Incorporating Better Register Allocation into Jikes

Jonathan Derryberry
Manfred Lau

December 4, 2003

Abstract

In dynamically compiled code, it is important not only that the compiler produce fast code, but also that the compiler itself costs little overhead to the runtime application. These two competing goals motivate the importance of accurate cost-benefit analysis for any optimizations performed in a dynamic compilation system. The Jikes RVM, which performs Just-In-Time (JIT) compilation of Java bytecodes, is an example of such a dynamic compiler. One of the most important components of a JIT compiler is register allocation because memory accesses are becoming increasingly expensive relative to register accesses. Jikes has opted to use exclusively a linear scan register allocator in its optimizing compiler because the linear scan algorithm is fast and provides performance nearly as good as more complicated register allocation algorithms. This paper seeks to improve the runtime performance of applications run by the Jikes RVM by incorporating a more exhaustive register allocation scheme for use on code for which the benefit would be large enough to justify the extra compilation effort. This was accomplished with partial success by adding an additional optimization level to the Jikes RVM and calibrating the necessary cost-benefit parameters in the configuration files. Our results suggest that the modifications we made improve the performance of the Jikes RVM on computationally intensive code without sacrificing much performance on smaller jobs.

1 Introduction

With the cost of memory accesses growing relative to the cost of accessing registers, performing good register allocation is becoming an increasingly important challenge for modern compilers. This is especially true for dynamic compilers, which compile and optimize code at runtime.

In such compilers, to minimize the time that it takes programs to complete their jobs, it is important to employ cost-benefit analysis for all optimizations and compilation choices that are made to determine the benefit of performing each optimization versus the cost of enacting it. Thus, to make the best decisions possible, it is necessary to obtain information about where the code is spending most of its time so that optimization efforts can be focused on where they will reap the greatest rewards. As such, the runtime optimizations are tailored in a way that is impossible statically, unless profiling is done ahead of time. In theory, it is even possible to optimize frequent control flow paths at the expense of infrequent ones, but the emphasis of this paper is on providing uniform improvements to all paths via better register allocation.

This project concerns the improvement of register allocation in the Jikes RVM, which dynamically compiles Java, by deploying such cost-benefit analysis to determine whether a graph-coloring-based register allocation scheme we implemented is worth performing on particular methods during runtime.

2 The Jikes RVM

This section provides an overview of the Jikes RVM as is relevant to our project, illuminating important key points that are necessary to understand what we did while skipping confusing details. For more detailed information, see [1] and [2].

The Jikes RVM provides Just-In-Time (JIT) compilation for Java classes, because static compilation and analysis cannot be carried out as a result of the Java specification, which allows for dynamic class loading. Because compilation occurs at runtime, the goals of the compilation system are significantly tempered by the fact that execution time includes not only the time needed to execute the code of the currently running application, but also the time needed to run the code of the JIT compiler on the code that it compiles. This phenomenon adds an extra dimension of complexity to compiler development.

To address this complexity, two compilers were created for compiling code that is run by the Jikes RVM. The first compiler, the *baseline* compiler, performs no optimizations and simply emulates Java's operand stack. This compiler costs virtually no overhead and is used for code that is not expected to be run very many times (e.g. code that is running for the first time). The second compiler, the *optimizing* compiler, creates an internal representation of the code and performs standard optimizations on the code at various levels of optimization, each level costing more time to execute, but theoretically making the resulting code run faster.

In this paper, we are modifying the Jikes register allocation functionality, which resides solely in the optimizing compiler. Currently, the Jikes RVM runs the same register allocation algorithm at all levels of optimization. Jikes uses the linear scan register allocation algorithm described in [3]. The linear scan algorithm first sorts the live intervals in ascending order according to the depth first numbering of the first instruction in the interval. Then, iterating through the resulting list, it greedily allocates registers and frees registers when their intervals have ended. It spills a register when there are more live intervals than registers, in which case the register with the lowest estimated spill cost is spilled. The linear scan algorithm was chosen because it executes quickly and generally produces register allocations that are as good or nearly as good as more complicated graph coloring algorithms.

3 The Theory Behind Our Improvements

By only providing a linear scan register allocation algorithm, which the creators of Jikes admit is not quite as good as graph coloring, there is the opportunity to squeeze out increased performance by providing a more exhaustive register allocation algorithm for code that is guessed to be run for an extremely long time. Ideally, performance would not suffer at all for short running programs, which never have any of their methods compiled at high enough optimization levels that the exhaustive register allocation is performed.

With an NP-hard optimization problem, such as register allocation, the best algorithms for solving them tend to exhibit a tradeoff between time spent searching for a good solution and the quality of the solution. Typically the quality of the solution, $f(t)$, increases monotonically throughout time t, asymptotic to the horizontal line representing the optimal solution (for register allocation, $f(t)$ represents the rate at which the method runs, such that a value twice as large implies that the method typically takes half as much time to finish running). As such, the benefit of spending Δt more time compiling a method is given by

$$B(T_m, t) = T_m \left(1 - \frac{f(t)}{f(t + \Delta t)}\right) - \Delta t, \tag{1}$$

where T_m represents the expected addition time spent running the code and Δt is subtracted to account for the compilation cost.

If optimizations can be performed on a small enough time scale and we first divide through by ΔT, the above equation can have its limit taken as $\Delta t \to 0$ and we have

$$\frac{\partial B(T_m, t)}{\partial t} = T_m \cdot \frac{f'(t)}{f(t)} - 1, \tag{2}$$

which represents the *marginal benefit of recompilation*, or the rate at which the application approaches completion as the result of an infinitesimal amount of recompilation. The full rate at which an application approaches completion is given by

$$R(T_m, t, \alpha) = (1 - \alpha) \cdot f(t) + \alpha \cdot \frac{\partial B(T_m, t)}{\partial t}, \tag{3}$$

where α represents the fraction of system resources appropriated to compilation. The optimal value of α could be found by differentiating $R(T_m, t, \alpha)$ with respect to α and setting the result equal to zero.

4 The Mechanism Jikes Has for Implementing the Theory

Although the theory for implementing register allocation using cost-benefit analysis is simple and pure, the Jikes RVM is a complex system and does not provide a framework for performing perfect cost-benefit analysis and infinitely fine grained precision over recompilation decisions at incrementally higher levels. This section describes the practical decisions that were made in the Jikes RVM in the spirit of the theory in Section 3 as well as some key details about how register allocation is performed and evaluated the the Jikes RVM.

4.1 Built-In Cost-Benefit Analysis

Because the Jikes RVM compiles code at run time, whichever compilation decisions that it makes should be based on an analysis which attempts to determine whether the code will finish execution more quickly if it expends the extra effort to perform sophisticated optimizations on the code to make it run more quickly. In fact, Jikes already uses such cost benefit analysis when making decisions to recompile code at higher and higher optimization levels [2].

Because the Jikes RVM performs all of its compilation at the method level, it uses its built in sampling functionality and heuristics to compute the expected amount of time T_m each method will spend executing before the currently running application exits. From this, it uses statically determined estimates to compute how much recompilation will decrease overall runtime, $T_m(1 - \frac{r_1}{r_2})$, where r_1 is the current rate of execution of the method and r_2 is the predicted rate if compilation were performed. This savings is compared with the cost of recompilation, which is also computed using static information. The cost is defined to be the number of bytes in the method times the rate of bytes compiled per second for the current optimization level (clearly this can be erroneous if the optimization algorithms are not all linear time).

In sum, Jikes provides a practical means of applying cost-benefit analysis at a coarse level of granularity.

4.2 Performing Register Allocation

Some aspects of register allocation have already been taken care of in the Jikes RVM register allocation framework. For example, there are already mechanisms for computing, and optionally splitting, live intervals. There are existing methods for computing the costs of spilling the live intervals to memory, which take into account the hotness of the code in which the register accesses lie (this theoretically provides an even better heuristic for the cost of spilling a register than static techniques, which typically use the nesting depth of the register accesses). Also, there are already register and spill coalescing graphs that aid in performing coalescing if desired.

5 Modifications to the Jikes RVM

We made several modifications to the Jikes RVM in order to try to harvest some of the theoretical improvements that appeared possible for register allocation in the Jikes RVM.

5.1 Adding an Extra Optimization Level

It would have been nice if it were possible to operate on the level of pure theory, and perform incremental register allocation at a fine grained level using a progressive search for better and better ways to allocate the registers. However, register allocation was a very complicated task in the Jikes RVM and it relied on many other phases of the optimizing compiler. Therefore, to implement a new register allocator without having to figure out all of the details of the Jikes RVM, we opted to use a similar framework to what was used in the other optimization passes.

We added an extra optimization level by making changes to the Jikes RVM options file so that in addition to the existing optimization levels, 0, 1, and 2, there would be optimization level 3, which would run the same phases as level 2, but substitute our new register allocator for linear scan. Also, we had to modify the class `VM_CompilerDNA` so that it would have cost-benefit information for how long the new optimization level would take to complete and how fast it would speed up code. This was done in order for Jikes to find the right opportunities to run optimization level 3.

5.2 Implementing A More Exhaustive Register Allocator

With the decision to add only one additional level of optimization made as a result of underestimates of the difficulty of successfully modifying the Jikes RVM under time pressure, we opted to implement a relatively simple and well-established graph-coloring-based register allocation scheme, using the ideas found in [4].

Our register allocation algorithm builds an interference graph among all of the symbolic registers that need to be allocated and records all potential colors for each node, which are determined by the type of register each node needs (e.g. non-volatile). The interference graph is constructed using the compound intervals that were created during a previous interval analysis phase. Each compound interval represents a set of live intervals that all belong to the same register, symbolic or physical. Using the interference graph and the potential colors for each node, we add all nodes that have fewer neighbors than potential colors to a stack, and delete them from the graph. If we encounter a situation in which there are registers left to color but there are no nodes that have fewer neighbors than children, we use the standard heuristic of choosing as a potential spill the remaining node that has the highest ratio of degree to spill cost, push it on the stack, and mark it as a potential spill.

When there are no nodes left to push onto the stack, we pop the stack and attempt to color each node as it comes off. If the existing Jikes code finds a good coalescing candidate, then that register is used. Note that only potential spills may have to spill, and even they will not necessarily spill depending on how the nodes above them in the stack were colored. If a spill is required, then the existing Jikes register allocation code is used to find a good spill interval by trying to find one that will coalesce spills. All in all, the register allocation phase we implemented was simple, but nevertheless theoretically more powerful than linear scan although it runs asymptotically more slowly. In practice, however, it was found to run within an order of magnitude of the running time of the linear scan algorithm.

Ultimately, we found that the best way to ensure the correctness of our code was to use all of the same register allocation infrastructure (e.g. live analysis, spill cost estimation, spill code generation, et cetera) and figure out the black-box effect of the linear scan pass so that we could make the necessary changes so that the following passes would insert the correct spill code and make the correct changes from fake, symbolic registers to real, physical registers.

5.3 Calibrating the Jikes Optimizing Compiler

As mentioned in Section 5.1, there was cost-benefit data that had to be calibrated in the class `VM_CompilerDNA`. More specifically, for the new optimization tier, level 3, we had to enter its compilation rate, which was based on the number of bytes compiled per second as described in Section 4.1.

Therefore, we collected a set of test cases, mostly gathered from the Jikes RVM test case directory, and measured the average total compile time of each method over 5 trials, via system calls in Java after configuring Jikes not to run compilation in a background thread. This test plan was executed for each of the four optimization levels, 0, 1, 2, and 3 (we modified the cost-benefit parameters in each case to make Jikes run each level of optimization exclusively). The results of these tests for each optimization level appear in Tables 1, 2, 3, and 4.

The median compilation rate was computed for each optimization level, and the `VM_CompilerDNA` class was configured accordingly so that the compilation rate of optimization level 3 was normalized to the same scale as the existing values (with some interpolation for incongruences with the existing values in `VM_CompilerDNA`).

Next, the speedup rate for optimization level 3 had to be calibrated. This was accomplished by recording the execution time for test cases compiled using each optimization level. We use the term *running time* to denote the total time spent running code including the time spent compiling the running application's methods, and the term *execution time* to denote the time spent running the actual methods of the application (i.e. the running time minus the time spent compiling methods). The resulting execution times and corresponding normalized speedup values are reported in Tables 5 and 6. The median speedups were used, just as the median compilation rates, to interpolate a good speedup value to use in `VM_CompilerDNA`.

During these tests, we noticed that the resulting values did not fit into accord exactly with the Jikes normalized calibration values. This was to be expected as performance gains naturally vary across different programs, but our tests at times showed seemingly unrealistic speedups in some cases. In the most prominent example, we recorded a speedup of 8.91 for optimization level 1 in one of the test cases. It seemed unlikely to us that this was the result of pathological cases that allowed certain optimizations to reap order-of-magnitude-sized benefits.

We were unable to track down the source of this variance, although we would hypothesize that our method for splitting apart measured compile time from measured execution time was not completely robust. Although we configured Jikes not to run compilation in a background thread, it may have nevertheless interleaved the execution of optimization passes with the execution of the test cases. This may not have greatly affected compile time measurements, which may have been *uniformly* slowed down so that normalization masked the error, but it would have affected the measured execution speedup because in these test cases, a significant fraction of execution time was spent in compilation. Given more time, we could investigate this hypothesis by creating very long running test cases so that compile time would comprise an insignificant portion of the total running time of the programs.

6 Measuring the Effect of the Changes

After calibrating the Jikes RVM as described in Section 5.3, we performed several tests to determine whether we had succeeded in improving runtime performance.

Our test setup involved running two test cases, `Hanoi` and `Perm` (See Section A), on successively larger inputs. Although this is a narrow test set to use, it nevertheless provides insight into the effect of our additions to the Jikes RVM on tasks requiring a range of computational requirements. After adding the compilation rate and speedup rate for our additional optimization level to `VM_CompilerDNA`, we measured total running time (including compilation) for increasingly large inputs to programs. The speedup of running the programs with our new level over the speed of

running Jikes without our additional level was graphed relative to the size of the problem (which caused a corresponding exponential increase in the running time of both `Hanoi` and `Perm`). See Figures 1 and 2 for these results.

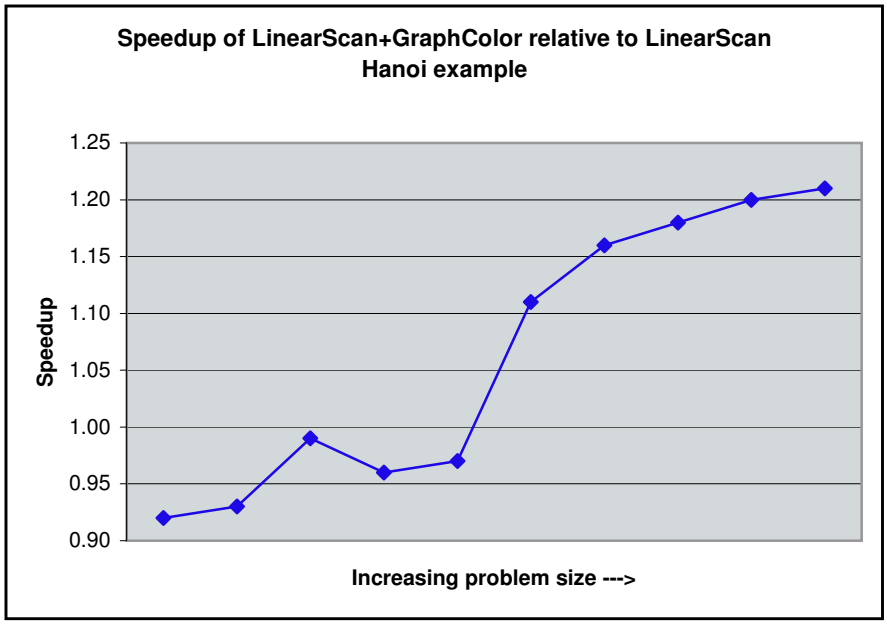

Figure 1: Relative speedup resulting from adding our optimization level to the Jikes RVM for the `Hanoi` test case on successively larger inputs.

Because the data we collected while calibrating our optimization level's cost-benefit parameters did not fully agree with the parameters already listed, we tried setting the parameters according to the data we gathered and re-ran the `Hanoi` test cases. The resulting graph, shown in Figure 3, shows that there was not a large effect on each of the test cases, and performance even suffered significantly for the smallest test case.

To summarize the results from our tests, we found encouraging results that showed that it was possible to speed up the running time of Jikes by adding an extra optimization level. For computationally intensive jobs, our addition appeared to speedup the running time of the code, while smaller jobs did not suffer greatly in most cases. However, although the downside of adding our level appeared to have been mitigated by setting the cost-benefit parameters appropriately, the persistent performance decline by about 10 percent on small test cases suggests that further attention must be given to make the cost of adding an additional optimization level imperceptible for such cases.

7 Future Work

The results of this project suggest the various additions that could be tried for the Jikes RVM.

For example, in addition to the more exhaustive graph-coloring-based register allocator that was created during this project, it would be interesting to see if performance on small jobs could be improved by creating a very lightweight optimization level that used an even faster register allocation scheme than linear scan, or a simpler version of linear scan than the relatively rich version used in Jikes.

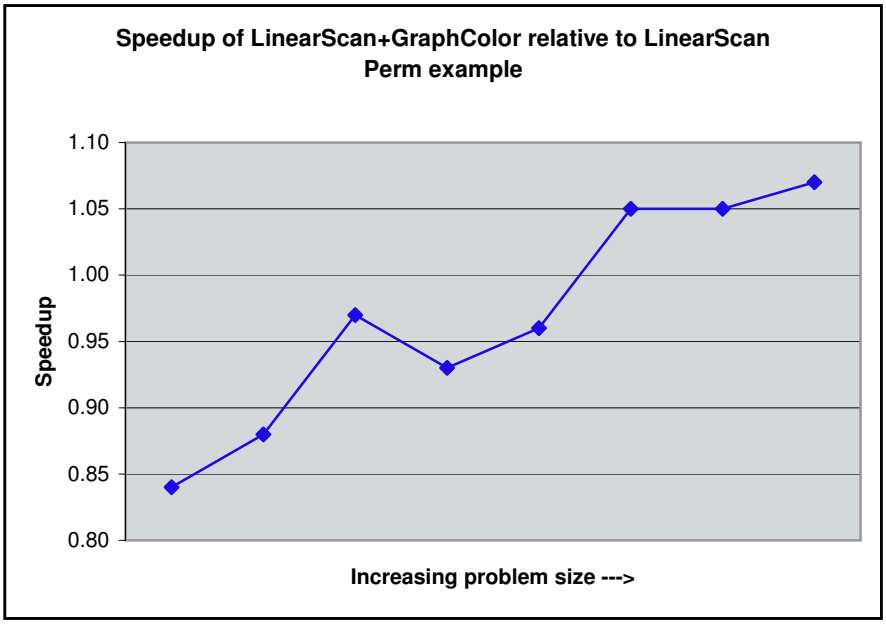

Figure 2: Relative speedup resulting from adding our optimization level to the Jikes RVM for the `Perm` test case on successively larger inputs.

Also, because register allocation is so important, it would be interesting to see if additional optimization levels could be added that did not require all of the other parts of the compiler to be executed so that simple tweaks could be made to the register allocations without redoing the high-level optimizations. This would require better knowledge about the Jikes RVM system as a whole.

8 Contributions and Accomplishments

In this project, we have accomplished the following salient goals:

- Added an additional dynamic optimization level to the Jikes RVM that uses a more exhaustive register coloring algorithm to perform register allocation.
- Used performance analysis along with the Jikes cost-benefit framework to minimize the down-side of the additional optimization level (e.g. time wasted compiling programs that will reap little benefit) while keeping most of the upside (e.g. time saved from the recompilation of methods for which there will be a significant benefit).

In addition to these contributions, we surmounted the complexity of the Jikes RVM, which had kept us behind schedule, and revised our goals described in the project proposal found at

http://www-2.cs.cmu.edu/~jonderry/15745project/project_proposal/proj_prop.pdf

so that we could realistically get a potential improvement to the Jikes RVM working correctly and have time to test and hone its performance.

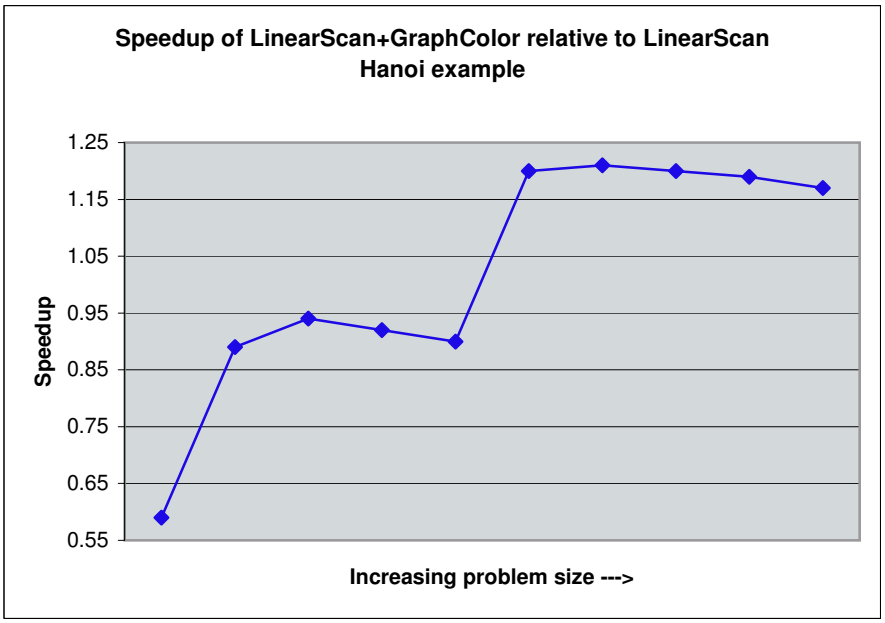

Figure 3: Relative speedup resulting from adding our optimization level to the Jikes RVM for the `Hanoi` test case on successively larger inputs when we modified the cost-benefit parameters of all optimization levels, not just ours, to conform to the ratios that we observed.

9 Acknowledgments

Thanks to Todd Mowry for providing guidance that helped us come up with the specific idea for our project. Also, Pedro Artigas and Jeff Stylos gave us general advice about the Jikes system and enlightened us with their knowledge of how it worked. Finally, thanks to Stephen Fink, one of the core developers of the Jikes RVM, for answering a few questions by email regarding where we should make changes to the Jikes RVM to accomplish our goals.

References

[1] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[3] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

A Results of Cost-Benefit Calibration Tests

The following tables show the results of our calibration tests. Descriptions of the test files are as follows:

- `Fibo` computes a specified Fibonacci number
- `Hanoi` solves the towers of Hanoi problem on a specified input size
- `Matmul` multiplies matrices of specified size together for a specified number of iterations
- `Perm` computes all permutations of an array of specifiable size
- `Repeatfoo` repeats a simple function 1 billion times
- `Sieve` computes the number of prime numbers up to a given number
- `Testsort` generates a specifiable number of random arrays of specifiable size and sorts them
- `Whet` is the Whetstone benchmark for testing floating point performance.

Test File	Bytesize	Compile Time (ms)	Rate (bytes/ms)
Fibo	48.8	94.2	0.52
Hanoi	96.6	151.8	0.64
Matmul	153.2	225.6	0.68
Perm	354.6	404.8	0.88
Repeatfoo	281.4	273.8	1.03
Sieve	432.6	760.4	0.57
Testsort	976.2	2446.2	0.40
Whet	2452.0	2302.4	1.06
Median			**0.66**

Table 1: Average over 5 trials for total bytesize of compiled code, total compile time, and bytes compiled per ms, for optimization level 0.

Test File	Bytesize	Compile Time (ms)	Rate (bytes/ms)
Fibo	93.2	754.2	0.12
Hanoi	89.0	183.4	0.49
Matmul	275.2	2177.2	0.13
Perm	598.4	2306.0	0.26
Repeatfoo	642.2	4704.0	0.14
Sieve	678.8	4806.4	0.14
Testsort	1053.8	6169.4	0.17
Whet	2865.4	6864.0	0.42
Median			**0.16**

Table 2: Average over 5 trials for total bytesize of compiled code, total compile time, and bytes compiled per ms, for optimization level 1.

Test File	Bytesize	Compile Time (ms)	Rate (bytes/ms)
Fibo	423.6	8253.4	0.05
Hanoi	514.4	10133.2	0.05
Matmul	809.4	13499.4	0.06
Perm	5416.6	50685.2	0.11
Repeatfoo	1852.0	21852.0	0.08
Sieve	3079.0	31134.6	0.10
Testsort	7742.2	72919.6	0.11
Whet	2647.2	13827.8	0.19
Median			**0.09**

Table 3: Average over 5 trials for total bytesize of compiled code, total compile time, and bytes compiled per ms, for optimization level 2.

Test File	Bytesize	Compile Time (ms)	Rate (bytes/ms)
Fibo	162.0	2826.4	0.06
Hanoi	657.4	11951.8	0.06
Matmul	748.0	13750.0	0.05
Perm	6239.4	62538.0	0.10
Repeatfoo	1327.0	18954.8	0.07
Sieve	2599.2	30632.0	0.08
Testsort	6898.4	70506.6	0.10
Whet	2589.6	14406.4	0.18
Median			**0.08**

Table 4: Average over 5 trials for total bytesize of compiled code, total compile time, and bytes compiled per ms, for optimization level 3 (using our graph coloring register allocator).

Test File	Opt. Level 0	Opt. Level 1	Opt. Level 2	Opt. Level 3
Fibo	3649.8	3477.8	2852.6	2985.6
Hanoi	4554.2	3912.6	2492.8	2018.2
Matmul	4004.4	3634.8	3816.6	2852.0
Perm	19787.2	19258.0	6832.8	3676.0
Repeatfoo	4106.2	4484.0	2992.0	2537.2
Sieve	6743.6	6036.6	1561.4	1876.0
Testsort	51187.8	42786.6	14456.4	23429.4
Whet	3315.6	372.0	2886.2	3087.6

Table 5: Total execution time (running time minus compile time) averaged over 5 trials.

Test File	Opt. Level 0	Opt. Level 1	Opt. Level 2	Opt. Level 3
Fibo	1.00	1.05	1.28	1.22
Hanoi	1.00	1.16	1.83	2.26
Matmul	1.00	1.10	1.05	1.40
Perm	1.00	1.03	2.90	5.38
Repeatfoo	1.00	0.92	1.37	1.62
Sieve	1.00	1.12	4.32	3.59
Testsort	1.00	1.20	3.54	2.18
Whet	1.00	8.91	1.15	1.07
Median	**1.00**	**1.11**	**1.60**	**1.90**

Table 6: Normalized execution time (relative to optimization level 0) averaged over 5 trials.