

# A Region-Based Compilation Technique for a Java Just-In-Time Compiler

Toshio Suganuma      Toshiaki Yasue      Toshio Nakatani  
IBM Tokyo Research Laboratory  
1623-14 Shimotsuruma, Yamato-shi, 242-8502 Japan  
{suganuma,yasue,nakatani}@jp.ibm.com

## ABSTRACT

Method inlining and data flow analysis are two major optimization components for effective program transformations, however they often suffer from the existence of rarely or never executed code contained in the target method. One major problem lies in the assumption that the compilation unit is partitioned at method boundaries. This paper describes the design and implementation of a region-based compilation technique in our dynamic compilation system, in which the compiled regions are selected as code portions without rarely executed code. The key part of this technique is the region selection, partial inlining, and region exit handling. For region selection, we employ both static heuristics and dynamic profiles to identify rare sections of code. The region selection process and method inlining decision are interwoven, so that method inlining exposes other targets for region selection, while the region selection in the inline target conserves the inlining budget, leading to more method inlining. Thus the inlining process can be performed for parts of a method, not for the entire body of the method. When the program attempts to exit from a region boundary, we trigger recompilation and then rely on on-stack replacement to continue the execution from the corresponding entry point in the recompiled code. We have implemented these techniques in our Java JIT compiler, and conducted a comprehensive evaluation. The experimental results show that the approach of region-based compilation achieves approximately 5% performance improvement on average, while reducing the compilation overhead by 20 to 30%, in comparison to the traditional function-based compilation techniques.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors – *incremental compilers, optimization*

## General Terms

Performance, Design, Experimentation, Measurement, Languages

## Keywords

Region-based compilation, dynamic compilers, partial inlining, on-stack replacement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'03, June 9-11, 2003, San Diego, California, USA.  
Copyright 2003 ACM 1-58113-662-5/03/0006...\$5.00.

## 1. INTRODUCTION

Dynamic compilation systems can exploit the profile information from the current execution of a program. This allows dynamic compilers to find opportunities for better optimization, and thus is a significant advantage over traditional static compilers. Many of today's Java Virtual Machines (JVMs) and Just-In-Time (JIT) compilers indeed use some form of online profile information, not only for focusing the optimization efforts on programs' hot spots, but also for better optimizing programs by exploiting various kinds of runtime information such as call edge frequency and biased branch behaviors [1][2][10][21][24][25].

Typically these systems have multiple execution modes. They begin the program execution using the interpreter or the base-line compiler as the first execution mode. When they detect the program's frequently executed methods or critical hotspots, they invoke the optimizing compiler for those selected methods to run in the next execution mode for higher performance. Some systems employ several different optimization levels to select from based on the invocation frequency or relative importance of the target methods.

Method inlining and data flow analysis are two major components for effective program transformations in the higher optimization levels. However, methods often contain rarely or never executed paths, as shown in the previous studies [5][29], and this can cause some adverse effects that reduce the effectiveness of these optimizations. For example, method inlining can be restricted due to the excessive code size caused by the rarely executed code in the target method. Some methods may include a large portion of rarely executed code, and this may prevent them from being inlined at the corresponding call sites. Others can grow large from the cumulative effects of sections of rare code after several stages of inlining, and this may prevent other hot methods from being inlined. Also data flow analysis is often hindered by kill points existing in those rarely executed paths, whose control flow may merge back to non-rare paths, and this can prevent the propagation of accurate data flow information on non-rare paths.

The problem here is that we implicitly assume methods are the units for compilation. Even if we perform inline expansion, we either inline or do not inline the entire body of a target method, regardless of the structure and dynamic behavior of the target method. Method boundaries have been a convenient way to partition the process of compilation, but methods are not necessarily a desirable unit to perform optimizations. If we can eliminate from the compilation target those portions that are rarely or never executed, we can focus the optimization efforts only on non-rare paths and this would make the optimization process both faster and more effective.

In this paper, we describe the design and implementation of a region-based compilation (RBC) technique in our dynamic compilation system. In this framework, we no longer treat methods as the unit of compilation, as in traditional method-based or function-based compilation (FBC), and select only those portions that are identified as non-rare paths. The term *region* refers to a new compilation unit, which results from collecting code from several methods of the original program but excludes all rarely executed portions of these methods. Regions are inherently inter-procedural, rather than intra-procedural, and thus the region selection needs to be interwoven with the method inlining process.

The notion of region-based compilation was first proposed in [14], as a generalization of the profile-based trace selection approach [19], with some experimental evidence of the potential impact by allowing the compiler to repartition the program for desirable compilation units. An improved region formation algorithm was then proposed by combining region selection and the inlining process [26][27]. The goal of this prior work was to expose as many scheduling and other optimization opportunities as possible for an ILP static compiler without creating an excessively large amount of code due to the aggressive inlining. In a dynamic compilation environment, however, this technique is especially useful for several reasons. First, dynamic compilers can take advantage of runtime profile information from currently executing code and use this information for the region selection process. Second, they are very sensitive to the compilation overhead, and this technique can significantly improve the total compilation time and code size. Third, they can avoid generating code outside the selected regions until the code is actually executed at runtime.

At the same time, we have other kind of challenges that are different from those in static compilers. For example, dynamic online profiles are not always available for all target methods, unlike a static compilation model using offline profile results. Even if they are available, we cannot expect complete profile results that cover all basic blocks of the target method. The profiling is usually performed only for selected methods and on a sampling basis in order to reduce the runtime overhead. Thus we cannot rely solely on the profile information and need to come up with a different region selection strategy by combining some static heuristics. Also the cost of handling region exits can be high in a dynamic compilation system, with recompilation and possibly stack frame replacement, so the region selection may need to be more conservative.

The key components for the RBC approach are region selection, partial inlining, and the region exit handling. For region selection, we employ both static heuristics and dynamic profiles to identify seed blocks of rare code and then propagate them to form rare code sections. The region selection process and method inlining can affect each other, in the sense that method inlining exposes another target for region selection, and the region selection process in turn conserves the inlining budget. Thus the inlining process can be performed for parts of a method, not for the entire body of the method. When the program attempts to exit from a region boundary at runtime, we trigger recompilation and rely on on-stack replacement (OSR) [15], which is a technique to dynamically replace a stack frame from one form to another, in order to continue the execution from the corresponding entry point in the recompiled code.

Notions related to RBC include deferred compilation [7][13], uncommon traps [15][21], and partial method compilation [29]. The

fundamental difference between these prior techniques and ours is in the integration of the region selection process with the method inlining decisions. That is, the previous techniques first perform inlining normally and then eliminate rarely executed code from the resulting code. The major disadvantage with that approach is that possibly large portions of the body of a once inlined method can be simply eliminated as rare in later phases, and it may miss some optimization effects that could be achieved with more method inlining if those eliminated rare code sections were not included in the inlining process.

The following are the contributions of this paper:

- **Design and implementation of a region-based compilation technique for a dynamic compiler:** We present a simple and effective technique consisting of region selection, partial inlining, region exit handling, and other RBC-related optimizations (such as partial escape analysis and partial dead code elimination), as implemented in our dynamic compilation system.
- **Detailed experimental evaluation of the effectiveness of region-based compilation in a dynamic compilation environment:** We present detailed evaluation results for the impact on both performance and compilation overhead by this technique, based on the actual implementation on a production-level Java JIT compiler.

The rest of this paper is organized as follows. The next section is an overview of our dynamic compilation system as used in this study, and describes our implementation of on-stack replacement for safely executing the exit path from the selected region. Section 3 describes our design and implementation of the region-based compilation technique, including intra-method region identification, partial inlining, and other optimization techniques that are aware of region information. Section 4 presents the experimental results on both performance and compilation overhead by applying this technique. Section 5 discusses the results and possible future research. Section 6 summarizes related work, and finally Section 7 presents our conclusions.

## 2. BACKGROUND

In this section, we describe the basic infrastructure required for the RBC approach in a dynamic compilation system. Section 2.1 gives the brief overview of the recompilation system with dynamic profiling mechanism<sup>1</sup>. Section 2.2 describes our implementation of OSR as the basic support for handling region exit points.

### 2.1 System Overview

Our system is a multilevel compilation system, with a mixed mode interpreter (MMI) and three compilation levels (level-0 to level-2). Initially all methods are interpreted by the MMI. A counter for counting both method invocation frequencies and loop iterations is provided for each method. When the counter exceeds a threshold, the method is considered as frequently invoked or computationally intensive, and the first compilation is triggered.

The dynamic compiler has a variety of optimization capabilities. The level-0 compilation employs only a very limited set of optimi-

---

<sup>1</sup> The overall system architecture of our dynamic optimization framework is described in detail in [24][25].

zations. Method inlining is considered only for small target methods. The devirtualization of method calls is applied based on class hierarchy analysis and type flow analysis, and produces either guarded or unguarded code [17]. Preexistence analysis is also performed to safely remove guard code and backup paths without requiring OSR [12]. Most of the data flow optimizations are disabled except for very basic copy and constant propagation. The level-1 compilation enhances level-0 by adding additional optimizations, including more aggressive full-fledged method inlining, and a wider set of data flow optimizations. The level-2 compilation is augmented with all the remaining optimizations available in our system, such as escape analysis and stack object allocation, code scheduling, and DAG-based optimizations.

The level-0 compilation is invoked from the MMI and is executed as an application thread, while level-1 and level-2 compilations are performed by a separate compilation thread in the background. The priority of the compilation thread is set equal to that of the application threads. The upgrade recompilation from level-0 compiled code to level-1 or level-2 optimized code is triggered on the basis of the compiled method hotness level as detected by a timer-based sampling profiler. The sampling profiler periodically monitors the program counters of application threads, and keeps track of methods in threads that are using the most CPU time. Depending on the relative hotness level, the method can be promoted from level-0 compiled code to either level-1 or directly to level-2 optimized code.

There is another profiler, the instrumenting profiler, used when detailed information needs to be collected from the target method. The instrumentation code is initially generated in level-0 compiled code but disabled. When a method is identified as hot and is a candidate for promotion to a higher optimization level, the controller enables the instrumentation code in the target method to collect the required information. After a sufficient number of samples are collected, it is disabled to minimize the performance impact. The compiler can then take advantage of this online profile information in the higher optimization levels.

This instrumenting profile mechanism is used for collecting both virtual/interface call receiver type distributions and basic block execution frequencies. The receiver type profile drives inlining if it identifies the call site as dynamically monomorphic, and the basic block frequency profile provides the runtime rare code information. We apply RBC-based optimization on level-1 and level-2, using the rare code profile information collected from level-0 compiled code.

## 2.2 Region Exit Handling

There are several options to handle region exit points. The simplest option is code splitting [7], which was originally designed to reduce the overhead of message sends in the SELF programming language. It eliminates control flow merges from rarely executed code blocks by performing tail-duplication of conditional control flows to improve data flow information. To alleviate the potentially exponential code expansion problem of this technique, several improved versions have been proposed, such as reluctant splitting [7] and feedback-directed splitting [2]. With this option, we do not eliminate rarely executed paths, but keep that code in a separate section from non-rare paths. The implementation at a region boundary can be very simple, just a single jump instruction, without requiring a complex OSR mechanism. However, using

this option in our framework means we lose the majority of the benefit expected from the integration of inlining and region selection. Recall that our motivation for the RBC approach is not only to eliminate the control flow merges, but also to exploit mutual interaction between method inlining and region selection, and this is possible by actually removing rare regions from the compilation scope. Thus we eliminated this simple option from consideration.

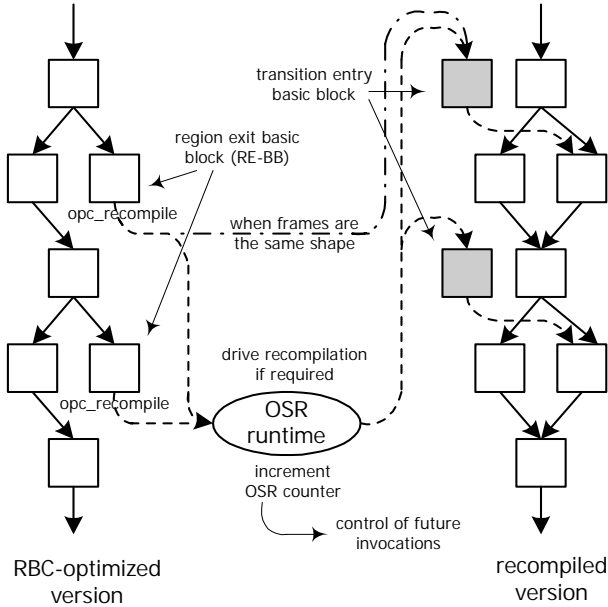
Some other options are listed below, all of which assume on-stack replacement as a pre-requisite technique.

1. Simply fall back to the interpreter. This is the option taken by the HotSpot server compiler [21] when class loading invalidates inlining or other optimization assumptions. This relies on the underlying recompilation system to promote the “de-compiled” interpreted method once again.
2. Drive recompilation with deoptimization. This is the policy used by the SELF-93 system [15] and Jikes RVM [13]. The deoptimized compiled code is used only for the current transition. If the optimization assumption turns out to be wrong and the uncommon cases happen frequently, the system reoptimizes the method under the new assumption, replacing the code for future invocations, by using the underlying recompilation system. Thus this option can produce two additional versions of the code.
3. Drive recompilation with the same optimization level. The recompilation is performed with the same scope for method inlining as in the original version. The recompiled version has several entry points corresponding to region boundaries in the original version, and is used for both current transitions and future method invocations.

The information we need to keep for restoring the JVM state at the region boundary should be basically the same for all three options above, but the way to reconstruct the new stack frame is different. In particular, the first two options need to create a set of stack frames according to the inlined context at the transfer point, while the third option can simply create a single new stack frame.

There are both advantages and disadvantages for each of these three options, and which options we should choose depends on the meaning of “rare code”. If we stay very conservative, saying that only extremely rare cases are rare, then the first option is probably the best choice, since the current version can still serve for future invocations and it is only necessary to handle transitions that may happen very infrequently. However, this means the compiler may miss some additional optimization opportunities for frequent cases due to the conservativeness. On the other hand, if we use an aggressive strategy for optimizing away rare code, these cases would actually occur frequently at runtime, and for the sake of overall performance it may be better to replace the compiled code as quickly as possible to avoid too many expensive OSR operations.

Ideally it might be better to choose from these options depending on the predicted execution frequency for each rare code block identified. For example, we could use the branch prediction from the dynamic profile information. In our current implementation, we use only the third option above, recompilation with the same optimization level. This is because we apply RBC-based compilation only at the higher optimization levels (level-1 and level-2), and thus those methods where region exits occur are already known to be hot and performance critical, and we do not want to

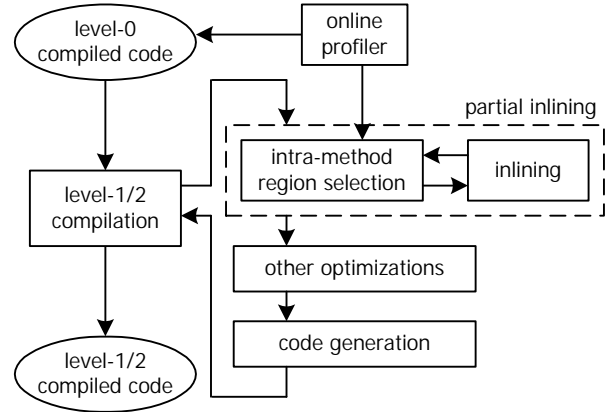


**Figure 1. An example of transitions from an RBC-optimized version to recompiled version. After replacing the current stack frame (OSR), it jumps to the corresponding entry point in the recompiled code. If the source and destination frames are of same shape, it directly jumps without an OSR operation. The OSR counter is used to determine when a future invocation is redirected to the recompiled version.**

deoptimize nor decompile them. In the recompilation, to avoid the recursive recompilation we do not apply RBC optimization. Figure 1 illustrates how a transition from a RBC-optimized version to a recompiled version occurs. The recompiled version prepares all the entry points for future possible transitions, not only the entry point for the current transition.

Since our region selection is speculative (based on static heuristics and dynamic profiles), it is possible that control frequently exits from certain region boundary points. However, if the frequency is small enough, we want the original RBC version to still serve for future invocations since it is better optimized. In order to determine when we should switch from the RBC-optimized version to the new recompiled version for future invocations, we provide a counter for each RBC version to count the number of actual OSR events. Initially the recompiled version is used for the region exit transitions only, not for future invocations. When the OSR counter exceeds a certain threshold, we conclude that our speculative region selection did not work well for this method, and redirect the control of future invocations to the recompiled version.

As in the previous implementation of OSR [13][29], we provide a first-class operator in our compiler’s intermediate representation, called `OPC_RECOMPILE`, at each region exit point. We keep all live variables at that program point in the given bytecode sequence as operands of this special operator for both local and stack variables so as to be able to restore the JVM state. This is done in a very early stage in the compilation, so that any optimizations in later phases can rely on the use of those variables across each region exit boundary. At the final stage of the compilation, we create a map indicating the final locations of those variables as well as other information for frame conversion, such as frame size, and



**Figure 2. The flow of region selection and inlining in level-1 and level-2 RBC-based optimizations. Both static heuristics and dynamic information provided by the online profiler are used in the intra-method region selection process.**

callee-saved registers. We also create a runtime structure indicating the inline context of the given method, and a pointer to the appropriate node in this structure is also recorded as part of the information for each exit point.

All registers holding live variables are first spilled out into memory at region exit points. One small optimization here is that when the source and destination stack frames turn out to be the same shape (frame size, callee saved registers, number and order of local variables, etc) in the first OSR after recompilation, we patch the instruction at the region exit points with an unconditional jump instruction directly into the corresponding entry points, so that we can skip the expensive OSR operation from the next time (see the shortcut line in Figure 1).

### 3. REGION-BASED COMPILATION

In this section, we provide a detailed description of our region-based compilation technique. Section 3.1 describes the intraprocedural region selection process using both static heuristics and dynamic profiles. Section 3.2 shows how the region identification and method inlining are interrelated to obtain more desirable targets as compilation units. Section 3.3 gives some useful optimizations we implemented to take advantage of the target code in the selected region.

As shown in Figure 2, region-based compilation is performed only in level-1 and level-2 compilation, using the profile information resulting from the instrumentation on level-0 compiled code. Since only level-0 compiled methods that are identified as hot and hence are candidates for promotion to the next optimization level are instrumented, profile information is not always available for all target methods considered for inlining at level-1 and level-2.

This is in contrast to the static compilation model using offline profiles. Even if the information is available, we cannot expect a complete profile that covers all the code within the target method. This is regarded as a common problem for dynamic compilers, where online profiles need to be collected selectively and on a sampling basis from the currently executing program to reduce the performance impact. Therefore we combine some static heuristics with incomplete profile information available for region selection.

### 3.1 Intra-Procedural Region Selection

Unlike the region formation algorithm shown in [14], we identify rarely executed portions that we want to remove from the original code, rather than choosing a preferred portion to be optimized together. This is because we have to be more conservative for region selection in a dynamic compiler than in a static compiler, considering the potentially high overhead of OSR that has to be performed when control exits from the selected region.

This is also a different strategy from that used in dynamic binary translation systems [3][6][9], where frequently executed paths (traces) are extracted as a unit for optimization to form a single-entry multiple-exit contiguous sequence. These systems dynamically reoptimize code on top of already (statically) optimized generic executable, which is quite a different compilation model from our system environment. We need to avoid escaping from the selected regions as much as possible, and thus have to employ wider regions that can contain arbitrary numbers of hot traces by removing rarely executed code blocks.

The intra-procedural region selection process is shown in Figure 3. We assume each method is represented as a control flow graph (CFG) at this point with a single entry block and a single exit block. The algorithm begins by marking a Gen set for some seed basic blocks as either non-rare or rare by employing both heuristics and dynamic profile results. We currently use the following heuristics.

- A backup block generated by compiler versioning optimization (such as devirtualization of method invocation<sup>2</sup>) is rare.
- A block that ends with an exception throwing instruction (OPC\_ATHROW) is rare.
- An exception handler block is rare.
- A block containing unresolved or uninitialized class references is rare.
- A block that ends with a normal return instruction is non-rare.

If the dynamic profile information is available and it shows that a block is never executed, then we mark the block as rare. If the profile count value is above a predetermined threshold, the block is marked as non-rare. The dynamic profile information is given priority when there are conflicts with the static heuristics.

In the iteration phase, we propagate this information along backward data flow until it converges for all of the basic blocks. The basic block is marked non-rare in its Out set if any of the successor's In sets is marked non-rare, otherwise it is marked rare when any of the In sets is marked rare. If two flags conflict on the same path between Gen set and Out set, the flag in Gen set is selected to propagate further. Thus a region of rare code can grow backward until it encounters a non-rare path, or a statically identified rare region can be blocked from growing by a profile-based non-rare block along its path. When converged, the rare regions should have reached the points where the branches are expected to be rarely taken from the non-rare paths.

<sup>2</sup> Loop versioning is another versioning optimization, but we currently do not treat its backup block as rare due to our implementation. See Section 5.

---

```

/* Rare-Static, Rare-Profile, NonRare-Static, and NonRare-Profile
are all defined as a bit vector representation. */

/* initialization phase (set flags in seed blocks) */
for each basic block bb {
  /* set flags in Gen set based on heuristics */
  if ( matched to static heuristics ) {
    set ( Gen ( bb ), Rare-Static ) or set ( Gen ( bb ), NonRare-Static );
  }
  /* set flags in Gen set based on dynamic profile information */
  if ( profile_info_available ( bb ) ) {
    if ( profile_count ( bb ) == 0 ) {
      set ( Gen ( bb ), Rare-Profile );
      unset ( Gen ( bb ), NonRare-Static );
    } else if ( profile_count ( bb ) > threshold ) {
      set ( Gen ( bb ), NonRare-Profile );
      unset ( Gen ( bb ), Rare-Static );
    }
  }
}

/* iteration phase (grow rare and non-rare regions) */
do {
  changed = false
  for each basic block bb {
    /* compute Out set from all successors' In set */
    Out ( bb ) =  $\cup$  In ( succ ( bb ) ) for all successors of bb;
    if ( is_set ( Out ( bb ), NonRare-Static | NonRare-Profile ) )
      unset ( Out ( bb ), Rare-Static | Rare-Profile );
    /* combine Out set and Gen set to update In set */
    new_set = Gen ( bb ) | Out ( bb );
    if ( is_set ( Gen ( bb ), NonRare-Profile ) )
      unset ( new_set, Rare-Static | Rare-Profile );
    else if ( is_set ( Gen ( bb ), Rare-Static | Rare-Profile ) )
      unset ( new_set, NonRare-Static | NonRare-Profile );
    if ( new_set  $\neq$  In ( bb ) ) {
      In ( bb ) = new_set
      changed = true
    }
  }
} while ( ! changed )

/* final phase (remove rarely executed block) */
find boundary points from a non-rare block to a rare block
perform live variable analysis
for each rare block entry bb {
  create a new bb (RE-BB)
  create a special instruction in the RE-BB holding live variables
  redirect incoming edges to the RE-BB
}

```

---

**Figure 3. Algorithm for intra-procedural region selection.**

The final phase simply traverses the basic blocks to determine the transitions from non-rare blocks to rare blocks, and marks those locations as rare block entry points. After performing live analysis to find the set of live variables at each rare block entry point, we generate a new region exit basic block (RE-BB) for each entry point and replace the original entry block by redirecting its incoming control flow edge to the new block. This new RE-BB contains a single special instruction OPC\_RECOMPILE, which holds all live variables at the rare block entry point as its operands, in order

to trigger recompilation and call a runtime routine that performs OSR when it is executed. All the rare blocks that originally existed following rare block entry points are no longer reachable from the top of the method and thus eliminated in the succeeding control flow cleanup phase.

### 3.2 Partial Inlining

Partial inlining begins by performing region selection for the root method as shown in Figure 4. It then builds a large call tree of possible inlined scopes from this root method with allowable sizes and depths using optimistic assumptions. The actual inlining pass proceeds by checking each individual decision in the given call tree against the total cost to come up with a pruned final tree. Specifically, it tries to greedily incorporate as many methods as possible using static heuristics until the predetermined budget is used up. Tiny methods are always inlined without qualification [25]. Otherwise the target method is first processed by region selection, and then it is determined whether or not it is inlinable based on this reduced code size. If inlinable, it performs inlining only for the non-rare part of the code, and the current cost is updated with the reduced size of the method.

Note that the devirtualization of the dynamically dispatched call sites is also performed during the inlining process based on class hierarchy analysis and the receiver type distribution profile collected in the previous runs, and region selection can successfully remove the backup path which otherwise needs to be generated.

One tricky part of this process is to update the live variable information in each special instruction provided in the RE-BB for the method being inlined. There are two things that need to be done. One is to rename those live variables by reflecting the mapping into the caller’s context, just like other local variable conversions when inlined. The other is to add the live variables at the call site to reflect the complete set of live variables at each point. This is necessary to automatically hide the effect of all optimizations such as copy propagation, constant propagation, and any other program transformations.

The direct advantage of the partial inlining in our framework is twofold:

1. Since we first remove the rarely executed paths before trying to find the next inlining candidate, we never inline methods at call sites within rare portions of code, since they are no longer included in the current scope. This avoids performing inlining at performance-insensitive call sites and can conserve the inlining budget.
2. Methods being inlined are first processed through intra-method region selection before actual inlining. Inlining is considered and carried out against the reduced target code after removing the rare portions of the code, and this contributes to conserving the inlining budget.

Assuming the current set of criteria for method inlining is reasonable and thus fixed, we can then use the saved budget from the above steps and try to inline other methods in the call tree, which is expected to be more effective and can contribute to further improving performance. Other indirect benefits due to partial inlining include that 1) instruction cache locality can be improved since non-rare parts of the code tend to be better packed into the same compilation unit, and 2) later optimizations in the compila-

---

```

/*setup for root method */
regionSelection ( root_method )
construct (a possibly large) call_graph G
set total_budget B, set current_cost C = 0

/* actual inlining pass */
do {
  select a call edge E in call_graph G {
    M = target method of E
    if ( is_tiny (M) ) {
      /* tiny methods are always inlined */
      perform inlining for M
    } else {
      /* otherwise, decision is made after region selection */
      M' = regionSelection (M)
      if ( inlinable (M') ) {
        perform inlining M'
        update live information for each RE-BB in M'
        C += cost (M')
      } }
    } }
} while ( C < B )

```

---

**Figure 4. Algorithm for interacting intra-procedural region selection and inlining process, leading to partial inlining.**

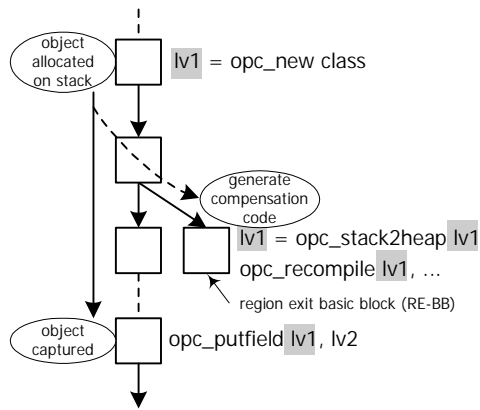
tion process can be more effective with increased optimization scope.

### 3.3 Optimizations

All of the analyses and optimizations that follow partial inlining can proceed normally. The operands in the special instruction OPC\_RECOMPILE provided in the RE-BB for each region exit point work as anchors to preserve the necessary variables, which can be renamed by another variables or replaced by constants during the optimization phases. The RE-BB serves as a placeholder for any optimizations that require generating compensation code for an exiting path. As special optimizations that can take advantage of the selected region, we implemented partial dead code elimination and partial escape analysis, as described in [29].

**Partial dead code elimination.** We have to keep all live variables (both local and stack) in the bytecode-level code at each RE-BB to be able to restore the JVM state correctly when a region exit occurs at runtime. This means the live range of some of those variables becomes longer than in the FBC approach. For example, common subexpression elimination can make some variables unreferenced, and they are eliminated as dead code in the FBC. But those variables may have to be passed on region exit if they are used later at the bytecode level. This problem is expressed in the final code as increased register pressure, extra instructions to spill into memory in the non-rare paths, and larger frame sizes.

Partial dead code elimination [18] can partly alleviate this problem by pushing computations that are only live in region exit paths into the RE-BBs. Our implementation of partial dead code elimination is a simple code motion followed by dead code elimination. We maintain two sets of live variables, one from the RE-BBs and the other from the non-rare paths. Using a standard code motion algorithm, we then move computations whose defined variables



**Figure 5. An example of partial escape analysis and the compensation code generation.**

are included in the set from the RE-BBs but not included in the other set. The computations are copied once into both the appropriate RE-BB and the non-rare path in the other branch direction, but the copy in the non-rare path can then be eliminated in the following dead code elimination phase.

**Partial escape analysis.** Escape analysis and its application to stack object allocation, scalar replacement, and synchronization elimination is very effective for improving performance. However, quite often it suffers from the fact that objects escape from only rarely executed code, especially from the backup path of a devirtualized method call, and thus its effectiveness with the FBC approach has been limited in practice. We modified the escape analysis to simply ignore region exit points, so that it can analyze whether or not objects can be escaped only for non-rare paths in the target code.

Our escape analysis is based on the algorithm described in [28]. It is a compositional analysis designed to analyze each method independently and to produce a parameterized analysis summary result that can be used at all of the call sites that may invoke the method. Hence the analysis result can be more precise and complete as more of the invoked methods are analyzed. However, if the analysis result is summarized based on the optimistic assumption with rare regions ignored, then some of its caller methods may conclude that the object passed as a parameter is non-escaping. When the execution exits from a region boundary, we have to recompile not only the current method but those caller methods that used the summary information as well. Therefore, we check whether arguments are included in the list of live variables at any region exit point within the method, and suppress generating summary information if that is the case.

In the final stage of the analysis, we check each of the objects identified as stack allocatable or replaced by scalar variables as to whether it is included in the list of live variables at each region exit point, and if it is, generate a special instruction `OPC_STACK2HEAP` in the RE-BB as compensation code. Figure 5 shows a graphic example of this. When executed, this instruction calls a runtime that performs 1) allocation of the object on the heap and its initialization, 2) copying of the object content from stack to heap or copying of the scalar-replaced variables which are also included in the list of live variables at the current exit point, and 3) synchronization on the allocated object, if that

operation has been eliminated in the non-rare paths but is necessary at the exit point.

## 4. EXPERIMENTAL RESULTS

This section presents some experimental results showing the effectiveness of the RBC in our dynamic compilation system. We outline our experimental methodology first, describing the configurations used in the evaluation, and then present and discuss our measurement results.

### 4.1 Benchmarking Methodology

All the performance results presented in this section were obtained on an IBM IntelliStation M Pro 6850 (Pentium4 Xeon 2.8 GHz uni-processor with 1,024 MB memory), running Windows XP, and using the JVM of the IBM Developer Kit for Windows, Java Technology Edition, version 1.3.1 prototype build. The benchmarks we chose are SPECjvm98-1.04 and SPECjbb2000-1.02 [23]. SPECjvm98 was run in interactive mode with the default large input size, and in autorun mode 10 times for each test, with the initial and maximum heap sizes of 128 MB. Each distinct test was run with a separate JVM. SPECjbb2000 was run in the fully compliant mode with 1 to 8 warehouses, with the initial and maximum heap sizes of 256 MB.

The threshold in the MMI to initiate level-0 compilation was set to 500. The timer interval for the sampling profiler for detecting hot methods was 3 milliseconds. The number of samples to be collected in the instrumentation-based profiler was set to 10,000. The threshold of the numbers of OSRs to redirect the future method invocations to recompiled code is set to 10.

The five configurations are compared, one with the FBC approach, and the others are RBC approaches with variations in optimizations and the region selection process as listed below. The baseline of the comparison is with the current FBC approach.

1. **RBC-noopt:** This is the RBC approach, but disabling partial escape analysis, partial dead code elimination, and partial inlining.
2. **RBC-nopi:** The same setting as RBC-noopt, but enabling the partial escape analysis and partial dead code elimination only. This is to evaluate the effectiveness of partial inlining. In this and the above cases, the region selection process is performed for the resulting code after normal method inlining.
3. **RBC-full:** This is the RBC approach, with all three optimizations enabled.
4. **RBC-offline:** This is the same as RBC-full, but using offline collected profile results instead of online results. This is to evaluate the maximum potential of the RBC approach if we could have complete profile information. There should be no recompilation or OSR events during execution.

In our current implementation of partial inlining, we perform region selection for an inline target method and estimate the cost for the reduced target code, but temporarily inline the entire body of the method. The rare portion of the code identified in the region selection is removed immediately after the whole inlining process. This should not greatly affect the compilation time or the code size, but the compile-time peak work memory usage can look larger than it should be with the complete implementation.

Benchmarks		mtrt	jess	compress	db	mpeg	jack	javac	SPECjbb
Total number of methods executed		455	734	325	321	495	561	1,085	2,818
Level-0 compiled methods		200	334	135	126	219	373	825	566
Level-1 / level-2 compiled methods		33	41	7	7	56	81	157	166
Region-based optimized methods		28	22	1	6	19	58	101	123
Number of region exit points	Profile identified rare path	3	12	1	7	2	7	14	35
	Devirtualized backup path	483	36	0	19	16	136	644	1,244
	Exception throwing path	19	22	0	7	33	114	169	171
	Handler block	2	3	0	6	0	69	54	96
	Uninitialized code path	0	0	0	0	0	1	0	4
Recompiled methods due to region exit		0	0	0	0	1	9	4	1
Number of on-stack replacement		0	0	0	0	6	52	34	10

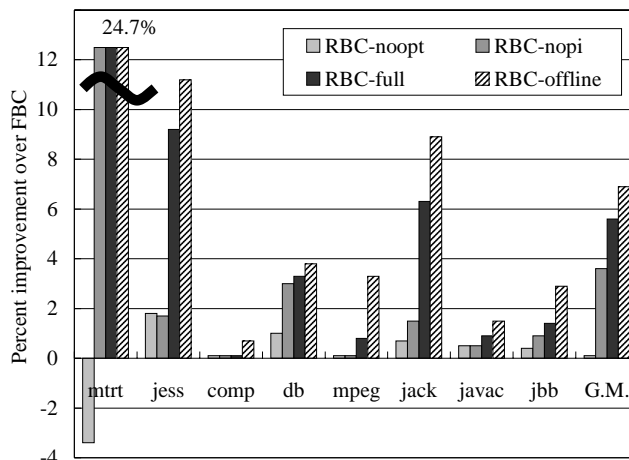
**Table 1. Statistics of the region-based compilation for benchmark runs with RBC-full. The top three rows show execution and compilation statistics, the middle six rows are numbers of RBC optimized methods and how those rare region are identified, and the bottom two show runtime behavior for recompilations and on-stack replacements.**

## 4.2 Statistics on Region-Based Compilation

Table 1 shows the statistics when running the benchmarks with RBC-full. The second to fourth rows are execution and compilation statistics, showing the total number of methods executed for each test, the total number of methods compiled with level-0, and the total number of methods compiled with level-1 and level-2, respectively. The last number, level-1 and level-2 compiled methods, are the target of RBC-based optimizations. Out of these methods, the fifth row shows the actual number of methods where rare regions were identified and RBC optimizations were performed. The next five rows show the breakdown of how those rare regions were identified in the region selection process, that is, the number of exit points for each rare type, based on profile results or on some heuristics. The bottom two rows show the number of methods recompilation has been triggered by region exits, and the total number of OSR events that occurred, respectively.

Except for *compress*, quite a few methods were RBC optimized. The number was roughly 50% to 80% of the level-1 and level-2 compiled methods (except for *mpegaudio*, which was about 30%), showing that many benchmarks contain rare regions even in hot methods and we can do some optimizations for these methods. The backup paths for the devirtualized method invocations is the most common kind of rare region identified, followed by the exception throwing path. The majority of rare regions are identified on the basis of static heuristics, and the numbers of profile identified rare paths are relatively small. This is because profile results are not always available for RBC optimization target methods as described in Section 3, and because we remain conservative when working from the profile results, considering the fact that our profiles are based on the samples collected for short intervals of program execution.

The numbers of recompiled methods and OSRs is relatively large in *jack* and *javac* compared to the other benchmarks. These benchmarks are known to frequently raise exceptions during the program execution, and control frequently exited from certain region boundary points. But the number of recompiled methods was still kept within a reasonable level, around 15% of the total num-



**Figure 6. Performance improvement with four RBC approaches over the FBC. Taller bars show better scores.**

ber of RBC-optimized methods. The number of OSRs is constrained, owing to the mechanism of dynamic OSR counting and the control of future invocations for the recompiled methods based on those counts. The other benchmarks show none or very small numbers of recompilations and OSRs.

## 4.3 Performance

Figure 6 shows the performance improvements of the four variations of the RBC over the current FBC approach. We took the best time from 10 repetitive autoruns for each test in SPECjvm98, and the best throughput from a series of successive executions from 1 to 8 warehouses for SPECjbb2000. The figure shows that both RBC-full and RBC-offline perform significantly better than FBC for some benchmarks, with a 25% improvement for *mtrt*, and 5 to 7% improvement on average.

The majority of the performance gain for *mtrt* comes from the elimination of backup paths for devirtualized method calls and its exploitation by the partial escape analysis. *Mtrt* has many virtual

method invocations, most of which are devirtualized and the target methods are inlined. These methods are never overridden and retaining the backup paths for dynamic class loading just prevents the escape analysis from working well. However, simply removing the backup paths as a part of region selection does not solve this problem. Without partial escape analysis, the performance actually degrades as shown in RBC-noopt, because the escape analysis now has to treat region exit points as globally escaping points for all live variables, not only for the variables passed as arguments as in the original virtual invocation call sites. As a result, more objects are analyzed as escaping, and the number of stack allocated or scalar variable replaced objects is decreased.

For the other benchmarks, the performance difference with RBC-noopt and RBC-nopi seems not very significant, meaning that simply eliminating data flow merge points from rare regions is not generally very effective by itself for a wide range of benchmarks. The partial inlining, on the other hand, can contribute to a significant improvement for some benchmarks, especially for `jess` and `jack`, as the difference from RBC-nopi to RBC-full shows.

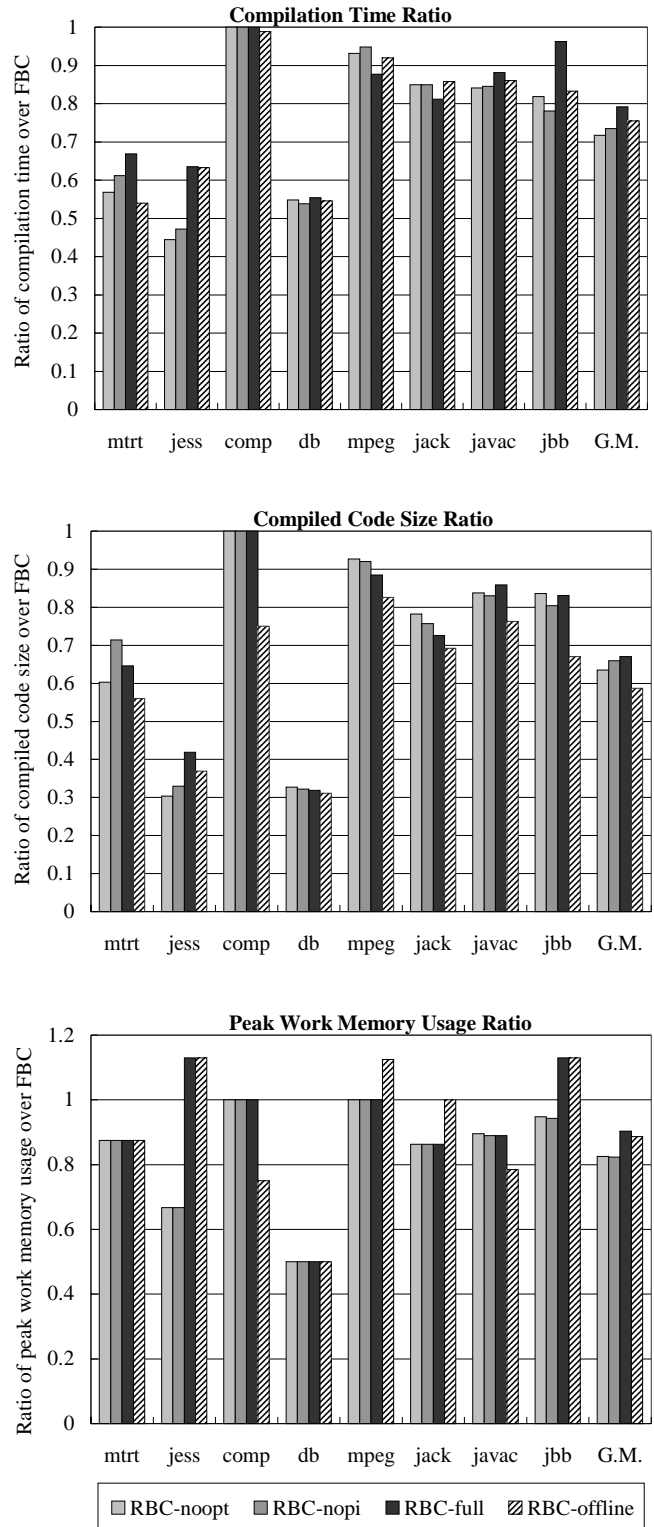
The large improvement with partial inlining in `jack`, for example, results from the additional inlining performed in the partial inlining process which then allows the escape analysis to recognize some frequently allocated objects as captured and makes those objects stack allocated in one of the core methods of the benchmark. This is a good example of the indirect effect of partial inlining.

#### 4.4 Compilation Overhead

Figure 7 shows the ratio of compilation overhead with the four RBC variations over FBC using three metrics: the compilation time, the compiled code size, and the compilation time peak work memory usage. Smaller bars mean better scores in these figures. We measured only level-1 and level-2 overhead, since level-0 is shared among all configurations. All these figures for the three RBC configurations using online profile include additional overhead that results from the recompilations if region exits occur at runtime. The peak memory usage is the maximum amount of memory allocated for compiling methods. Since our memory management routine allocates and frees memory in 1 Mbyte blocks, this large granularity masks the minor differences in memory usage and causes the results of the figure to form clusters.

Overall the RBC approaches show significant advantages over the current FBC approach in most of the benchmarks. In particular, the reductions sometimes exceed 60% (such as code size for `jess` and `db`), and are between 20% and 30% on average, depending on the benchmarks and metrics of the overhead. This significant reduction in compilation overhead is not surprising for RBC-noopt and RBC-nopi, since the rare regions identification and their removal from the target code is performed after completing the normal inlining process, and all the optimizations and code generation are executed for this smaller target code. The reduction of overhead for RBC-full is slightly less dramatic, but it still shows a significant reduction from FBC overall. An increase in peak work memory usage with RBC-full and RBC-offline for some benchmarks is caused by a problem in our current implementation of partial inlining, in temporarily inlining the entire body of target methods. We intend to fix this problem in the future.

The middle graph of Figure 7 shows the code size reduction over FBC, but RBC actually requires more runtime memory space for



**Figure 7. Ratio of three metrics of compilation overhead with four RBC approaches compared to FBC. Smaller bars mean better scores. The top figure shows the compilation time ratio, the middle shows the ratio of compiled code size, and the bottom shows the ratio of compilation time peak work memory usage.**

the map of each region’s exit points and for the runtime structure holding the inline context of each method. The structure showing the inline context is shared for other purposes in our implementation, such as by the exception handling system, so the additional overhead specific to the RBC is only the map. The size of the map depends on the number of live local and stack variables for each region exit point, but it typically requires around 50 to 70 bytes per exit point. Even if we take this map space into consideration, the total size is still well under the FBC code size for most of the benchmarks.

## 5. DISCUSSION

Overall, this study shows the advantages of the RBC approach in both performance and compilation overhead over the traditional FBC approach. It shows the potential for significantly reducing the compilation overhead, measured in time, work memory, and code size, and for improving performance. In a dynamic compilation environment, we have to be very careful in performing any optimizations that have significant impact on compilation overhead, so RBC is a promising strategy for dynamic compilers.

As expected, RBC-offline shows better performance with smaller compilation overhead over RBC-full for several benchmarks, but overall our current online RBC strategy seems to compete well against RBC using the offline profiles. The online profiling combined with static heuristics currently provides useful information to identify rarely or never executed blocks of code, as compared to the offline-collected profile results.

Currently we disable exception directed optimization (EDO) [20]. This is a technique to monitor frequently raised exception paths and to optimize them by inlining and converting exception throwing instructions to simple jump instructions into their corresponding handlers. This is complementary to our RBC approach, since it effectively eliminates frequently excepting instructions from the current control flow, before treating them as rare paths in our static region identification heuristics, and we can ensure the remaining exception throwing instructions are truly rare. As shown in Section 4.2, most of the recompilation and OSR occurring in our current implementation is due to region exits from exception paths, so this optimization is expected to decrease the probability of region exits without reducing the effectiveness of RBC approach.

As described in Section 2.2, it may be useful to support several options for OSR and employ them depending on the characteristics of each region exit point—how the rare paths are eliminated. For example, the current strategy of recompilation with the same optimization level works fine for the exit points of a devirtualized call site backup path, because once the control escapes from one of those exit points due to a dynamic class loading, it will most likely escape from this exit point in subsequent executions. On the other hand, when the region exit occurs from an exception throwing path, it may be sufficient to fall back to the interpreter, since exception paths (especially after EDO) need not be optimized, considering the inherently high overhead of runtime exception handling. We can still collect the dynamic counts of OSRs to identify from which region exit points the control is frequently escaping, and thereby drive recompilation, rather than waiting for the promotion to be performed by the underlying recompilation system. This mechanism may allow more aggressive rare path elimination than we currently use.

We can explore further opportunities for identifying rarely executed code to increase the effectiveness of the RBC approach. For example, loop versioning is an optimization technique for hoisting the array bound exception checking code for an individual array outside a loop by providing two copies of the loop: the safe loop, where exception checking code is retained as in the original loop, and an unsafe loop, where all array exception checking code is eliminated. Guard code is provided to examine the whole range of the index against the bound of the arrays accessed within the loop, and depending on the result of this test, either the safe or unsafe loop is selected at runtime. This is an effective optimization, but entails a significant code size increase. It is expected that the safe loop is rarely or never executed in most of the cases, and we could have a significant code size reduction if we can integrate this opportunity into the RBC strategy. We could even use on-the-fly safe loop generation when the test in the entry guard code fails.

Method splitting, or procedure splitting [22] is a technique that can complement our RBC strategy. This is to place relatively infrequent code apart from common code, typically in a separate page, in order to improve instruction cache locality. Our region selection process does not identify these relatively infrequent but still executed portions of the code, since over-aggressive region selection will lead to too many recompilations and can degrade performance. In other word, the selected region still contains relatively infrequent code. We can increase the code locality even more by integrating the method splitting technique into our framework.

## 6. RELATED WORK

As described earlier, Hank et al. [14] first described the problems of the conventional function-based compilation strategy and demonstrated the potential of the region-based compilation technique by showing several experimental results, including static code size savings. The proposed region formation was to perform an aggressive (possibly an over-aggressive) inlining pass first, followed by a partitioning phase that created new regions based on heuristics using offline profile results. Way et al. [26] improved this region formation algorithm to make it scalable by combining region selection and the inlining process, and reduced the compilation time memory requirements considerably. They also evaluated region-based partial inlining that was performed through partial cloning, and observed small performance improvements [27]. All this work was done in an ILP static compiler environment, so it required two additional steps, encapsulation and reintegration, to make regions look like ordinary functions for optimizations and then to reintegrate them into the containing function.

The SELF-91 system [7] uses a technique called deferred compilation for uncommon branches where a skewed execution frequency distribution can be expected. They use type information to defer compilation for messages sent to receiver classes that are presumed to be rare. When the rare path is actually executed, the compiler generates code for the uncommon branch extension, which is a continuation of the original compiled code from the point of failure to the end of the method. The extension is unoptimized to avoid recursive uncommon branches, and reuses the stack frame created for the original common-case version. It was demonstrated that the technique increases compilation speed significantly, by nearly an order of magnitude, but there were both performance and code size problems when the compiler’s uncom-

mon code predictions were wrong. The SELF-93 system [15] fixed these problems by treating the occurrence of uncommon cases as another form of runtime feedback and replacing overly specialized code with less specialized code rather than just extending the specialized code with an unoptimized extension code. This approach was made possible by introducing both an OSR technique and an adaptive recompilation system. The OSR allowed the dynamic deoptimization of the target code and the replacement of the stack frame containing the uncommon trap with several unoptimized frames. When it was found that this unoptimized compiled code was executed frequently, the recompilation system could optimize the method again based on the feedback from the unoptimized code.

The HotSpot server [21] is a JVM product implementing an adaptive optimization system with an interpreter to allow a mixed execution environment. As in the SELF-93 system, it also employed the uncommon trap mechanism to avoid code generation for uncommon cases. The system always falls back to the interpreter at a safe point after converting the stack frame when an uncommon path is actually taken. Both the SELF-93 and HotSpot systems have some similarities to ours, but the important differences are that our technique deals with a broader set of rare regions using both static heuristics and profiling results, not focusing only on uncommon virtual method targets and references to an uninitialized class, and that we perform partial inlining to make the region-selection process affect inlining decisions.

Whaley [29] described a technique for performing partial method compilation using basic-block-level offline profile information. The technique allows most optimizations to completely ignore rare paths and fully optimize the common cases. This system also assumes falling back to an interpreter when the rare path is taken. Since the technique is not fully implemented in a working system, he estimated the effectiveness of the technique by collecting basic-block-level profiles offline and then using this information to refactor the affected classes with a Bytecode Engineering Library tool. The interpreter transfer points at rare block entry are replaced with method calls to synthetic methods that contain all the code separated from the transfer point, so that the compilation of those rarely executed blocks can be avoided. Thus the result is an ideal case, since the compiler need not hold any information to restore the interpreter state.

Fink and Qian [13] described a new, relatively compiler independent mechanism for implementing OSR, and apply this technique to integrate the deferred compilation strategy in the Jikes RVM adaptive optimization system. Since they do not yet implement optimizations that take advantage of the deferred compilation, the performance improvement is small, but the compilation time and code size show modest improvements.

There are several binary translation systems for profile-based native-to-native reoptimization, such as Dynamo [3], its descendent DynamoRIO [6], HCO [11], and mojo [9]. These systems identify frequently executed paths (traces) and optimize them by exploiting code layout and other runtime optimization opportunities. Since the trace is a single-entry multiple-exit contiguous sequence, and can extend across static program boundaries, it has arbitrary sub-method and cross-method granularity as the unit of optimization, similar to the effect with our partial inlining. However, they operate only on a single trace at a time, and the optimization can be less effective when the selected trace is hot but not a dominant one.

This may not be a problem for these systems, since the statically optimized generic code is already there and a specialized reoptimized version is being generated. We need to be more conservative not to frequently exit from selected regions, and thus employ more general regions to contain arbitrary numbers of hot traces by removing rarely executed code blocks.

Trace scheduling [19] is a technique that predicts the outcome of conditional branches and then optimizes the code assuming the prediction is correct, but it can suffer from the complexity involved in the compensation code generation. Superblock scheduling [16] simplifies the complexity by using tail duplication to create superblocks, single-entry multiple-exit regions. Both of these techniques are to extend the scope of ILP scheduling beyond basic block boundaries to encompass larger units. Other classic optimizations are also extended to exploit superblocks in [8].

Bruening and Duesterwald [5] explored the issues of finding optimal compilation unit shapes for an embedded Java JIT compiler. They demonstrated that method boundaries are a poor choice for compilation. They did not implement a working JIT compiler for evaluation, and only provided estimates of the code size benefits when using trace-based and loop-based strategies for compilation units. They found that a majority of the code in methods is rarely or never executed, and concluded that code size can be reduced drastically with only a negligible change in performance.

Ball and Larus [4] proposed a heuristic approach for static branch prediction based on the data types and type of comparison used in the branch and the code in the target basic blocks. We also use program-based static heuristics for the region selection, and propagate the rare/non-rare information along backward data flow to identify rarely or never executed regions.

## 7. CONCLUSIONS

We have described the design and implementation of a region-based compilation technique in our dynamic compilation system for Java. We presented our design decisions for handling the region exit points and for the intra-procedural region selection and its integration in the inlining process. We implemented this RBC framework and evaluated it using the industry standard benchmarks. The experimental results show the potential to achieve better performance and improved compilation overhead in comparison to the traditional FBC approach. In the future, we plan to further study the effectiveness of region-based compilation techniques, especially by exploring other optimization opportunities that can take advantage of selected regions and by providing several options for region exit handling.

## 8. ACKNOWLEDGEMENTS

We would like to thank all the members of the Network Computing Platform group in IBM Tokyo Research Laboratory for helpful discussions and comments. In particular, we thank Motohiro Kawahito for implementing the partial dead code elimination routine. The anonymous reviewers also provided many valuable suggestions and comments to improve the presentation of the paper.

## REFERENCES

- [1] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimizations in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented*

- Programming, Systems, Languages & Applications*, pp. 47-65, Oct. 2000.
- [2] M. Arnold, M. Hind, and B.G. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 111-129, Nov. 2002.
- [3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-12, Jun. 2000.
- [4] T. Ball, and J.R. Larus. Branch Prediction For Free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 300-313, Jun. 1993.
- [5] D. Bruening, and E. Duesterwald. Exploring Optimal Compilation Unit Shapes for an Embedded Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Workshop on Feedback-Directed and Dynamic Optimization*, 2000.
- [6] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the ACM SIGPLAN Conference on Code Generation and Optimization*, pp. 265-275, Mar. 2003.
- [7] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 1-15, Oct. 1991.
- [8] P.P. Chang, S.A. Mahlke, and W.M. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software Practice and Experience*, 21(12), pp. 1301-1321, Dec. 1991.
- [9] W.K. Chen, S. Lerner, R. Chaiken, and D.M. Gillies. Mojo: A Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN Workshop on Feedback-Directed and Dynamic Optimization*, 2000.
- [10] M. Cierniak, G.Y. Lueh, and J.M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13-26, Jun. 2000.
- [11] R. Cohn, and P.G. Lowney. Hot Cold Optimization of Large Windows/NT Applications. In *Proceedings of 29<sup>th</sup> International Conference on Microarchitecture, MICRO-29*, pp. 80-89, Dec. 1996.
- [12] D. Detlefs, and O. Agesen. Inlining of Virtual Methods. In *the 13<sup>th</sup> European Conference on Object-Oriented Programming, ECOOP*, LNCS 1628, pp. 258-277, 1999.
- [13] S. Fink, and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Proceedings of the ACM SIGPLAN Conference on Code Generation and Optimization*, pp. 241-252, Mar. 2003.
- [14] R.E. Hank, W. Hwu, and B.R. Ran. Region-Based Compilation: An Introduction and Motivation. In *Proceedings of 28<sup>th</sup> International Conference on Microarchitecture, MICRO-28*, pp. 158-168, Dec. 1995.
- [15] U. Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Ph.D. Thesis, Stanford University, CS-TR-94-1520, Aug. 1994.
- [16] W.M. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellete, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1-2), pp. 229-248, May 1993.
- [17] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 294-310, Oct. 2000.
- [18] J. Knoop, O. Rüthing, and B. Steffen. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 147-158, Jun. 1994.
- [19] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2), pp. 51-142, Jan. 1993.
- [20] T. Ogasawara, H. Komatsu, and T. Nakatani. A Study of Exception Handling and Its Dynamic Optimization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 83-95, Oct. 2001.
- [21] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pp. 1-12, Apr. 2001.
- [22] K. Pettis and R.C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 16-27, Jun. 1990.
- [23] Standard Performance Evaluation Corporation. SPECjvm98 available <http://www.spec.org/osg/jvm98>, and SPECjbb2000 available at <http://www.spec.org/osg/jbb2000>.
- [24] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 180-194, Oct. 2001.
- [25] T. Suganuma, T. Yasue, and T. Nakatani. An Empirical Study of Method Inlining for a Java Just-In-Time Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '02)*, pp. 91-104, Aug. 2002.
- [26] T. Way, B. Breech, and L. Pollock. Region Formation Analysis with Demand-driven Inlining for Region-based Optimization. In *Proceedings of the Conference on Parallel Architecture and Compilation Technique*, pp. 24-36, Oct. 2000.
- [27] T. Way, and L. Pollock. Evaluation of a Region-based Partial Inlining Algorithm for an ILP Optimizing Compiler. In *Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2002)*, pp. 552-556, Jun. 2002.
- [28] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 187-206, Nov. 1999.
- [29] J. Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 166-179, Oct. 2001.