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. In this paper, we address 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. 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 using type-based alias analysis for Java. As motivation for dependence analysis, we discuss two phases of the Jalapeño dynamic optimizing compiler, instruction selection and instruction scheduling, that use the data dependence graph and benefit from more precise dependence analysis.

1 Introduction

Data dependence analysis is a fundamental program analysis technique used by optimizing and parallelizing compilers to identify constraints on code motion and instruction reordering [18]. 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 [9]. In addition, past research has studied dependence analysis for memory accesses, with a focus on array variables with affine subscripts (*e.g.,* [22]) and on dereferences of pointer variables (*e.g.,* [21]). In this paper, we address the problem of dependence analysis for Java [3] in the presence of Java's "non-traditional" language features such as exceptions, synchronization, and memory consistency.

As in past work, we represent the result of data dependence analysis as a data dependence graph We unify scalar, memory, exception, and synchronization dependences as dependences on "abstract locations". Each register or scalar temporary is its own abstract location. Abstract locations for memory are derived from the results of the alias analysis performed by the compiler. Exception-generating instructions are modeled as defs of an "exception state" abstract location, and thus sequenced in total order. To ensure proper memory consistency at synchronization points, we treat each `monitorenter` Java instruction as

* On sabbatical from U. Washington.

a write of all memory abstract locations, and each `monitorexit` Java instruction as a read of all memory abstract locations, in addition to the ordering constraints necessary to ensure that memory instructions remain within a critical section. 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 simplicity, this paper only addresses the data dependence analysis problem for a single basic block. This is not a major restriction because call instructions and exception-generating instructions do not terminate a basic block in our program model. In fact, the rules for the new classes of dependences introduced in this paper can also be used to compute dependences across basic blocks, either as explicit dependence edges in a global program dependence graph [13] or as implicit constraints on global code motion.

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 uses an example to illustrate the key aspects of our approach to dependence analysis. Section 4 describes the main algorithm used for data dependence analysis. Sections 5 and 6 discuss two applications of dependence analysis information in the Jalapeño optimizing compiler [4], namely instruction selection and instruction scheduling. Section 7 discusses related work, and section 8 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 [5], which describes how basic blocks are defined and global analysis is performed in the presence of exceptions. However, the work in [5] did not address the problem of dependence analysis in the presence of exceptions. These design decisions have been incorporated in the Jalapeño optimizing compiler [4], which provides the context for our research.

Exceptions in Java 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 not be present in the user-visible state of the program. Our goal is to compute the most precise (least constrained) set of dependences 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*otential *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 [18]. 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 PEI's are quite frequent in Java, the approach taken by the Jalapeño optimizing compiler for modeling control flow due to exceptions is that PEI's 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 [18] or a superblock), and can be significantly larger than basic blocks that must be terminated by PEI's.

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 implement checks for invoking virtual methods on null objects. Explicit instructions are generated for array-bounds and zero-divide checks, in a similar manner. 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. Even if some of the tests can be performed implicitly by hardware, such as implicit null pointer tests in the Jalapeño JVM [2], there is value in modeling the tests explicitly in the IR because doing so can enable more flexibility in code motion and instruction scheduling (as illustrated by the example in section 3). We include a post-pass that merges each remaining `null_check` instruction with the first following load or store instruction that uses its condition register result, when legal to do so. After this merging, the null-check becomes implicit in the load or store instruction, and is accomplished by an underlying hardware mechanism.

3 Example

In this section, we use a simple example to illustrate the key aspects of our approach to computing data dependences in a Java program. Consider method `foo()` in the Java program shown in Figure 1. 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 2 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 2. For example, the load of `x[i]` in Figure 1 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 [4], the HIR is actually generated from the bytecodes for method `foo()`, and not from the Java source code.

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. 1. An example Java program

```
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. 2. 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.

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" 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 2 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 [4]. All exception test instructions in figure 2 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 PEI's has been reduced from seven to three in figure 2 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 [4], the next step is to convert the optimized HIR in figure 2 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. In addition, a `bounds_check` instruction is converted to a `trap_if` instruction. Figure 3 shows the MIR (machine-level IR) obtained after performing instruction selection on the LIR using the approach outlined in section 5. 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 binary machine code.)

[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 [2], to support hardware null checking on the PowerPC architecture where low memory (including address 0) is not page-protected but high memory is.

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

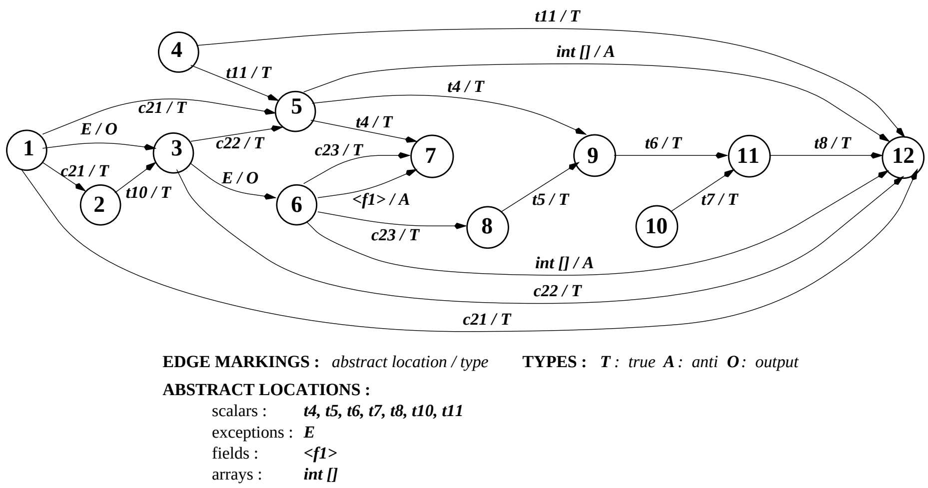

Fig. 4. Dependence graph of basic block 1 in method `foo()`

Figure 4 shows the data dependence graph obtained for the basic block in figure 3. Details of the algorithm for constructing the dependence graph are presented in section 4.

```
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. 5. 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.

Figure 5 shows the new sequence of target instructions obtained after instruction scheduling is performed on the MIR using the dependence graph in figure 4, and the algorithm outlined in section 6. 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 5, 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 3) to 25 cycles (when using the new ordering in figure 5). (Completion times were estimated assuming that the processor does not perform any out-of-order execution of instructions.)

Finally, figure 6 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

```
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. 6. Final MIR after merging `null_check` instructions in figure 5.

been successfully merged with load instructions. Note that the load instructions (`ppc_lwz`) are now marked as PEI's. No update is required to the exception handler tables when this merging is performed because the handler blocks are identical for all PEI's that belong to the same basic block.

4 Dependence Analysis

Traditionally, data dependence graphs represent dependences among register reads and writes, and memory reads and writes. Data dependences have been classified as *true* dependences (write to a location followed by a read of the same location), *anti* dependences (read of a location followed by a write of the same location) and, *output* dependences (write of a location followed by another write of the same location) [22]. Data dependences can be easily computed for (virtual or physical) registers, because registers are explicitly named as operands of instructions — there is no indirect addressing for registers. Computing data dependences for memory locations (object fields, array elements) is a harder problem, and is 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 the type of exception being thrown as *abstract locations*. We present a simple two-pass algorithm, given in Figure 7, 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 4.1 summarizes the abstract locations used in our approach. Section 4.2 contains a brief description of the dependence analysis algorithm in figure 7. Sections 4.3 to 4.7 outline how ALU instructions, memory instructions, exception-generating instructions, call instructions and synchronization instructions are modeled using abstract locations. Section 4.8 outlines the impact of Java's memory coherence assumption on dependence analysis. Section 4.9 outlines opportunities for more precise dependence analysis, but with the drawback of increasing the worst-case time complexity to beyond linear time.

4.1 Abstract Locations

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 data:

- Each field declaration has a unique 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. For example, fields `f1` and `f2` of class `LCPC` in figure 1 have distinct abstract locations, enabling references to `b.f1` and `b.f2` to be unordered.
- Each primitive array type (*i.e.,* `bool[]`^{N}, `short[]`^{N}, `int[]`^{N}, `long[]`^{N}, `float[]`^{N}, `double[]`^{N}, and `char[]`^{N}) has a unique abstract location modeling the concrete locations accessed by `aload` and `astore` instructions for an N-dimensional array of the corresponding type. An additional abstract location, `Object[]`^{N}, is used to represent all N-dimensional arrays of objects.
  These abstract locations are obtained from type-based aliasing; results of more extensive alias analysis are incorporated as further refinements/partitionings of these abstract locations.
- Each static data element (including static fields) has a distinct abstract location. (In the Jalapeño JVM, the abstract location is identified by a constant index into the "JTOC" table [2].)

In addition, there are three kinds of abstract locations that represent method-local data:

- Each symbolic register has a distinct abstract location.
- Each spill location has a distinct abstract location based on its constant offset in the stack frame. (This abstract location is only used when performing dependence analysis after register allocation.)

– There is a single abstract location, `exception_type`, that is used to model exception dependences. A PEI contains a pseudo-assignment of this abstract location with the value of its exception type.

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 4.2 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 sets will be constant in size, and the algorithm will take time linearly proportional to the number of instructions. More precise interprocedural information for `call` instructions can lead to larger `defs` and `uses` sets.

4.2 Linear-Time Dependence Analysis Algorithm

Figure 7 contains our algorithm for computing the dependence graph for a basic block in time and space that is linear in the size of the basic block *i.e.,* linear in the number of operands (and pseudo operands) across all instructions in the basic block.

The first pass of the algorithm traverses the instructions in the basic block in forward order. The key idea is to associate a "last definition" value, `last_def`, with each abstract location used in the basic 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 set to equal `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

```
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

    // create true dependence edges
    // 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

    // create output dependence edges
    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. 7. Algorithm to compute dependence graph of a basic block

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.,* [22]) 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 of extra edges created by using traditional dependence analysis algorithms.

4.3 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 traditional data dependence edges for registers. For example, the edge from node (4) to node (5) in figure 4 is a true dependence edge created for abstract location $\widehat{t11}$ (corresponding to symbolic register `t11`).

4.4 Memory Instructions

Memory instructions have abstract locations that represent the global data being accessed. The abstract locations are initially associated with HIR instructions based on the source bytecode operations, and are then carried forward as HIR memory instructions (`getfield`, `putfield`, `aload`, `astore`, etc.) get translated to MIR load and store instructions. This is important, because it is (in general) infeasible to recover the abstract location information from raw machine code. For example, the edge from node (5) to node (12) in figure 4 is an anti dependence edge created for abstract location `int[]` (corresponding to array `x`).

Our dependence analysis algorithm needs to know when two memory instructions can potentially reference common memory locations. Alias analysis is a rich research area, but fortunately it is separable from the dependence graph construction problem. Our algorithm assumes that the results of alias analysis can be expressed as refinements/partitionings of the abstract locations defined in section 4.1.

4.5 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 PEI's.

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 PEI's. Live variables analysis can compute the set of local variables that are live on entry to each handler (all global data would normally be considered to be live, by default). For example, instruction (4) in figure 4 writes to symbolic register `t11`, and can be freely moved before its preceding PEI's (instructions (1) and (3)) because the `uses` set for those PEI's do not contain any abstract locations for symbolic registers (since method `foo()` has no handler blocks).

In addition, we include in `p.defs`, a definition of the abstract location, `exception_type`. For example, the edge from node (3) to node (6) in figure 4 is an output dependence edge created for the abstract location, `exception_type` (denoted by E). This dependence edge ensures that the `bounds_check` for array `x` will be performed before the `null_check` for object `b`.

4.6 Call Instructions

The explicit operands of a call instruction identifies its arguments and result. In addition, pseudo-operands are introduced to represent abstract memory locations accessed by the call.

Wildcard summaries can be used for conveniently representing sets of abstract locations. For example, 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); this effect is modeled by a definition of an `all_globals` summary abstract location. (More precise sets of abstract locations could result from interprocedural side-effect analysis.) Wildcard summaries are restricted to cases for which the overlap with an abstract location can be tested very efficiently (in constant time).

Finally, for calls that can raise exceptions, the `defs` set should include the abstract location, `exception_type`.

4.7 Synchronization Instructions

The lock and unlock operations of a pair of monitor_enter and monitor_exit instructions define a critical section. A special synchronization abstract location is used to ensure that the original ordering of locak and unlock operations is preserved, similar to the ordering of PEIs. In addition, a load/store memory instruction is constrained to remain with its original critical section.

However, there are some additional dependences that must be enforced due to Java's memory model. When a lock is acquired (at a monitor_enter instruction), global memory is (conceptually) copied to local memory, enabling the local thread to see any remote updates. When a lock is released (at a monitor_exit instruction), local memory is copied back to global memory, enabling remote threads to see local updates. Thus the proper scheduling constraints on instructions around monitor_enter and _exit instructions can be achieved by treating each abstract location for a memory location as being in the `defs` set of a monitor_enter instruction and in the `uses` set of a monitor_exit instruction; these definitions and uses capture the effects of other concurrent threads. By exploiting an analysis of which objects or memory locations cannot be shared by other threads [6, 1], the set of used and defined abstract memory locations can be restricted to only the shared locations, enabling accesses to unshared locations to be moved across monitor boundaries.

4.8 Memory Coherence

The Java memory model enforces a strict memory coherence semantics for all accesses to the same location. 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 memory coherence assumption [15, 19]. 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, the dependence graph must contain an edge from the first `getfield` instruction to the next, so as to prevent this reordering.

Our solution is to provide two options when using the Jalapeño optimizing compiler:

1. *Strict consistency.* This is the default option for multithreaded programs; it endeavors to strictly obey the Java memory model. It treats each memory load instruction as a store, thus introducing more constraints in the dependence graph.
2. *Relaxed consistency.* This is the default option for single-threaded programs (as in the example in section 3). It can also be requested by the user for multithreaded programs. The relaxed consistency option supports a weaker consistency model that does not require memory coherence (*e.g.,* Location Consistency [14]), while still guaranteeing that properly synchronized (data-race-free) programs will have the same semantics as on the stricter Java memory model.

4.9 Beyond Linear-Time Dependence Analysis

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. More sophisticated pointer, alias, and array dependence analyses can also express their results in extensions of abstract locations, improving the flexibility of the constructed dependence graph still further. However, one has to be prepared for the worst-case time complexity to extend beyond linear time when using these more precise alias analysis algorithms.

Examples of these more precise analyses that we have designed thus far are:

- Exploitation of declared/discovered static type information for accesses to arrays of objects so as to obtain a more precise dependence graph. In this case, the `last_def` value for the abstract location corresponding to object arrays should be extended to a set of defining instructions, rather than a single defining instruction. Though this approach can contribute to a worst-case quadratic-time algorithm, we have designed a factored representation that will still guarantee a linear bound on space.
- It is sometimes possible for exception checks themselves to be reordered, if they are *indistinguishable* from the perspective of the exception handler. This can occur if both exception tests are for the same kind of exception (*e.g.,* both null pointer exceptions or both array bounds check exceptions), and they both report the exception at the same place in the program as determined by the resolution of the implementation's getStackTrace debugging native method, and the handler's live variables constraints discussed above are respected. Moreover, if the handler catches all exceptions and then ignores the exception object, then the exception itself isn't even live, and then all exceptions going to this handler are indistinguishable. In this case, the `last_def` value for the abstract location corresponding to exceptions should be extended to a set of multiple PEI's, which can also contribute to a worst-case quadratic-time algorithm.

5 Instruction Selection

This section briefly describes the instruction selection phase of the Jalapeño optimizing compiler, and how it uses dependence graphs. The job of instruction selection is to convert LIR (low-level IR) to MIR (machine-level IR). Our approach is to build a dependence graph for each basic block of LIR instructions, partition the dependence graph into trees, and apply a BURS-based tree-pattern-matching system [7] to perform instruction selection on each tree [4]. Trees are built from register-true dependencies edges in the dependence graph. Details of this approach can be found in [20].

Unlike previous approaches to partitioning DAGs of register-true dependence edges into trees (*e.g.,* [12]), our approach considers partitioning general dependence graphs with memory and exception dependences. We have defined legality constraints for this partitioning, and also developed a partitioning algorithm that incorporates code duplication.

Figure 8 shows a simple example of pattern matching for the PowerPC, and also illustrates our legality check for tree partitioning. Figure 8(a) shows three LIR instructions and its dependence graph. There is a register anti-dependence, shown in the figure as a dotted-arrow. The solid arrows form a single tree of register-true dependences that is used as the input to the BURS-based system. The output from tree-pattern-matching is shown in figure 8(b). LIR instructions S1 and S3 are "merged" into a single MIR instruction that performs an *andc* (and with complement) operation. LIR instruction S2 is simply translated to an equivalent MIR instruction.

Figure 8(c) shows the MIR-level dependence graph obtained by merging LIR instructions in the original dependence graph in figure 8(a). Note that this MIR-level dependence graph contains a cycle! Hence the original tree partitioning was illegal, and we have to break the cycle by "cutting" node S1, as shown in figure 8(d). The cut ensures that node S1 is first processed as a singleton tree, and then nodes S2 and S3 are processed as a separate tree. The output from tree-pattern-matching after this cut is shown in figure 8(e), and the corresponding MIR-level dependence graph is shown in figure 8(f). This MIR-level dependence graph does not contain a cycle, which indicates that the resulting instruction selection is legal.

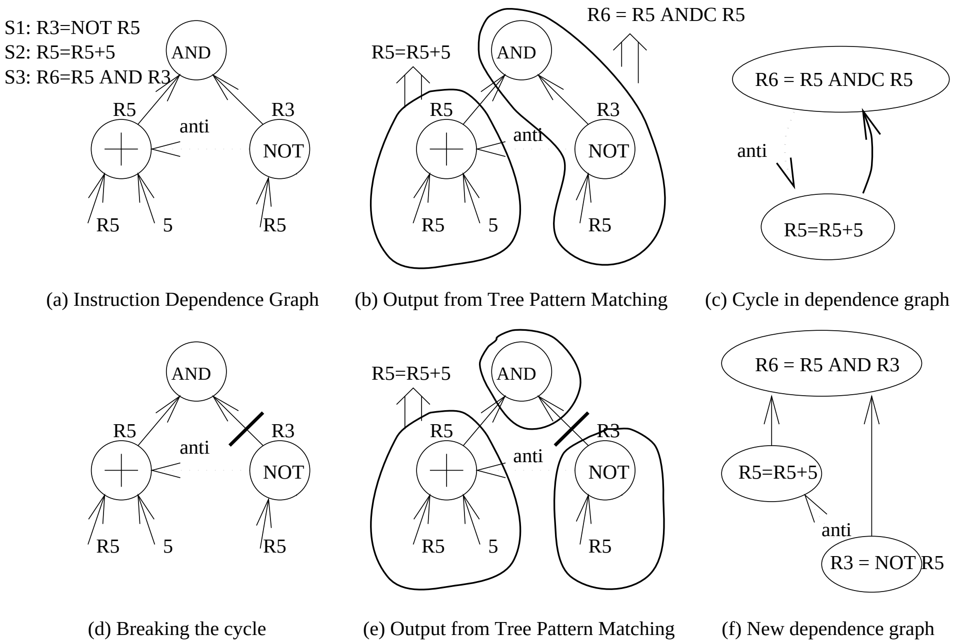

Fig. 8. Example of legality rule for tree pattern matching on dags for PowerPC

6 Instruction Scheduling

In this section, we briefly describe the instruction scheduling algorithm implemented in the Jalapeño optimizing compiler and its use of the dependence graph. The implementation of the algorithm is currently work-in-progress.

Machine Model. We specify the target machine as a collection of independent resources. The machine may have multiple distinct resource types, each of which can contain multiple identical resources. Instructions are grouped into instruction classes such that all instructions in the same class have the same resource usage pattern. A resource usage pattern is a list of resource reservations, each of which consists of a resource type and time range of usage relative to instruction issue time. When reserving a resource of a certain type, any resource of that type may be used, unless the instruction reserves *all* resources of that type.

The Jalapeño optimizing compiler currently targets the PowerPC architecture, in particular the PowerPC 604e processor. This processor contains the following types of functional units: FXU (fixed point unit), FPU (floating point unit), BRU (branch unit), LDST (load/store unit), and FXUC (complex fixed point unit). This processor contains multiple instances of the FXU. To model a multiple-issue machine, we added a pseudo-resource class ISSUE (issue slot).

Scheduling Algorithm. The scheduling algorithm used is a greedy list scheduler, with instructions arranged in a priority list. The construction of the priority list is deliberately left unspecified in the scheduling algorithm. The scheduler picks the next available instruction from the priority list, and finds the first slot in the schedule for which there is no resource conflict, starting with the instruction's earliest issue time. The dependence graph is used to compute earliest issue times for all instructions. The schedule is implemented as a 2-dimensional bitmap, so that bitwise operations can be used to detect resource conflicts. The algorithm is run on each basic block of a method.

Priority List Scheduling. The first priority function that we have implemented in the Jalapeño optimizing compiler is a simple topological numbering of the dependence graph. The idea is to use this priority function as a baseline to test the correctness of our implementations of dependence analysis and instruction scheduling. We are in the process of experimenting with a variety of priority functions *e.g.,* priority functions based on critical path lengths and slack times, and priority functions that incorporate register pressure [17].

7 Related Work

Dependence analysis for scalar variables has been well understood for a long time *e.g.,* def-use chaining [18] was an early technique for identifying true dependences within and across basic blocks. The same principle has been used to compute global anti and output dependences for scalar variables as well. Scalar renaming [8] is an effective technique for eliminating anti and output dependences, and SSA form is a popular approach to obtain a canonical renaming of scalars [9].

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.,*see [22]); 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 [21] is a notable exception). More recently, it has been observed [11] 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, [19] 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 4 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.,* [10]) 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 PEI's within the same basic block.

8 Conclusions and Future Work

In this paper, we addressed 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. We introduced new classes of edges in a dependence graph to model code motion constraints arising from these language features. We presented a linear-time algorithm for constructing this augmented dependence graph 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.

Some of the directions that we are pursuing as future extensions to this work are as follows:

- Improve the precision of the dependence information using the techniques outlined in Section 4.9.

- There are two kinds of ordering constraints for supporting garbage collection (GC) [16] that the Jalapeño optimizing compiler will need to enforce in the near future: 1) ordering constraints that ensure the integrity of interior pointers with respect to "GC points", and 2) ordering constraints that ensure the integrity of "write barriers" used to support generational GC algorithms.
- We are investigating the use of *escape analysis* [4, 6] to determine cases in which the synchronization object is known to be "thread-local". Identification of thread-local objects will help remove some of the dependence edges inserted due to the synchronization and memory model semantics of Java.
- To further reduce constraints in the dependence graph, we are exploring the possibility of allowing a guard instruction for an arithmetic exception (*i.e.,* 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 MIR by removing the condition register from the MIR instructions when the guard need not precede the compute instructions. This idea can be generalized to encompass other forms of speculation.

References

1. Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Eliminating unnecessary synchronization from java programs. In *to appear in Static Analysis Symposium (SAS'99)*, September 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 1999 Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999. Also available as INRIA report No. 0228, March 1999.
3. Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
4. 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.
5. Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs, 1999. To appear at PASTE '99.
6. Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for Java. 1999. To appear at ACM SIGPLAN OOPSLA'99.
7. R.R. Henry C.W. Fraser and T.A. Proebsting. Burg — fast optimal instruction selection and tree parsing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.
8. 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.
9. 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.
10. 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.
11. Amer Diwan, J. Eliot Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, October 1996.
12. M. Anton Ertl. Optimal code selection in DAGs. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1999.
13. J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
14. Guang R. Gao and Vivek Sarkar. Location Consistency: Stepping Beyond the Memory Coherence Barrier. *International Conference on Parallel Processing*, August 1995.
15. 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.
16. Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.
17. Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining Register Allocation and Instruction Scheduling (Technical Summary). Technical report, Courant Institute, New York University, July 1995. TR 698.
18. Steven S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.
19. William Pugh. Fixing the Java Memory Model. In *ACM Java Grande Conference*, June 1999.
20. Vivek Sarkar, Mauricio J. Serrano, and Barbara B. Simons. "Retargeting Optimized Code by Matching Tree Patterns in Directed Acyclic Graphs", Patent Application, submitted in December 1998.
21. 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.
22. 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.