Eliminating Exception Constraints of Java Programs for IA-64

Kazuaki Ishizaki, Tatsushi Inagaki, Hideaki Komatsu, Toshio Nakatani

IBM Research, Tokyo Research Laboratory

<http://www.trl.ibm.com/projects/jit/index_e.htm>
Goal of the Paper

- **Motivation**
  - Enable to perform code motion to exploit *instruction level parallelism* (ILP) of IA-64 for Java
  - Enable to perform only beneficial speculative code motion
- **Our approach** "*exception speculation*" using speculative code motion
  - Perform exception speculation on *directed acyclic graph* (DAG)
- **Experimental results**
- **Summary**
A Running Example

Java program and bytecode

Java program

```java
int foo(int a[], int i) {
    return a[i] + 1;
}
```

Bytecode

```
iaload
iconst_1
iadd
ireturn
```

PEI (Potentially Excepting Instruction) may throw a Java exception.
Intermediate Representation

Java language introduces many exception checks

Java program
```java
int foo(int a[], int i) {
    return a[i] + 1;
}
```

Intermediate Representation (IR)
```
nullcheck a
len v1=[a]
add v2=a,16
shiftl v3=i,2
boundcheck i<v1
ld v4=[v2+v3]
add v5=v4,1
ret v5
```

Bytecode
```
iaload
iconst_1
iadd
ireturn
```
Problems in Java

- An exception dependence between exception check and load suppresses code motion

Cannot move load instructions across exception checks

```
nullcheck a
len v1=[a]
add v2=a,16
shiftl v3=i,2
boundcheck i<v1
ld v4=[v2+v3]
add v5=v4,1
ret v5
```
An speculative load instruction allows dependant loads to issue a load earlier before the conditional branch is resolved.

Before performing control speculation:
- `ld t1= ...
- ...
- br.cond`
- `ld t1= ...
- ...
- br.cond`

After performing control speculation:
- `ld.s t1= ...
- ...
- br.cond`
- `chk.s t1 ...
- ...
- br.cond`

May defer a hardware exception.

Id moved across br.cond

Control Dependence

Check a deferred hardware exception.

Chk.s takes many cycles to redo the load if ld.s t1= defers an exception.
Our Approach - Exception Speculation

- Eliminate exception dependence edges from each load

Move load instructions across exception checks

```
len.s v1=[a]
nullcheck a
chk.s v1
add v2=a,16
shiftl v3=i,2
ld.s v4=[v2+v3]
boundcheck i<v1
chk.s v4
add v5=v4,1
ret v5
```
Why We Distinguish Between Control and Exception Speculation

- **Reduce the size of IR by not splitting basic blocks**
  - We do not handle exception dependence as control dependence.
  - In our experiments, # of basic blocks can be increased by a factor of four without using exception dependence edges.

- **Estimate the benefit of exception speculation along the exception dependence edge.**
  - The code can be moved speculatively only when it is beneficial on the DAG.
Where We Perform Exception Speculation

- Translate bytecode to IR
- Dataflow optimizations
- Build DAG
- Loop optimizations
- Exception speculation
- DAG scheduling
- Register allocation
- Code generation
- Native code
Algorithm Outline

1. Decide whether a load can be moved speculatively
   - When Delay(n) is set only by exception dependence, where n is an instruction.
     \[ \text{Delay}(n) = \max_{m \in \text{Pred}(n, \text{DAG})} \text{Delay}(m) + \text{Latency}(m) \]

2. Determine a speculative chain
   - Load and the succeeding instructions w/o side effect

3. Eliminate and connect exception dependence edges
   - Restructure a DAG to issue a load earlier.

4. Create dependence edges
   - Maintain edges to preserve the correctness
Our DAG for Exception Speculation

Before eliminating exception dependence edge

Java Program
int foo(int a[], int i) {
    return a[i] + 1;
}

Intermediate Representation
nullcheck  a
len  v1 = [a]
add  v2 = a, 16
shiftl  v3 = i, 2
boundcheck  i < v1
ld  v4 = [v2+v3]
add  v5 = v4, 1
ret  v5

DAG
nullcheck
len
boundcheck
add
shiftl
ld
add
ret
Data dependence
Exception dependence
Calculate the maximum possible delay to execute *load*

- Perform exception speculation if the time set by exception dependence is the slowest.

\[ 0 = 0 \]

non-beneficial
Decision to Perform Exception Speculation

- Calculate the maximum possible delay to execute `load`
  - Perform exception speculation if the time set by exception dependence is the slowest.

\[
2 + 0 > 1
\]

beneficial
Decision to Perform Exception Speculation

- Calculate the maximum possible delay to execute load
- Perform exception speculation if the time set by exception dependence is the slowest.

\[2 + 0 > 1\] beneficial
Determine Speculative Chain

Determine a chain of instructions that have no side effects as a speculative chain

Speculative chain

a=arg0
i=arg1
nullcheck
len
add
shiftl
boundcheck
ld
add
ret
Exception Dependence Edge

- Eliminate an edge from ld
- Decompose ld into ld.s and chk.s
- Connect an edge to a chk.s.

A. For executing nullcheck and boundcheck before executing recovery code
Dependence Edges

- **Create edges to chk.s**
  A. Source variables of chk.s
  B. Variables referenced in the recovery code

- **Create an edge from chk.s**
  C. To each instruction that uses any variable defined in the recovery code.
Experimental Results

- **Measurements for:**
  - Performance improvement
  - Code size expansion

- **Benchmarks**
  - Java Grande Benchmark Version 2.0
  - SPECjvm98

- **Environments**
  - IBM Developers Kit for IA-64, Java Technology Edition, 1.3
  - 2-way 800MHz Itanium with 2GB memory
  - Windows XP Advanced Server
Exception speculation is effective in programs with many array accesses.

Improve performance with an average of 2.0%.

Higher bars are better.
Code Size Expansion

- Increase code size with an average of 2.6%.

Smaller bars are better

Code expansion over no Exception Speculative

Series  LUFact  HeapSort  Crypt  FFT  SOR  SparseMatmul  compress  jess  db  javac  mpegaudio  mtrt  jack
Summary

- Propose a new solution "exception speculation"
  - Eliminate constraint of a load instruction by exception dependences on a DAG representation.
  - Perform speculative code motion based on cost-benefit analysis.
- Show the effectiveness using a production Java JIT compiler
Thanks !!

- Let's take a coffee break.

Just In Time...