented the Briggs et al [9] colouring-based register allocator, which is an improvement to the original colouring algorithm by Chaitin et al. [6].

The Chaitin algorithm is conceptually divided into three phases, and its outline is as follows:

1. Build interference graph.
2. Simplify graph: if there exists a node n with $deg(n) < k$, where k is the number of available colours,

```
PriorityColourer(program)
  for each method body
    do liveness analysis and build interference graph

    Constrain(locals)
    while there are unassigned constrained nodes
      get interferences of constrained nodes
      for each node,
        if number of coloured neighbours >= number of remaining colours
          increase k, the number of available colours
          Constrain(locals)
        otherwise
          AssignPriority(node)
    Color(constrained)
    Color(unconstrained)

subroutine Constrain(locals) returns sets: constrained, unconstrained locals
  for each local
    if coloured
      add to list of coloured locals
      if color value > k, the number of available colours
        set k to the colour value
      otherwise,
        if the number of interferences of this local < k
          set aside local as unconstrained
        else
          add to list of constrained locals

subroutine Color(local)
  for each local in order by priority
    get interfering colours
    colour node with lowest non-interfering colour
```

Figure 10: Pseudocode for the priority-based colourer.
Developed by Chow and Hennessey [8], with my modifications for the Java Virtual Machine.

remove n ad place it on the stack. Otherwise choose a node m for spilling, remove m from the graph and mark it for spilling.

3. Assign colours to nodes from stack: For each node n in stack order, insert n into interference graph, and assign n a colour not equal to the colour of its neighbours.

Briggs et al. developed improvements both in the simplification of the graph, and the selection of nodes for spilling. In the Briggs version of the simplify stage, we select for removal the lowest degree node and push it onto the stack. This intuitively makes sense, as the fewer neighbours it has, the easier it will be to colour that node, and so we can defer selection of a colour for that node until all nodes with greater degree have been coloured.

The selection of the lowest-degree node at each step can be made efficient by storing all nodes in an array N, where $N[i]$ is the first element of a linked list of nodes that have i neighbours. If there are no such neighbours, $N[i]$ is null. At fist N is searched from the beginning until the first non-null cell $N[i]$ is found. The node at the head of the list at $N[i]$ is removed, and each of its neighbours are moved one index down in N. Since we removed a node with degree i, this might have created nodes with degree $i-1$, and we can start the search in the next iteration at index $i-1$, as we know that no nodes of lower degree are available.

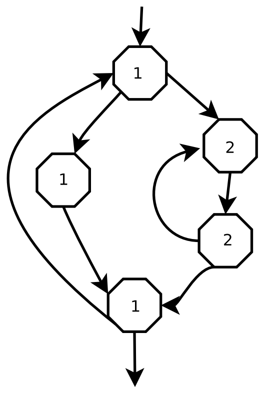

Figure 11: Control flow graph strongly connected component illustration.
The nested depth algorithm first finds all loops of level 1, then removes the entry points of these loops and repeats on the resulting graph. Shown here is the labeling that would result were this control flow graph fed as input to the NestDepth algorithm.

```
set depth of all nodes in the control flow graph to 0
find strongly connected components in the control flow graph

for each strongly connected component with size > 1
  find entry points, ie. nodes with predecessors not in the component
  increment the depth of nodes in this component
  remove the entry points of this component from the graph
repeat until there are no strongly connected components of size > 1
```

Figure 12: The NestDepth algorithm
for finding the depths at which each local is used or defined. A strongly connected component of a graph is a set of nodes in which there exists a path between any two members of the set.

Because my work is on the Java Virtual Machine, I do not have to worry about spilling. If there is no node n with $deg(n) < k$, then the algorithm can not find a k-colouring of the graph, and k merely has to be incremented, resulting in increasing the locals limit for this method body.

5 Experimental Framework

Experiments were designed and run to show the relationship between runtime of test programs and the

During the development of the allocation algorithms it proved valuable to compile test programs and benchmarks with the allocators and disassemble the class files, comparing the resulting allocation to the original Soot FastColorer allocation, even when the classfile produced did not pass muster in terms of verifiability. Produced, verifiable classfiles were run on the JVM to test the correctness of the algorithms, checking for differences in program behaviour.

Some small test cases were output to the screen through Soot's verbose logging facility and intermediate representation tagger, and hand-verified on paper. This is especially true for the NestDepth locals weighting algorithm, since the preliminary results did not lend themselves for testing in any other way.

Finally, a suite of benchmarks was compiled with Sun Javac, ABC: the AspectBench compiler and AJC, where applicable and their runtimes were recorded. The compiled classfiles were then transformed with the standard Soot FastColorer as well as the priority-based colourer and the Briggs colourer, and their runtimes were recorded. Each transformed version of each benchmark was timed seven times, the highest and lowest times were discarded and the remaining five times averaged.

	Compiled	Soot	Priority	Briggs
Average	37.5468 s	38.0503 s	38.0364	37.6004
Ratio/compiled	1.0000	0.9868	0.9872	0.9986
Ratio/Soot	1.0134	1.0000	1.0004	1.0120

Table 1: Comparison of runtimes averaged over all benchmarks.
Shown are the averages for each set of classfiles in seconds, the relative improvement compared to the Sun javac-compiled runtime and the relative improvement compared to the standard Soot allocation.

6 Results

I show that although the performance gains are highly dependent on the benchmark application, there is on average a 1.2% performance gain compared to the standard soot allocator, and on average the same running time as classfiles compiled with the Sun Java compiler, as is apparent from Figure 6. One particular benchmark, *Scimark*, showed a 10% increase in performance when using the priority-based colourer compared to the Java compiler and original soot packers, while another benchmark, *chromosomes*, took a 15% hit in performance when using the priority-based allocator. Figure 6 shows that the Briggs allocator comes out best overall, with most timings being equal in performance to the Sun javac timings while a select few (*asac* and *fractal*) finish about 5% earlier.

6.1 Experimental environment

All final benchmarks were performed on a Pentium 4 2.66GHz 512MB RAM running Gentoo GNU/Linux with kernel version 2.6.7. Each benchmark program was compiled using ajc or, in the non-aspect case, plain Sun javac. Then, each benchmark was transformed using the latest development version of unmodified Soot, and also transformed using the PriorityColorer and the BriggsColorer. This produced four sets of classfiles for each benchmark. Each set of classfiles for each program was timed 7 times with the standard Unix program time(1). The highest and lowest times were discarded, and the remaining 5 times were averaged to produce the final benchmark timing.

6.2 Benchmark programs

For my testing I used the McGill COMP-621 class of 2005's benchmarks. There were 13 benchmarks in total, each with their own special characteristics. Here I give a brief summary of the notable characteristics of each benchmark program.

- *AORecoveryBlock* - Aspect-oriented implementation of a recovery block for software fault-tolerance. Uses a lot of reflection.
- *asac* - Aspect-oriented multithreaded comparison of three sorting algorithms.
- *bookstore* - Aspect-oriented bookstore simulation with lots of reflection.
- *chromosomes* - Aspect-enabled genetic algorithm to maximize a function $f(x)$ over some domain.
- *eigenv* - Computes the eigenvalues and eigenvector of a singular matrix created by computing the Fibonacci number of each entry of the original matrix recursively. Array and integer-heavy.
- *Fractal* - Draws a Fractal tree to a certain depth as specified by the user. Aspect-enabled for profiling. Lots of recursion.
- *Imaging* - Transforms an input image by using the Java Imaging Utilities library. Loop and array-heavy.
- *Rasterizer* - A triangle rasterizer sampling randomly-coloured triangles with 9X anti-aliasing. CPP2Java compiled, interesting program structure.

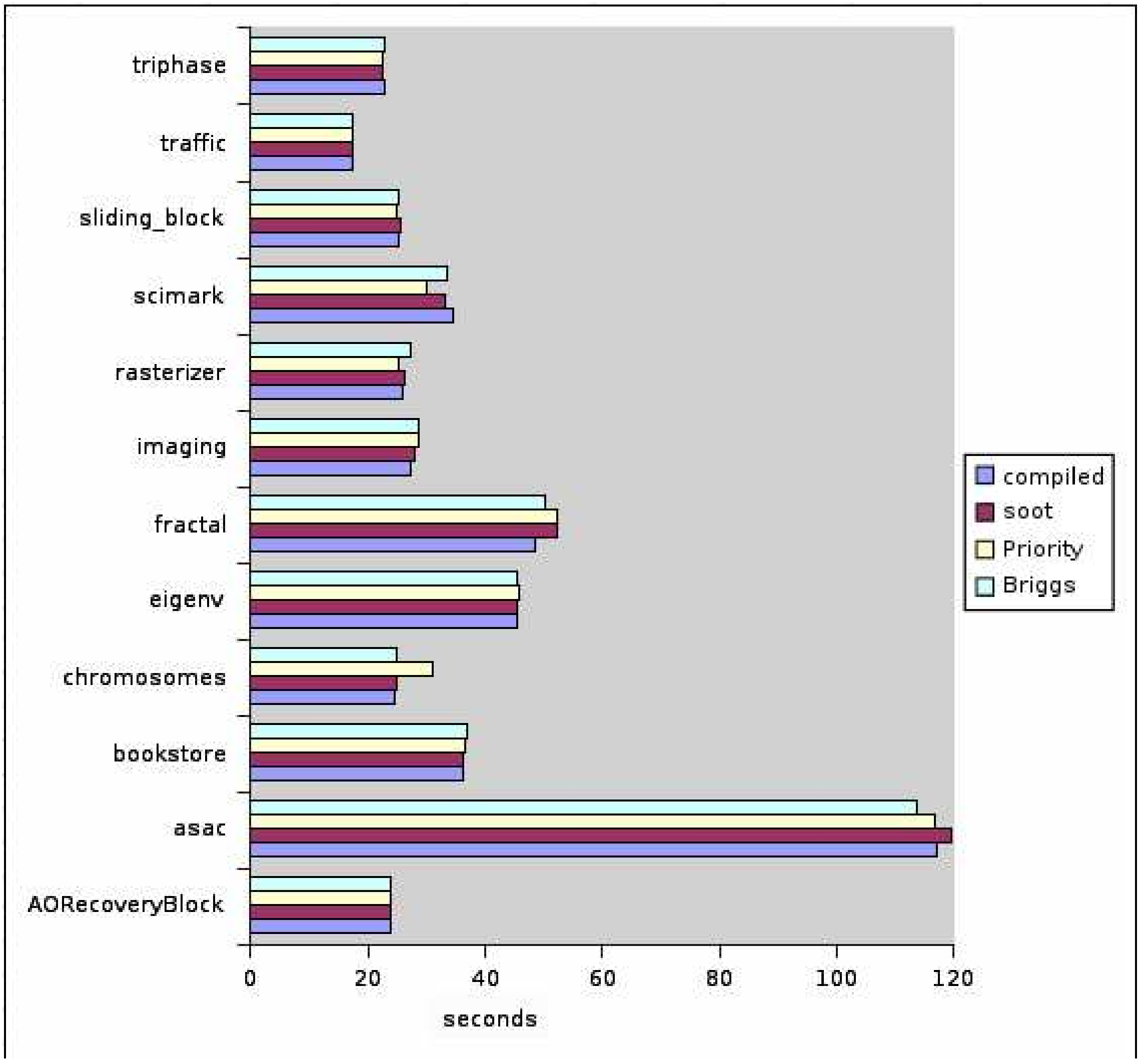

Figure 13: Comparison of benchmark runtimes.
Shown are runtimes for benchmarks compiled with Sun javac, transformed with unmodified soot, transformed with the priority-based colouring algorithm and transformed with the Briggs allocation strategy.

- *Scimark* - Well-known numerical computing benchmark, including FFTs, Jacobi SOR, matrix-multiply, Monte Carlo integration and LU-factorization. Matrix and floating-point heavy.
- *sliding_block* - A program to solve the sliding block puzzle with A* exhaustive search. Memory-heavy and aspect-enabled.
- *traffic* - Simulates cars moving past intersections on roads. Multithreaded with heavy use of semaphores, condition variables and monitors.
- *tribench* - Three-part benchmark.
  1. Gaussian elimination with partial pivoting.
  2. Massively-multithreaded matrix multiplication.
  3. Multithreaded Sieve of Eratosthenes.

  Aspect-enabled for profiling.
- *wig11* - Compiler implementation that compiles a web gateway source language into python. Gigantic switch statement for control flow in the auto-generated parser.

6.3 Results and analysis

We can see that the results depend heavily on the type of benchmark program. Priority-based reordering of locals, and Briggs improved colouring heuristics both clearly have an effect on execution time, which can be said to be positive overall. Seeing a 1.2% improvement overall over the original Soot locals packer is relief, as my approach is far less simplistic. This is a good sign that the reordering of locals is doing what it is supposed to be doing: placing the most frequently accessed four locals on top of the locals stack for each method body. Overall, my locals packing algorithms, and the Briggs colourer in particular bring the Soot Framework closer in terms of performance to Sun javac-produced bytecode.

We see in Figure 6 that while the PriorityColourer produces the highest improvements - up to a whole 10% on *Scimark*, the strategy also produces the poorest results for other benchmarks: a 15% increase in running time on the *chromosomes* benchmark. In contrast, the Briggs colourer performs rather evenly at a little above the Soot packer, with modest improvements of 1% to 5% on select benchmarks.

The *traffic* benchmark shows no improvement because it spends 98% of its running time waiting for locks. We also do not see any significant changes in running time for *triphase*, *sliding_block*, *eigenv* nor *AORecoveryBlock*, possibly because these benchmarks are all balanced and large enough to negate the effects of the heuristics. Although the differences are marginal, both *bookstore* and *imaging* show a slight increase in running time for the classfiles that employ my new locals allocation. The Briggs colourer shines in *fractal* and *asac*, while the priority colourer in both cases keeps up with the pack; in the *fractal* case, it does as well as the Soot colourer, while in the *asac* case, both allocators outperform the Sun-built classfile.

The largest differences can be seen in *scimark* and *rasterizer*, where the priority-based colourer clearly find a better ordering on locals, and in *chromosomes*, where the same strategy fails pitifully, blowing up with a 15% increase in running time over the three other alternatives.

On a final note I would like to remark that the Wig11 benchmark never finishes compiling with the new locals packers - the compilation phase simply runs far longer than the author's patience. Preliminary profiling suggests that the problem may be in the implementation of NestDepths somehow blows up on the extremely dense control flow graph of the auto-generated parser that is part of Wig11, because that's where the execution stalls and cannot proceed.

7 Conclusions and Future Work

I have presented experimental results showing that execution time may be improved on several architectures by reordering locals so that the most used ones receive the first four stack locations. I have modified two existing register allocation algorithms to work with the Java Virtual Machine, and I have implemented these algorithms in the context of the Soot Optimizing Framework. I have shown that assigning priorities to locals can measurably affect performance on all architectures, improving the performance of programs compiled with Soot by 1.2% on average, and in particularly well-conditioned cases by up to 10%.

The never-terminating compilation of the Wig11 benchmark is a stumbling block that would have to be resolved before the allocation strategies developed in this paper could be put to general use. The weight function could be improved to take into account for example the relative benefits of int and double types on the Intel architecture, to prioritize these types over for example longs, which do not see any improvement on any architecture/operating system combination. A constant propagation analysis can be used to better determine the number of times a loop is executed, which could lead to a better approximation for the weighting of locals.

Due to time constraints, investigations into the merit of reordering locals were not completed: it would be helpful to see, for each class file or program, the sum of weights over type, and the maximum depth that each type appears in. It would further be helpful to examine whether the new allocators show improvement in benchmark execution time on virtual machines other than the Sun JVM. Also, it would be interesting to see if the register allocators significantly improve execution on the Sun Sparc architecture, as promised by the preliminary results.

A Appendix

A.1 Source code and online copy of project report

The *PriorityColorer*, *BriggsColorer* and all auxiliary source files, along with a patch for the latest version of soot and a tarball can be downloaded from `http://kacper.doesntexist.org/optimizing/`. An online copy of this document can also been found on the same page.

A.1.1 A Better Mutable Graph

During the course of development, I found the need to extend the existing Soot *MutableGraph* to supply breadth-first-search and depth-first-search traversals of the graph, along with a conversion algorithm that, given any *DirectedGraph*, will produce an equivalent *MutableGraph*.

A.2 A note on the spelling of "colour"

The author finds the British spelling of colour to be the most natural. However, the existing Soot implementations refer exclusively to "colors", so for consistency it was not possible to in good faith use a different spelling for the implementations in this project. It is for this reason that this report uses the British spelling except where referring to the names of objects that are part of or additions to the Soot Framework.

References

[1] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 125–135. IBM Press, 1999.

[2] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[3] Andrew W. Appel. Chapter 11, register allocation. In *Modern Compiler Implementation in C*, pages 235–264. Cambridge University Press, 1998.

[4] V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. 1998.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[6] G.J. Chaitin, M. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, , and P. Markstein. Register allocation via coloring. In *Computer Languages*, pages 6:47–57, 1981.

[7] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[8] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 222–232. ACM Press, 1984.

[9] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 275–284. ACM Press, 1989.

[10] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[11] Jonathan Derryberry and Manfred Lau. Incorporating better register allocation into jikes. 2003.

[12] J. Hummel, A.Azevedo, D.Kolson, and A. Nicolau. Annotating the java bytecodes in support of optimization. In *Concurrency: Practice and Experience, 9(11)*, pages 1003–1016, 1997.

[13] Raja Vallée-Rai. Profiling the kaffe jit compiler. Technical report, 1998.

[14] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of java interfaces: Invokeinterface considered harmless. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 108–124. ACM Press, 2001.

[15] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 176–191. Springer-Verlag, 1992.