REGISTER ALLOCATION OPTIMIZATION IN A KAFFE BASED DYNAMIC COMPILATION

Amudha.G
Associate Professor, RMD Engineering College

Abstract
Dynamic compilation and optimization are widely used in heterogeneous computing environments, in which an intermediate form of the code is compiled to native code during execution. An important tradeoff exists between the amount of time spent dynamically optimizing the program and the running time of the program. In this paper, we explore this trade-off for an important optimization – global register allocation. I present a graph-coloring register allocation in a Kaffe based JIT that has been redesigned for runtime compilation. Compared to Chaitin- Briggs, a standard graph-coloring technique, the reformulated algorithm requires considerably less allocation time and produces allocations that are only marginally worse than those of Chaitin-Briggs. The experimental results indicate that the allocator performs better than the linearscan and Chaitin-Briggs allocators on most benchmarks in a runtime compilation environment. By increasing allocation efficiency and preserving optimization quality, the presented algorithm increases the suitability and profitability of a graphcoloring register allocation strategy for a runtime compiler.

Introduction

Although Java has for a long time been criticized for its slow execution, recent advances in better Java class libraries and Just-in-Time (JIT) compilation techniques have greatly boosted the performance of Java. Some results indicate that Java can deliver a performance in the 65–90% range of the best Fortran performance for a variety of benchmarks and can compete with the performance of C++ .A "Just-In-Time" (JIT) compiler generates native code from Java byte code at runtime. It must improve the runtime performance without compromising the safety and flexibility of the Java language. . Just-intime compilers are invoked during application execution and therefore need to ensure fast compilation times. Consequently, runtime compiler designers are averse to implementing compile-time intensive optimization algorithms. Instead, they tend to select faster but less effective transformations Kaffe is an open source implementation of the Java Virtual Machine specifications. The Kaffe JVM has been ported to a lot of different systems, probably more than any other JVM including SUN's. The Kaffe VM is a C based implementation. It also includes implementation of the Java API's. The API is also part of the implementation without which the Java platform is incomplete.

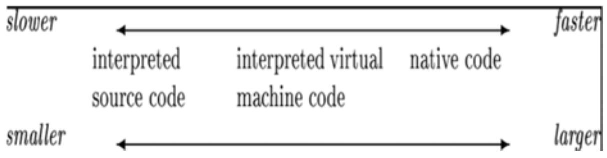
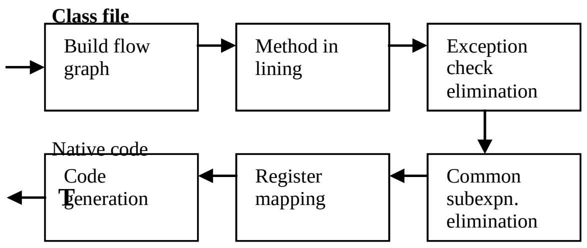


Fig. 1. The time-space tradeoff.

The Java Compilation Process Programs written in Java are first compiled into machine-independent byte codes using a Java bytecode compiler, such as *javac*. These bytecodes are stored in class files to be later read by a *Java Virtual Machine* (JVM). The process of executing a Java method is as follows: A JVM reads in the bytecodes for each method as each method is invoked. The JVM transforms the bytecodes into an *intermediate representation*

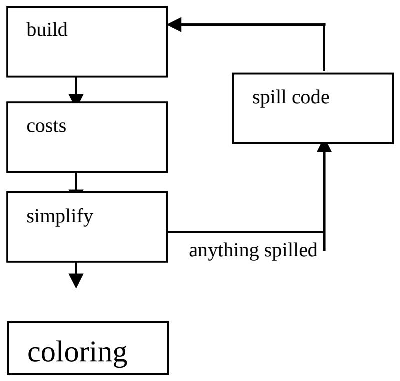
(IR) and a variety of optimizations are applied. The IR is then transformed into assembly code which is then transformed into machine code and executed.

The core of a JVM implementation contains an execution engine that executes the byte codes. There are two popular ways of implementing the execution engine, interpretation and *Just-In-Time* (JIT) compilation. In interpretation, a loop repeatedly fetches, decodes, and executes the next byte code. The interpreter has code to interpret, i.e., to accomplish the effect of each form of byte code instruction. Many Java JIT systems provide an interpreted mode as the initial mode of execution or for infrequently invoked methods. For *hot* (frequently executed) methods, typically JVMs also provide a JIT that compiles those methods to native codes, possibly with optimizations. Jikes RVM is unusual among JVMs in that it does not include a byte code interpreter, but always compiles to native code any method that gets invoked. Java is implemented by static compilation to byte code instructions for the Java virtual machine, or JVM. Early JVMs were only interpreters, resulting in less than-stellar performance: Interpreting byte codes is slow. Mere translation from byte code to native code is not enough code optimization is necessary too.



Overview of JIT compiler

In order to run efficiently, today's microprocessors must carefully take advantage of the registers which reside on the chip. These are the closest storage units to the CPU's pipeline and therefore have the fastest access time. In a minimal compilation sequence, the front end parses the source high-level language code, performing lexical and semantic analyses, and converting the program into an intermediate representation (IR). The compiler backend now takes over, translating the intermediate representation into instructions in the target machine's native instruction set. After this phase of code generation, register allocation is performed. At this point, the instructions contain references to three kinds of operands: the program's symbolic variables, compiler generated temporaries, and machine registers that come pre-assigned due to architectural conventions. Since accessing the registers is much faster than accessing memory (even if the request hits in the first or second level caches), the object of the register allocation phase is to decide which variable and compiler temporaries are *assigned* to registers and which are *spilled* to memory. We seek such a register allocation that minimizes spilling — the traffic between the registers and memory. We distinguish between *local* and *global* register allocation. Local register allocation seeks to find an assignment of variables to registers within a single basic block — a linear sequence of instructions that are always executed together without interruption. Global register allocation seeks to find an assignment of variables to registers over a procedure's entire control flow graph — a data structure made up of basic blocks, showing the conditional and looping structure of the program code. This thesis considers techniques for global register allocation. Register allocation has been shown to be *NP*-complete. The bin packing formulation of register allocation is in fact equivalent to the 0-1 knapsack problem. Both the graph coloring and the 0-1 knapsack problems are classical *NP*-complete problems Within these two formulations, researchers have suggested heuristics in an attempt to reduce the complexity of the algorithms. Jikes RVM performs register allocation for each method, using intra procedural allocation. Before register allocation the intermediate representation uses an unbounded number of registers, called *symbolic registers*. The task of the register allocator is to pack symbolics into *available machine registers*, while satisfying some important constraints: If two symbolics are live at the same time, i.e., both have been loaded or computed and both have possible future uses, then the allocator must assign them to different machine registers. Integers and pointers must go into integer registers and floating point numbers into floating point registers. Some values are bound to specific machine registers, notably arguments to a method and the method's result if any. Also, some registers are reserved for special use (e.g.,the frame pointer). It is not always possible to assign every symbolic to a register while satisfying these constraints. In that case some symbolics are *spilled*, i.e., assigned a location in memory. Spilled symbolics must be fetched from memory (into one of two registers reserved specifically for holding spilled operands) each time they are used, and immediately written to memory each time their value is updated.



Flow graph of register allocators

Graph-Coloring Register Allocation:

In order to find an assignment of register candidates to machine registers, graph-coloring register allocators use information about *liveness* and *interference*. A variable is said to be live at a program point if there is a path to the exit along which its value may be used before it is redefined . It is *dead* if there is no such path. A straight-forward dataflow analysis pass can be used to compute liveness information. Roughly speaking, two variables are said to interfere if they are simultaneously live at some program point. A graph-coloring allocator summarizes the liveness information relevant to the register allocation problem in an interference graph, where nodes represent register candidates and edges connect two nodes whose corresponding candidates interfere. For a *k*-register target machine, finding a *k*-coloring of the interference graph is equivalent to assigning the candidates to registers without conflict. The standard graphcoloring method, adapted for register allocation by Chaitin et al. , iteratively builds an interference graph and heuristically attempts to color it. If the heuristic succeeds, the coloring results in a register assignment. If it fails, some register candidates are spilled to memory, spill code is inserted for their occurrences, and the whole process repeats. In practice, the cost of the graph-coloring approach is dominated by the construction of successive graphs, which is potentially quadratic in the number of register candidates. Since a single compilation unit may have thousands of candidates (variables and compiler-generated temporaries), coloring can be expensive. In designing our algorithm, we wished to construct the interference graph once and then use incremental methods to update the graph after spilling and coalescing. To achieve a substantial decrease in allocation time, we were willing to accept some loss in allocation proficiency. To this end, we decided to augment the representation of the interference graph. The unmodified interference graph is represented by two major data structures – a bit matrix and a collection of edge sets. The bit matrix indicates whether two nodes in the graph interfere. Each node in the graph, N, holds an edge-set that lists the nodes with which it interferes. In this allocator, we added additional information to the edge-sets – each edge originating from a node contains a tag indicating the type of the edge. We classified every edge in the graph as a definition edge or use edge. To comprehensively define these terms. The procedure for building an interference graph traverses the program, identifies live ranges, and adds interferences between them. A careful examination of the algorithm shows that there exist three scenarios when an interference edge is added. If the algorithm added an edge between live range L1 and L2, then either:

1. The algorithm discovered that L2 is live at a definition point of L1, in which case the edge < L1, L2 > gets classified as a definition edge, or
2. The algorithm discovered that L1 is live at a definition point of L2, in which case the edge < L2, L1 > gets classified as a definition edge, or

for every block B in the procedure iterate through every inst. I in B if a load is needed for temporary reg. T locate the last def. in B prior to I if such a def. D is found add the edge (T, D) to the graph set D to its name before renaming for every def. edge <D, E> add the edge (T, E) to the graph if D is a copy inst., add an edge between the source and T. else if no such def. exists for every value L in LiveIn(B) add edge (T, L) to the graph if a store is needed for reg. T let D = the name of T before renaming for every def. edge <D, E> add the edge (T, E) to the graph mark all edges added to T as def. edges if the load services multiple instructions. Add interferences between T and all definitions till the last use of T remove spilled nodes from graph

Related Work on Register Allocation

The first to implement a graph coloring algorithm to solve register allocation was Chaitin. The approach builds an interference graph that represents the interferences or overlaps between the live ranges of the variables in the code for which one is allocating registers. The live range of a variable is the contiguous region of the program where definitions flow to uses of that live variable. Formally, a variable, V is live at point *P* if there exists a control path that includes *P* consisting of a definition of *V* before *P* which reaches a use of *V* after *P* .The algorithm then tries to find a *k*-coloring for the interference graph where *k* represents the number of registers available on the processor being targeted. Determining whether a *k*-coloring exists is known to be an *NP*-complete problem for *k* _ 3, therefore Chaitin's algorithm, as in all global register allocation algorithms, uses heuristics to find a *k*-coloring or to change the graph in order to make it *k*-colorable. The algorithm proceeds by repeatedly removing all those nodes (and their corresponding edges) that have degree less than *k* and placing them in a stack to be colored later. If nodes remain after this graph reduction phase, the graph cannot easily be *k*-colored. The algorithm therefore *spills*, i.e., copies values to memory, one or more live ranges in order to transform the graph into one that has a simple *k*-coloring. Thus, each spill causes a successive graph to be built. The cost of the algorithm is dominated by the building and processing of these successive graphs, which is in the worst case quadratic in the number of register candidates. Briggs *et al.* use the same order as Chaitin to remove nodes, but they continue removing nodes and pushing them on the stack even if they have *k* neighbors. This optimistic algorithm hopes some neighbors will get the same color so that this node can still trivially get a color. During coloring if there is not a color available (for a node with *k* neighbors), Briggs *et al.* spill, rebuild the interference graph, and try again. It has the same worst case complexity as Chaitin but tends to perform better in practice. Recently Omri Traub *et al.* formulated the register allocation problem as a binpacking problem. This algorithm allocates registers for a code sequence by traversing the sequence in a linear scan. When a temporary variable is encountered it is placed in a bin. The constraint on a bin is that it can contain only one valid value at any given point in a program's execution. If a new temporary is encountered and all bins are full, one of the values in a bin must be spilled. The spilled value marks a split in the temporary's live range. Picking a spill candidate employs a heuristic that looks at the distance to the candidate's next reference. This algorithm improves on a previous bin-packing allocation implementation by being able to allocate and rewrite the instruction stream in a single pass. Vegdahl describes a modification to Chaitin's algorithm that leads to a reduction in the number of colors (and therefore number of registers needed) to color an interference graph. Chaitin's coloring algorithm blocks during the simplification stage, that is after all 27 nodes that have degree less than *K* have been removed from the graph. At this point, the algorithm splits a live range (which introduces spilling), or (in Briggs's algorithm) optimistically continues hoping *K*-coloring will still be found. Vegdahl observed that merging two nodes in the graph that are not neighbors, but that share common neighbors, causes a reduction in the degree of any node that had been adjacent to both. In some cases, this reduction can allow simplification to continue without introducing spilling. Also, Vegdahl noticed that there were two aspects in Chaitin's algorithm that were non-deterministic. First, during simplification there may be many nodes that have less than degree *K*, and the order of their removal indicates what color they will get. Second, during coloring, there may be more than one color to choose from when coloring a node. Thus, there are many different colorings of the same graph and these colorings can differ in the number of colors that are used. This led to the insight of applying Chaitin's algorithm repeatedly to the interference graph, and using random choices whenever a non-deterministic choice was available. Node merging colors interference graphs with fewer colors than Chaitin's algorithm and applying Chaitin's algorithm repeatedly by 8% and 0.6% respectively. The fact that node merging and applying Chaitin's algorithm repeatedly produce improvements over Chaitin's original algorithm are evidence that applying heuristics in several phases of the original algorithm proves beneficial.

Experimental Setup

For the experiments we used the KAFFE JVM compiler infrastructure since it was modular, flexible, and very well documented. The JIT

compiles a procedure to native code upon the first invocation of the procedure. We implemented both the classic Chaitin-Briggs algorithm and our allocator in KAFFE JVM. We also use JIKES RVM with the GNU classpath for our experiment. Now JIKES use linear scan register allocation algorithm which will be modified by Graph coloring algorithm. We are decided to compile and evaluate our benchmarks on an Intel Pentium 4, 3.2GHz processor with 1 GB of main memory running Linux.
The Pentium 4 has 7 allocatable integer registers and 8 allocatable floating-point registers. We decided to evaluated the allocators on benchmarks from the SPEC CINT2000 suite. We selected these benchmarks since they perform mostly integer computations. Also normal Java programs in interpreted mode are evaluated using Specjvm98 benchmark programs which will be compared with the JIT compiled programs.

Conclusion

The optimized graph coloring algorithm when implemented under Kaffe JIT surely the runtime will be reduced at a minimum of factor of two. Here we decided to optimize the register allocation in JIKES RVM in a Kaffe based JIT by applying graph coloring algorithm. In future more concentration can be done on the code generation part.

References

[1] John aycock " A brief history of Just-in-Time" In ACM computing surveys in JUNE 2003

[2] Keith.D.Cooper "Tailoring Graph coloring register allocation for runtime compilation" in the Proceedings of the International Symposium on code generation and optimization.

[3] Briggs, P. 1992. Register allocation via graph coloring Ph.D. thesis, Dept. of Computer Science, Rice Univ., Houston.

[4] Briggs, P., Cooper, K. D., And Torczon,L, 1992. Coloring Register Pares ACM Lett, Program Lang, Syst. 1, 1(Mar.), 3-13.

[5] CHAITIN, G, J. 1982. Register allocation and spilling via graph coloring, In Proceedings of the ACM SIGPLAN 82 Symposium on Compder Construction. SIGPLAN Not. 17. 6 (June),
www.Kaffe.org
www.jikes.sf.net

[6] Chaitin, G. J., Auslander, M. A., Chandra, A K , Cocke, J., Hopkins, M. E., And Markstein) P. W. 1981. Register allocation via coloring. Comput. Lang. 6, 1 (Jan.), 47-57.

[7] Toshia Suganama "Design Implementation and Evaluation of optimization in a just-in-time compiler " in the proceedings of ACM 1999.

[8] Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., Andwolczko, M. 1997. Compiling Java Just In Time. *IEEE Micro 17*, 3 (May/June)

[9] Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.Y.,Parikh, V.M., And Stichnoth, J.M. 1998. Fast, Effective Code generation in a just-in-time Java compiler. In *PLDI '98*. 280–290.

[10] Bik, A. J. C., Girkar, M., And Haghighat, M. R. 1999. Experiences With Java Jit Optimization. In *Innovative Architecture for Future Generation High-Performance Processors and Systems.* IEEE Computer Society Press, Los Alamitos, CA, 87–94.

[11] Burke,M.G.,Choi, J.-D., Fink, S.,Grove,D.,Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., And Srinivasan, H. 1999. The Jalape~no dynamic optimizing compiler for Java. In *Proceedings of JAVA '99.* 129–141.

[12] Plezbert, M. P. And Cytron, R. K. 1997. Does "Justin Time" = "Better Late Then never"? In *Proceedings of POPL '97.* 120–131.

[13] John Cavazos "Automatically Constructing Compiler Optimization Heuristics Using supervised learning" Doctor Of Philosophy February 2005 Department Of Computer Science

[14] Michael Matz " Design and Implementationof a Graph Coloring Register Allocator for GCC"