Combining Offline and Online Optimizations: Register Allocation and Method Inlining

Hiroshi Yamauchi and Jan Vitek

Department of Computer Sciences, Purdue University,
{yamauchi,jv}@cs.purdue.edu

Abstract. Fast dynamic compilers trade code quality for short compilation time in order to balance application performance and startup time. This paper investigates the interplay of two of the most effective optimizations, register allocation and method inlining for such compilers. We present a bytecode representation which supports offline global register allocation, is suitable for fast code generation and verification, and yet is backward compatible with standard Java bytecode.

1 Introduction

Programming environments which support dynamic loading of platform-independent code must provide supports for efficient execution and find a good balance between responsiveness (shorter delays due to compilation) and performance (optimized compilation). Thus, most commercial Java virtual machines (JVM) include several execution engines. Typically, there is an interpreter or a *fast compiler* for initial executions of all code, and a profile-guided optimizing compiler for performance-critical code.

Improving the quality of the code of a fast compiler has the following benefits. It raises the performance of short-running and medium length applications which exit before the expensive optimizing compiler fully kicks in. It also benefits long-running applications with improved startup performance and responsiveness (due to less eager optimizing compilation). One way to achieve this is to shift some of the burden to an offline compiler. The main question is what optimizations are profitable when performed offline and are either guaranteed to be safe or can be easily validated by a fast compiler.

We investigate the combination of offline analysis with online optimizations for the two most important Java optimizations [11]: register allocation and method inlining, targeted for a fast compiler. The first challenge we are faced with is a choice of intermediate representation (IR). Java bytecode was designed for compactness, portability and verifiability and not for encoding offline program optimizations. We build on the previous work [15, 17, 16, 2, 8, 10, 9] and propose a simplified form of the Java bytecode augmented with annotations that support offline register allocation in an architecture independent way. We call it SimpleIR (or SIR). A SIR program is valid Java bytecode and can thus be verified and used in any JVM. We then evaluate offline register allocation heuristics [2, 9] and propose novel heuristics. Another challenge is performing method

inlining offline is not always effective because of separate compilation and architecture independence (e.g., dynamic class loading over network, dynamic bytecode generation, platform-dependent (standard) library modules). Thus, we ask the question: can we combine offline register allocation with online method inlining? The contributions of this paper are as follows:

- **Backward-compatible IR for offline register allocation:** *We propose a simplified form of Java bytecode with annotations which supports encoding offline register allocation, fast code generation and verification, and backward compatibility.*
- **Evaluation of offline register allocation heuristics:** *We directly compare two previously known register allocation heuristics and two new heuristics.*
- **Register allocation merging technique:** *which quickly and effectively computes register allocation for inlined methods based on offline register allocation for individual methods.*
- **Empirical evaluation:** *We have implemented our techniques in a compiler and report on performance results and compilation times for different scenarios.*

2 Intermediate Representations

Alternative architecture independent code representations have been explored in the literature. They can be categorized into three groups according to their level of abstraction and conceptual distance from the original format. The first category is annotated bytecode using the existing features of Java bytecode format. This approach is backward compatibile as any JVM can run the code by simply ignoring the annotations. The work of Krintz et al. [10], Azevedo et al. [2], and Pominville et al. [13] are good examples. The second category can be described as optimization-oriented high-level representations. These representations do not necessarily bear any resemblance to Java bytecodes. An example is SafeTSA [1] which is a type safe static single assignment based representation. The last category is that of fully optimized low-level architecture dependent representations with certain safety annotations, such as the typed assembly language (TAL) [12].

2.1 An IR for Offline Register Allocation

We propose an IR for offline register allocation which is a simplified form of the Java bytecode (called SIR). We motivate our design choices and contrasts them with previous results.

Backward compatibility with Java. SIR is a subset of Java bytecode and thus backwards compatibile. This is important: Any JVM can run SIR code with the expected semantics. Existing tools can be used to analyze, compile, and transform SIR code. This is in contrast to [1, 17] which proposes an incompatible register-based bytecode or a SSA form. Offline register allocation results are encoded in annotations following [2, 8–10].

Local variables as virtual registers. We follow Shaylor [15] who suggested mapping local variables in the Java bytecode to (virtual) registers. In contrast, [2, 8, 9] suggest using a separate annotation stream. Directly mapping locals to registers has the advantage that no verification is needed. Other formats must ensure that annotations are consistent. Any additional verification effort will increase the (online) compilation time and thus reduce the usefulness of offline optimizations.

Cumulative register allocation. We refer to local variables as *virtual registers* since they are candidates for physical registers. We adopt a *cumulative* register allocation strategy. This means that the allocation decision for K physical registers is computed on top of that for $K-1$ registers by adding an additional mapping from the Kth register to some locals that were previously not allocated to registers. It produces a 'priority list' of locals variables. Cumulative allocation aims to support an arbitrary number of physical registers while trying to minimize the degradation in allocation quality when the number of available registers is unknown offline. [8] doesn't discuss how registers are allocated. [10] simply encode the static counts of variable occurrences as hints. [15] limits allocation to the first nine local variables.

Register tables We store our register allocation annotations in a *register table* which associates local variables with their *scores* in the decreasing order of scores, in the form $\{(l_1, s_1), (l_2, s_2), ...\}$. Scores indicate the desirability of allocating a given variable to a physical register. In our implementation, these scores are weighted reference counts of variables (count $\texttt{10}^{\texttt{d}}$ in a loop of depth `d`). The fast online compiler takes as many local variables as the available physical registers on the target architecture from the top of the register table and assign them to the physical registers. There is a separate table for each of integers, object references, longs, floats and doubles. A register table bears some similarity to a *stack map* that is used to store the types of local variables at different program points for fast bytecode verification [14]. Register tables tend to be smaller than the parallel register annotations of [2, 8, 9] (space overheads of more than 30% have been reported).

Simplified control and data flow. In SIR, subroutines (the `jsr` and `ret` instructions) of the Java bytecode are disallowed. Subroutines are notorious for making code analysis, optimization and verification slower and more complex. Furthermore, the operand stack must be empty at basic block boundaries. This allows single-pass code generation. For example, if a loop head is only reachable from the backward edge, a single-pass code generator (like ours) cannot know the height of the evaluation stack without a second pass. SIR requires the evaluation stack to be empty between core operations (such as arithmetic operations, method calls, etc.) Operands must always be loaded from local variables and result stored to a local variable. This essentially means that we treat bytecode

as a three-address IR, following [15]. When a method is called, arguments reside in the first part of the local variables array. For backwards compatibility, we treat these locals specially. We do not consider them to be virtual registers and exclude them from the register table. We insert a sequence of moves (loads and stores) at the method entry to copy arguments to local variables. Furthermore, we restrict local variable to hold only one type for the entire method. This simplifies the mapping local variables to physical registers, If the same local was use at different types, we would have to differentiate between types that can be stored in general purpose registers and, e.g., one that must be stored floating point registers.

Verification. It is easy to check if bytecode is SIR and can be performed in a single pass. It is simply a matter of making sure that restricted instructions (e.g., `jsr`, `ret`, `swap`) do not appear, that the local variables are not used to hold more than one type and that they match the type of the register table, that there is a store (or a `pop`) after each instruction that produces a value and that the evaluation stack is empty at branch instructions. We do not verify the scores in register tables because the correctness of the scores does not affect the safety of the code. However, incorrect scores may influence the performance.

3 Offline Register Allocation

3.1 Cumulative Assignments

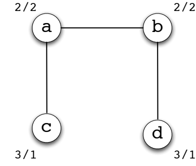
We formulate offline register allocation in terms of *cumulative register assignment* where an assignment for K physical registers is reused for $K+1$ registers by adding an assignment for the $(K+1)th$ register without changes for the first K. There are two benefits of cumulative assignments: architecture independence as any number of physical registers can be matched to the top K virtual registers. Second, cumulative assignments are more space efficient than an alternative approach where separate assignments for each possible value of K are stored in the IR. Cumulative allocation can be viewed as a packing problem where an ordered list of containers (virtual registers) and items (live ranges of data values) must be packed into as few containers as possible and as densely toward the first container as possible so that interfering items (data values whose live ranges overlap) will not be put in the same container.

A fast compiler can use cumulative assignment as follows. If K physical registers are available, the top K virtual registers in the register table will be mapped to physical registers. Several scratch registers have to be reserved for loading and spilling the virtual registers that are not assigned to physical registers and for micro operations hidden in the bytecode. The drawback of a cumulative register assignment is that for any K, a cumulative assignment may not be optimal.

There are two potentially conflicting goals in cumulative allocation: minimize the number of virtual registers in an assignment. Secondly maximize density in terms of the packing problem (the sum of the score of the variables that are

```
mov c, 1
add a, c, c
add b, a, 2
add d, b, 1
call m(d)
ret d
```

(a) Code



(b) Interference graph

K	Assignment	Score
1	(v0 ↦c,d)	6
2	(v0 ↦c,d), (v1 ↦a)	8
3	(v0 ↦c,d), (v1 ↦a), (v2 ↦b)	10

(c) Cumulative register assignment

K	Assignment	Score
1	(v0 ↦c,d)	6
2	(v0 ↦a,d), (v1 ↦c, b)	10

(d) Traditional assignment

Fig. 1. An example of register allocation. (a) is the code. (b) is the interference graph for the code. (c) is the (cumulative) offline register allocation result generated by offline allocator IGC. (d) is the normal non-cumulative register allocation result generated by traditional allocator GC.

mapped to physical registers). There may be a situation where obtaining the highest density possible in the first virtual registers leads to needing an extra register to assign to all variables. Conversely, minimizing the number of virtual registers needed to assign to all variables may cause lower density in the first virtual registers.

3.2 Example

Figure 1 illustrates cumulative allocation. The code is in register transfer form for the example. The interference graph shows the presence of interference between the variables by edges between nodes. The fraction on the side of each node is the ratio of the score (weighted reference count) to the current degree (the number of edges) of the node. Cumulative allocation results are in Figure 1 (c). Each row represents the result with one additional register on top of the previous row. Three virtual registers are needed in total. The allocation result for row K includes the results for the rows 0 to $K-1$. The 'score' column shows the sum of the scores (weighted reference counts) of the virtual registers assigned in each row. We see that with one register ($K = 1$), variables `c` and `d` are assigned to virtual register `v0`. The combined score for `c` and `d` is 6 because `c` and `d` appear in the code six times in total. With two registers ($K = 2$), that assignment is extended with `v1` assigned to `a` and the combined score is, thus, 8. Finally, with three registers ($K = 3$), all variables are assigned to virtual registers with the combined score 10. Figure 1 (d) shows the result of non-cumulative allocation. The differences are that in the non-cumulative case, the mappings for different numbers of registers do not completely intersect, which means that one cannot encode assignments for all possible number of K in one mapping. Second, the

non-cumulative allocator only needs two registers to assign to all the variables in this example.

4 Offline Register Allocation Heuristics

Offline register allocation heuristics assign virtual registers to live ranges of data, called *webs*. The live ranges are not identical to variables since multiple definitions of the same variable can be renamed and each separate live range of the variable can be assigned to different virtual registers. A web is a transitive closure of def-use chains that share a definition or use point and is often used as the targets of register allocation in order to avoid inserting unnecessary moves between registers. A web makes a consecutive multi-entry multi-exit control flow range which starts at the definition points and ends at the use points. We assign virtual registers to webs in offline register allocation. The actual offline register allocation consists of finding webs, computing the interference graph of the webs, making an ordered list of the sets of non-interfering webs based on one of the offline register allocation heuristics described later in this section, and renaming the local variables in the original bytecode so that the webs in the same set are assigned to the same local variable number in the order (local variable 1 is assigned to the first web set in the list, local variable 2 is assigned to the second, and so on, ignoring locals used to pass arguments).

Some of the heuristics described below are based on the optimistic allocator [4] which repeats allocation whenever a spill occurs and the interference graph changes. This repetitive part is omitted in the the heuristics because spilling is handled by the online compiler.

4.1 Linear packing (LP)

Webs are sorted into a list in non-increasing order of scores (weighted reference counts). By linear-scanning over the list, we merge together the webs that are consecutive in the list and do not interfere with each other. At the end, we obtain a list of web sets where consecutive web sets interfere with each other. We sort the list according to the combined scores of the web sets. The ith virtual register is assigned to the ith web set in the list. This heuristic has a $O(n \log n)$ time complexity where n is the number of webs (due to sorting).

4.2 Greedy packing (GP)

This heuristic is equivalent to that of Azevedo et al. [2] and Sites [18]. As with `LP`, webs are sorted into a list in non-increasing order of scores. We iterate the following process until all webs are picked: Keep picking the web with the highest score in the list that does not interfere with the already picked webs in this iteration, until there is no more such web left in the list. Each iteration produces a set of webs. Eventually, we obtain a list of web sets. We sort the list according to the combined scores of the web sets. The ith virtual register

is assigned to the ith web set in the list. This heuristic has a quadratic time complexity in the number of webs.

4.3 Exact graph coloring (EGC)

This heuristic is based on the optimistic graph coloring allocator. We merge webs by performing a binary search for the minimum number of virtual registers needed to assign to all webs, using the optimistic graph coloring allocator. We obtain a set of web sets, each of which is to be assigned to the same virtual register. We sort the set into a list of web sets according to the combined scores of the web sets. The ith register is assigned to the ith web set in the list. Even with the binary search technique, this heuristic may have to run the underlying graph coloring allocator many times and take a relatively long time. The heuristic of Jones et al. [9] based on [6, 3] seems comparable to `EGC`.

4.4 Incremental graph coloring (IGC)

This heuristic is also based on the optimistic graph coloring allocator. We merge webs by incrementally running the optimistic graph coloring allocator for only one register at each iteration. After each iteration, we remove the allocated webs out of the interference graph. We obtain a list of web sets in the end. We sort the list according to the combined scores of the web sets. The ith register is assigned to the ith web set in the list.

To compare the cumulative register allocation heuristics above with a traditional non-cumulative register allocation heuristic, we also include the non-cumulative register allocator in the measurements, called `GC`.

5 Method Inlining and Register Table Merging

Register table merging (RTM) is a technique used to perform online method inlining by reusing offline register assignments for individual methods. It is similar to the merge sort algorithm and described as follows. We have a register table for each method computed offline. These tables contain virtual registers sorted in the order of non-increasing scores. When we inline method B (callee) into another method A (caller), we combine the register tables of A and B into a single register table using the following algorithm.

First, we multiply the scores of the virtual registers in B's register table by $\texttt{10}^{\texttt{depth}}$ where `depth` is the loop nesting depth of the inlining site in A. We then repeat the following process until we reach the end of either A's or B's register table: We pick the virtual register with the higher score between A's top register and B's top register and append it to the register table of the combined method. After the above loop, if there are some registers left either in A's or B's register table, the remaining registers are appended to the combined table. The combined register table is already sorted and the virtual registers are renamed. We update the combined method body with the new register names to obtain

the final combined method body. RTM has a time complexity of $O(a+b)$ where a and b are the size of A's and B's register table, respectively. The algorithm is described in Figure 2.

Figure 3 shows an example of RTM. Suppose that there is an inlining site where a method (callee) is inlined in another method (caller). The inlining site is in a nested loop (the loop depth is 2). The caller and the callee have the register tables shown in Figure 3 (a) and (b), respectively. These register tables are computed offline. The merging result is shown in Figure 3 (c). This table shows the list of virtual registers of the combined method, their scores, and, the mapping from the two register sets of the caller and the callee to the merged register set of the combined method.

This merging algorithm, of course, does not in general give as good register allocation results as redoing full-scale register allocation *after* method inlining.

Input:
a (virtual register array sorted by score for method A (caller))
b (virtual register array sorted by score for method B (callee))
depth (loop nest depth of the inlining site in A)

Output:
c (virtual registers for combined method)

```
map := make a hash map
ia := 0, ib := 0, ic := 0
while  ia < a.length and ib < b.length
    if  a[ia].score ≥ b[ib].score * 10^depth
        c[ic] := make a new virtual register with score: a[ia].score
        put (a[ia], c[ic]) into map
        ia := ia + 1, ic := ic + 1
    else
        c[ic] := make a new virtual register with score: b[ib].score * 10^depth
        put (b[ib], c[ic]) into map
        ia := ia + 1, ib := ib + 1
if  ia < a.length
    for  i := ia to a.length-1
        c[ic] := make a new virtual register
                 with score: a[i].score
        put (a[ia], c[ic]) into map
        ia := ia + 1, ic := ic + 1
if  ib < b.length
    for  i := ib to b.length-1
        c[ic] := make a new virtual register with score: b[i].score * 10^depth
        put (b[ib], c[ic]) into map
        ib := ib + 1, ic := ic + 1
update virtual registers in the combined method body using map
```

Fig. 2. Inlining and Register Table Merging.

regs	v-registers	score
1	v0	1001
2	v1	800
3	v2	753
4	v3	3

(a) Caller's register table

regs	v-registers	score
1	v0'	11
2	v1'	10
3	v2'	5

(b) Callee's register table

regs	v-registers	score	old v-reg
1	v0	1100	v0'
2	v1	1001	v0
3	v2	1000	v1'
4	v3	800	v1
5	v4	753	v2
6	v5	500	v2'
7	v6	3	v3

(c) Merged register table

Fig. 3. An example of register table merging. (a), (b), and (c) show the register allocation result for a caller method, a callee method, the combined method after the callee method is inlined in the caller method, respectively, provided that the inlining site is in a nested loop.

However, our goal is to achieve non-optimal but acceptable level of allocation quality for fast compilers in exchange for short online compilation time. We will see below how this algorithm performs in comparison with results from doing a full-scale register allocation after method inlining.

Performing some optimizations (e.g., constant propagation) after method inlining in order to further optimize the combined method is optional because we focus on fast compilers and additional data flow analysis and optimizations would increase compilation time considerably for fast compilers.

6 Experimentations

We compare the four different compile scenarios shown in Figure 4. The first is online allocation (referred to as `ONR`) where the register allocation is performed without annotations. The second scenario is offline register allocation (`OFR`) where an offline compiler annotates the program and a fast compiler performs allocation using these annotations. The third scenario is online register allocation with method inlining (`ONRI`) – as `ONR` except that the compiler inlines methods *before* applying register allocation. The last scenario is offline register allocation with method inlining (`OFRI`) which is the same as `OFR` except that the fast compiler performs online method inlining with RTM. In addition, we also consider the baseline scenario where the compiler does not perform any optimizations and generates code that literally emulates the evaluation stack and the local variables as in a bytecode interpreter.

The objectives of the measurements are to evaluate the offline scenarios versus the online scenarios in terms of code size, performance, compilation time, and to compare the four offline register allocation heuristics. We only consider compilers that use only one intermediate representation (i.e., Java bytecode which

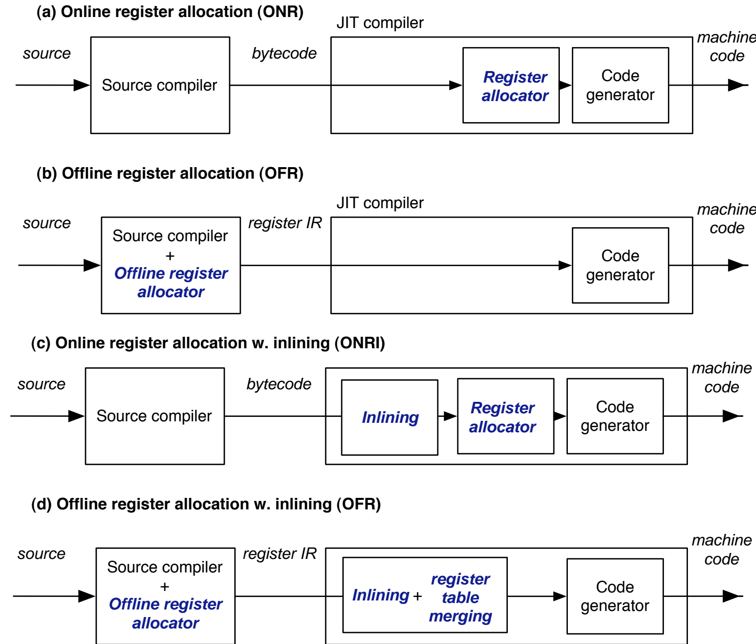


Fig. 4. The four compile scenarios: in (a) and (c) register allocation (and method inlining in (c)) is performed online by the fast compiler; in (b) and (d) register allocation is performed offline, the online compiler uses the register assignments to generate code (and inlining in (d)).

includes SIR) because we are focusing on fast compilers, rather than optimizing compilers that can afford to build other intermediate representations.

The experimental environment is the following. We use the SimpleJIT compiler in the Ovm virtual machine framework [19] on a 1.33 Ghz PowerPC G4 processor with 2 GB of RAM, running Mac OS X.

6.1 Code size

We measure the space overhead of SIR versus bytecode. There are two sources of space overhead, additional loads and stores and register tables. Figure 5 shows the space overhead in terms of the class file size for each of the offline register allocation heuristics. The overall space overhead is 18-20%. Previous work reports overheads of 100% ([2]) and 31% ([9]).

6.2 Offline translation time

We measure the time for offline translator to convert Java bytecode into SIR. This involves: (a) Eliminate subroutines (by duplicating the subroutine body

(Kbytes)	Total	Ovm	SPECjvm98
# classes	3,317	2,767	550
Original	8,854	7,291	1,563
LP	10,568(19.4%)	8,472(16.2%)	2,095(34.0%)
GP	10,434(17.8%)	8,374(14.9%)	2,059(31.7%)
EGC	10,435(17.9%)	8,375(14.9%)	2,060(31.8%)
IGC	10,423(17.7%)	8,366(14.8%)	2,056(31.5%)

Fig. 5. The code size overhead of SimpleIR

(sec)	Total	Ovm	SPECjvm98
# classes	3,317	2,767	550
LP	629	395	234
GP	676	426	250
EGC	948	541	407
IGC	668	437	230

Fig. 6. The offline bytecode translation time

for each subroutine call site). (b) Insert loads and stores to make the evaluation stack empty between core instructions and at basic block boundaries. (c) Introduce extra local variables to let local variables have only one consistent type for the entire method. (d) Perform several simple dataflow optimizations such as constant propagation, copy propagation, and dead code elimination. (e) Apply one of the offline register allocation heuristics (including liveness analysis needed) (f) Compute register tables. (g) Write the converted code into class files.

Figure 6 shows the offline translation time for the four different offline allocation heuristics. `EGC` takes the longest translation time because it needs to iterate the graph coloring heuristic to find the minimal number of virtual registers. `LP` is the fastest overall due to its simplicity. The two other heuristics came relatively close.

6.3 Performance

Next, we measure the execution time of the seven benchmarks of SPECjvm98 to evaluate the steady-state performance of the generated code. The PowerPC G4 processor has 32 32-bit general purpose registers (GPR) and 32 64-bit floating point registers (FPR). We use 15 GPRs for the register allocation of the integer and reference type virtual registers (local variables) and 13 FPRs for the float and double type virtual registers. We do not assign registers for *long* typed virtual registers. The rest of the registers are used for argument and scratch uses. To share a single set of physical registers (e.g., GPRs) for multiple virtual register sets (e.g., integer and reference virtual registers), we merge two register tables into one register table before assigning virtual registers to physical registers.

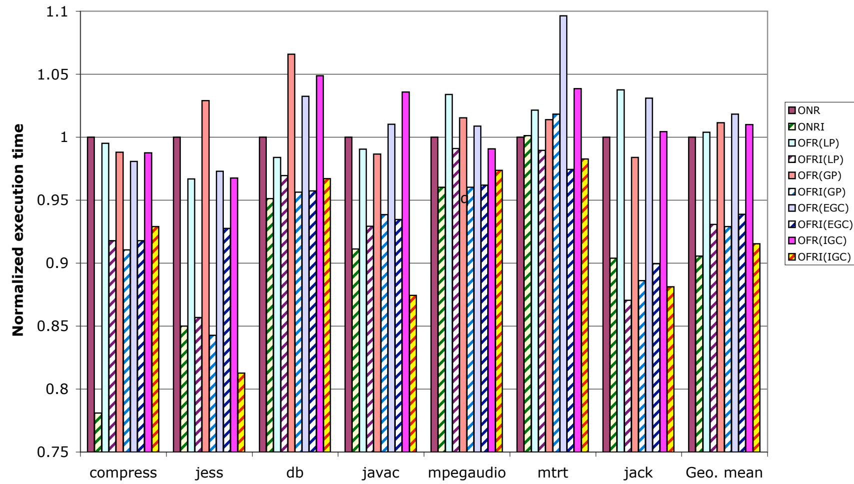


Fig. 7. The normalized SPECjvm98 execution time (steady state performance) of the two online scenarios and the eight offline scenarios.

The method inlining heuristics we use is the following. We inline all methods (callees) that are private, static, or final, and whose bytecode size is less than or equal to 27 bytes. The maximum inlining depth is 5. The maximum caller code size is 240 bytes. We do not attempt to perform devirtualization or inlining of non-final virtual methods.

The results for the two online scenarios (`ONR` and `ONRI`) and the eight offline scenarios (`OFR` and `OFRI` for each of `LP`, `GP`, `EGC`, and `IGC`) are shown in Figure 7. They are the averages of nine runs. For the online scenarios, we used the standard optimistic graph coloring allocator `GC`. The right-most bars show the geometric means over the seven benchmarks. Method inlining contributes about 10% overall performance gain. As expected, with or without method inlining, the online scenarios `ONR` and `ONRI` resulted in the best overall performance. However, the difference between the online scenarios and the offline scenarios are small. This leads to the two following observations. First, the quality difference between the (online) traditional register allocation, which needs the fixed number of registers, and the (offline) cumulative register allocation, which does not, turns out small. Second, the quality difference between register allocation *after* inlining (`ONRI`) and register allocation *before* inlining (`OFRI`), that is, the phase ordering problem between register allocation and method inlining is minimized by RTM.

Comparing the four heuristics (new heuristics `LP` and `IGC`, and `GP` and `EGC` from the previous work), we can derive the following observations. First, without inlining, `LP` overall performed the best. This is surprising because of `LP`'s simplicity. This may imply that, without inlining, we have a sufficient number of physical registers for many methods (i.e., small methods) and the quality differences among the heuristics do not stand out. This is actually supported by

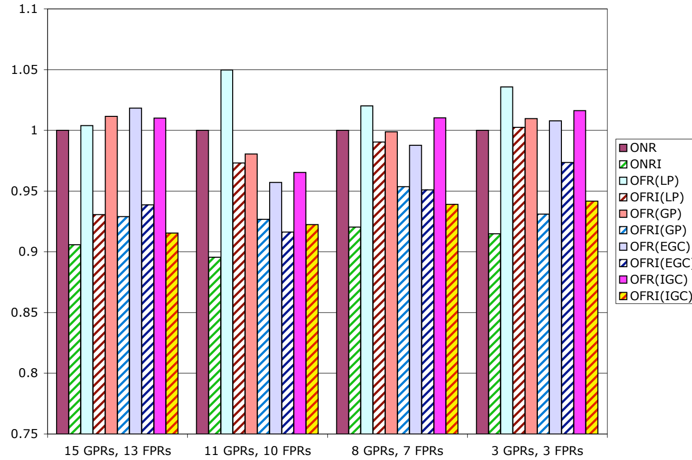


Fig. 8. The SPECjvm98 geometric means with varying numbers of allocated registers.

Figure 8, which shows the overall SPECjvm98 results with varying numbers of registers used for allocation. Second, with inlining, `IGC` achieved the best overall results. This implies that `IGC` tends to work better for large methods that are produced after inlining. Third, unexpectedly, `EGC` did not work as well. However, Figure 8 shows that `EGC` works well in some cases with fewer registers.

We also compare the four major scenarios: the baseline compilation, the online register allocation (`ONR` and `ONRI`), the offline register allocation (`OFR` and `OFRI` with `IGC`), and the optimizing compilation. The baseline compilation does not perform any optimizing and the optimizing compilation performs the highest level of optimizations. Figure 9 shows the results. The online or offline scenarios achieve about 2.5 times lower performance than the optimizing scenario whereas the baseline compilation is about 5 times lower.

6.4 Compilation time

We measure the online compilation time in an ahead-of-time compilation setting. This is not what we advocate since a fast compiler is ideally invoked in a just-in-time manner, but simply a mechanism we use to evaluate the compilation time in this paper.

Figure 10 shows the compilation time results. The compilation times are in milliseconds and shown with the relative lengths compared to the baseline compilation time. The baseline compilation and the offline scenario compilations scans the input bytecode literally once for code generation. The code generators are written using Visitor Pattern (double dispatch) rather than a switch statement. The offline scenarios in addition needs to parse the register table annotations

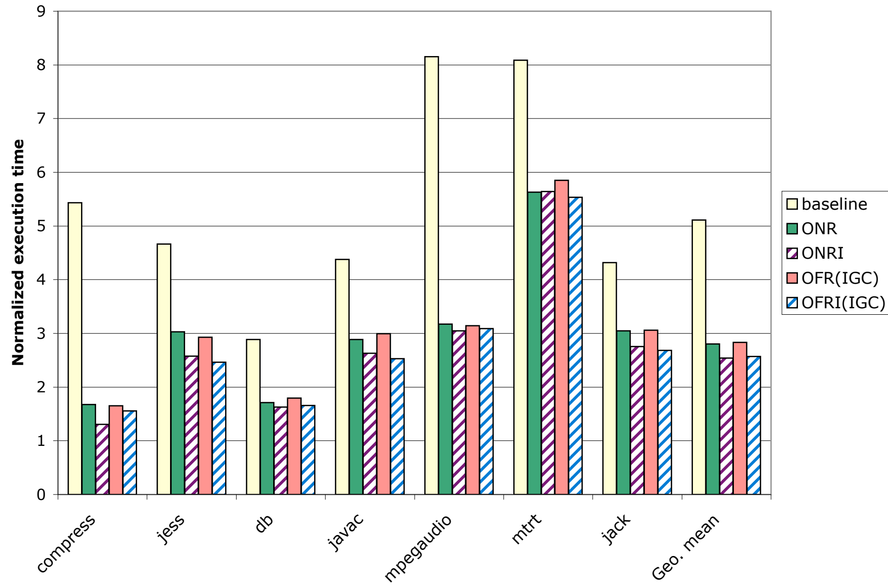


Fig. 9. The normalized SPECjvm98 execution time (steady state performance) of the baseline scenario, the two online scenarios, the two offline scenarios (IGC), and the scenario with the optimizing compiler. The bars are normalized against the optimizing scenario.

(msec)	Total	Ovm	SPECjvm98
# of methods	9523	6761	2762
baseline	6880 (1.00)	4526 (1.00)	2354 (1.00)
ONR	71984(15.90)	27014(5.97)	44970(19.10)
ONRI	87153(19.25)	33768(7.46)	53385(22.67)
OFR(LP)	8242 (1.20)	4136 (0.91)	4106 (1.74)
OFRI(LP)	16082 (2.34)	8644 (1.91)	7438 (3.16)
OFR(GP)	8504 (1.24)	4553 (1.01)	3951 (1.68)
OFRI(GP)	15159 (2.20)	8202 (1.81)	6957 (2.96)
OFR(EGC)	8431 (1.23)	4233 (0.94)	4198 (1.78)
OFRI(EGC)	15567 (2.26)	8367 (1.85)	7200 (3.06)
OFR(IGC)	8288 (1.20)	4231 (0.93)	4057 (1.72)
OFRI(IGC)	15363 (2.23)	8162 (1.80)	7201 (3.06)

Fig. 10. The online compilation time

from the class files, merge the register tables (e.g., integer and reference tables) before code generation, and perform the light-weight code verification to check that the input bytecode complies with SIR during code generation. With-

out inlining, the four heuristics have the overall compilation overhead of about 20-25% from the baseline. The compilation overhead is higher for SPECjvm98 (72-78%) because there are some large methods whose register table tends to be large. With inlining, the overall compilation time is about 2.2-2.34 times longer than the baseline. SPECjvm98 needs up to 3.2 times longer compilation time with inlining. The online scenario compilation time includes the time to perform optional inlining (if `ONRI`), the traditional graph coloring allocation (`GC`), and a single-pass code generation. The overall online scenario compilation time is longer than the baseline scenario by a factor of 15.9 (19.3 with inlining). We believe that the offline scenario compilations achieve high cost-performance considering the compilation time overhead and the performance gain, compared to the baseline compilation.

7 Related Work

Sites [18] used a simple packing heuristic equivalent to `GP` on Pascal U-Code. Gupta et al. [7] and Callahan et al. [5] proposed compositional graph coloring register allocation heuristics. The interference graph of a procedure is decomposed into subgraphs. Subgraphs are colored separately and are combined into a single graph. These two approaches are analogous to register table merging in our work in the sense that subparts of programs are register-allocated separately and the allocation results are combined together. However, these two approaches are different from our work because their approaches are for online register allocation where the number of available colors is known at allocation time.

8 Conclusions

We investigated the interplay of two of the most effective optimizations in object-oriented programs: register allocation and inlining in a combination of offline and online compilation, for fast dynamic compilers. With offline register allocation heuristics, SIR, and RTM, we achieved performance very close to that of the online allocation scenarios, with significantly shorter online compilation time. Compared to the baseline compilation, the offline scenarios achieved good cost-performance with about 80% (99% with inlining) better performance, 20% of overall code size overhead and 25% (a factor of 2.3 with inlining) online compilation overhead.

References

1. Wolfram Amme, Nial Dalton, Jeffrey von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Conference on Programming Language Design and Implementation (PLDI)*, 2001.

2. Ana Azevedo, Alex Nicolau, and Joe Hummel. Java annotation-aware just-in-time (AJIT) compilation system. In *Java Grande Conference*, 1999.
3. D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Conference on Programming language design and implementation (PLDI)*, 1989.
4. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocatoion. *Transaction on Programming Languages and Systems*, 1994.
5. David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Conference on Programming language design and implementation (PLDI)*, 1991.
6. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Journal of Computer Languages*, 6:47–57, 1981.
7. Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. In *Conference on Programming language design and implementation (PLDI)*, 1989.
8. Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, 1997.
9. Joel Jones and Samuel Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, 2000.
10. Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Conference on Programming language design and implementation (PLDI)*, 2001.
11. Han Lee, Daniel von Dincklage, Amer Diwan, and J. Eliot B. Moss. Understanding the behavior of compiler optimizations. Technical Report Technical Report CU-CS-978-04, University of Colorado at Boulder, 2004.
12. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. In *Symposium on Principles of Programming Languages (POPL)*, 1998.
13. Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *Compiler Construction (CC)*, 2001.
14. E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm"*, 1998.
15. Nik Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Java Virtual Machine Research and Technology Symposium (JVM)*, 2002.
16. Nik Shaylor, Douglas N. Simon, and William R. Bush. A Java virtual machine architecture for very small devices. In *Conference on Language, compiler, and tool for embedded systems, (LCTES)*, 2003.
17. Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In *Conference on Virtual execution environments (VEE)*, 2005.
18. R. L. Sites. Machine-independent register allocation. In *Symposium on Programming Language Design and Implementation*, 1979.
19. Purdue University. The ovm virtual machine framework.