Marmot: an Optimizing Compiler for Java

R.Fitzgerald, T.B.Knoblock, E.Ruf,

B. Steensgaard, D. Tarditi

Microsoft Research MSF-TR-99-33

June 1999

A Prolangs Overview - October 28, 1999

Marmot

- **Research compiler for large subset of Java**
  - **optimizing static native-code compiler**
    - **scalar optimizations as for Fortran**
    - **OO optimizations as static dispatching based on CHA**
  - **runtime system supports threads, synchronization and exceptions, garbage collection**
  - **implemented in Java**

Marmot

- **Claimed results**
  - **well-known optimizations can produce good performance comparable to other Java systems**
  - **reduces safety checks to 5-10% of execution time**
  - **generational garbage collection works, especially with bounded object lifetime analysis**
- **Multi-level IR conversion from Java to native x86 assembly code**

Marmot

Figure 1: The Marmot compilation process.

Java class files converted to JIR, conventional virtual register based intermediate form; presumes class files are verifiable.

Conversion to JIR - step 1

- Temporary-variable-based IR
  - bblocks are multiple exit and not terminated at function call boundaries
  - special exception edges used
    - labeled with class of exception, bound variable in handler, bblock to transfer control to if exception occurs
- Worklist algorithm converts all reachable classes
  - build temp variable model of stack operations
  - makes explicit some implicit byte code operations
    - e.g., class initialization

Conversion to JIR - step 2

- SSA conversion uses Lengauer/Tarjan dominator tree algm and Sreedhar/Gao phi placement algorithm
  - special exception edges complicate this process
  - phi nodes are eventually eliminated after high-level optimization is complete using copies

Conversion to JIR - step 3

- **Infers types from info implicit in byte code**
  - **Types of local vars and stack cells are unspecified**
  - **Values represented as small ints (e.g., booleans) are mixed in class files**
- **Produces strongly-typed IR, with all conversions explicit and all operator overloading resolved.**
  - **Per method type elaboration**
  - **Can recover some legal typing of the code, although may not be original typing**
  - **cf Gagnon/Hendren Sable algorithm**

Findings

- **Type elaboration can be expensive**
- **Some details of source (e.g., inner classes) are lost in byte code**
  - **Need source-level optimizations**
- **Need for cleanup transformations**
- **Claim get larger bblocks with their exception edges**
  - **Vortex approach: annotate each possible exception point with success and failure successor**

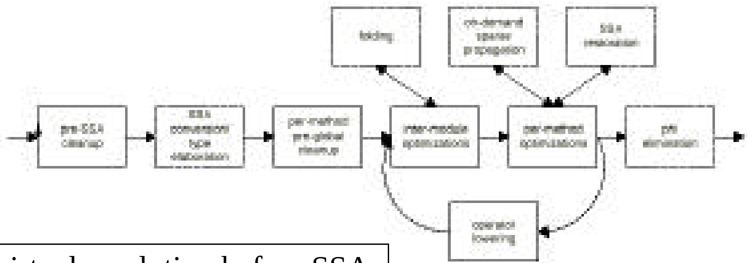
High-level Optimization

- Standard optimizations
  - cse and copy prop
  - dead-assignment/dead variable elimination
  - array bounds check optimization
  - control opts (e.g.,branch removal, unreachable code)
  - intermodule inlining
  - loop invariant code motion, strength reduction
- OO optimizations
  - reference null check removal
  - stack allocation of objects
  - redundant type test elimination
  - uninvoked method elimination

High-level Optimization

- Java optimizations
  - bytecode idiom recognition
  - redundancy elimination and loop-invar code motion of field and array loads
  - synchronization elimination

Phase Ordering



do virtual resolution before SSA;

inter-module: reresolve virtuals, inline, fold
inline when result of inlining is estimated
smaller than original

operator-lowering translates
high-level cast checks into JIR codes

Findings

- **Exceptions complicates the dataflow analysis**
  - **Implicit and explicitly thrown exceptions are problems**
  - **Limit code motion to provably effect-free non-throwing oprations (can't do PRE)**
- **SSA rep benefits analysis/transformation, but complicates transformation complexity**
  - **need to keep SSA graph up-to-date as transform**
- **Local type propagation dependent on their RTA info which may be too imprecise**

Code Generation

- JIR --> MIR, a low-level IR
- Cleanup of converted code
  - dead-code elimination, copy and constant propagation
- Register allocation performed
  - Chaitin/Briggs style allocator for 8 available regs
- Redundant jumps eliminated
- No instruction scheduling due to exceptions!

Runtime Support

- Written in Java
  - cast, array store, `instanceof` checks thread synchronization, interface call lookup
- Three garbage collectors tried
  - conservative, copying, generational (2)
- Libraries (specified as in 1.1)
  - use native code sparingly
    - 51K LOC of Java plus 11.5K LOC C++, 3K LOC of C++ headers, 2K LOC assembler

Performance Measurement

Name	LOC	Description
impcompress	2962	The IMPACT transcription of the SPEC95 compress_129 benchmark, compressing and decompressing large arrays of synthetic data.
impdes	561	The IMPACT benchmark DES encoding a large file
impgrep	551	The IMPACT transcription of the UNIX grep utility on a large file
impli	8864	The IMPACT transcription of the SPEC95 li_130 benchmark, on a sample lisp program
impcmp	200	The IMPACT benchmark cmp on two large files
imppi	171	The IMPACT benchmark computing π to 2048 digits
impwc	148	The IMPACT transcription of the UNIX wc utility on a large file
impsort	133	The IMPACT benchmark merge sort of a 1MB table
impsieve	64	The IMPACT benchmark prime-finding sieve

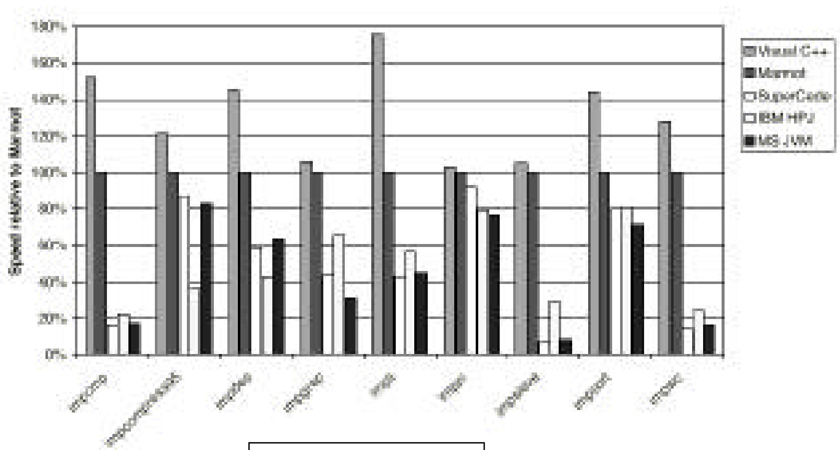
Figure 5: Small-to-medium benchmark programs that have implementations in both Java and C++.

Benchmark suite, mostly compiler benchmarks translated from C++ to Java by IMPACT/NET, and modified some by MS.

Comparisons

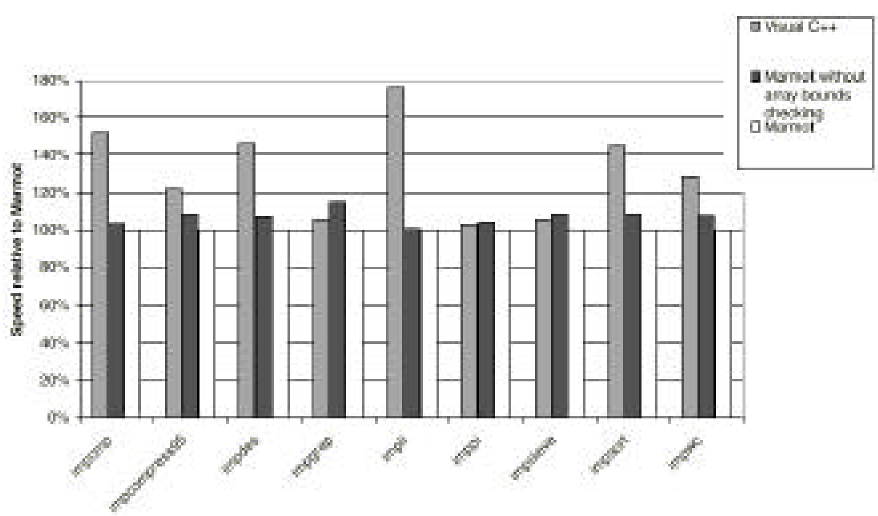
- **Compilers**
  - **JIT: MS Java VM**
  - **Commercial: SuperCede**
  - **Research: IBM HPJ (high performance compiler for Java)**
- **Used Pentium II-300 Mhz PC running Windows NT4.0, 512Mb memory**
  - **ran programs inside loops for timings**

Executed C++/Java Benchmark Speeds

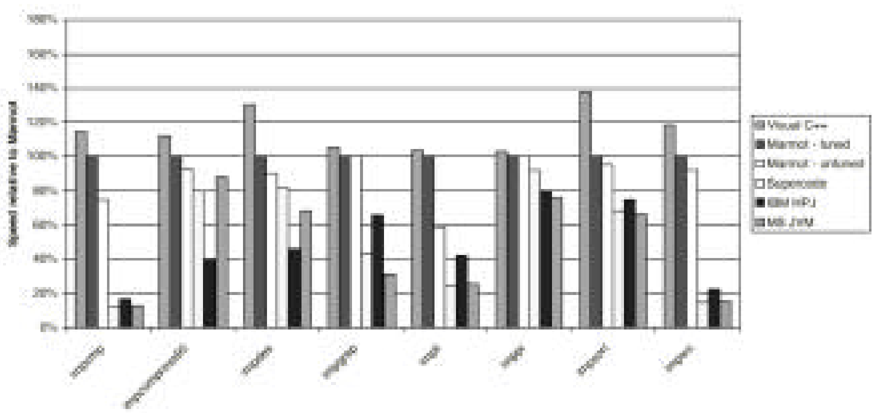


Marmot is 100%

Effect of Bounds Checks



Tuned Benchmarks



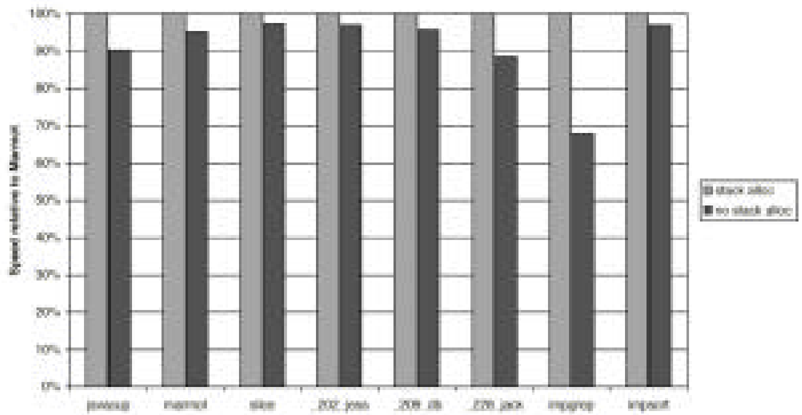
B.G. Ryder, Fall 1999 19

Findings

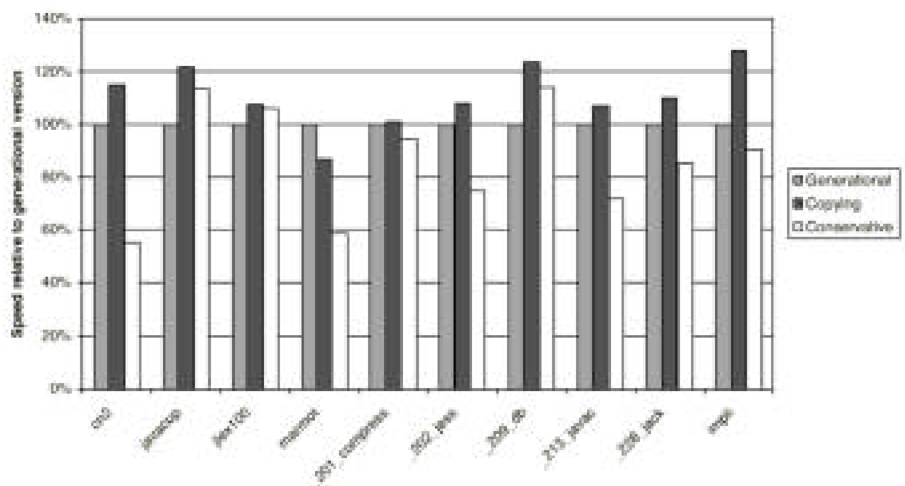
- Marmot compared well to Supercede, IBM HPJ, MS JVM in compiled code speed
- Combined cost of array store, null pointer, dynamic cast checks is insignificant (relative to running times with all checks on)
  - for 80% of programs is less than 10% of time
- Synchronization elimination has effects which are very program specific
- Stack allocation reduced execution time as much as 11%

B.G. Ryder, Fall 1999 20

Stack Allocation Effect



GC Choice



speed normalized on use of generational gc for each program; benchmarks run w/o safety checks

Conclusions

- Marmot: native-code compiler, runtime system, library for Java
- Focus: to create research platform, concentrating on extending known optimizations to Java
- Lessons
  - Java bytecode is inconvenient as an IR
  - Normal optimizations required extensions for exceptions, multi-threaded storage

Conclusions

  - SSA hard model for exceptions
  - Instruction scheduling hindered
- Achieved performance comparable to other Java systems and approaching C++
- Reduced cost of safety checks to about 4%
- Simple synchronization removal saved ~30% on larger benchmarks
- Storage management a real runtime cost
  - Stack allocation reduced time by <= 11%