Marmot: an Optimizing Compiler for Java

Robert Fitzgerald, Todd B. Knoblock, Erik Ruf,
Bjarne Steensgaard, and David Tarditi

Microsoft Research*
Draft of October 29, 1998.

Abstract

Performance optimizations for high level languages are best developed and evaluated in the context of an optimizing compiler and an efficient runtime system. To this end, we have constructed Marmot, a native compiler and runtime system for Java. Marmot is a complete system, using well-known scalar, object, and low-level optimizations, without reliance on external (fixed) compilers, back ends, runtime systems, or libraries. Initial performance results demonstrate that Marmot is competitive with existing Java systems, and suggests targets for future optimization research.

1 Introduction

Our long-term research focuses on the study of performance tradeoffs in the implementation of high-level programming languages. We are particularly concerned with implementation choices affecting the performance of large applications.

To this end, we have built Marmot, a bytecode-to-native-code compiler, runtime system, and library for Java. Marmot is

Current: Marmot implements both standard scalar optimizations (of the sort found in Fortran, C and C++ [Muc97]) and basic object-oriented optimizations such as call binding based on class hierarchy analysis [BS96, DGC95]. Modern representation techniques such as SSA and type-based compilation [CFR+89, Tar96] are also used. These improve optimizations and support tracing garbage collection.

Complete: Marmot implements all Java language features except dynamic class loading and reflection[1]. It supports a useful subset of the Java libraries: most of java.lang, java.util, java.io and java.awt, but not java.applet, java.beans, java.text, java.net, etc.

Flexible: Marmot implements all phases of compilation and runtime support except source-to-bytecode translation, with no reliance on outside back ends, garbage collectors, etc. This makes Marmot a good testbed for studying implementation techniques for high-level languages. Indeed, to support ease of modification, the Marmot compiler, libraries, and runtime system are implemented in Java wherever possible.

In this paper, we describe the architecture of the Marmot system in detail. We present preliminary performance measurements demonstrating that Marmot-generated executables are competitive with those produced by other research and commercial Java systems. We also compare Marmot with a C compiler via transliterated benchmark code.

To our knowledge, Marmot is the first project to systematically apply traditional compiler technologies to a native-code compiler built specifically for Java and to report the results. Other projects have relied on compilation to C or adaptation of existing backends for other languages. Implementors of commercial Java compilers have neither described their architectures in detail nor released performance results for even medium-sized programs (5K+ lines).

Marmot is intended primarily as a testbed for studying tradeoffs in high-level language implementation techniques. We chose to study Java because it has many features that are representative of high-level languages, such as object-orientation, strong typing, automatic storage management, multi-threading support, and exception handling. Java's popularity makes it likely that large applications will become available for empirical studies.

Marmot is current and complete to avoid skewing implementation studies. If aspects of the system are substandard, the relative value of additional implementation efforts may be over- or understated. For example, if basic, well-understood optimizations are not applied, then experimental optimizations may find opportunities

*Corresponding author: Todd Knoblock, Microsoft Research, Redmond, WA 98052, tel. (425) 936-4852, fax (425) 936-7329, email toddk@microsoft.com.

[1]These features are of secondary importance for our chosen domain of applications: large static applications. Reflection would be relatively simple to support. Dynamic class loading would require additional effort, principally to permit inter-class optimizations in the presence of changes to the class hierarchy.

for improvement which would have been eliminated anyway via a standard technique. Similarly, if the language semantics are simplified, one may erroneously find optimization opportunities which would not be legal under the full semantics. For example, Java's precise exception semantics limits code motion opportunities.

The remainder of the paper is organized as follows. Section 2 describes the architecture of Marmot, including the internal representations, compiler phases, runtime system, and libraries. Section 3 presents performance results. The paper concludes with a discussion of related work and directions for future research.

2 System Architecture

Marmot is organized as a sequence of transformations which operate on several different program representations (see Figure 1). The following sections describe the more important representations and optimizations following the order of their use.

2.1 Bytecode

Marmot takes verified Java bytecode [LY97] as input. Compiling a Java program begins with a class file containing the *main* method. This class is converted and all statically referenced classes in it are queued for processing. The conversion continues from the work queue until the transitive closure of all reachable classes have been converted.

Using bytecode has several advantages. The Java bytecode specification has been stable for the past year, whereas the Java language specification has continued to evolve. Compiling bytecode avoids the construction of a front end for Java (lexical analysis, parser, constant folder, and typechecker). Finally, it allows the system to compile programs where only the class files, and not the Java source files, are available.

By further restricting the input language to *verified* bytecode, Marmot is able to make a number of simplifying assumptions during processing which rely on the bytecode being well behaved. Since Java-to-bytecode compilers are supposed to generate verifiable bytecode, this is not a significant restriction for our purposes.

Bytecode is a high-level intermediate format that retains most of the information present in the source files, save formatting, comments, and some variable names and types. However, its irregular, redundant, stack-oriented nature makes it unattractive for use as an internal representation in an optimizing compiler. Marmot translates bytecode to a more conventional virtual register based intermediate form, reconstructing source-level type information that was omitted in the bytecode.

2.2 High-Level Intermediate Form

The Java intermediate representation, *JIR*, is the high-level intermediate form used in Marmot. It has the usual characteristics of a modern intermediate form: It is a temporary-variable based, static single assignment, 3-address representation. In addition, it is also strongly typed.

A method is represented as a control flow graph with a distinguished root (entry) block. Each graph node (basic block) consists of a sequence of *effect statements* and concludes with a *control statement*. An effect statement is either a *side effect statement* or an *assignment statement*. A side effect consists of an expression, and represents a statement that does not record the result of evaluating the expression. An expression consists of one of the following:

- A local variable, `v`.
- A constant, `c`.
- A type, `T`.
- A field reference, `v.f`.
- A static field reference, `f`.
- An array index expression, `v[i]`.
- A direct application of an operator to operands, $op(v_1, \ldots, v_n)$.
- An indirect (virtual) application of an operator to operands, $op(v_1, \ldots, v_n)$.

An assignment is comprised of two expressions, the *source* and *destination* of the assignment. The destination must be a variable, field, static field, or array index expression. All well formed statements are *linear*, meaning that they contain at most one of the following expression forms: field reference, array index, or function application.

Each basic block concludes with a control statement which specifies the succeeding basic block to execute under normal control flow. A control statement is one of the following forms:

`goto` Unconditional jump.

`if(v)` Conditional transfer

`return` Method exit (for void methods)

`return(v)` Method exit (for non-void methods)

`throw(v)` Raise exception.

`switch(v)`$[t_1, \ldots, t_n]$ Multiway transfer, $t_1, \ldots, t_n$ are the case tag constants.

2.2.1 Representing Exceptions and Handlers in JIR

In order to represent exception handlers, the basic blocks of JIR differ from the classic definition (e.g., [ASU86, App98, Muc97]) in that they are single entry, but multiple exit. In addition, basic blocks are not terminated at function call boundaries. If a statement causes an exception either directly, or via an uncaught exception in a function call, execution of the basic block terminates.

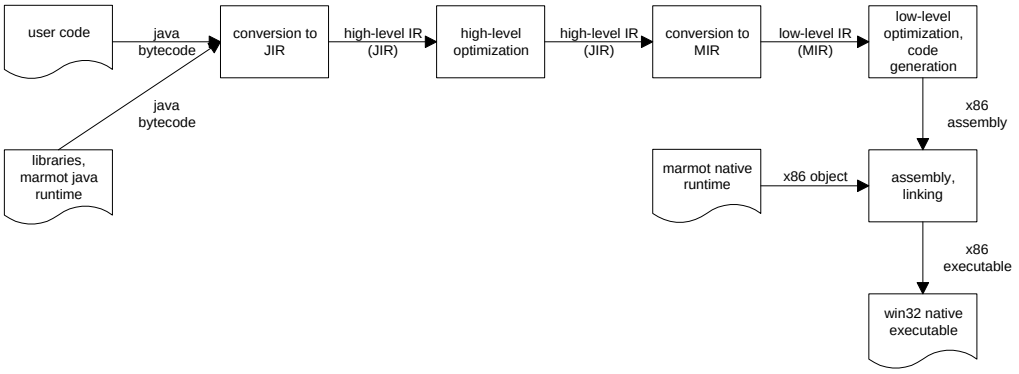

Figure 1: The Marmot compilation process.

JIR models Java exception handling by labeling basic blocks with distinguished exception edges. These edges indicate the class of the exceptions handled, the binding variable in the handler, and the basic block to transfer control to if that exception occurs durring execution of the guarded statements. The presence of an exception edge does not imply that the block will throw such an exception under some execution path.

The intuitive dynamic semantics of basic block execution are as follows. Execution proceeds sequentially through the statements unless an exception is raised. If an exception is encountered, then the ordered list of exception edges for the current basic block is searched to determine how to handle the exception. The first exception edge $e : E$ such that the class E matches the class of the raised exception is selected. The exception value is bound to the binding variable, e, and control is transferred to the destination of the exception edge. If no matching exception edge exists, the current method is exited, and the process repeats recursively.

Exception edges leaving a basic block differ semantically from the normal edges. While a normal edge indicates the control destination once execution has reached the end of the basic block, an exception edge indicates that control may leave the basic block anywhere in the block, e.g., before the first statement, or in the middle of executing a statement. This distinction is especially important while traversing edges backwards. While normal edges may be considered to have a single source (the very end of the source block), exception edges have multiple sources (all statements in the source basic block which might throw an instance of the edge's designated exception class). In this sense, JIR basic blocks are similar to superblocks [CMCH91].

The conversion of bytecode to JIR proceeds in three steps:

1. Initial conversion to a temporary-variable based intermediate representation.
2. Conversion to static single assignment form.
3. Type elaboration.

The next three sections describe these steps.

2.3 Initial Conversion

The initial conversion from bytecode to JIR uses an abstract interpretation algorithm [CC77]. An abstraction of the bytecode VM stack is built and the effects of the bytecode execution stream are modeled. A temporary variable is associated with each stack depth, $temp_0$ for the bottom-of-stack value, $temp_1$ for depth 1, and so forth.

Bytecode instructions that manipulate the stack are converted into effects on the stack model along with JIR instructions that manipulate temporary variables. For example, given an abstract stack of depth 28 with top two values `x` and `y`, the bytecode `iadd` is translated to

```
temp_27 = x + y
```

and a new stack model with `x` and `y` popped, and `temp_27` as the new top-of-stack value. The assumption that the source bytecode is verifiable assures that all abstract stack models for a given control point will be compatible.

Some care must be exercised in modeling the multiword long and double bytecode operations, which are represented as multiple stack elements in the bytecode. Conversion reassembles these split values into references to multiword constants and values. Once again, verifiability of the bytecode ensures that this is possible.

Some simplification and regularization of the bytecode occurs during the conversion. For example, the various `if_icmp<cond>`, `if_acmp<cond>`, `ifnull`, and `ifnonnull` operations are all translated to JIR `if` control statements with an appropriate boolean test variable. Similarly, the various `iload_n`, `aload_n`, `istore_n`, `astore_n`, etc., are translated as simple references to local variables. Where possible, `fcmp`, `dcmp`, and `lcmp` are translated to simpler boolean comparisons. Reducing the number of primitives in this way simplifies subsequent processing of JIR.

The initial conversion makes manifest some computations that are implicit in bytecode. This include operations for class initialization and synchronized meth-

ods. This lowering to explicitly represented operations is done to make the operations available for further analysis and optimization.

2.4 Static Single Assignment Conversion

The second step of converting from bytecode to JIR is conversion to static single assignment (SSA) form [CFR+89, CFRW91]. The conversion is based upon Lengauer and Tarjan's dominator tree algorithm [LT79] and Sreedhar and Gao's phi placement algorithm [SG95]. Conversion is complicated by the presence of exception-handling edges, which must be considered during the computation of iterated dominance frontiers. Such edges may also require that their source blocks be split to preserve the usual one-to-one correspondence between phi arguments and CFG edges.

After high level optimization is complete, the phi nodes are eliminated before translation to the low-level intermediate form. Phi elimination is implemented using a straightforward copy introduction strategy. The algorithm uses edge splitting to limit the scopes of copies, but does not attempt to reuse temporaries; that optimization is subsumed by the coalescing register allocator.

2.5 Type Elaboration

The third and final step in constructing JIR from bytecode is *type elaboration*. This pass infers type information left implicit in the bytecode, and produces a strongly-typed intermediate representation in which all variables are typed, all coercions and conversions are explicit, and all overloading of operators is resolved.

Java source programs include complete static type information, but some of this information is not included in the bytecode:

- Local variables do not have type information.
- Stack cells are untyped (and so are the corresponding temporaries in JIR at this stage).
- Values represented as small integers (booleans, bytes, shorts, chars, and integers) are mixed within bytecode methods.

The bytecode does preserve some type information, namely:

- All class fields contain representations of the originally declared type.
- All function formals have types.
- Verified bytecode implies certain internal consistency in the use of types for locals and stack temporaries.

Although it is not possible in all case to recover the user's typings, type elaboration can always recover *a* legal type elaboration of the intermediate code.

- Standard optimizations
  - Array bounds check optimization
  - Common subexpression elimination
  - Constant propagation and folding with conditional branch elimination
  - Copy propagation
  - Dead-assignment and dead-variable elimination
  - Inlining
  - Loop invariant code motion, including loop rotation
  - Loop induction variable elimination and strength reduction
  - Unconditional jump elimination
  - Unreachable block elimination
- Object-oriented optimizations
  - Stack allocation of objects
  - Static method binding
  - Type test and cast elimination
  - Uninvoked method elimination and abstraction
- Java-specific optimizations
  - Array store check elimination
  - Cmp operator elimination
  - Synchronization elimination

Figure 2: Optimizations performed by the high-level optimizer.

In addition to its usefulness in analysis, garbage collection, and representation decisions, the type information serves as a consistency check for JIR. All optimizations on the JIR are expected to preserve correct typing, allowing *type checking* to be used as a pre- and post-optimization semantic check. This is useful in debugging Marmot.

2.6 High-Level Optimization

The high-level optimizer implements the optimizations shown in Figure 2.

The high-level optimizer transforms the intermediate representation (JIR) while preserving its static single assignment and type-correctness properties. Figure 2 lists the transformations performed. These can be grouped into three categories: (1) scalar optimizations for imperative languages, (2) general object-oriented optimizations, and (3) Java-specific optimizations.

2.6.1 Standard Optimizations

The standard scalar and control-flow optimizations are performed on a per-method basis using well-understood intraprocedural dataflow techniques. Some analyses (e.g., induction variable classification) make extensive use of the SSA use-def relation, while others (e.g., availability analysis) use standard bit-vector techniques.

The remainder of this section presents a summary of our experience implementing these optimizations under an SSA-based representation in the face of Java's ubiquitous exception handlers.

The presence of exceptions complicates dataflow analysis because the analysis must model the potential transfer of control from each (implicitly or explicitly) throwing operation to the appropriate exception handler. The JIR representation models these transfers coarsely via outgoing exception edges on each extended basic block. Marmot bit-vector analyses achieve greater precision by modeling the exception behavior of each operation in each basic block, and using this information to build edge-specific transfer functions relating the block entry to each exception arc. The SSA-based analyses do not need to perform this explicit modeling, as the value flow resulting from exceptional exits from a block is explicitly represented by phi expressions in the relevant handler blocks.[2]

Java's precise exception model further complicates transformation by requiring that handlers have access to the exact observable program state that existed immediately prior to the throwing operation. Thus, not only is the optimizer forbidden to relocate an operation outside of the scope of any applicable handlers (as in most languages), it is also unable to move a potentially throwing operation past changes to any local variable or storage location live on entry to an applicable handler without building a specialized handler with appropriate fixup code.[3] The present optimizer limits code motion to effect-free, non-throwing operations. Also because of this, Marmot implements only full redundancy elimination, and not partial redundancy elimination, in our base system; fully redundant throwing expressions may be removed without such complicated code motion. Good partial redundancy elimination is likely to require dynamic information to justify the extra analysis effort and code expansion for specific code paths.

While the explicit SSA representation benefits both analysis (e.g., flow-sensitive modeling without the use of per-statement value tuples) and transformation (e.g., transparent extension of value lifetimes), it significantly increases the implementation complexity of many transformations. The main difficulty lies in appropriately creating and modifying phi operations to preserve SSA invariants as the edge structure of the control flow graph changes.[4] This transformation-specific phi-operator maintenance is often the most difficult implementation task (and largest compilation-time cost) in Marmot optimizations.

Briggs et al. [BCHS88] noted that systems treating all phi operations in a basic block as parallel assignments may require a scheduling pass to properly serialize these assignments during the phi elimination phase. The Marmot optimizer avoids this issue by giving phi-assignments the normal sequential semantics of statements, including the ability of a loop-carried definition to kill itself. This requires some extra care in copy propagation but does not affect other analyses, and has the benefit of simplifying the phi elimination phase of SSA.

Unlike most of the optimizations shown in Figure 2, array bounds check optimization is not yet a standard technique, particularly in the face of Java's exception semantics. Several techniques for array bounds check optimization have been developed for Fortran [MCM82, Gup90, Asu91, Gup93, CG95, KW95] and other contexts [SI77, XP98].

Marmot employs the folklore optimization that the upper and lower bounds checks for a zero-origin array may be combined into a single unsigned check for the upper bound [App98]. Also, the common subexpression elimination optimization removes fully redundant checks. The remaining bounds checks are optimized in two phases.

First, the available inequality facts relating locals and constants in a method are collected using a dataflow analysis. Sources of facts are control flow branching, array creation, and available array bounds checks. To these facts are added additional facts derived from bounds and monotonicity of induction variables.

Second, an inequality decision procedure, Fourier elimination [DE73, Duf74, Wil76], is used at each array bounds check to determine if the check is redundant relative to the available facts. If both the lower- and upper-bound checks are redundant, then the bounds check is removed. See Knoblock [Kno98] for additional information on array bounds check optimizations in Marmot.

2.6.2 Object-Oriented Optimizations

Marmot's object-oriented optimizations are implemented using a combination of inter-module flow-insensitive and per-method flow-sensitive techniques.

The instantiation and invocation analysis, *IIA,* simultaneously computes conservative approximations of the sets of instantiated classes and invoked methods. Given an initial set of classes and methods known to be instantiated and invoked, it explores all methods reachable from the call sites in the invoked method set (subject to the constraint that the method's class be instantiated), adding more methods as more constructor invocations are discovered. This is similar to the Rapid Type Analysis algorithm of Bacon [Bac97], except that IIA does not rely on a precomputed call graph, eliminating the need for an explicit Class Hierarchy Analysis [DGC95] pass. We use an explicit annotation mech-

[2] Due to the invariant that a basic block may not contain multiple SSA variable definitions reaching a single phi node in a handler block (c.f. Section 2.4), it is always possible to precisely represent the statement-level exception information at the basic block level.

[3] Performing such code motion requires a sophisticated effect analysis and may require significant code duplication; e.g., if the throwing operation is moved out of a loop.

[4] Using SSA as our sole representation denies us the standard option of simply reanalyzing the *base* representation to reconstitute the SSA hypergraph. After initial conversion, that base no longer exists. Since converting out of and back into SSA form on each CFG edit would be prohibitively expensive, we are forced to write custom phi-maintenance code for each CFG transformation. Choi et al. [CSS96] describe phi maintenance for several loop transformations, but do not give a solution for general CFG mutation.

anism to document the invocation and instantiation behavior of native methods in our library code.

Marmot uses the results of this analysis in several ways. A *treeshake* transformation rebinds virtual calls having a single invoked target and removes or abstracts uninvoked methods.[5] This not only removes indirection from the program, but also significantly reduces compilation times (e.g., many library methods are uninvoked and thus do not require further compilation). Other analyses use the IIA to bound the runtime types of reference values, or to build call graphs. For example, the inliner may use this analysis to inline all methods having only one call site.

Local type propagation computes flow-sensitive estimates of the runtime types (e.g., sets of classes and interfaces) carried by each local variable in a method, and uses the results to bind virtual calls and fold type predicates. It relies upon the flow-insensitive analysis to bound the values of formals and call results but takes advantage of local information derived from object instantiation and type casting operations to produce a more precise, program-point-specific, result. This information allows the optimizer to fold type-related operations (e.g., cast checks and `instanceof`), as well as statically binding more method invocations than the flow-insensitive analysis could alone. Type operations not folded by the type propagator may still be eliminated later by other passes.

The stack allocation optimization improves locality and reduces garbage collection overhead by allocating objects with bounded lifetimes on the stack rather than on the heap. It uses an inter-module escape analysis to associate object allocation sites with method bodies whose lifetime bounds that of the allocated object. The allocation is then moved up the call graph into the lifetime-dominating method, while a storage pointer is passed downward so that the object can be initialized at its original allocation site. See Gay and Steensgaard [GS98] for details of this optimization.

2.6.3 Java-Specific Optimizations

To date, work on Marmot has concentrated on efforts to implement fairly standard scalar and object-oriented optimizations in the context of Java; thus, the present version of the optimizer contains relatively few transformations specific to the Java programming language.

Java's covariant array subtyping rule requires that writes to arrays of reference types must be checked at runtime. The intermediate representation makes these checks explicit so they can be optimized independently of the array assignments (i.e., we can remove checks while still performing the store to the array). Marmot eliminates such checks in two ways: (1) by eliminating fully redundant checks in the CSE pass, and (2) by removing checks when the local type propagator can prove that the value being stored cannot be of a more general runtime type than the array's runtime element type. In isolation, the latter technique is not very effective due to the imprecision of the IIA (e.g., `java.lang.Object[]` could conceivably denote any instantiated reference array type). Marmot addresses this limitation by adding a simple global analysis (similar to type-based alias analysis [DMM98]) which computes a flow-insensitive alias relation on instantiated array types. The resulting improvement in precision allows the removal of additional array store checks.

Because Java programs may execute multiple threads simultaneously, many of the methods in the standard library guard potential critical sections with synchronization operations. The cost of these operations is then paid by multi-threaded and single-threaded programs alike. Marmot optimizes the single-threaded case by using the IIA to detect that no thread objects are started, allowing it to remove all synchronization operations from the program before further optimizations are performed. Similar analyses appear in Bacon [Bac97] and Muller et al. [MMBC97].

2.6.4 Phase Ordering

Figure 3 shows how the optimizations are organized. We briefly describe the phases here.

Because SSA conversion and type elaboration are relatively expensive, it is profitable to run the treeshake optimization to remove unused methods from the representation prior to conversion. This pass also detects single-threaded code and removes synchronization operations if possible. This is the only high-level optimization pass which operates on an untyped, non-SSA representation.

Before performing inter-module optimizations, Marmot applies a suite of simple optimizations on each method. These optimizations remove significant numbers of intermediate variables, unconditional jumps, and redundant coercions introduced by the bytecode translation and type elaboration phases. Doing so reduces code size, speeding later optimizations, and also improves the accuracy of the inliner's code size heuristics.

The inter-module optimization phase runs the treeshake pass again, followed by multiple passes of inlining, alternating with various folding optimizations. Stack allocation invokes the inliner on individual call sites where the transformed target is known to be small. Finally, a variety of optimizations are applied to each method. Some optimizations avoid the need for re-folding entire methods by performing on-demand value propagation and folding on the SSA hypergraph. Others momentarily transform the representation in violation of the SSA invariants, and rely on *SSA restoration* utilities to reestablish these invariants.

2.7 Low-Level Intermediate Form

The low-level intermediate form, *MIR*, is in most ways a conventional low-level intermediate representation. MIR shares the control-flow graph and basic block representations of JIR, enabling reuse of algorithms that

[5] Method abstraction is required when a method's body is uninvoked but its selector continues to appear in virtual calls. We do not prune the class hierarchy, as uninstantiated classes may still hold invoked methods or be used in runtime type tests.

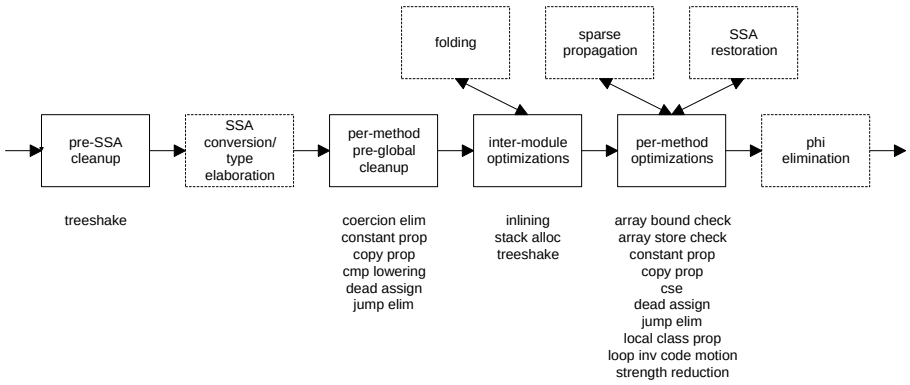

Figure 3: Optimization phases. The boxes with solid outlines are optimization phases; while those with dashed outlines are utilities invoked by these phases.

operate on the CFG.

The instruction set of MIR includes a subset of the instruction set of the Intel x86 processor family, which is the current target architecture of Marmot. There are six distinct kinds of low-level MIR instructions: data instructions with no operands, data instructions with one operand, data instructions with two operands, machine-level function calls (the 80x86 `call` instruction), control instructions with no operands, control instructions with one operand. All machine-level function calls are annotated with their register use-def information. Control instructions may optionally be annotated with register use-def information, allowing MIR to encode use-def information for non-local transfers of control (such as function return or throwing an exception). The instruction set of MIR also includes high-level instructions for function call, return, and throwing exceptions; these are replaced by actual machine instructions during register allocation. This simplifies the translation from JIR to MIR. It also allows decisions about interprocedural register usage to be deferred until the register allocation phase, which will make it easier to incorporate interprocedural register allocation techniques into Marmot.

The operands of MIR are registers, constants (integers or symbolic addresses), effective addresses, register pairs (which are used to represent 64-bit integers when these integers are passed or returned from a function), and floating-point register stack locations (x86 floating point instructions use a register stack).

MIR differs from conventional low-level intermediate formats by keeping *representation information* about each operand. Representation information is a simplified version of type information that distinguishes between:

- traceable pointers,
- pointers to code,
- pointers to static data,
- interior pointers (that point to locations within an object)
- 8, 16, 32, and 64-bit signed integers, and 16-bit unsigned integers
- 32 and 64-bit floating point numbers

Traceable pointers point to the base of an object, while interior pointers point into the middle of an object. Tracking interior pointers is particularly useful because garbage collection implementations often do not handle them. For these implementations, it may be verified that code generation never produces code where interior pointers are live at program points where garbage collection may occur without a corresponding traceable pointer.

2.8 Conversion to MIR

To convert JIR to MIR, Marmot first determines explicit data representations and constructs meta-data. It then converts converts each method using the data representations chosen during the first phase.

2.8.1 Constructing Meta-Data

Marmot implements all meta-data, including virtual function tables (vtables) and `java.lang.Class` instances, as ordinary Java objects. These classes are defined in the Marmot libraries and their data layouts are determined by the same means used for all other classes. Once the meta-class layout has been determined, MIR conversion is able to statically construct the required meta-data instances for all class and array types.

2.8.2 Converting Methods to MIR

For each JIR basic block in the CFG and updates the JIR basic block

Marmot converts methods to MIR procedures using syntax-directed translation [ASU86]. For each JIR block in the JIR CFG, it creates a corresponding MIR block, then translates each statement in the JIR block to one or more MIR statements in the MIR block. Most

JIR statements map to two or more MIR instructions; a one-to-one mapping cannot be created because most MIR instructions have one fewer addresses than their corresponding JIR statements. For example, consider the translation of a three-address statement such as `a = b op c`, where `a`, `b`, and `c` are local variables. Marmot translates this to MIR as

```
mov tmp,b
op tmp,c
mov a,tmp
```

where `tmp` is a new pseudo-register and `mov tmp,c` assigns the value of `c` to `tmp`. The translation of two-address JIR statements similarly introduces a temporary pseudo-register. We rely on the register coalescing phase of the register allocator to remove unnecessary moves and temporaries.

The translation of JIR statements to MIR instructions is for the most part routine. Some aspects of the translation deserve further comment:

- Run-time type operations. JIR provides several operations that use run-time type information: `checkcast`, `instanceof`, and `checkarraystore`. We have implemented all these operations as Java functions that use run-time type information stored in VTable instances. Marmot translates each operation as a call to the appropriate Java function.

- Exception handlers. JIR exception handling blocks may have two kinds of edges coming into them: exception edges and normal edges. Exception edges assign a variable, while normal edges represent just transfer of control. The conversion to MIR splits every block that is the target of an exception edge into two blocks. The first block contains code which implements the variable assignment and the second block contains code corresponding to the original JIR block. The first block jumps to the second block. The conversion redirects all normals arcs to the original JIR block to the second block. It redirects all exception arcs to the first block.

- Switch statements. Marmot converts dense switches to jump tables and small or sparse switches to a chain of conditional branches. A switch is defined to be dense if it contains cases for more than half of the integers between the smallest and largest integer cases of the switch. A switch is small if there are three or fewer cases for the switch.

- Field references. Marmot maps field references to effective addresses which are then used as operands in instructions. It does not assume a RISC-like instruction set where load and store instructions must be used for memory accesses.

- Long integer operations. The x86 architecture does not support 64-bit integers natively. Marmot translates 64-bit integer operations to appropriate sequences of 32-bit integer operations. It places all these sequences inline except for 64-bit multiplication and division, for which it generates calls to runtime functions.

  Marmot maps 64-bit JIR variables to pairs of 32-bit pseudo-registers. Most uses of these pairs disappear, of course, as part of the above translation, but the pairs do appear in MIR functions in the high-level call and return instructions and formal argument lists. The register allocator eliminates these occurrences.

- Long comparisons: `fmpg`, `fcmpl`, `lcmp`. The conversion to JIR eliminates most occurrences of these operators, replacing them with simpler boolean comparison operations. Marmot generates inline code for the remaining occurrences of each operation.

2.9 Low-Level Optimization

After Marmot converts a JIR method to an MIR method, it performs several low-level optimizations. First, it eliminates unnecessary assignments to boolean variables that are used immediately in conditional branches. These assignments arise because JIR conditional statements only take boolean variables as arguments; they do not incorporate comparisons. For example, Marmot translates `if (a>b) ...` to JIR of the form `t = a>b; if (t) ...`. Marmot translates this JIR to MIR of the form `t = a>b` to `cmp a,b; setg t` and `if (t) ...` to `test t,t; jne ...`. Marmot improves the whole code sequence by looking for code of the form `setcc t; test t,t; jne ...` and changing it to `setcc t; jcc ...`, where `cc` represent the appropriate comparison condition code. If the `setcc` turns out to be unnecessary, it will be eliminated by dead-code elimination.

Second, Marmot replaces unconditional jumps to control instructions and conditional branches to unconditional jumps. Third, it eliminates dead and unreachable code. Finally, it does peephole optimizations. Marmot does not yet do instruction scheduling.

2.10 Register Allocation

Marmot uses graph-coloring register allocation in the style of Chaitin [CAC+81, Cha82], incorporating improvements to the coloring process suggested by Briggs et al. [BCT94]. The allocator has five phases:

1. The first phase eliminates high-level procedure calls, returns, and throws. It does this by introducing appropriate low-level control transfer instructions and making parameter passing and value return explicit as moves between physical locations and pseudo-registers.

2. The second phase eliminates unnecessary register moves by coalescing pseudo-registers. It coalesces registers aggressively and does not use the more conservative heuristics suggested by [BCT94,

GA96]. The phase rewrites the intermediate form after each pass of coalescing and iterates until no register coalesces occur.

3. The third phase, which is performed lazily, estimates the cost of spilling each pseudo-register. It sums all occurrences of each pseudo-register, weighting each occurrence of a register by 10^n, where n is the loop-nesting depth of that occurrence.

4. The fourth phase attempts to find a coloring using optimistic coloring [BCT94]. If at some point coloring stops because no colors are available (and hence a register must be spilled), the phase removes the pseudo-register with the lowest spilling cost from the interference graph and continues coloring. If the phase colors all registers successfully, it applies the mapping to the program and register allocation is finished.

5. The fifth phase, which inserts spill code, is especially important because the target architecture has so few registers. The phase creates a new temporary pseudo-register for each individual occurrence of a spilled register and inserts load and store instructions as necessary. It attempts to optimize reloads from memory: if there are several uses of a spilled register within a basic block; it will use the same temporary register several times and introduce only one load of the spilled register[6]. If this optimization does not apply, this phase attempts to replace the spilled register with its stack location. Doing so avoids using a temporary register and makes the program more compact by eliminating explicit load and store instructions.

After the fifth phase completes, register allocation returns to the fourth phase and tries to color the new intermediate program again. This process iterates until all registers are successfully colored.

Tracing garbage collection requires that the runtime system accurately find all memory locations outside the heap that contain pointers into the heap. To support this, each function call is annotated with the set of stack locations that contain pointers and are live across the call. These sets are empty at the beginning of register allocation and are updated during the introduction of spill code. For each pointer-containing register that is live across a function call and is being spilled, the corresponding stack location is added to to the set for the function call.

2.11 Runtime Support

The majority of the runtime system code is written in Java, both for convenience and to provide a large, complex test case for the Marmot compiler. Operations including cast, array store and instanceof checks, `java.lang.System.arraycopy()`, thread synchronization and interface call lookup are implemented in Java.

[6]See [Bri92] for a detailed description of when this can be done.

2.11.1 Data Layout

Every object has a *vtable* pointer and a *monitor* pointer as its first two fields. The remaining fields contain the object's instance variables, except for arrays, where they contain the length field and array contents.

The vtable pointer points to a `VTable` object that contains a virtual function table and other per-class metadata. These include a `java.lang.Class` instance, fields used in the implementation of interface calls (see Section 2.11.2), and size and pointer tracking information describing instances of the associated class or array types.

The monitor pointer points to a lazily-created extension object containing infrequently-used parts of the per-instance object state. The most prominent is synchronization state for synchronized statements and methods and for the `wait()` and `notify()` methods of `java.lang.Object`. It also incorporates a hash-code used by `java.lang.Object.hashCode()`. Bacon et al. [BKMS98] describes a similar scheme to reduce space overhead due to synchronization.

2.11.2 Interfaces

Marmot implements interface dispatch via a per-class data structure called an interface table, or *itable*. A class's vtable contains one itable for each interface the class implements. Each itable maps the interface's method identifiers to the corresponding method entry points. The vtable also contains a mapping from the `Class` instance for each interface to the position of its corresponding itable within the vtable. Itables are shared where possible.

Invoking an interface method consists of calling a runtime lookup function with the `Class` instance for the interface as an argument. This function uses the interface-to-itable mapping to find the offset for the itable within the vtable. It then jumps through the itable to the desired method.

Marmot saves space by sharing itables. If an interface I has direct superinterfaces S_1, S_2 and so on, it positions the itable for S_1 followed by the itable for S_2, and so on. Any method m declared in I that is declared in a superinterface can be given a slot of m from a superinterface.[7] All new methods declared in I can be placed after all the itables for the direct superinterfaces of I.

2.11.3 Exceptions

Marmot uses the same kind of program-counter-based exception handling mechanism that Java bytecode [LY97] uses. Memory containing Marmot-generated machine code is divided into ranges, each of which is associated with a list of exception handlers. When an exception occurs, the runtime system finds the range containing the throwing program point and finds the appropriate handler in the list.

[7]Note that more than one direct superinterface of I may declare a method m, so the itable for I may have multiple slots for m.

Marmot implements stack unwinding by creating a special exception handler for each function body. Each such handler catches all exceptions. When it is invoked, it pops the stack frame for the function and rethrows the exception. Thus, no special-case code is needed in the runtime system to unwind the call stack when an exception occurs.

Marmot does not add special checks for null pointer dereferences or integer divide-by-zero. Instead, it catches the corresponding operating system exception and throws the appropriate Java exception.

2.11.4 Threads and Synchronization

Each Java thread is implemented by a native (Win32) thread. Monitors and semaphores are implemented using Java objects which are updated in native critical sections. The mapping from native threads to `java.lang.Thread` objects uses Win32 thread-local storage.

2.11.5 Garbage Collection

At present, Marmot offers a choice of two garbage collection schemes: a conservative collector and a copying collector. The conservative collector uses a mark-and-sweep technique, while the copying collector is a semi-space collector using a Cheney scan. We are working on making the copying collector generational.

For both collectors, all heap-allocated objects must be valid Java objects that contain vtable pointers as their first field. The conservative collector uses information in the `VTable` object concerning the size of the object. If the object is an array, it also uses type information to determine whether or not the array contains pointers. The copying collector uses additional fields in the `VTable` object to derive the locations of all pointers in an object.

2.11.6 Native Code

Marmot can use the same alignment and calling conventions as native x86 C/C++ compilers. A C++ class declaration corresponding to a Java class is straightforward to build manually because Marmot adds fields to objects in a canonical order.

Native code must interact with the garbage collector. Before executing blocking system calls, a thread must put itself in a safe state to allow the collector to run during the call. A safe state means that the call does not use any heap data structures because the structures may move during garbage collection. Native code may not retain any Java pointers across garbage collection points. The per-thread state includes fields where native code may place pointers that must be updated by the garbage collector. It is not a general solution, but it has sufficed for our library development.

2.12 Libraries

Marmot uses a set of libraries written from specifications of the Java 1.1 class libraries [CL98a, CL98b]. The `java.lang`, `java.util`, `java.io`, and `java.awt` packages are mostly complete. Individual classes in other packages have been implemented as required. Some methods have not yet been updated to provide full UNICODE support. The AWT library currently only supports the Java 1.1 event model.

The libraries are predominantly written in Java. Native code is used only when functionality could not be reasonably expressed in Java; it is *not* used for performance reasons. C++ methods are used as interfaces to graphics and I/O primitives. Assembly code is used to implement some of the math libraries (e.g., trigonometric functions). The libraries are comprised of 33KLOC of Java code, 4.6KLOC of C++ code (plus 2.3KLOC C++ header files), and 1KLOC of assembly code.

3 Results and Analysis

This section presents preliminary performance results for Marmot. It compares these against three other Java systems, and provides some performance comparisons against native C compilation.

3.1 Marmot Versus Java Compilers

To evaluate the quality of the Marmot compiler relative to other available native-code Java compilers, we chose three other native-code Java compilers from different categories:

- Just-in-Time compilers: Microsoft Visual J++ Version 6.0, VM (jview) version 5.00.2922. Currently considered a state-of-the-art JIT compiler [Nef98].
- Commercial static compilers: SuperCede for Java, Version 2.03, Upgrade Edition.
- Research static compilers: IBM Research High Performance Compiler for Java (HPJ).

To measure overall performance, we compiled and executed a number of small- to medium-sized Java programs (described in Figure 4) using each compiler, and measured the execution time on an otherwise unloaded dual Pentium II-300 Mhz PC running Windows NT 4.0 SP3 in 512MB of memory. The running times were "utime" averaged over 5–50 runs of each program, with added loops around some of the short-running programs to avoid problems due to the granularity of the clock. This measurement methodology is used in all comparisons in this paper.

The graph in Figure 5 shows the relative performance of all four compilers on the benchmark programs (results for the benchmarks with C equivalents are given in Figure 6). The results indicate that Marmot is at least competitive with the other compilers in terms of generating code for medium-sized programs.

To determine whether Marmot is competitive across a range of program features, and not simply dependent on a superior implementation of a single feature we used micro-benchmarks (not shown). We started with the

Name	LOC	Description
marmot	88K	Marmot compiling itself
jessmab	14K	Java Expert Shell System solving "Bananas and Monkeys" problem.
jessword	14K	Java Expert Shell System solving the "Word game" problem.
jlex	14K	JLex generating a lexer for sample.lex.
javacup	8760	JavaCup generating a Java parser
parser	5581	The JavaCup generated parser parsing Grm.java
SVD	1359	Singular-Value Decomposition (3x4 and 100x600 matrices)
plasma	648	A constrained plasma field simulation/visualization
cn2	578	CN2 induction algorithm
slice	989	Viewer for 2D slices of 3D radiology data
linpack	679	Linpack 10*1000
impdes	561	The IMPACT benchmark DES encoding a large file
impgrep	551	The IMPACT benchmark grep on a large file
impli	8864	The IMPACT benchmark li 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 benchmark wc on a large file
impsort	113	The IMPACT benchmark merge sort of a 1MB table
impsieve	64	The IMPACT benchmark prime-finding sieve

Figure 4: Small-to-Medium Benchmark Programs. The programs below the horizontal line have implementations in both Java and C.

UCSD Benchmarks for Java [GP97], modified them to prevent the compilers from doing inlining, loop hoisting, etc., and added tests for other things we wanted to measure. Marmot generated code was generally comparable with code generated by the other systems. Marmot floating point performance trailed that of the IBM HPJ and SuperCede systems by a factor of two (matching Microsoft Visual J++). For a throw-and-catch combination Marmot was roughly twice as fast IBM HPJ and SuperCede, and 70 times faster than Microsoft Visual J++.

We also wrote micro-benchmarks to measure the performance of the garbage collectors by updating and deleting nodes from a large tree data structure. When deletion of nodes was uniform with respect to the age of objects, Marmot performed 10 times better than Microsoft Visual J++, 2 timers better than SuperCede, and 1/3 faster than IBM HPJ. When young objects were deleted at a much faster rate than old objects were deleted, the performance of Microsoft Visual J++ improves by a factor of two (still 5 times slower than Marmot).

3.2 Marmot Versus C Compilers

Another way to characterize the performance of Marmot-generated executables is to compare their run times with those of similar programs written in a lower-level language and compiled with an optimizing compiler for that language. The IMPACT/NET project [HGH96, HCJ+97] has transliterated a variety of C/C++ benchmark programs to Java, making such comparisons possible. Figure 6 shows the performance of Java programs compiled by Marmot relative to corresponding C/C++ programs compiled by Microsoft Visual C++ version 6.0. For some of these benchmarks, Marmot performance approaches the performance of C/C++, but for most of them it is significantly worse.

3.3 Marmot Performance Breakdown

One purpose of the Marmot baseline compiler is to direct our future optimization research in profitable directions. To do so, we needed to understand understand where Marmot-generated executables spend their time. We also wanted to determine the overhead incurred by garbage collection and the runtime checks mandated by the Java language semantics.

The time spent in the garbage collector can be measured by instrumentation of the garbage collector. Marmot allows us to selectively disable various nonstandard optimizations, such as synchronization elimination, as well as runtime checks required by the Java semantics, such as array bounds checking. By disabling particular optimizations or checks, we can determine their benefits or costs, respectively. There are, of course, interactions between these settings, so the order in which they are disabled is significant. For this study, we have simply disabled features in a fixed order: stack allocation, synchronization elimination, dynamic cast check, array bounds check, array store check, and null pointer check.

Figure 7 shows the resulting performance breakdowns for several benchmark programs. None of the runtime checks fail during execution of these programs, so eliminating the checks does not influence the behavior of the programs (on the specific input data). Eliminating all the checks provides an upper bound on how much performance can be gained by semantics-preserving optimizations to eliminate the checks.

It is interesting to note that for the smaller micro-benchmarks, array bounds checks consume 8-10% of execution time, consistent with earlier observations [HGH96]. However, as program sizes increase and more diverse data structures are used, the relative significance of bounds checks decreases.

In many small SPEC-like benchmark programs, the cost of array store checks and dynamic casts was quite

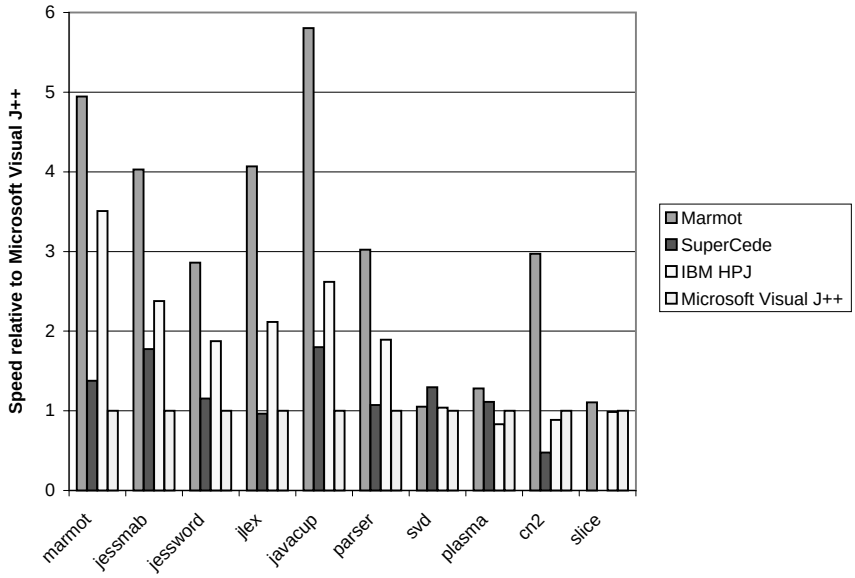

Figure 5: Relative performance of compiled code on small-to-medium benchmarks (normalized: Microsoft Visual J++ = 1). We were unable to generate an optimized executable of Marmot with the IBM HPJ compiler; we used a debug executable instead for that single data point.

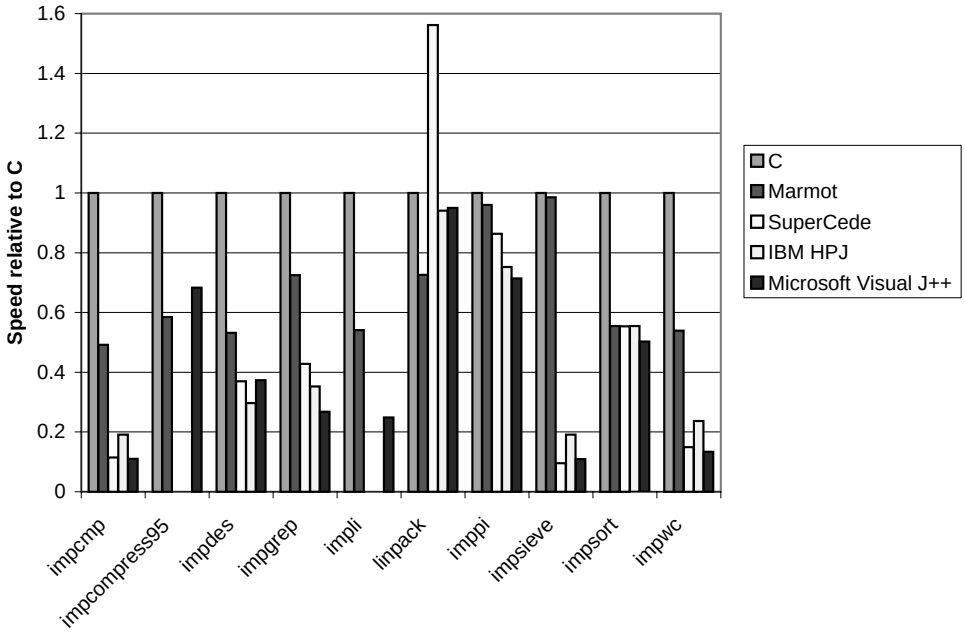

Figure 6: Relative performance of compiled code on benchmarks having both C and Java implementations (normalized: Visual C++ Version 6.0 = 1).

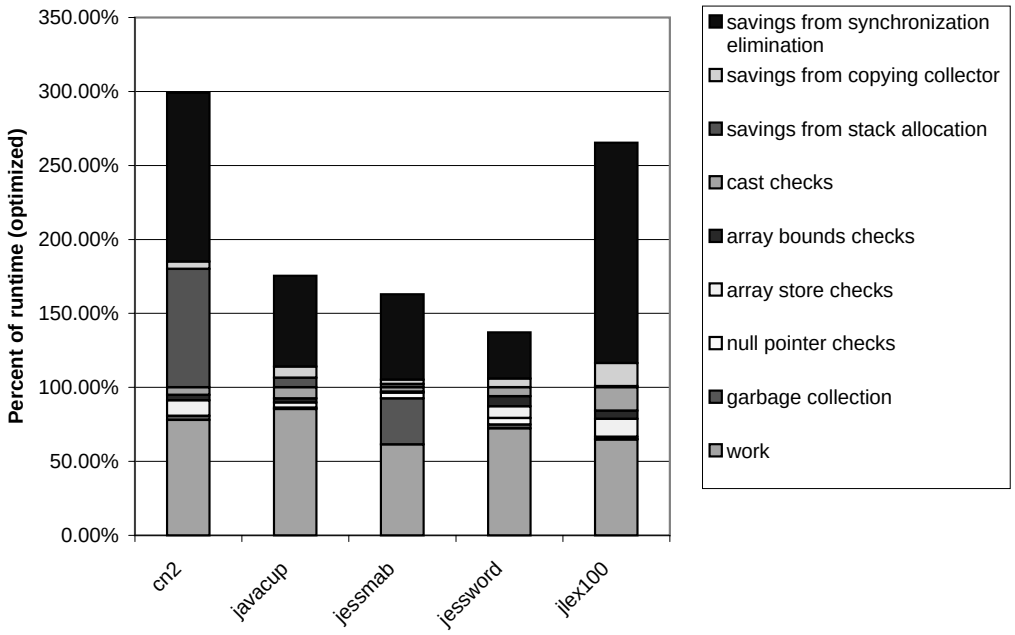

Figure 7: Performance breakdowns for Marmot-compiled code. The cost of garbage collection is obtained by instrumentation. The other costs are deduced by running programs compiled with various optimizations and runtime checks enabled and disabled. All costs are relative to the cost of running the programs generated by default. Synchronization elimination and stack allocation benefits occur above the 100% line because the default is to enable these optimizations.

large: as much as 1/4 of the total running time of the JLex benchmark. To determine that this was not due to poor implementation we added a micro-benchmark to measure the cost of such checks (not shown). Marmot's costs for dynamic type checks are no worse than those generated by other compilers. This suggests further exploring the removal of dynamic type checks via static analysis.

For the (single-threaded) benchmarks, the synchronization elimination optimization was quite effective. This result indicates that multithreaded programs, even those where little or no lock contention takes place, often spend more time in synchronization operations than in any other form of runtime checking. As with array store checks, we have verified that Marmot's implementation of synchronization is competitive with that of other systems (not shown). This suggests that Java implementors should investigate the elimination of synchronization primitives in multithreaded programs.

4 Related Work

Several Java compiler projects statically compile either Java source code or Java bytecodes to C, using C as a portable assembly language. The most complete implementations known to us are Harissa [MMBC97], Toba [PTB+97], and TurboJ [Ope98]. Harissa and Toba both have their own runtime systems using the Boehm-Demers-Weiser conservative garbage collector. TurboJ incorporates compiled code into the Sun Microsystems Java runtime system using JNI.

Other Java compiler projects statically compile Java source or Java bytecodes via an existing back end which is already used to compile other languages. The IBM High Performance Compiler for Java uses the common back end from IBM's XL compilers. `j2s` from UCSB [KH97] and `j2s` from University of Colorado [MDG97] both use the SUIF system. The IMPACT NET compiler [HGH96, HCJ+97] uses the IMPACT optimizing compiler back end. The Java compiler from the Cecil/Vortex project [DDG+96] uses the Vortex compiler back end. The Vortex runtime system includes a non-conservative garbage collector tuned for Cecil programs; the other systems retrofit (conservative) garbage collectors into their runtime systems since the systems were not designed for tracing collection.

Most Java compilers compiling directly to native code are part of commercial development systems. Tower Technology's TowerJ [Tow98] generates native code for numerous platforms (machine architectures and operating systems) while SuperCede [Sup98] and Symantec's Visual Café [Sym98] generate native code for x86 systems. These systems all include customized runtime systems.

Sun Microsystem's HotSpot compiler [HBG+97] uses technology similar to that of the Self compilers. Optimizations are based on measured runtime behavior, with recompilation and optimization being performed while the program is running.

Instantiations' Jove compiler [Ins98] and NaturalBridge's BulletTrain compiler [Nat98] both employ static whole-program analysis and optimization. They include their own runtime systems. Although little published information is available about these two systems, it appears that of all the Java compilers mentioned they are the closest in design to the baseline Marmot system.

5 Conclusion

To provide a realistic baseline for future optimization research, we have built Marmot, an optimizing native compiler and runtime system for Java. Since Marmot is a research vehicle, we have focused on ease of implementation and modification rather than on compilation speed, compiler storage usage, debugging support, library completeness, or other requirements of production systems. Examples of this approach include our decisions to implement the compiler in Java, to separately perform optimizations that could have been engineered into a single pass, to produce and link assembly code rather than generating native executables directly, and to implement library functionality as needed and in Java where possible.

We have shown that, for a variety of benchmarks, Marmot-compiled executables achieve performance comparable or superior to those generated by other research and commercial Java systems. We believe, however, that there is much room for future improvement. The preliminary execution time breakdowns of Section 3 indicate that a large amount of potentially removable overhead remains, particularly in garbage collection, synchronization, and safety checking. We are actively pursuing optimization research in these areas.

Acknowledgements

We would like to thank Cheng-Hsueh Hsieh and Wen-mei Hwu of the IMPACT team for sharing their benchmarks with us.

References

[App98] Andrew W. Appel. *Modern Compiler Implementaton in Java*. Cambridge University Press, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[Asu91] Jonathan M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, June 1991.

[Bac97] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, U.C. Berkeley, October 1997.

[BCHS88] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 1(1), January 1988.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 324–341, October 1996. Published as Proceedings OOPSLA '96, ACM SIGPLAN Notices, volume 31, number 10.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CFR+89] Ron Cytron, Jeanne Ferrante, Berry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, January 1989.

[CFRW91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CG95] Wei-Ngan Chin and Eak-Khoon Goh. A reexamination of "optimization of array subscript range checks". *ACM Transactions on Programming Languages and Systems*, 17(2):217–227, March 1995.

[Cha82] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[CL98a] Patrick Chan and Rosanna Lee. *The Java Class Libraries*, volume 1. Addison-Wesley, second editon edition, 1998.

[CL98b] Patrick Chan and Rosanna Lee. *The Java Class Libraries*, volume 2. Addison-Wesley, second editon edition, 1998.

[CMCH91] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-guided automatic inline expansion for c programs. *Software: Practice and Experience*, 22(5):349–369, May 1991.

[CSS96] Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. Incremental computation of static single assignment form. In *CC '96: Sixth International Conference on Compiler Construction*, LNCS 1060, pages 223–237, April 1996.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 83–100, October 1996. Published as Proceedings OOPSLA '96, ACM SIGPLAN Notices, volume 31, number 10.

[DE73] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings ECOOP'95*, LNCS 952, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, June 1998.

[Duf74] R. J. Duffin. On Fourier's analysis of linear inequality systems. In *Mathematical Programming Study 1*, pages 71–95. North Holland, New York, 1974.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[GP97] William G. Griswold and Paul S. Phillips. Bill and paul's excellent ucsd benchmarks for java (version 1.1). http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html, October 1997.

[GS98] David Gay and Bjarne Steensgaard. Stack allocating objects in Java. In preparation, 1998.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.

[Gup93] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, March–December 1993.

[HBG+97] Urs Hölzle, Lars Bak, Steffen Grarup, Robert Griesemer, and Srdjan Mitrovic. Java on steroids: Sun's high-performance Java implementation. Presentation at Hot Chips IX, Stanford, California, USA, August 1997.

[HCJ+97] Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C. Gyllenhaal, and Wen-mei W. Hwu. Optimizing NET compilers for improved Java performance. *Computer*, 30(6):67–75, June 1997.

[HGH96] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *IEEE Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.

[Ins98] Instantiations, Inc. Jove: Super optimizing deployment environment for Java. http://www.instantiations.com/javaspeed/jovereport.htm, July 1998.

[KH97] Holger Kienle and Urs Hölzle. j2s: A SUIF Java compiler. In *Proceedings of the Second SUIF Compiler Workshop*, August 1997.

[Kno98] Todd B. Knoblock. Array bounds check optimizations for Java. In preparation, 1998.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[MCM82] Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 114–119, June 1982.

[MDG97] Sumith Mathew, Eric Dahlman, and Sandeep Gupta. Compiling Java to SUIF: Incorporating support for object-oriented languages. In *Proceedings of the Second SUIF Compiler Workshop*, August 1997.

[MMBC97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the Third Conference on Object-Oriented Technologies and Sytems (COOTS '97)*, 1997.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[Nat98] NaturalBridge, LLC. Bullettrain Java compiler technology. http://www.naturalbridge.com/, 1998.

[Nef98] John Neffenger. Which Java VM scales best? *JavaWorld*, 3(8), August 1998.

[Ope98] The Open Group. TurboJ high performance Java compiler. http://www.camb.opengroup.com/openitsol/turboj/, February 1998.

[PTB+97] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Harman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (WAT) compiler. In *Proceedings of the Third Conference on Object-Oriented Technologies and Sytems (COOTS '97)*, 1997.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, January 1995.

[SI77] Norishisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 132–143, Janury 1977.

[Sup98] SuperCede, Inc. SuperCede for Java, Version 2.03, Upgrade Edition. http://www.supercede.com/, September 1998.

[Sym98] Symantec Corporation. Visual Café Pro. http://www.symantec.com/, September 1998.

[Tar96] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, Carnegie Mellon University, December 1996.

[Tow98] Tower Technology. TowerJ, release 2. http://www.twr.com/, September 1998.

[Wil76] H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Conbinatorial Theory (A)*, 21:118–123, 1976.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, page unknown pages, 1998.