Dependence Analysis for Java

Craig Chambers*, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan

IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598, USA
{chambers, igor, vivek, mserrano, harini}@watson.ibm.com

Abstract. We describe a novel approach to performing data dependence analysis for Java in the presence of Java's "non-traditional" language features such as exceptions, synchronization, and memory consistency. We introduce new classes of edges in a dependence graph to model code motion constraints arising from these language features. We present a linear-time algorithm for constructing this augmented dependence graph for an extended basic block.

1 Introduction

Data dependence analysis is a fundamental program analysis technique used by optimizing and parallelizing compilers to identify constraints on data flow, code motion, and instruction reordering [11]. It is desirable for dependence analysis to be as precise as possible so as to minimize code motion constraints and maximize opportunities for program transformations and optimizations such as instruction scheduling. Precise dependence analysis for scalar variables is well understood; *e.g.,* an effective solution is to use SSA form [6]. In addition, much previous work has studied dependence analysis for memory accesses, typically focusing on references to array variables with affine subscripts (*e.g.,* [15]) and on dereferences of pointer variables (*e.g.,* [14]). In this paper, we address the problem of data dependence analysis for Java [2], focusing on Java's "non-traditional" language features such as exceptions and synchronization.

As in past work, we represent the result of data dependence analysis as a data dependence graph. We unify all dependences (due to scalars, memory locations, exceptions, synchronization, and the memory model) as dependences on *abstract locations.* Each register or scalar temporary is its own abstract location, and abstract locations for memory are derived from the results of an earlier alias analysis, such as type-based alias analysis [8]. We handle Java's plethora of instructions that may potentially raise exceptions by (a) constructing the dependence graph over extended basic blocks, unbroken by *potentially excepting instructions* (PEIs), (b) introducing a new kind of dependence (on an

* On sabbatical from the Dept. of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA; chambers@cs.washington.edu.

"exception state" abstract location) to sequence PEIs, and (c) treating each PEI as reading all scalar/memory locations live in the exception handler for the PEI. Instructions that do not read or write any abstract locations live in the exception handler of a PEI will end up being unrelated to the PEI in the dependence graph, and thus can be moved across the PEI during scheduling. (Previous Java implementations have routinely considered PEIs as barriers to code motion for all instructions.) We also separate the tests for exception conditions (such as null-pointer and array-bounds checks) from the instructions doing the work (such as field and array load and store operations), keeping only a special "validated" dependence edge from the test instruction to the work instruction, so that they can be optimized independently and scheduled more flexibly. To ensure proper memory consistency at synchronization points, we treat each acquire-lock Java instruction (`monitorenter`) as a write of all memory abstract locations, and each release-lock Java instruction (`monitorexit`) as a read of all memory abstract locations. Overall, these dependence constraints preserve the correct semantics of Java programs, but still allow significant scheduling and code motion flexibility. We present a linear-time algorithm for constructing this dependence graph for for a single extended basic block. This algorithm has been implemented in the Jalapeño optimizing compiler, where it is used for performing instruction selection and instruction scheduling [3].

The rest of the paper is organized as follows. Section 2 provides an overview of our design decisions for modeling basic blocks and control flow due to Java exceptions. Section 3 describes the main algorithm used for data dependence analysis. Section 4 presents an example illustrating our approach to dependence analysis. Section 5 discusses related work, and section 6 contains our conclusions.

2 Modeling Basic Blocks and Control Flow due to Java Exceptions

In this section, we provide an overview of our design decisions for modeling basic blocks and control flow due to Java exceptions. A detailed description of this approach is given in [4], which describes how the Jalapeño optimizing compiler builds extended basic blocks and performs global analysis in the presence of exceptions. The focus of the work in [4] was program analysis; it did not address the problem of code motion (dependence analysis) in the presence of exceptions.

Exceptions in Java are *precise*. When an exception is thrown at a program point, (a) all effects of statements and expressions before the exception point must appear to have taken place; and (b) any effects of speculative execution of statements and expressions after the exception point should not be present in the user-visible state of the program. Our goal is to efficiently compute the most precise (least constrained) set of dependences that we can, while obeying Java's exception semantics. For checked exceptions and runtime exceptions, the Java language specification identifies all statements/operations that can potentially throw an exception. In the remainder of this paper, we use the term *PEI* (*P*otentially *E*xception-throwing *I*nstruction) to denote these statements/operations.

Traditionally, a basic block consists of a set of instructions that is sequentially executed: if the first instruction of the basic block is executed, then each subsequent instruction in the basic block will be executed in turn [11]. Thus, in the standard model of a basic block as a single-entry, single-exit region of the control flow graph, any PEI will signify the end of its basic block. Since PEIs are quite frequent in Java, the approach taken by the Jalapeño optimizing compiler is that PEIs do not force the end of a basic block. Therefore, a basic block can be a single-entry multiple-exit sequence of instructions (similar to an extended basic block [11] or a superblock), and can be significantly larger than basic blocks that must be terminated by PEIs.

Another key decision in the design of the Jalapeño optimizing compiler is to separate out the checking performed in a PEI from its actual work. For example, a `getfield` (or `putfield`) instruction is split into an explicit `null_check` instruction followed by a `load` (or `store`) instruction. `null_check` instructions are also used to guard against accessing null arrays and invoking virtual methods on null objects. Explicit instructions are generated for array-bounds and zero-divide checks, in a similar manner. The ordering relationship between the test and worker instructions is captured by having the test instruction contain a pseudo-assignment of a scalar temporary, and having the later worker instruction contain a pseudo-read of this temporary. (Strictly speaking, the temporary does not hold a boolean condition, but instead a "validated" signal. The value in the temporary is immaterial to the worker instruction; all that matters is that the test has been successfully performed.) The advantage of creating explicit test instructions is that they are eligible for redundancy elimination via global analysis, just like other instructions in the IR (Intermediate Representation).

Even if some of the tests can be performed implicitly by hardware, such as implicit null pointer tests in the Jalapeño JVM [1], there is value in modeling the tests explicitly in the IR because doing so can enable more flexibility in code motion and instruction scheduling. Additionally, virtual method invocations can be optimized (such as through static binding and inlining) without worry that a required `null_check` might be lost. To reclaim the efficiency of implicit hardware-performed checks, we include a post-pass that merges each remaining `null_check` instruction with the first following load or store instruction (when legal). After this merging, the `null_check` becomes implicit in the load or store instruction, and is accomplished by an underlying hardware mechanism.

3 Dependence Analysis

Traditionally, data dependence graphs represent dependences among register reads and writes, and memory reads and writes. A data dependence can be classified as a *true* dependence (a write to a location followed by a read of the same location), an *anti* dependence (a read of a location followed by a write of the same location), or an *output* dependence (a write of a location followed by another write of the same location) [15]. Data dependences can be easily computed for (virtual or physical) registers, because registers are explicitly named as operands of instructions. Computing data dependences for memory locations (*e.g.,* object fields and array elements) is a harder problem, and exact solutions are undecidable in general. The difficulty arises from the *aliasing* problem, where syntactically different expressions may nevertheless refer to the same memory location. In addition to pointer-induced aliases, certain reads and writes of locations must be ordered so as to obey the semantics of *exceptions*, *synchronization*, and the *memory model* in Java. Finally, exceptions themselves must be properly ordered in order to ensure that the view of program state as needed by the corresponding exception handlers is preserved.

Our approach integrates all these new kinds of dependence constraints into a single framework, based on true, anti, and output dependences. We model registers, memory locations, and even exception and synchronization states as *abstract locations.* We present a simple two-pass algorithm, given in Figure 2, that iterates over the instructions in each (extended) basic block to construct the dependence graph. The execution time and space for the algorithm is linear in the size of the basic block, *i.e.,* linear in the number of defs and uses of abstract locations across all instructions in the basic block.

The rest of this section is organized as follows. Section 3.1 summarizes the abstract locations used in our approach. Section 3.2 contains a brief description of the dependence analysis algorithm in figure 2. Sections 3.3 to 3.7 outline how the scheduling constraints of ALU instructions, memory instructions, exception-generating instructions, call instructions, and synchronization instructions are modeled using abstract locations. Section 3.8 outlines the impact of Java's memory coherence assumption on dependence analysis. Section 3.9 discusses some extensions to our basic algorithm.

3.1 Abstract Locations

As illustrated in Figure 1, we use type information to partition concrete memory locations into *abstract locations* (location types). Each abstract location represents a "may-aliased" equivalence class of concrete locations, *i.e.,* any two references to the same abstract location can potentially interfere, but two references to distinct abstract locations cannot interfere.

The following abstract locations represent different cases of global or heap-allocated data:

Fields: Each field declaration has a corresponding abstract location; all loads of that field *use* its abstract location, and stores of that field *define* its abstract location. Distinct fields have distinct abstract locations.

Array Elements: Each primitive array type (*i.e.,* `bool[]`, `short[]`, `int[]`, `long[]`, `float[]`, `double[]`, and `char[]`) has a unique abstract location modeling the concrete locations of its elements, accessed by `aload` and `astore` instructions for an array of the corresponding type. An additional

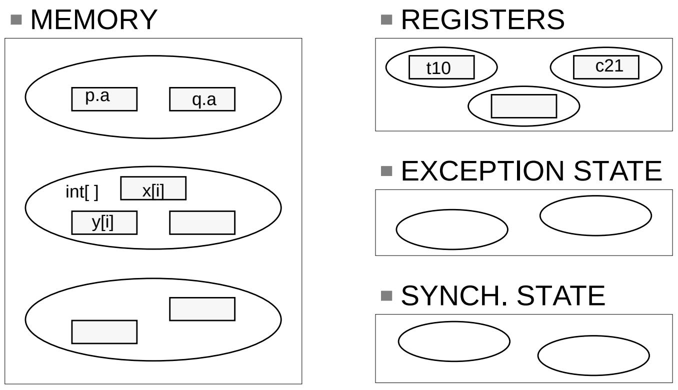

Fig. 1. Examples of abstract locations

abstract location, `Object[]`, is used to represent the locations of elements of all arrays of objects, including arrays of arrays.
This refinement of array element abstract locations based on the type of the elements of the arrays reflect a simple kind of type-based alias analysis [8]. Results of more extensive alias analysis can be incorporated as further refinements of these abstract locations, so long as the resulting abstract locations partition the underlying concrete locations.

Statics: Each static data element (including static fields) has a distinct abstract location.

In addition, there is a distinct abstract location associated with each of the following that represent method-local data:

– A symbolic register.
– A spill location, based on its constant offset in the stack frame. (This abstract location is only used when performing dependence analysis after register allocation.)
– `exception_type`, that is used to model all exception dependences. A PEI contains a pseudo-assignment of this abstract location, assigning it the class of the raised exception.

Each operand of an IR instruction accesses an underlying abstract location, as described above. In some cases, this requires creating pseudo-operands that specify defs/uses of abstract locations, even though these defs/uses are not part of the concrete semantics of the instruction. For example, a load instruction has an explicit operand that is a use of the symbolic register that contains the load address, and a pseudo-operand that represents the abstract location being accessed in global memory.

In this paper, `p.uses` will be used to denote the set of abstract locations used (*i.e.*, read) by instruction `p`, and `p.defs` will be used to denote the set of abstract locations defined (*i.e.*, written) by `p`. It is important to note that the execution time of the dependence analysis algorithm presented in section 3.2 is linear in the sum of the sizes of the `p.uses` and `p.defs` sets for all instructions, `p`, in the basic block. When conservative intraprocedural analysis information is used to determine these sets, the size of each set will be bounded by a constant, and the algorithm will take time linearly proportional to the number of instructions. More precise interprocedural information for `call` instructions can lead to larger, non-constant-sized `defs` and `uses` sets.

3.2 Linear-Time Dependence Analysis Algorithm

Figure 2 contains our algorithm for computing the dependence graph for an (extended) basic block, in time and space that is linear in the size of the block, *i.e.,* linear in the number of operands (and pseudo-operands) across all instructions in the block.

The first pass of the algorithm traverses the instructions in the block in forward order. The key idea is to associate a "last definition" value, `last_def`, with each abstract location used in the block. For a given abstract location, `loc`, `loc.last_def` is initialized to NULL. As this pass proceeds, `loc.last_def` is set to the most recent definition operand that performs a write on abstract location `loc`. In general, when a use `u` is encountered with `loc` $=$ `u.location` such that `loc.last_def` $\neq$ NULL, a flow dependence edge is created from `loc.last_def .instruction` to `u.instruction`. As with all dependence edges created by this algorithm, this edge may represent a register, memory, or exception dependence, depending on the underlying abstract location. Similarly, when a def `d` is encountered with `loc` $=$ `d.location` such that `loc.last_def` $\neq$ NULL, an output dependence edge is created from `loc.last_def.instruction` to `d.instruction`. In addition, `loc.last_def` is updated to `d`.

The second pass of the algorithm traverses the instructions in the reverse order. All `last_def` values are reinitialized to NULL. As this pass proceeds, `loc.last_def` is set to the most recent definition operand (in a backwards traversal) that performs a write on abstract location `loc`. When a use `u` is encountered such that `u.location.last_def` $\neq$ NULL, an anti dependence edge is created from `u.instruction` to `u.location.last_def.instruction`.

The use of a single `last_def` value for each abstract location guarantees the linear-sized time and space complexity for this algorithm. Consider the following IR fragment as an example:

```
s1:     putfield a.x := t1
s2:     putfield b.x := t2
s3:     t3 := getfield c.x
```

All three instructions access the same abstract location (field `x`). Therefore, our algorithm will only insert an output dependence edge from `s1` to `s2` and a flow dependence edge from `s2` to `s3`.

In contrast, traditional dependence analysis algorithms (*e.g.,* [15]) will also insert a flow dependence edge from `s1` to `s3` (assuming that object references `a`, `b`, `c` can potentially be aliased). In general, there can be quadratic number

```
Dependence_Graph_Construction(BasicBlock bb) {

  foreach abstract location loc in bb.locations do
    loc.last_def := NULL      // clear last defs
  end for

  for each instruction p in bb.instructions in forward order do
    let pnode := dependence graph node corresponding to instruction p

    // Abstract location determines if register/memory/exception dependence
    for each use operand u in p.uses do
      let loc := u.location
      if loc.last_def != NULL then
        create TRUE dependence edge from loc.last_def.instruction to pnode
      endif
    end for

    for each def operand d in p.defs do
      let loc := d.location
      if loc.last_def != NULL then
        create OUTPUT dependence edge from loc.last_def.instruction to pnode
      endif
      loc.last_def := d  // record last def
    end
  end

  foreach loc in bb.locations do
    loc.last_def := NULL  // clear last defs
  end

  for each instruction p in bb.instructions in backward order do
    let pnode := dependence graph node corresponding to instruction p
    // record last def
    foreach def operand d in p.defs do
      let loc := d.location
      loc.last_def := d
    end

    // create anti dependence edges
    foreach use operand u in p.uses do
      let loc := u.location
      if loc.last_def != NULL then
        create ANTI dependence edge from pnode to loc.last_def.instruction
      endif
    end
  end
}
```

Fig. 2. Algorithm to compute dependence graph of an extended basic block

of extra edges created by using traditional dependence analysis algorithms. Our algorithm avoids creating such transitively implied edges by ensuring that each use or def is the target of at most one edge in the dependence graph.

3.3 ALU Instructions

Simple arithmetic instructions have some number of register operands and some number (typically one) of register destinations. Each register is modeled in our system by a unique abstract location, distinct from any other abstract location, representing the fact that a register is not aliased with any other register or memory location. Given this definition of abstract locations, our algorithm will construct data dependence edges for registers.

3.4 Memory Instructions

Memory instructions have abstract locations that represent the global data being accessed. A load instruction has a use pseudo-operand that reads the abstract location representing the area of memory possibly read by the load, while a store instruction has a def pseudo-operand that writes to the abstract location representing the area of memory possibly written by the store.

The appropriate abstract locations for memory pseudo-operands are determined by an alias analysis preceding dependence graph construction. Alias analysis is a rich research area, but fortunately it is separable from the dependence graph construction problem. Our dependence graph construction algorithm only assumes that the results of alias analysis can be expressed as a partitioning of concrete locations, as discussed in section 3.1.

3.5 Exception Instructions

A conservative approach to dependence analysis of exception instructions would simply prevent any write operation (including updates to scalar local variables) from moving above or below a PEI. However, this would greatly limit opportunities for scheduling, and largely defeat the purpose of scheduling across PEIs.

Given a PEI `p`, our approach is to include in `p.uses` abstract locations for all variables and memory locations live at the entry of the exception handler to which the exception would be routed. Then, write operations that do not modify an abstract location in `p.uses` can be free to move across the PEI `p`. Live variable analysis can compute the set of local variables that are live on entry to each handler (all global and heap-allocated data would normally be considered to be live, by default).

In addition, we include in `p.defs` the `exception_type` abstract location. This ensures that PEIs are contrained to be executed in order via the output dependences between them generated by the defs of `exception_type`.

3.6 Call Instructions

The explicit operands of a call instruction identify the call's arguments and result. In addition, pseudo-operands are introduced to represent abstract memory locations possibly read and/or written by the callee(s). In the absence of interprocedural analysis information, a `call` instruction is assumed to define all abstract locations that represent memory locations (*i.e.,* all fields, arrays, and static data). More precise sets of abstract locations could result from interprocedural use and side-effect analysis.

In addition, for calls that can raise exceptions, the rules for PEIs described in section 3.5 should be followed.

3.7 Synchronization Instructions

The `monitorenter` and `monitorexit` synchronization instructions define a critical section. Java's memory model defines the effect of these instructions on memory locations from the point of view of a particular thread as follows: at a `monitorenter` instruction, the thread updates its local view of memory based on the "true" or global view, and at a `monitorexit` instruction, the thread flushes its local view of memory back to the global view.

To construct a safe dependence graph, we expand a `monitorenter` instruction into a `lock` operation and an `update` operation, and a `monitorexit` instruction into a `publish` operation and an `unlock` operation, as shown in the example in Figure 3. A single abstract location S (representing the global synchronization state) is used to serialize all `lock`/`unlock` instuctions within a thread (by treating each of them as a def of S). The memory model semantics is captured by treating the `update` operation as a write of all abstract locations (more precisely, all locations that can be written by another thread), and the `publish` operation as a read of all abstract locations (more precisely, all locations that can be read by another thread). Finally, an abstract location for the synchronized object (*lock1* in Figure 3) is used to ensure that all memory read/write operations remain within their original critical section (by treating `lock` and `unlock` as defs of *lock1*, and memory operations in the critical section as uses of *lock1*).

Note that instructions that access only scalars can be moved freely across synchronization instructions *e.g.,* see the defs of scalars `a` and `b` in Figure 3. A memory read operation (*e.g.,* the read of `r.z` in Figure 3) can be moved above a `monitorexit` if so desired (but not above a `monitorenter`); further, because of the presence of the *lock1* abstract location, the code motion can be reversed by moving the memory read back below the `monitorenter` without exhibiting the anomaly described in [13]. Finally, the expansion of the `monitorenter` and `monitorexit` instructions enables synchronization elimination to be performed by removing the `lock`/`unlock` operations, while still retaining the memory model requirements that are captured by the `update` and `publish` operations.

```
    a = ...
S, lock1 = monitorenter
   ... = p.x ‹lock1›
 q.y = ... ‹lock1›
    b =
S, lock1 = monitorexit
   ... = r.z
```

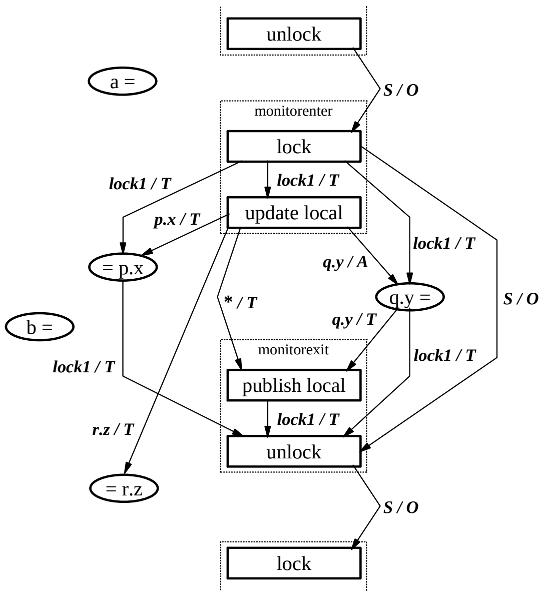

Fig. 3. Example of modeling dependences due to monitorenter and monitorexit

3.8 Memory Coherence

The Java memory model enforces a strict memory coherence semantics for all accesses to the same (concrete) location, even when not guarded by synchronization instructions. It has been observed that reordering of potentially aliased load instructions can be an illegal transformation for multithreaded programs written for a memory model that includes the strict memory coherence assumption [10, 13]. For example, consider the following code fragment:

```
getfield p.x
getfield q.x
```

Suppose p and q may be aliased. While one thread (T1) executes the above code fragment, another thread (T2) can execute multiple `putfield p.x` instructions. The Java memory model requires that, if p and q point to the same object in memory, then thread T1 should not see the values written by thread T2 out of order. Since the compiler must conservatively assume the possible existence of a thread like T2, it would be illegal for the compiler to reorder the above `getfield` instructions. Therefore, under this strict memory coherence model, the dependence graph must contain an edge from the first `getfield` instruction to the next, so as to prevent this reordering.

Unfortunately, adding read dependences between all possibly aliased loads can significantly constrain opportunities for instruction reordering. As an alternative to the default, strict memory coherence model, the Jalapeño optimizing compiler allows users to select a *weak consistency model* that does not require memory coherence *e.g.,* Location Consistency (LC) [9]. This memory model guarantees that properly synchronized programs will have the same execution semantics as in the default model, without imposing additional constraints on the dependence graph. Recently, there has been a proposal for a new memory model for Java [12] that is based on the LC model.

3.9 Extensions

The dependence analysis algorithm described in this section takes linear time because of the use of a single `last_def` value for each abstract location. This algorithmic property relies on the fact that abstract locations do not overlap. Requiring abstract locations to be disjoint can limit the precision and expressiveness of more sophisticated alias analyses. Our algorithm can be extended to handle abstract locations that form a lattice of possibly overlapping regions, but the worst-case time complexity of the algorithm may degrade to quadratic time (although a factored representation of the resulting dependence graph can still require only a linear amount of space).

PEIs, calls, and synchronization instructions may all be considered to read and/or write all global memory abstract locations. Representing all these abstract memory locations explicitly would greatly increase the size of the basic block and the resulting dependence graph. To reduce the number of such abstract memory locations, only those abstract memory locations referenced explicitly by some load or store instruction in the same (extended) basic block need to be included explicitly; all other abstract memory locations can be summarized by a single "rest of memory" abstract location (this extra abstract location is only needed in basic blocks that contain no explicit loads or stores, but do contain synchronization instructions and/or calls). The resulting abstract locations still form a partition of the set of concrete locations accessed by that block.

Currently, PEIs are considered to write to the special `exception_type` abstract location, ensuring that PEIs are serialized. However, two PEIs with the same exception handler can be *indistinguishable* from the perspective of the exception handler, if (a) both PEIs raise the same kind of exception (*e.g.,* both are null-pointer tests or both are array-bounds checks) or if the handler always handles whatever exceptions both PEIs raise and the handler otherwise ignores the exception object it is passed, (b) both report the exception at the same place in the program as determined by the resolution of the Java implementation's `getStackTrace` debugging native method, and (c) both have the same dependence constraints on the set of abstract locations live in the handler. Indistinguishable PEI instructions can be freely reordered, up to any other dependence constraints on their operands. This reordering ability can be implemented by extending our algorithm to track the *value* assigned to the special `exception_type` abstract location, where this value reflects the kind of exception raised by the PEI (if the handler cares) and the debugging location information for the PEI. Then a PEI's assignment to the `exception_type` abstract location leads to an output dependence with the location's `last_def` PEI only if they assign different values to the location. Otherwise the new PEI is inserted in the dependence graph "in parallel" with the previous PEI, forming a kind of equivalence class in the dependence graph of PEIs that are indistinguishable. A straightforward implementation of this idea would lead to a quadratic-time and -space algorithm, but a more sophisticated algorithm exists, based on maintaining a factored representation of the dependence graph, that has only linear time and space complexity.

4 Example

Java source program:

```
    public class LCPC {
        int f1, f2;
        static void foo(int[] x, int i, LCPC b, int q) {
            if (q != 0)
                x[i] = ((b.f1 = x[i]) + b.f2 + i/q);
        }
    }
```

Fig. 4. An example Java program

In this section, we use a simple example to illustrate the key aspects of our approach to computing data dependences in a Java program, and how this algorithm fits into the rest of the compiler framework. Consider method `foo()` in the Java program shown in Figure 4. In the remainder of this section, we will focus our attention on the main basic block for the `q != 0` case in method `foo()`. Figure 5 shows the (unoptimized) HIR (high-level IR) for this basic block[1]. The "PEI" annotation is used to identify exception-generating instructions.

Note that all exception tests are explicit in Figure 5. For example, the load of `x[i]` in Figure 4 is translated into three HIR instructions — `null_check`, `bounds_check`, and `int_aload`. If an exception test fails, then control is transferred from the exception test instruction to a runtime routine in JVM; this runtime routine examines the exception table (not shown in the IR figures) to determine how the exception should be handled (either by transferring to a handler block or exiting from the method). If an exception test succeeds, the instruction(s) dependent on the exception test can be enabled for execution.

[1] As described in [3], the HIR is actually generated from the bytecodes for method `foo()`, and not from the Java source code.

```
Bytecode
Offset      Operator         Operands
------      --------         --------
 4          LABEL1           B1@4
 9     PEI  null_check       c21 = a0
 9     PEI  bounds_check     c22 = a0, a1, c21
 9          int_aload        t4 = @{ a0, a1 }, [c21,c22]
11     PEI  null_check       c23 = a2
11          putfield         a2, t4, <LCPC.f1>, c23
15     PEI  null_check       c24 = a2                 . .     (*)
15          getfield         t5 = a2, <LCPC.f2>, c24  . . --> c23
18          int_add          t6 = t4, t5
21     PEI  int_zero_check   c25 = a3                 . .     (*)
21          int_div          t7 = a1, a3, c25         . . --> none
22          int_add          t8 = t6, t7
23     PEI  null_check       c26 = a0                 . .     (*)
23     PEI  bounds_check     c27 = a0, a1, c26        . .     (*)
23          int_astore       t8, @{ a0, a1 }, [c26,c27] . . --> [c21,c22]
            END_BBLOCK       B1@4
```

Fig. 5. HIR generated from bytecode for basic block 1 (bytecodes 4 ... 23) in method `foo()`. a0 ... a3 are the method arguments. t4 ...t7 are temporaries. c21 ... c27 are condition registers. Lines marked with (*) are removed by subsequent optimizations, and condition codes are replaced by the ones following the arrows.

```
Bytecode                                             Issue   Dependence
Offset      Operator         Operands                Time    Graph Node
------      --------         --------                -----   ----------
 4          LABEL1         B1@4
 9     PEI  null_check     c21 = a0                    0       (1)
 9          ppc_lwz        t10 = @{ -4, a0 }, c21      0       (2)
 9     PEI  ppc_tw         c22 = ppc_trap <=U, t10, a1 2       (3)
 9          ppc_slwi       t11 = a1, 2                 2       (4)
 9          ppc_lwzx       t4 = @{ a0, t11 }, [c21,c22] 3      (5)
11     PEI  null_check     c23 = a2                    5       (6)
11          ppc_stw        t4, @{ -16, a2 }, c23       5       (7)
15          ppc_lwz        t5 = @{ -20, a2 }, c23      6       (8)
18          ppc_add        t6 = t4, t5                 8       (9)
21          ppc_divw       t7 = a1, a3                 9       (10)
22          ppc_add        t8 = t6, t7                29       (11)
23          ppc_stwx       t8, @{ a0, t11 }, [c21,c22] 30      (12)
            END_BBLOCK     B1@4
-----------------------------------------------------------------------
Completion time:                                             34 cycles
```

Fig. 6. MIR generated for basic block 1 in method `foo()`. c21 ... c23 are condition registers. Issue times assume no out-of-order execution.

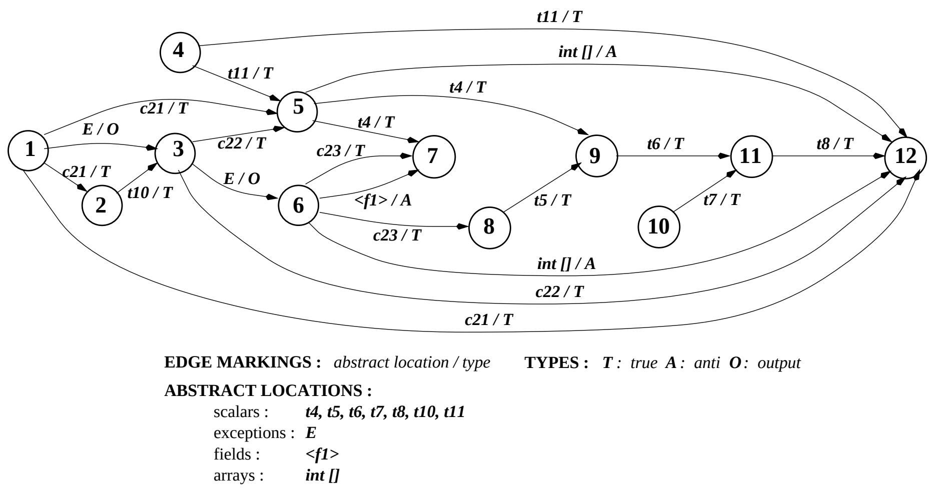

Fig. 7. Dependence graph of basic block 1 in method `foo()`

To make explicit the connection between an exception test and the instructions that it guards in our IR, an exception test is assumed to compute a "condition register" temporary as a result. Each instruction that depends on the success of the exception test takes the condition register as an extra input operand. These condition registers do not appear in the final machine code, but their presence in the IR ensures that IR optimizations will not cause a guarded instruction to be executed prior to its exception test. For example, condition register `c23` in figure 5 captures the result of a null-pointer check for argument `a2` (*viz.,* object reference `b`). `c23` is then used as an input operand by the `putfield` instruction. For future work, we are exploring the possibility of allowing a guard instruction for an arithmetic exception (*e.g.,* the `zero_check` guard for a divide-by-zero exception) to be performed after its corresponding compute instruction. These possibilities are processor-specific and can be expressed in the IR by removing the condition register from the IR instructions when the test instruction need not precede the compute instructions.

Redundant check elimination is performed during global optimizations on the HIR [3]. All exception test instructions in figure 5 that are marked `(*)` are redundant and can be removed. The instructions marked `-->` are then modified to use condition registers from equivalent exception tests. Note that the number of PEIs has been reduced from seven to three in figure 5 after optimization. This is one of the key benefits of making exception tests explicit in our IR. After elimination of redundant tests, we are left with one `null_check` for each of `x` and `b`, and one `bounds_check` for `x`. Note that the `zero_check` instruction for the divide instruction was eliminated because of the `q != 0` test in method `foo()`. Further optimization of the remaining tests might be possible with knowledge of interprocedural calling context information for method `foo()`.

As described in [3], the optimized HIR in figure 5 is next converted to LIR (lower-level IR). High-level operations such as `aload`, `astore`, `getfield`, putfield are expanded into multiple low-level operations that are closer to machine code. After LIR optimizations are performed, the LIR is translated to MIR (machine-level IR) by performing instruction selection; the resulting MIR is shown in figure 6. MIR instructions belong to the target instruction set architecture (in this case, the PowerPC) with two minor extensions. First, null_check instructions are allowed to be present in the MIR because their presence gives greater flexibility for instruction scheduling. (After scheduling, the null_check instructions will be merged with load/store instructions when possible, so as to be performed implicitly[2].) Second, the condition register operands are included in the MIR instructions so as to enable precise dependence analysis for instruction scheduling. (The condition registers will be discarded prior to generation of machine code.)

Next, our dependence graph algorithm is applied to construct the dependence graph over MIR instructions. Figure 7 shows the resulting data dependence graph for the extended basic block from figure 6.

```
Bytecode                                                     Issue
Offset      Operator           Operands                      Time
------      --------           --------                      -----
 4          LABEL1             B1@4
 9     PEI  null_check         c21 = a0                       0
 9          ppc_lwz            t10 = @{ -4, a0 }, c21         0
 9          ppc_slwi           t11 = a1, 2                    0
21          ppc_divw           t7 = a1, a3                    0
 9     PEI  ppc_tw             c22 = ppc_trap <=U, t10, a1    2
11     PEI  null_check         c23 = a2                       2
15          ppc_lwz            t5 = @{ -20, a2 }, c23         2
 9          ppc_lwzx           t4 = @{ a0, t11 }, [c21,c22]  3
11          ppc_stw            t4, @{ -16, a2 }, c23          5
18          ppc_add            t6 = t4, t5                    5
22          ppc_add            t8 = t6, t7                   20
23          ppc_stwx           t8, @{ a0, t11 }, [c21,c22]   21
            END_BBLOCK         B1@4
-----------------------------------------------------------------------
Completion time:                                             25 cycles
```

Fig. 8. MIR generated after instruction scheduling for basic block 1 in method foo(). This schedule assumes that the order of exception-generating instructions must be preserved. Issue times assume no out-of-order execution.

[2] Jalapeño uses a layout where an object's fields and an array's length are stored at negative offsets from the object's reference [1], to support hardware null checking on the PowerPC architecture where low memory (including address 0) is not page-protected but high memory is.

```
Bytecode
Offset      Operator          Operands
------      --------          --------
 6          LABEL1            B1@4
 9     PEI  ppc_lwz           t10 = @{ -4, a0 }
 9          ppc_slwi          t11 = a1, 2
21          ppc_divw          t7 = a1, a3
 9     PEI  ppc_tw            ppc_trap <=U, t10, a1
15     PEI  ppc_lwz           t5 = @{ -20, a2 }
 9          ppc_lwzx          t4 = @{ a0, t11 }
11          ppc_stw           t4, @{ -16, a2 }
18          ppc_add           t6 = t4, t5
22          ppc_add           t8 = t6, t7
23          ppc_stwx          t8, @{ a0, t11 }
            END_BBLOCK        B1@4
```

Fig. 9. Final MIR after merging `null_check` instructions in figure 8.

Next, instruction scheduling is applied to the dependence graph over MIR instructions. Figure 8 shows the new sequence of target instructions obtained when applying a simple list scheduling algorithm to the dependence graph from figure 7. The use of explicit exception tests enables more reordering to be performed than would otherwise have been possible. For example, since `b.f2` is used as an input to a (time-consuming) divide operation, it is beneficial to move the load of this field as early as possible. As shown in figure 8, the divide instruction and the load of `b.f2` (at stack offset -20) are both moved prior to the load of `x[i]` and the store of `b.f1`. This reordering would not have been possible if exception tests were not modeled explicitly in the IR. To appreciate the impact of instruction scheduling for this example, note that the completion time for the basic block was reduced from 34 cycles (when using the instruction ordering in figure 6) to 25 cycles (when using the new ordering in figure 8). (Completion times were estimated assuming that the processor does not perform any out-of-order execution of instructions.)

Finally, figure 9 contains the MIR obtained after merging each `null_check` instruction with the first following load or store instruction that uses its condition register result. In this example, the two `null_check` instructions have been successfully merged with load instructions. Note that the load instructions (`ppc_lwz`) are now marked as PEIs. No update is required to the exception handler tables when this merging is performed because the handler blocks are identical for all PEIs that belong to the same basic block.

5 Related Work

Dependence analysis for scalar variables has been well understood for a long time; for example, def-use chaining [11] was an early technique for identifying true dependences within and across basic blocks. The same principle has also been used to compute anti and output dependences for scalar variables. Scalar renaming [5] is an effective technique for eliminating anti and output dependences, and SSA form is a popular approach to obtain a canonical renaming of scalars [6].

The bulk of past research on dependence analysis for memory accesses has focused on array variables with affine subscripts and on pointer dereferences. The advent of vectorizing and parallelizing compilers led to several data dependence analysis techniques being developed for array accesses with affine subscript expressions (*e.g.,* [15]); this is a special case that is important for optimization of scientific programs written in Fortran and C. A lot of attention has also been paid to "points-to" alias analysis of pointer variables, with the goal of improving the effectiveness of compiler optimizations of pointer-intensive programs written in C and C++. Points-to analysis of general C and C++ programs is a hard problem and the experience thus far has been that most algorithms for points-to analysis consume excessive amounts of time and space (the algorithm in [14] is a notable exception). More recently, it has been observed [8] that type-based alias information can be used to obtain simple and effective dependence analysis of object references in statically-typed OO languages such as Modula-3 and Java. Finally, [13] has identified restrictions due to Java's memory model that must be imposed on compiler optimizations for multithreaded Java programs.

This paper addresses the problem of data dependence analysis for Java in the presence of Java's "non-traditional" language features such as exceptions, synchronization, and memory consistency. The "abstract locations" introduced in section 3 can be viewed as an generalization of type-based alias analysis to also deal with exceptions. Most previous compilers for object-oriented languages (*e.g.,* [7]) modeled exceptions as branch instructions that terminate basic blocks, and hence did not have to deal with dependence analysis for exceptions. In contrast, our compiler builds extended basic blocks that can include multiple PEIs within the same basic block.

6 Conclusions and Future Work

In this paper, we addressed the problem of data dependence analysis for Java in the presence of features such as exceptions, synchronization, and memory consistency. We introduced dependences on new classes of abstract locations to model code motion constraints arising from these language features. We presented a linear-time algorithm for constructing this augmented dependence graph for an extended basic block (using type-based alias analysis for Java). As motivation for dependence analysis, we discussed two phases of the Jalapeño dynamic optimizing compiler, instruction selection and instruction scheduling, that use the data dependence graph. An interesting direction for future work is to use the dependence rules for abstract locations presented in this paper to enable code motion transformaction across regions that are larger than a single extended basic block.

References

1. Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño — a Compiler-Supported Java Virtual Machine for Servers. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999. Also available as INRIA report No. 0228, March 1999.
2. Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.
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 Java Grande Conference*, June 1999.
4. Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Toulouse, France, September 1999.
5. Ron Cytron and Jeanne Ferrante. What's in a Name? Or the Value of Renaming for Parallelism Detection and Storage Allocation. *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.
6. 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.
7. Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, San Jose, CA, October 1996.
8. Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, May 1998.
9. Guang R. Gao and Vivek Sarkar. Location Consistency: Stepping Beyond the Memory Coherence Barrier. *International Conference on Parallel Processing*, August 1995.
10. Guang R. Gao and Vivek Sarkar. On the Importance of an End-To-End View of Memory Consistency in Future Computer Systems. *Proceedings of the 1997 International Symposium on High Performance Computing, Fukuoka, Japan*, November 1997.
11. Steven S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.
12. William Pugh. A new memory model for Java. Note sent to the JavaMemoryModel mailing list, http://www.cs.umd.edu/ pugh/java/memoryModel, October 22, 1999.
13. William Pugh. Fixing the Java Memory Model. In *ACM Java Grande Conference*, June 1999.
14. Bjarne Steensgaard. Points-to analysis in almost linear time. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 32–41, January 1996.
15. Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.