Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs

Jong-Deok Choi David Grove Michael Hind Vivek Sarkar

IBM Research
Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
Email: {jdchoi,groved,hindm,vsarkar}@us.ibm.com

Abstract

The Factored Control Flow Graph, *FCFG*, is a novel representation of a program's intraprocedural control flow, which is designed to efficiently support the analysis of programs written in languages, such as Java, that have frequently occurring operations whose execution may result in exceptional control flow. The FCFG is more compact than traditional CFG representations for exceptional control flow, yet there is no loss of precision in using the FCFG. In this paper, we introduce the FCFG representation and outline how standard forward and backward data flow analysis algorithms can be adapted to work on this representation. We also present empirical measurements of FCFG sizes for a large number of methods obtained from a variety of Java programs, and compare these sizes with those of a traditional CFG representation.

1 Introduction

Over the last four years, Java [11] has been rapidly gaining importance as a popular object-oriented programming language for networked applications as well as for general-purpose programming. Although some aspects of Java (such as strong typing and restricted pointers) simplify the task of program analysis and optimization, other aspects (such as support for exceptions, threads, synchronization, and garbage collection) can lead to complications in program analysis.

This paper addresses the problem of performing efficient and accurate program analysis in the presence of precise exceptions. To preserve correctness, such an analysis must accurately model exceptional control flow. We use the term *PEI* (*P*otential *E*xception-throwing *I*nstruction) as a shorthand for operations that can (potentially) throw an exception. PEIs are quite frequent in Java: common operations such as instance variable reads and writes, array loads and stores, method calls, and object allocations are all PEIs. In a traditional control flow graph (CFG) representation, each *PEI* terminates a basic block, and outgoing CFG edges are added to all potential target exception handler blocks and/or the exit of the method, as well as the regular successor blocks. Two drawbacks exist with the traditional approach: (1) it reduces the size of basic blocks, which reduces the scope for local analysis and local summaries, and (2) it increases the number of nodes and edges in the CFG. If PEIs occur frequently, as in Java, both factors can contribute to significant increases in program analysis time. This poses a problem for environments in which compile-time overhead is a concern e.g., static analysis of large programs or dynamic compilation in a Java virtual machine.

To address the problem of efficient and precise program analysis in the presence of exceptions, we introduce the Factored Control Flow Graph (FCFG) representation and outline how standard forward and backward data flow analysis algorithms can be adapted to work on this representation. The FCFG is more compact than traditional (unfactored) CFG representations for exceptional control flow. Our empirical measurements show that the FCFG representation leads to significantly larger basic blocks and smaller CFGs, compared to the traditional CFG representation.

The rest of the paper is organized as follows. Section 2 reviews the Java exception model and discusses its implications for correct program analysis and optimization. Section 3 introduces the FCFG representation. Section 4 describes how the FCFG representation impacts local analyses, i.e., data flow analyses performed within a basic block. Section 5 describes the extensions necessary for global (intraprocedural) analysis with two concrete examples — reaching definitions (a forward analysis) and live variable analysis (a backward analysis). Section 6 summarizes how the FCFG representation can be used in an interprocedural analysis framework. Section 7 contains a brief description of the *Jalapeño optimizing compiler* [3], which is the infrastructure used to obtain the empirical measurements and is also the primary motivation for this work; it also presents empirical measurements of FCFG sizes for a large number of methods obtained from a variety of Java programs, and compares these sizes with those of a traditional CFG representation. Section 8 discusses related work, and Section 9 contains our conclusions.

2 Background

In this section we review the Java exception model [11], and discuss its implications for correct program analysis and optimization. Java exceptions are *precise*. When an exception is thrown at a program point (1) all effects of statements and expressions before the exception point must appear to have taken place; and (2) any effects of speculative execution of statements and expressions after the exception point should

To appear in the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99). Toulouse, France, September 6, 1999.

Java Source Code

```
   public T foo(T p, T[] a) {
      try {
1:       int n = Example.bar();
2:       p = Example.baz(a[n]);
      }
      catch (NullPointerException e) {
3:       .  .  .
      }
      catch (Exception e) {
4:       .  .  .
      }
5:    return p;
   }
```

Figure 1: An example Java method. Statements 1 and 2 are both static calls to methods of the class named `Example`.

Intermediate Representation

```
                    label          B0
1a:      PEI        call_static    n = Example.bar()
2a:      PEI        null_check     a
2b:      PEI        bounds_check   a, n
2c:                 ref_aload      t0 = @{a, n}
2d:      PEI        call_static    p = Example.baz(t0)
                    end_block      B0

                    label          B1
5a:                 ref_return     p
                    end_block      B1

         (java.lang.NullPointerException handler)
                    label          B2
3a:                 .  .  .
3b:                 goto           B1
                    end_block      B2

         (java.lang.Exception handler)
                    label          B3
4a:                 .  .  .
4b:                 goto           B1
                    end_block      B3
```

Figure 2: Intermediate representation for Java source in Figure 1. For convenience, we use the source-code names for the symbolic registers that represents locals `n`, `a`, and `p`. `t0` represents a temporary. Two exceptions can occur at each call: one if the class `Example` cannot be loaded and the other if an exception is propagated from the called method.

not be present in the user-visible state of the program. A correct program analysis or optimization must observe both these properties of Java's precise exception semantics.

When an exception is thrown during program execution, an exceptional transfer of control flow occurs from the operation/statement that throws the exception. The destination of this control flow transfer may be a handler block in the method or the exit of the method.[1] A transfer to a handler block is simply an intraprocedural control flow; this occurs when a handler block is found that matches the type of the exception that was thrown. This handler block must be associated with a `try` block that contains the statement that caused the exception. If no matching handler block is found, the exception is said to "escape" the method. In that case, a transfer to the exit of the method occurs that results in interprocedural control flow — after control reaches the exit of the method, its caller is examined to see if it contains a matching handler block, and so on.

Figures 1 and 2 contain an example method as Java source code and in a quadruple-like intermediate representation.[2] In Figure 1, statement 2 dereferences the array variable `a`, and thus may throw the `NullPointerException` if `a` is null. In this case, control would transfer to the catch block containing statement 3 and then to statement 5.[3] If the `NullPointerException` was not caught by any handler associated with the `try` block, control would proceed to the exit block, i.e., the distinguished block where all control flow leaves a method. Likewise, if statement 2 were located outside the `try` block, a `NullPointerException` would also cause control to flow to the exit block.

Note that the user-visible state when an exception is thrown depends on whether the exception escapes. If the exception causes control to be transferred to a handler block in the same method, then the handler block may examine the local variables of the method as well as any global data. However, if the exception causes control to be transferred to the exit block, then the local variables of the method that throws the exception will not be visible to its caller. A correct program analysis must accurately model the impact of these exceptional control flow transfers in conjunction with the program's regular intraprocedural control flow in the absence of exceptions.

Java exceptions fall into one of four classes [11]:

- *Checked exceptions* — a checked exception can only be thrown by an explicit *throw* statement.
- *Runtime exceptions* — a runtime exception is an unchecked exception from which "ordinary programs may wish to recover from" [11].
- *Errors* — an error is an exception from which "ordinary programs are not ordinarily expected to recover" [11].
- *Asynchronous exceptions* — an asynchronous exception occurs when an error exists in the Java Virtual Machine implementation.

Of these four classes, it is necessary for a compiler or program analysis tool to precisely model the control flow due to the first two classes; the remaining classes do not require support for precise exceptions when compiling or analyzing ordinary Java programs.

For checked and runtime exceptions, the Java language specification identifies all statements/operations that can (potentially) throw an exception, i.e., that are PEIs. For example, statements 1a, 2a, 2b, and 2d in Figure 2 are all PEIs. PEIs pose a challenge to precise and efficient program analysis because they represent a transfer of control. Further, though identification of PEIs is straightforward, type analysis is necessary to identify the potential targets of an exception. Identifying potential handlers for an exception is similar to virtual function resolution: any handler block is a potential target if it catches exceptions of a type compatible with the PEI's exception type. If the PEI is a throw statement or a call statement, the dynamic type of its exception

[1] The try-catch-finally construct is transformed into equivalent try-catch blocks by the Java compiler.

[2] The Java source is presented for expository reasons. The data flow analyses described in Sections 4, 5, and 6 are performed on the intermediate representation.

[3] We assume that no PEIs are present in ". . ." code.

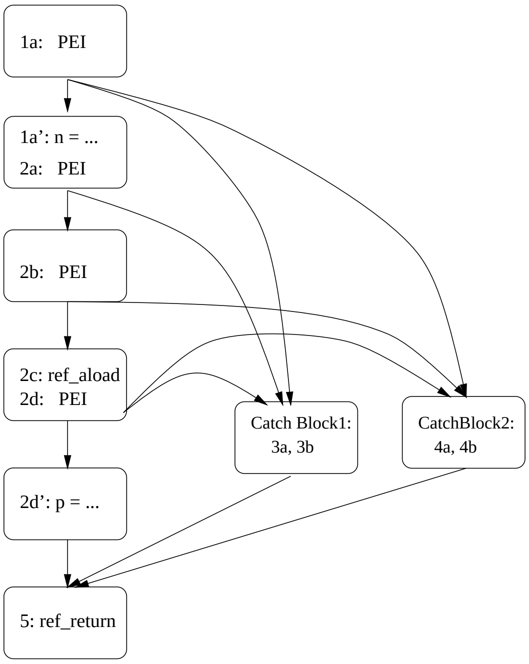

Figure 3: Traditional control flow graph

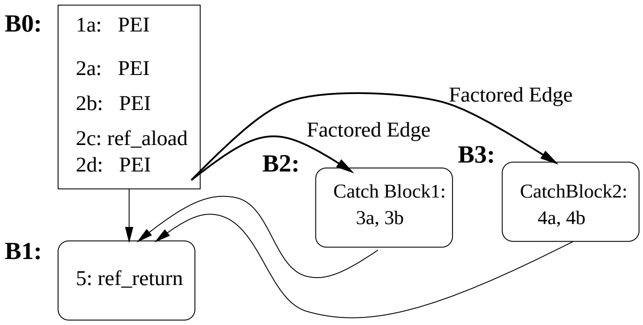

Figure 4: Factored control flow graph

object may be a subtype of its declared static type. Hence, improved precision in type analysis (e.g., via flow-sensitive type propagation) can be used to refine the set of potential handlers for a PEI.

In summary, precise and efficient analysis of Java programs is a challenge due to Java's precise exception semantics. Efficient program analysis is usually predicated on compact (linear-size) control flow graphs and large basic blocks. However, PEIs are frequent and may have multiple potential handlers. This can lead to control flow graphs with small basic blocks and large numbers of nodes and edges, all of which can greatly increase the execution time of standard program analysis and optimization algorithms.

3 The Factored Control Flow Graph

In this section we describe the Factored Control Flow Graph, *FCFG*, which is designed to efficiently support precise and correct analysis of programs written in languages such as Java that contain a large number of PEIs. First, we outline the abstraction of the intermediate representation (IR) assumed in this paper. Then, we describe the differences between the traditional and factored control flow graphs and illustrate them with an example. Finally, we provide some key details of the FCFG construction algorithm.

The abstract IR we use is intended to be general enough to accommodate a variety of concrete IRs; almost any IR based on quadruples or three-address code [15] will satisfy these assumptions. In our abstraction, an IR instruction consists of an operator and some number of input and output operands. IR instructions are grouped into basic blocks, which in turn are connected to form a control flow graph. Figure 2 illustrates the IR abstraction for the example method in Figure 1. Details on the Jalapeño optimizing compiler's concrete IR and how it is constructed from Java bytecodes can be found in [3].

Conceptually, constructing the FCFG for a Java method entails three main tasks: generating the IR instructions from the bytecodes, grouping the IR instructions into basic blocks, and adding edges between basic blocks to represent both regular and exceptional control flow. After contrasting characteristics of the FCFG and the traditional CFG, we briefly describe the later two steps, which differ for the two representations.

Traditionally, every basic block in the control flow graph consists of a set of instructions that is sequentially executed. Each instruction *dominates* [15] all instructions that follow it in the basic block and *post-dominates* [15] all instruction that precede it in the basic block. In other words, a basic block represents a single-entry, single-exit region of the control flow graph. Thus, in a traditional CFG a PEI must terminate its basic block because the potential for exceptional control flow arising from the PEI prevents any instructions that might come after it in the basic block from post-dominating the PEI.

If PEIs were infrequent in practice, terminating a basic block at a PEI might not be a concern. However, PEIs are quite frequent in Java: common operations such as instance variable reads and writes, array loads and stores, method calls, and object allocations are all PEIs. Therefore, in the FCFG, PEIs do not force the end of a basic block. Instead, exceptional control flow from multiple PEIs in the same basic block is summarized by a *factored* control flow edge from the basic block to target exception handler blocks or the exit block, as appropriate. Thus, a basic block in the FCFG may be a single-entry multiple-exit sequence of instructions, similar to an extended basic block [15]. Each instruction dominates the instructions that follow it in the basic block, but it does not necessarily post-dominate the instructions that precede it in the basic block. This leads to a significant reduction in the number of control flow edges since (for example) a single factored edge for a `NullPointerException` can be used to denote control flow from all PEIs in the same basic block that can throw a `NullPointerException`.

We refer to an edge representing exceptional control flow from a PEI to the exit block as an *exit* edge. These edges are not explicitly created. We categorize these edges, and the edges that are encoded by factored edges, as *implicit* edges. We refer to edges that are actually created as *explicit* edges. These edges include factored and regular control flow edges.

The IR example from Figure 2 contains 4 PEIs whose potential exceptions could be caught by 2 distinct catch blocks. Figure 3 gives the traditional CFG for this example, which requires 8 basic blocks and 13 edges to represent this instruction sequence. The exception associated with `2d` can occur after the call, but before the assignment to `p`. Thus, the basic block containing `2c` and the call part of `2d` must terminate after the call, but before the assignment, to allow two edges to emanate from the block of `2d`, one regular and one exceptional. This results in the creation of an additional basic block containing the new instruction `2d´`. A similar situation occurs with instruction `1a`, but in this case the assignment part of instruction `1a` (i.e., `1a´`) is moved into the next basic block. In general, instructions that are both PEIs and assignments to non-temporary variables can be problematic for a traditional CFG because the two parts of the instruction must be in different basic blocks and may necessitate the introduction of temporary variables and new instructions.

Figure 4 gives the FCFG for the IR example from Figure 2. The FCFG contains only 4 basic blocks and 5 edges, compared to 8 basic blocks and 13 edges with the traditional CFG. Because PEIs do not terminate basic blocks in the FCFG, the two parts of instructions `1a` and `2d` do not need to be split and thus can remain in the same FCFG basic block. However, a consequence of the compact representation used in the FCFG is that program analysis algorithms might need to be extended to pay special attention to PEIs — this is the subject of Sections 4, 5 and 6.

In the FCFG representation new basic blocks are only created for the following reasons:

- Definite (Regular) Control Flow. Instructions such as branches, `jsrs`, and `throws` represent a definite transfer of control and thus end a basic block.
- Single Entry. The FCFG requires that a basic block be a single-entry region. Therefore, if an instruction is the target of more than one definite or potential (exceptional) control flow branch, then the instruction must be the first instruction in its basic block.[4]
- Handler Scoping. To simplify the generation of exception mapping tables, we currently maintain the invariant that all instructions in the same basic block have the same set of in-scope exception handlers, i.e., they all came from the same source-level try block. This restriction could be relaxed by allowing blocks to be coalesced that have the same set of reachable in-scope exception handlers.[5]

Once the basic blocks are constructed, the edges corresponding to regular control flow (*regular exits*) can be added in the same manner for both the traditional and the factored CFG. The ordered set of exceptional edges for a given PEI are computed as follows: Given the ordered set of in-scope exception handlers derived from the class file, the set of reachable exception handlers can be computed by considering each handler block in turn and adding an edge if the type of exception caught by the handler is a subtype of the exception(s) potentially thrown by the PEI. If a handler is found that is guaranteed to catch the exception (its exception type is equal to or a supertype of the PEIs exception type), then this search can be terminated. In the traditional CFG, a distinct edge is created from each PEI to each handler block that it can reach, and also to the exit block if necessary. In the FCFG, a single factored (summary) edge is created from an extended basic block to each handler block that can be reached by PEIs in the extended basic block. Thus, a PEI represents an implicit *side exit* from an extended basic block in the FCFG.

Given the ordered set of factored edges for a FCFG block constructed above and a particular PEI that occurs in the block, the subset of the handler blocks reachable from that PEI can easily be decoded on demand by the same search algorithm. Thus, no precision is lost by factoring the edges.

4 Local Analysis

A local analysis is an analysis that is performed purely within a basic block using some conservative assumptions (typically $\perp$) at the starting point (first instruction for a forward analysis, last instruction for a backward analysis). Such an analysis exploits the sequential control flow within a basic block for efficiency.

From the viewpoint of local analysis, there are two significant differences between a basic block in the FCFG and in the traditional CFG representations:

1. For typical Java programs, FCFG basic blocks are bigger than traditional CFG basic blocks. (Section 7 contains empirical measurements that quantify this difference.)
2. Control may exit from the middle of an FCFG basic block due to the presence of PEIs, but control only exits from the end of a basic block in a traditional CFG.

The first attribute makes local analysis more effective in FCFGs than in traditional CFGs because it leads to more opportunities for local optimizations, such as local common subexpression elimination, constant folding, and bounds check elimination. The second attribute only impacts local backward data flow analyses in an FCFG.

More specifically, the presence of PEIs is of no concern to a local forward analysis because a local analysis does not propagate information outside of a basic block. Thus, for local forward analyses the FCFG provides the advantages of large basic blocks, with no disadvantages. This makes local forward analyses (which tend to be inexpensive) attractive, particularly in the context of a dynamic optimizing compiler, where the goal is to generate reasonably optimized code as quickly as possible.

A backward local analysis on the FCFG will need to be adjusted because a PEI may have more than one control flow successor. In a backward local analysis, the data flow information assumed at the point that follows a PEI will be the *meet* [15] of the information obtained from its successor instruction in the basic block and the (conservative) information assumed to hold at the start of the backward analysis, i.e., $\perp$. Because $\perp \wedge x = \perp$, for all x, the data flow value after each PEI can simply be set to $\perp$, which is typically a constant-time operation. This operation would also be required in a traditional CFG that terminates a basic block at a PEI.

5 Global Analysis

An intraprocedural global analysis considers control flow within a method, and propagates information within basic blocks and across basic block boundaries. Propagation across basic blocks follows the edges of the CFG. In the FCFG, edges resulting from PEIs are implicit and hence intraprocedural analyses need to be extended to deal correctly with this implicit control flow.

[4] A less aggressive construction algorithm could create a new basic block whenever a branch to the instruction is encountered.

[5] If a basic block does not contain a PEI that may throw a particular exception type, then the presence or absence of a handler that is applicable to that exception type is irrelevant and should not prevent basic block coalescing.

Persistent variables, initially empty
For each Block B
  In_B: set of definitions that reach the entry of B
  Out_B: set of definitions that reach the exit of B
  $Kill_B$: set of definitions that are killed in B
  Gen_B: set of definitions in B that are not later killed in B

Summarization phase for Block B
  For each instruction, i, in B, in forward order loop
    If i contains a definition d then
      $Kill_B = Kill_B \cup$ alldefs(d)
      $Gen_B = Gen_B -$ otherdefs(d) $\cup \{d\}$
    end if
  end loop

Fixed Point phase for Block B
  $In_B = \bigcup_{p \in predecessors(B)} Out_p$.
  $Out_B = In_B - Kill_B \cup Gen_B$

Local Propagation phase for Block B
  $Current = In_B$
  For each instruction, i, in B, in forward order loop
    $Current_i = Current$
    If i contains a definition d then
      $Current = Current -$ otherdefs(d) $\cup \{d\}$
    end if
  end loop

alldefs(d) contains all definitions of the variable defined at d.
otherdefs(d) = alldefs(d) $- \{d\}$.

Figure 5: Reaching definitions algorithm using a CFG.

Persistent variables, initially empty
For each Block B with E *nonexit* out edges
  In_B: set of definitions that reach the entry of B
  Out_B: set of definitions that reach the exit of B
* $Kill_e$: $\forall\, e \in E$, set of defs killed on paths through B exiting at e
* Gen_e: $\forall\, e \in E$, set of defs generated and not killed later
*   on paths through B exiting at e
Summarization phase for Block B
* $CurrentGen = \{\}$ $CurrentKill = \{\}$
  For each instruction, i, in B, in forward order loop
    If i contains a definition d then
      $CurrentKill = CurrentKill \cup$ alldefs(d)
      $CurrentGen = CurrentGen -$ otherdefs(d) $\cup \{d\}$
    end if
*   If i is a PEI or the last instruction of B then
*   For each *nonexit* out edge, e, associated with i, loop
*   If $Kill_e = \{\}$ then
* $Kill_e = CurrentKill$
*   else
         // This is redundant because $Kill_e \subseteq CurrentKill$
* $Kill_e = Kill_e \cap CurrentKill$
*   end if
* $Gen_e = Gen_e \cup CurrentGen$
*   end loop
*   end if
  end loop
Fixed Point phase for Block B, with *nonexit* out edge e
* $In_B = \bigcup_{p \in predecessors(B)} Out_j$, where edge $j = \langle p, B \rangle$
* $Out_e = In_B - Kill_e \cup Gen_e$
Local Propagation phase for Block B
  $Current = In_B$
  For each instruction, i, in B, in forward order loop
    $Current_i = Current$
    If i contains a definition d then
      $Current = Current -$ otherdefs(d) $\cup \{d\}$
    end if
  end loop

alldefs(d) contains all definitions of the variable defined at d.
otherdefs(d) = alldefs(d) $- \{d\}$.

Figure 6: Reaching definitions algorithm using an FCFG. Additional statements from the Figure 5 algorithm are marked with "*". This algorithm degenerates to the Figure 5 algorithm when the analyzed method contains no handlers.

In a traditional CFG, information is propagated between basic blocks at block entry and exit points. In a FCFG, information can flow out from PEIs for a forward analysis, and in to PEIs for a backward analysis. Although an explicit CFG edge is not present, the information can be recovered by inspecting the PEIs and the factored edges, as described in Section 3. Sections 5.1 and 5.2 present examples of forward and backward data flow problems using the FCFG.

Consider a PEI r. We define a *PEI region* r to be the instructions from the beginning of the basic block to the execution point of PEI r that may throw an exception. For example, in the instruction "`x = y / z`", the assignment to `x` would be part of the next PEI region because it occurs after the potential `ArithmeticException`.[6] For convenience, we use the same symbol r to denote a PEI or the PEI region. For two regions (or PEIs) r and s of the same FCFG block, we say $r < s$ if r is smaller than s: i.e., PEI r precedes PEI s in instruction order. For convenience, we also assume that the block is terminated by a dummy PEI b. When a reference is made to the "handler targets" of this dummy PEI, it should be assumed to denote the regular (nonexception) FCFG successors of the block. Note that PEI region b will be larger than any other PEI region in the block.

5.1 Reaching Definitions

This section illustrates how a forward intraprocedural data flow analysis can be performed on the FCFG. The analysis that we consider is *reaching definitions*, which computes the set of definitions (assignments) of variables that may reach an instruction. This information is useful when constructing def-use chains and is representative of forward analyses on the FCFG. The algorithm consists of three phases. The first phase, *summarization*, summarizes the Gen and Kill sets (for reaching definitions) for each basic block. The second phase, *fixed-point*, uses these summaries to compute a fixed-point solution for each basic block entry. The third phase, *local propagation*, propagates the fixed-point solution to instructions within a basic block. Figure 5 provides an overview of these three phases for a traditional CFG.

The implementation for a FCFG differs from a CFG implementation in the summarization and fixed-point phases. In a traditional CFG, reaching definition information holding at the exit of a basic block is propagated from the basic block to all its successors. In an FCFG some successors of a basic block are reachable only through a side exit, and some successors are reachable through multiple side exits and/or the regular exit. Propagating reaching definition information holding at the exit of an FCFG basic block equally to

[6]Some instructions, such as `invokevirtual`, can have more than one PEI due to class loading, dereferencing the receiver object, and method invocation. Our analysis correctly identifies the exceptional control flow from each of these PEIs.

Persistent variables, initially empty
For each Block B
  In_B: set of variables that are live at the entry of B
  Out_B: set of variables that are live at the exit of B
  $Kill_B$: set of variables that are killed in B
  Gen_B: set of variables that have upward-exposed uses in B

Summarization phase for Block B
  For each instruction, i, in B, in reverse order loop
    If i contains a must definition of some variable, v, then
      $Kill_B = Kill_B \cup \{v\}$
      $Gen_B = Gen_B - \{v\}$
    end if
    For each use of some variable v, at i, loop
      $Gen_B = Gen_B \cup \{v\}$
    end loop
  end loop

Fixed Point phase for Block B
  $Out_B = \bigcup_{s \in successors(B)} In_s$
  $In_B = Out_B - Kill_B \cup Gen_B$

Local Propagation phase for Block B with n PEIs
  $Current = Out_B$
  For each instruction, i, in B, in reverse order loop
    If i contains a must definition of some variable, v, then
      $Current = Current - \{v\}$
    end if
    For each use of some variable v, at i, loop
      $Current = Current \cup \{v\}$
    end loop
    $Current_i = Current$
  end loop

Figure 7: Live variable analysis algorithm using a CFG

all successors can be either conservative or incorrect.

To correctly compute reaching definition information, we compute the Gen and Kill information not per basic block as in a traditional CFG, but per nonexit edge of the FCFG. During the summarization phase, we compute $Kill_e$ and Gen_e for a nonexit edge e as given in Figure 6. The two equations used during the fixed-point phase are modified to use the edge-based Gen and Kill sets as shown in Figure 6. The local propagation phase is the same as with a CFG. In this figure statements that differ from the CFG-based analysis are marked with "*".

5.2 Live Variable Analysis

This section illustrate how a backward intraprocedural data flow analysis, live variable analysis, can be performed on the FCFG. This analysis determines for a particular program point those variables that are *live*, i.e., may be subsequently used before they are assigned. This information is useful for register allocation and the computation of garbage collection reference maps. This analysis differs from reaching definitions analysis because it tracks variables instead of definitions of variables.

Like the reaching definitions algorithm described in the previous section, the high-level structure of the live variable algorithm consists of three phases. The first phase, *summarization*, summarizes the Gen and Kill sets for live variable analysis for each basic block. The second phase, *fixed-point*, uses these summaries to compute a fixed point solution for each basic block exit. The third phase, *local propagation*, propagates the fixed point solution to instructions within a

Persistent Variables, initially empty
For each Block B with n PEIs
  In_B: set of variables that are live at the entry of B
  Out_B: set of variables that are live at the exit of B
  Gen_B: set of variables that have upward-exposed uses in B
  $Kill_B$: set of variables that are killed in B
* $Kill_r$: forall $r \leq n$
*   set of variables that are killed in B before PEI r,

Summarization phase for Block B
  For each instruction, i, in B, in reverse order loop
    Let r be the PEI region for i
    If i contains a must definition of some variable, v, then
      $Gen_B = Gen_B - \{v\}$
      $Kill_B = Kill_B \cup \{v\}$
*   For each x, where $x \geq r$ loop
* $Kill_x = Kill_x \cup \{v\}$
*   end loop
    end if
    For each use of some variable v, at i, loop
      $Gen_B = Gen_B \cup \{v\}$
    end loop
  end loop

Fixed Point phase for Block B with n PEIs
  $Out_B = \bigcup_{s \in regularSuccessors(B)} In_s$
* For $i \leq n$, let Out_i be the union of the In sets of the handlers
*   that can catch exceptions at PEI, i.
  $In_B = (Out_B - Kill_B)$
* $\cup\, (Out_n - Kill_n) \ldots \cup (Out_1 - Kill_1)$
      $\cup\, Gen_B$

Local Propagation phase for Block B with n PEIs
  $Current = Out_B$
  For each instruction, i, in B, in reverse order loop
    Let r be the PEI region for i
    If i contains a must definition of some variable, v, then
      $Current = Current - \{v\}$
    end if
    For each use of some variable v, at i, loop
      $Current = Current \cup \{v\}$
    end loop
*   If i is a PEI then
* $Current = Current \cup \bigcup_{h \in handler(i)} In_h$
      // $handler(i)$ is the set handler successors of B
      // that can catch the potential exception(s) of i
*   end if
    $Current_i = Current$
  end loop

Figure 8: Live variable analysis algorithm using an FCFG. Additional statements from the algorithm in Figure 7 are marked with "*". The above algorithm degenerates to the Figure 7 algorithm when the analyzed method contains no handlers.

basic block. Figure 7 provides an overview of these three phases for a traditional CFG.

During the CFG-based summarization phase, the analysis traverses the instructions of a basic block, B, in reverse order tracking two sets, Gen_B and $Kill_B$. A (definite) write triggers the analysis to place the variable in $Kill_B$; the variable is no longer live before this instruction. An instruction with a read triggers the analysis to place the corresponding variable in Gen_B and remove it from $Kill_B$; the variable is live before this instruction regardless of whether it is live after the last instruction.

Figure 8 provides an overview of the FCFG-based live variable analysis, where statements that differ from the CFG-based analysis are marked with "*". Live variable analysis for an FCFG differs from a CFG implementation in all three phases. However, these differences are only necessary for methods that contain handler blocks. When analyzing local variables and temporaries, no additional processing is required for methods that do not contain handlers, which are the majority of methods found in practice (see Section 7). (Additional processing may be required if interprocedural liveness analysis is attempted on global variables.)

As described in Section 3, instructions in an FCFG basic block do not necessarily post-dominate previous instructions in the block. Thus, a write instruction that is preceded by at least one PEI is not guaranteed to occur on all paths through the basic block. To provide the same level of precision as a traditional CFG, an FCFG-based implementation can record a Kill set for each PEI in an FCFG basic block.[7] We define $Kill_r$ to be the set of variables that are killed in PEI region r.[8] These sets are computed during the summarization phase and are used in the fixed-point phase of FCFG-based algorithm as shown in Figure 8. Figure 9 gives the Gen and Kill sets for the intermediate representation given in Figure 2. The assignment of a return value (if any) occurs after the PEI, i.e., only if the exception is not thrown will the assignment be performed. For this reason, n $\notin Kill_{1a}$ and p $\notin Kill_{2d}$.

During the fixed-point phase, the In set for each basic block is reevaluated whenever an In set of its successor is updated. Although this remains the same in an FCFG-based analysis, the manner in which the In set is computed differs. A CFG-based analysis computes the In set using the standard equation

$$In_B = Out_B - Kill_B \cup Gen_B$$

as shown in Figure 7. The multiple exit points of an FCFG-based basic block result in different ways in which successor handler blocks' In sets are incorporated and the manner in which Kill information for the block is used. Figure 8 presents the details of this computation, which provides the same level of precision as a CFG-based analysis. Figure 9 presents the results of the propagation phase for each block for the example in Figure 2, and elaborates on the computation of In_{B0} by illustrating the intermediate computations.

The local propagation phase for a CFG block uses the fixed-point solutions for all successors to the block to determine what is live at the exit of the block (Figure 7). This information is then used in a manner similar to the summarization phase; variables are added and removed from the

[7] This results in the same number of Kill sets as a traditional CFG implementation because a PEI terminates a basic block in a traditional CFG.

[8] Since $Kill_i \subseteq Kill_j$, for all $i < j$, storage enhancements can be employed.

Summarization Phase

Block	Kill	Gen
B0	{n, t0, p}	{a}
B1	{}	{}
B2	{}	{}
B3	{}	{p}

PEI Region	Kill
1a	{}
2a	{n}
2b	{n}
2d	{n, t0}

Fixed-point Phase

Block	Out	In
B0	{p}	{p, a}
B1	{}	{p}
B2	{p}	{p}
B3	{p}	{p}

$$\begin{array}{lll} In_{B0} = & (\{p\} - \{n, t0, p\}) & // \ Out_{B0} - Kill_{B0} \\ & \cup\ (\{p\} - \{n, t0\}) & // \ Out_{2d} - Kill_{2d} \\ & \cup\ (\{p\} - \{n\}) & // \ Out_{2b} - Kill_{2b} \\ & \cup\ (\{p\} - \{n\}) & // \ Out_{2a} - Kill_{2a} \\ & \cup\ (\{p\} - \{\}) & // \ Out_{1a} - Kill_{1a} \\ & \cup\ \{a\} & // \ Gen_{B0} \end{array}$$

Figure 9: Results from summarization and fixed-point phases of live variables analysis for the example in Figure 2.

Current set based on reads and writes, to obtain the final solution at each instruction.

In an FCFG implementation two modifications to the local propagation phase are required. First, only the solutions for regular successors are unioned and used as the initial data flow information for the block. Second, at a PEI i, the In set(s) for all handlers that may catch the exceptions at i are unioned into the set of live variables before i as shown in Figure 8.

6 Interprocedural Analysis

A flow-sensitive interprocedural analysis utilizes an intraprocedural analysis to propagate information at call sites to method entry points and at method exit points to the instruction following a call. Thus, the FCFG modifications described in Section 5 are also applicable to a flow-sensitive interprocedural analysis.

The FCFG can support a flow-sensitive interprocedural analysis that propagates method summaries in the same manner as in the traditional CFG, or it can be used to build an *interprocedural FCFG* (IFCFG), in the spirit of the *interprocedural CFG* [16, 19, 4, 14, 12].

In building the IFCFG, an interprocedural path must exist from a PEI with an exit edge to any handlers in calling methods that might catch the exception. A spectrum of precision exists in creating this path. A simple approach would be to create edges from the PEI to the exit node in the called routine, relying on the exception edge in the called routine (to a handler or its exit node) to complete the appropriate path. A more precise approach would create an edge from the PEI in the called routine directly to the handler in the appropriate caller's routine, if one exists. This approach avoids the potential loss of precision that results from merging information at the exit of the called method from regular and exception control flow. An intermediate approach would create two exit nodes, one for regular control flow and the other for all exceptional control flow. An edge from the regular exit in the callee to the node following the

```
class T {
   public void m1(T f) {
      try {                          public void m2(T f) {
           . . .                          . . .
S1:        m2(f);              S3:    G.v = ...
           . . .                          . . .
      }                        S4:    f.m3();
      catch (...)  {                . . .
S2:        ...  = G.v;         S5:    G.v = ...
           . . .                          . . .
      }                              }
   }
                   public void m3() {
                        . . .
               S6:    G.v = ...
                        . . .
               S7:    PEI
                        . . .
               S8:    G.v = ...
               S9:    G.w = ...
                   }
}
```

Figure 10: Example for interprocedural analysis

call under regular control flow is created, as well as an edge from the exception exit in the callee to all handler nodes for the call.

Consider Figure 10, where `G.v` and `G.w` are static (global) variables. The simple approach would have implicit edges to the exit node of `m3` from the basic blocks of both `S9` and `S7`, and an edge from the exit node back to the basic block after `S4`. The more precise approach would retain the regular edge from the basic block of `s9` to the exit node, but would create an edge from the basic block of `S7` to the handler in `m1`, eliminating the unrealizable path starting at `S7`, passing through the exit nodes of `m3` and `m2`, and ending at the `S2` handler.

In the following we illustrate how the FCFG can support a flow-sensitive interprocedural analysis that propagates method summaries from the callee to the caller. Using this approach, the presence of control flow due to exceptions does not alter the interprocedural call graph: i.e., when an exception "escapes" a method, it follows the normal procedure return path by returning to its caller. Hence, the only extension required for correct interprocedural analysis in the presence of exceptions is to ensure that the intraprocedural summary information is correct in the presence of exceptions.

Once again, consider Figure 10. The summary Gen set for reaching definitions information for method `m3` is {`S6`, `S8`, `S9`}: i.e., the definitions of `G.v` at statements `S6` and `S8`, and the definition of `G.w` at `S9` can all reach the exit of method `m3`. The definition at `S6` reaches the exit of `m3` due to the PEI at `S7`, which does not have a handler block in `m3`. The summary Kill set for `m3` includes all definitions of `G.v` due to the definition at `S6`. `S9`, however, does not contribute to the summary Kill set for `m3` because it can be bypassed by the PEI at `S7`.

The call at `S4` expands into three PEIs, one to ensure the declared class of `f` is loaded, one for the null check of `f`, and one for the possible exceptions propagated from `m3`. When the Gen and Kill sets of `m3` are propagated to `S4` in `m2`, the definition at `S3` no longer reaches the point after the third PEI of `S4` because of the Kill set for `m3`. However, because the first two PEIs (a class loaded check and a null check) occur before the method call, a path exists to the exit node of `m2` that does not include the execution of method `m3`, and thus, the definition at `S3` may reach the exit node of `m2`. Thus, the summary Gen set for `m2` is {`S3`, `S5`, `S6`, `S8`, `S9`}. The Kill set for `m2` is the same as `m3`, all definitions of `G.v`, due to the definition at `S3`. When this summary information is propagated to `S1` in `m1`, which is itself a PEI, the Gen information is propagated to `S2`, the first statement of the handler block for the PEI at `S1`. At this point, the interprocedural analysis identifies that the definitions reaching the use of `G.v` at `S2` are `S3`, `S5`, `S6`, and `S8`.

Using the most precise version of the IFCFG discussed above, where PEI edges go directly to their handlers even if they are in other methods, would result in only the definitions of `G.v` at `S3` and `S6` reaching `S2`, and only the definition at `S5` reaching the point after the call to `m2` at `S1`. Exploiting fully the most precise version of IFCFG in general requires identifying *feasible (realizable)* interprocedural control flow [19], which is beyond the scope of this paper.

7 Experimental Results

This section presents an empirical comparison of the factored and traditional control flow graphs generated for methods drawn from a suite of Java programs. The Jalapeño optimizing compiler, the infrastructure used to obtain the empirical measurements, is briefly summarized in Section 7.1. The actual measurements are presented in Section 7.2.

7.1 The Jalapeño Optimizing Compiler

The Jalapeño optimizing compiler is a key component of the Jalapeño JVM, a new research Java Virtual Machine being built at the IBM T.J. Watson Research Center [2, 1]. A detailed description of the Jalapeño optimizing compiler can be found in [3]. A key distinguishing feature of the Jalapeño JVM is that it uses a *compile-only* approach to program execution with two dynamic compilers — a non-optimizing *quick* compiler that has fast compile-times, and an *optimizing* compiler that contains a production-strength optimizing back-end. The Jalapeño JVM reduces dynamic compilation overhead by judiciously mixing the use of the two compilers: the optimizing compiler is only invoked on those methods that significantly contribute to the application's execution time.

Another distinguishing feature of the Jalapeño JVM is that all its subsystems (including the compilers, runtime routines, and garbage collector) are implemented in Java and run alongside the Java application. Although it is written in Java, the Jalapeño JVM is self-bootstrapping; i.e., it does not need to run on top of another JVM. One of the many advantages of a pure Java implementation is that we can dynamically self-optimize the Jalapeño JVM.

The Jalapeño optimizing compiler first translates Java bytecodes to HIR, the high-level intermediate representation. HIR is a register-based IR in which each instruction corresponds to a Java bytecode instruction. Unlike Java bytecodes, the inputs and outputs of instructions in our HIR are explicitly represented as register operands. After optimizations are performed on the HIR, the HIR is converted to LIR, a machine-independent low-level IR, in which high-level operations are expanded into low-level operations that are representative of machine instructions e.g., all the address arithmetic for field and array accesses is made explicit in the LIR. Optimizations are performed at

Metric	HIR		MIR	
	CFG	FCFG	CFG	FCFG
Total Number of Basic Blocks	253,762	72,343	202,570	134,869
Total Number of Explicit CFG Edges	285,146	101,918	257,838	195,939
Median Number of Instructions per Basic Block	1	3	4	6
Basic Blocks with 0 Handler Edges	98.26%	96.06%	98.98%	94.71%
Basic Blocks with 1 Handler Edge	1.51%	3.44%	0.88%	4.55%
Basic Blocks with 2 Handler Edges	0.22%	0.48%	0.13%	0.70%
Basic Blocks with 3 Handler Edges	0.01%	0.02%	0.01%	0.04%

Table 1: Comparison of FCFG and CFG representations

the LIR level, and then the LIR is converted to MIR, the machine-specific IR that contains target instructions. Finally, machine-specific optimizations are performed on the MIR, followed by generation of binary code. All three levels of IR are based on the FCFG; thus the techniques described in this paper are implemented throughout the Jalapeño optimizing compiler.

7.2 Empirical Measurements

This section empirically evaluates the FCFG representation by comparing it against a traditional CFG representation on 10,461 methods drawn from 14 Java programs. To gather the data, the programs were run on the Jalapeño JVM in a special configuration in which all methods of dynamically loaded classes were compiled by the optimizing compiler, which in turn was instrumented to collect the desired statistics. FCFG data was collected simply by examining the FCFGs constructed during compilation. Data for the traditional CFGs was derived by analyzing the FCFGs to determine where additional basic blocks and edges would be required.[9] Although this derivation can be performed precisely, the MIR data may understate the relative effectiveness of the FCFG representation because all of the optimizing compiler's local analyses and optimizations were actually performed on the larger basic blocks enabled by the FCFG.

The benchmark programs consisted of three micro-benchmark suites (`CaffeineMark`, `ByteMark`, and `Symantec`), four programs from the `SPECjvm98` suite (`compress`, `javac`, `db`, and `jack`), six other moderate-size applications (a Java lexer, a Java parser, two Java compilers, a lisp interpreter, and a business object benchmark), and the Jalapeño JVM, which includes the optimizing compiler. All methods were compiled with optimization (including inlining of "small" static and final methods). Note that due to inlining, a single control flow graph may contain instructions and handler blocks derived from multiple source methods.

Table 1 presents data comparing the traditional CFG and FCFG representation of the HIR and MIR for all methods. As shown in the first two rows of the table, the FCFG significantly reduces representation space costs in comparison to the traditional CFG. For example, at the HIR level, using our factored representation yields a 71% reduction in basic blocks and a 64% reduction in control flow graph edges. This space savings is an important consideration for a dynamic compiler. Furthermore, as reported in the third row of the table, the FCFG representation increased basic block size: at the HIR level, the median-size basic block in the traditional CFG contained one instruction whereas the median-size basic block in the FCFG contained three instructions.

Of the 10,461 generated control flow graphs, 9,769 (93.4%) contained no handler blocks.[10] The lower half of Table 1 illustrates that even though 6.6% of the control flow graphs contained at least one handler block, very few basic blocks were actually linked to handler blocks. In the HIR FCFGs, 96.06% of the blocks had no outgoing edges to handler blocks, 3.44% were linked to one handler, 0.48% were linked to two handlers, and 0.02% were linked to three handlers. No basic block was linked to more than three handler blocks.

Figure 11 provides more detail on the impact of the FCFG representation on basic block size. The graphs show the cumulative percentage of basic blocks containing x or fewer instructions for both traditional and factored CFGs. For example, 46% of all basic blocks in HIR FCFGs contained two or fewer instructions. Note that for a given x value, smaller y values are "better" because they indicate that a greater percentage of basic blocks have more than x instructions. Overall the FCFG representation had a substantial impact on basic block size. For example, in the HIR FCFG, 29% of all basic blocks contained more than five instructions, but only 2% of the blocks in the HIR traditional CFG contained more than five instructions. The results for the MIR FCFGs show similar trends.

8 Related Work

Three representations similar to the FCFG are *extended basic blocks* [15], *superblocks* [13], and *traces* [9]. An extended basic block is a sequence of consecutive basic blocks that contain no join nodes other than the first block [15]. Superblocks and traces are variations of an extended basic block. A set of consecutive basic blocks on a frequently executed path is chosen, based on profiling, to form a superblock (or a trace) to be treated as a single unit for some optimization and scheduling decisions. Side exits of a superblock (or a trace) correspond to the branches taken less frequently than the ones taken more frequently. Duplications of some of the basic blocks in a superblock might be required to eliminate entrances to the middle of the superblock, called *side entrances*. Side exits for a FCFG block, however, correspond to exceptional control flows, and are chosen statically.

The side exits of a superblock are explicitly represented, with their targets also explicitly represented at each side

[9] Although modifying the optimizing compiler to initially construct a traditional CFG would be straightforward, preserving the traditional CFG throughout the compilation process would have required substantial effort. In particular, the lowering of HIR to LIR and LIR to MIR may result in a single PEI e.g., a call instruction, being expanded into multiple PEIs, each of which would need to be placed in a separate basic block in the traditional CFG.

[10] A handler block includes both catch blocks declared in the source code and those introduced by the compiler to support `finally` and `synchronized` blocks.

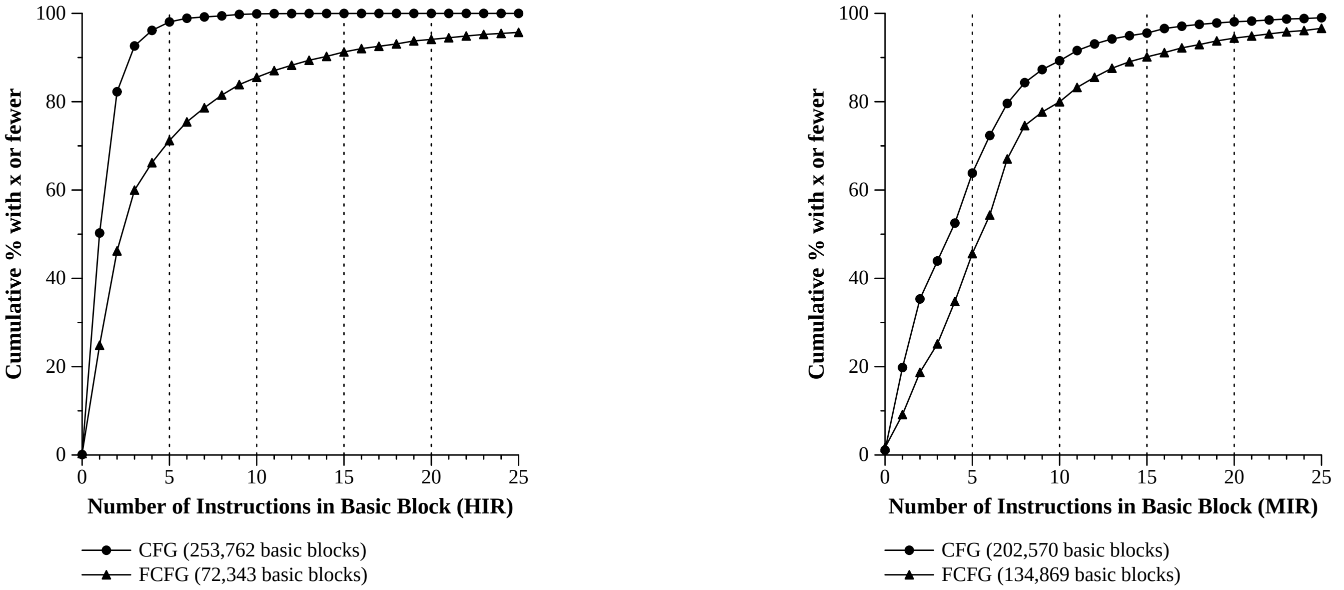

Figure 11: Impact on basic block size (HIR and MIR instructions)

exit. The side exits in FCFG are, however, implicitly represented as PEIs, with their targets represented only at the end of the basic block. The target set of a PEI is decoded as described in Section 3. In our experiments, this factoring effect resulted in a 36% reduction in the number of explicit control flow graph edges in HIR used to represent side exits.

The primary purposes of an extended basic block are (1) to increase the number of instructions to consider for certain optimizations such as instruction scheduling; and (2) to correctly compute any *compensatory* actions to be taken in the presence of *incorrect* speculative scheduling across the side exits. The primary purpose of the FCFG is to correctly handle side exits due to potential exception throws and catches, without penalizing the analysis time of the majority of methods with no exception handlers in them. This is important in a dynamic compiler, which has to balance the costs and benefits of an analysis or optimization.

The Marmot system [10] is a static optimizing compiler (and associated runtime library support) for a subset of the Java language. Some portions of the Marmot optimizing compiler use a CFG representation that appears to be similar to our FCFG. Just as in the Jalapeño optimizing compiler, Marmot's basic blocks are not terminated by calls or PEIs and it also represents exceptional control flow by "distinguished exception edges" which are analogous to the factored edges used in our representation. Many of Marmot's analyses and optimizations operate on SSA form [7] rather than the CFG. Their algorithm for placing SSA ϕ-nodes can force basic blocks to be split (thus "unfactoring" the exception edges) to preserve the one-to-one correspondence between ϕ-nodes and CFG edges [10].

A large body of prior work has applied traditional control flow graph representations to support the analysis of programs written in languages that have frequently occurring exceptional control flow. The Vortex compiler [8] used a straightforward extension of a traditional control flow graph to enable both intraprocedural and interprocedural analysis of Java, Modula-3 [17], and Cecil [5] programs. Chatterjee et al. [6] have developed an extension of Landi and Ryder's conditional-points-to [14] that can precisely model exceptional control flow in the ICFG [16, 19, 4, 14, 12]. Sinha and Harrold [20] detail the subtleties of correctly capturing the semantics of Java's `try-catch-finally` structure in a traditional CFG representation. The Jex tool of Robillard and Murphy [18] constructs method level summaries of a program's exception structure. By examining these summaries, software developers can reason about the cross-module flow of exceptions and identify regions where error-handling policies are not being followed.

9 Conclusions

In this paper, we introduced the Factored Control Flow Graph (FCFG) representation and outlined how standard forward and backward data flow analysis algorithms can be adapted to work on this representation. The FCFG is a novel representation of a program's intraprocedural control flow that is designed to efficiently support the precise analysis of programs written in languages such as Java that have frequently occurring operations whose execution may result in exceptional control flow. Our empirical measurements showed that the FCFG representation leads to significantly larger basic blocks and smaller CFGs, compared to the traditional CFG representation. We believe that an alternative to the traditional CFG representation, such as the FCFG representation introduced in this paper, will be necessary for effective analysis of programs written in Java or in any other language with similar exception semantics.

We consider, as future work, extending the PEI-based extended-basic-block analysis framework to superblocks by treating any infrequently executed branch direction (with runtime information available from the Jalapeño optimizing compiler) the same as an exceptional control-flow branch due to a PEI. Any optimization across a PEI, be it a regular exception PEI or an infrequent-branch PEI, should provide for compensatory actions for the case a side exit is taken at the PEI.

Acknowledgments

We would like to thank Craig Chambers, Igor Pechtchanski, and Harini Srinivasan for their contributions to technical discussions on modeling exceptions in the Jalapeño optimizing compiler. We are also grateful to the anonymous reviewers and to G. Ramalingam, Frank Tip, and Laureen Treacy for their feedback on an earlier draft of this paper. We also acknowledge the Jalapeño team members for providing the infrastructure to capture the empirical measurements reported in Section 7.

References

[1] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[2] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño — a compiler-supported Java virtual machine for servers. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 36–46, May 1999. INRIA Technical Report #0228.

[3] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[4] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, July 1988. *SIGPLAN Notices, 23(7)*.

[5] Craig Chambers. The Cecil language: Specification and rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993. Revised, March 1997.

[6] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Complexity of concrete type-inference in the presence of exceptions. In *Lecture Notes in Computer Science, 1381*, pages 57–74. Springer-Verlag, April 1998. Proceedings of the *European Symposium on Programming*.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] Jeffrey Dean, Greg DeFouw, David Grove, Vass Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–100, October 1996.

[9] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 7(C-30):478–490, July 1981.

[10] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.

[11] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[12] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition – use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[13] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. In *The Journal of Supercomputing*, pages 229–248. Kluwer Academic Publishers, 1993.

[14] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992. *SIGPLAN Notices* 27(6).

[15] Steven S. Muchnick. *Advanced Compiler Design and Imlementation.* Morgan Kaufmann, 1997.

[16] Eugene W. Myers. A precise inter-procedural data flow algorithm. In *8th Annual ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[17] Greg Nelson. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[18] Martin P. Robillard and Gail C. Murphy. Analyzing exception flow in Java programs. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1999.

[19] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[20] Saurabh Sinha and Mary Jean Harrold. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*, pages 348–357, November 1998.