Marmot: An Optimizing Compiler for Java

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

Microsoft Research

June 16, 1999

Technical Report
MSR-TR-99-33

Microsoft Research
1 Microsoft Way
Redmond, WA 98052

Marmot: An Optimizing Compiler for Java

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

Microsoft Research

June 16, 1999

Abstract

The Marmot system is a research platform for studying the implementation of high level programming languages. It currently comprises an optimizing native-code compiler, runtime system, and libraries for a large subset of Java. Marmot integrates well-known representation, optimization, code generation, and runtime techniques with a few Java-specific features to achieve competitive performance.

This paper contains a description of the Marmot system design, along with highlights of our experience applying and adapting traditional implementation techniques to Java. A detailed performance evaluation assesses both Marmot's overall performance relative to other Java and C++ implementations and the relative costs of various Java language features in Marmot-compiled code.

Our experience with Marmot has demonstrated that well-known compilation techniques can produce very good performance for static Java applications—comparable or superior to other Java systems, and approaching that of C++ in some cases.

1 Introduction

The Java™ programming language [GJS96] integrates a number of features (object-orientation, strong typing, automatic storage management, multithreading support, and exception handling, among others) that make programming easier and less error-prone. These productivity and safety features are especially attractive for the development of large programs, as they enable modularity and reuse while eliminating several classes of errors. Unfortunately, the performance penalty associated with such features can be prohibitive. While existing interpreter and just-in-time-compilation (JIT) based Java implementations compete favorably with Web technologies such as scripting languages on small Internet "applets," they are far less competitive with the static C++ compilers traditionally used in the development of larger, more static programs such as servers and stand alone applications.

We seek to bring the benefits of Java's high level features to larger programs while minimizing the associated performance penalty. Our approach is to adapt well-known implementation techniques from existing language implementations to Java, integrating them with new Java-specific technology where necessary. We have constructed a system, Marmot, that includes a static optimizing compiler, runtime system, and libraries for a large subset of Java. The native-code compiler implements standard scalar optimizations of the sort found in Fortran, C and C++ compilers [Muc97]) and basic object-oriented optimizations such as call binding based on class hierarchy analysis [BS96, DGC95]. It uses modern representation techniques such as SSA form and type-based compilation [CFR+89, Tar96] to improve optimization and to support precise garbage collection. The runtime system supports multithreading, efficient synchronization and exception mechanisms, and several garbage collectors, including a precise generational collector. Marmot is implemented almost entirely in Java and is one of its own most demanding test cases.

Our experience with Marmot demonstrates that well-known compilation techniques can produce very good performance for static Java applications—comparable or superior to other Java systems, and ap-

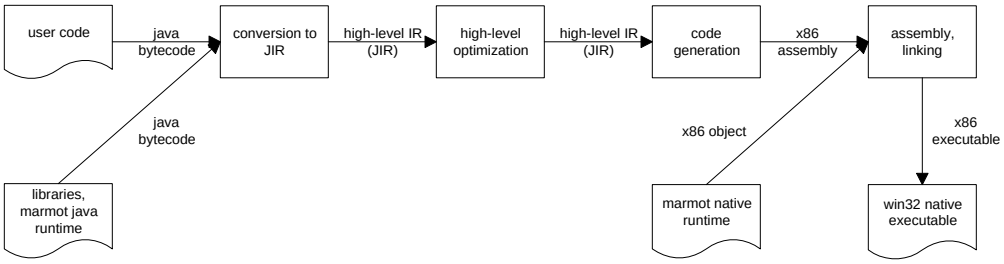

Figure 1: The Marmot compilation process.

proaching that of C++ in some cases. Marmot reduces the costs of safety checks in many programs to a modest 5–10% of execution time. Synchronization costs can be quite significant, and optimization techniques to reduce these costs are worth pursuing. Recognizing single threaded programs and optimizing them accordingly can be an important special case. Although Marmot includes a quality generational collector and further reduces the cost of storage by allocating objects with appropriately bounded lifetimes and statically known sizes on the stack, storage management costs can still be significant.

We begin this paper by describing and motivating the subset of Java supported by Marmot. Sections 3-5 describe the design and implementation of the native-code compiler, along with highlights of our experience applying and adapting modern implementation techniques for other languages to Java. The runtime system and library are described in Sections 6 and 7. Section 8 analyzes the performance of Marmot-compiled applications. Finally, related work is described in Section 9 and conclusions are presented in Section 10.

2 Supported Language Features

Marmot presently implements a significant subset of Java language and runtime features. This subset was chosen to be appropriate for studying the performance of larger Java applications. It contains the complete core language, including Java's precise exception handling and multithreading semantics. These features represent some of the important differences between Java and C++, provide useful functionality to the application developer, and provide interesting and significant semantic constraints on the optimizer and runtime.

Marmot includes a quality generational garbage collector. This was included to properly account for the costs of automatic storage management and as a basis for future study.

The Marmot runtime has its own implementation of a large subset of the Java libraries, including most of `java.lang`, `java.util`, and `java.awt` and parts of `java.net`, `java.applet`, and `java.text`. Missing library functionality is implemented as needed to support additional test applications.

The most significant omission from Marmot is support for dynamic class loading. This was omitted because of its implementation burden and because of the significant constraints it places on static optimization. It was also judged to be a feature of secondary importance for the class of applications being studied.

Some other Java features are not presently supported in Marmot, but could in principle be supported without undue effort. These include object finalization, JNI and reflection, although Marmot does support `Class.forName` and new instance creation for classes known at compile time. Marmot implements a static schedule for class initialization that is more eager than the Java specification suggests.[1]

3 Conversion to a High Level Intermediate Representation

The native-code compiler translates a collection of Java class files to native machine code. Figure 1 shows the organization of the compiler. It is divided into three parts: conversion of Java class files to a high level intermediate representation called JIR, high level optimization, and code generation. The code generation phase converts JIR to Intel x86 assembly code. The next three sections describe each part of the compiler.

Java programs are usually distributed as class files containing Java bytecode [LY97]. Java front end compilers such as `javac` translate from Java source to class files. These class files have much of the semantic information present in the original source program, except for formatting, comments, and some variable names and types. However, the class files employ an instruction set that is stack-oriented, irregular, and redundant. This makes the instruction set unattractive for use as an internal representation in an optimizing compiler.

Thus, the first part of the Marmot compiler converts Java class files to JIR, a conventional virtual-register-based intermediate form. The conversion proceeds in three steps: initial conversion to a temporary-variable based intermediate representation, conversion to static-single assignment form, and type elaboration. Type elaboration reconstructs some of the static type information that was lost in the conversion from Java source files to class files.

Starting from class files instead of Java source code has several advantages. It avoids the construction of a front end for Java (lexical analysis, parser, constant folder, and typechecker). This is especially significant because the Java class file specification has been stable, whereas the Java language specification has continued to evolve. Accepting class files as input also allows Marmot to compile programs for which the Java source files are not available.

Marmot restricts its input language to *verifiable* class files. This allows a number of simplifying assumptions during processing that rely on the code being well behaved. Because Java-to-bytecode compilers are expected to generate verifiable class files, this is not a significant restriction for our purposes.

3.1 High Level Intermediate Representation

JIR has the usual characteristics of a modern intermediate representation: 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. Each basic block concludes with a control statement that specifies the succeeding basic block, if any, to execute under normal control flow.

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.

JIR models Java exception handling by labeling basic blocks with distinguished exception edges. These edges indicate the class of the exceptions handled, the bound variable in the handler, and the basic block to transfer control to if that exception occurs during 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 this occurs, the ordered list of exception edges for the current basic block is searched to determine how to handle the exception. The first exception edge with label $e : E$ such that the class E matches the class of the raised exception is selected. The exception value is bound to the 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.

[1] Most existing Java interpreters and JITs also vary slightly from the details of the Java class initialization specification.

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 completes, 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].

3.2 Initial Conversion

Compiling a Java program begins with a class file containing the `main` method. This class file 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 has been converted.

The initial conversion from class files to JIR uses an abstract interpretation algorithm [CC77]. An abstraction of the virtual machine stack is built and the effects of instruction execution on the stack 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. Virtual machine instructions that manipulate the stack are converted into effects on the stack model along with JIR instructions that manipulate temporary variables. The assumption that the source class file 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 virtual machine instructions because longs and doubles are represented as multiple elements on the stack. Conversion reassembles these split values into references to multiword constants and values. Once again, verifiability of the class file ensures that this is possible.

Some simplification and regularization of virtual machine instructions occurs during the initial 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. operations are translated as simple references to local variables. Reducing the number of primitives in this way simplifies subsequent processing of JIR.

The initial conversion makes manifest some computations that are implicit in the virtual machine instructions. This includes operations for class initialization and synchronized methods. This lowering to explicitly represented operations is done to make the operations available for further analysis and optimization.

A problematic feature of Java bytecode is the subroutine used to encode `try-finally` constructs. Marmot initially used a complex encoding of these as normal control flow (similar to that described in Freund [Fre98], but with additional mechanism for SSA form). This proved complex and inefficient in the normal case, and we eventually adopted Freund's minimalist approach of expanding `try-finally` handlers in line.

3.3 Static Single Assignment Conversion

The second step of converting from class files 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.

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

3.4 Type Elaboration

The third and final step in constructing JIR from class files is *type elaboration*. This process infers type information left implicit in instructions, 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 class files:

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

Class files do preserve some type information, namely:

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

Type elaboration operates per method. It begins by collecting constraints on the omitted types. The result is a set of type constraints over type variables. This system is solved by factoring the constraints into strongly connected components and solving each component in depth-first order. The result of type elaboration is a strongly typed intermediate representation in which all variables are typed, all coercions and conversions are explicit, and all overloadings of operators are resolved.

Because type information was lost in the initial translation from Java to bytecode, it is not always possible to recover the user's type declarations. However, type elaboration can always recover *some* legal typing of the intermediate code.

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 has proved to be useful in debugging Marmot. Gagnon and Hendren have also developed a type inference algorithm for Java bytecode for their Sable compiler [GH99].

3.5 Experience

Marmot takes verified bytecode as input in lieu of Java source. While this has the advantages outlined above, it also has costs:

1. Type elaboration is potentially computationally expensive (one natural formulation of the problem is NP-complete).
2. Certain Java language features are encoded into bytecode. For example, inner classes, finalization, and some synchronization primitives are expanded into a low level bytecode representation. Optimizations targeted at these high level features may require work to rediscover their encoded use that would have been simple or unnecessary if the compiler had started from Java source.
3. The multiple translation steps, Java to bytecode to JIR, demand cleanup optimizations. This is a minor point because most of these optimizations are desirable anyway. However, quirks of the initial bytecode representation may degrade the resulting JIR. For example, simple boolean control source expressions can become large chunks of code employing multiple statements, temporaries, and blocks.

- Standard optimizations
  - Array bounds check optimization
  - Common subexpression elimination
  - Constant/copy propagation with conditional branch elimination
  - Control optimizations (branch correlation/removal, dead/unreachable block elimination)
  - Dead-assignment and dead-variable elimination
  - Inter-module inlining
  - Loop invariant code motion
  - Loop induction variable elimination and strength reduction
  - Operator lowering/reoptimization
- Object-oriented optimizations
  - Null check removal
  - Stack allocation of objects
  - Static method binding
  - Type test (array store, cast, `instanceof`) elimination
  - Uninvoked method elimination and abstraction
- Java-specific optimizations
  - Bytecode idiom recognition
  - Redundancy elimination and loop-invariant code motion of field and array loads.
  - Synchronization elimination

Figure 2: Optimizations performed by the high level optimizer.

Basic blocks serve to group statements with related control. Marmot's use of distinguished exception edges allows larger blocks than would have been possible otherwise. One alternative would have been to annotate each possible exception point with a success and failure successor, as was done in Vortex [DDG+96]. This could lead to an explosion in edges in the control flow graph, and to very small blocks. In SSA form, this is potentially much worse since each edge may have a corresponding phi argument in each of the successor blocks.

4 High Level Optimization

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) standard optimizations for imperative languages, (2) general object-oriented optimizations, and (3) Java-specific optimizations.

4.1 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. Most of these optimizations are straightforward; complications due to the Java exception model and the explicit SSA representation are discussed in Section 4.5.

Unlike most of the optimizations shown in Figure 2, array bounds check optimization is not yet a widely used 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 include 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. If only the upper bound check is redundant, the check is rewritten to a lower-bound-only test, which eliminates a reference to the array length.

4.2 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. Marmot uses an explicit annotation mechanism 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.[2] 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. Local type information is also used in constructing a more precise call graph, which drives traditional optimizations such as inlining. Type operations not folded by the type propagator may still be eliminated later by other passes.

A type-based interprocedural analysis finds provably non-null expressions. The results are used directly to eliminate null checks and to fold conditionals, as well as by other optimizations such as load hoisting.[3]

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

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

[3] The loop-invariant code motion optimization currently hoists only those loads which cannot throw an exception; e.g., those which have been proven to have non-null base addresses.

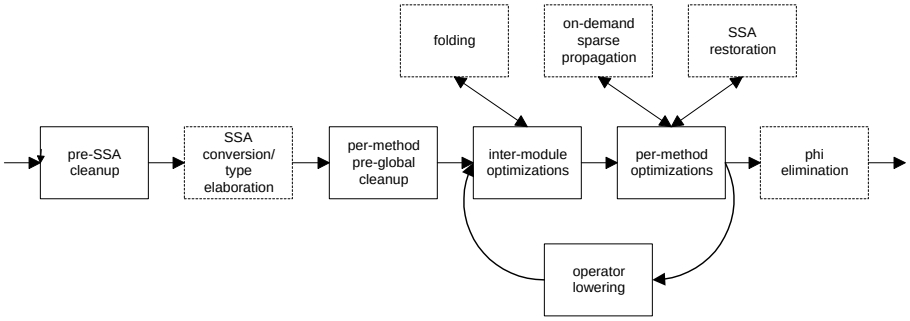

Figure 3: Optimization phases. The boxes with solid outlines are optimization phases; while those with dashed outlines are utilities invoked by these phases.

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 [GS99] for details of this optimization.

4.3 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.

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 multithreaded 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].

While techniques for the removal of redundant loads and the hoisting of loop-invariant loads exist in the literature [CL97, LCK+98], applying them to Java programs requires an extra step. The language semantics require that global memory be read (written) whenever a `monitorenter` (`monitorexit`) operation is executed, effectively invalidating any cached loads at such points. Marmot performs a call-graph-based interprocedural analysis to determine which method invocations may execute synchronization operations, and modifies the kill sets of the load analyses appropriately.

Marmot also performs several transformations that eliminate artifacts of the Java bytecode representation from the JIR. Where possible, the ternary-result comparison operators `fcmp`, `dcmp`, and `lcmp` are translated to simpler boolean comparisons. Bytecode idioms in which boolean operations are implemented via control (e.g., `y = !x` becomes the equivalent of `if (x) y=false; else y=true;`) are recognized and reduced to corresponding boolean operations. Integer bitwise operations on integral operands known to represent booleans[4] as are also reduced to boolean operations.

4.4 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 prior to conversion to remove unused methods from the representation. Similarly, converting

[4]Because the Java bytecode lacks a boolean primitive datatype, it encodes boolean operations as bitwise integral operations. Converting to the underlying boolean operations enables additional operator folding.

control operations to value operations as described in the previous subsection significantly simplifies control flow. At the same time, Marmot recognizes single threaded code and removes synchronization operations if possible. These are the only high level optimization passes that operate 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. Although the inliner includes more advanced capabilities including the ability to use profile information to make space and time tradeoffs, for the purposes of this paper, a very conservative inline control strategy was employed. It inlines where the result of inlining is estimated to be smaller than the original call sequence.

After the inter-module operations, 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.

The operator lowering phase translates high level operations such as cast checks and monitor operations into corresponding JIR code (which may include calls to Java and native code in the runtime system). Marmot then runs the inter- and intra-module optimizations as appropriate to simplify the internals of the lowered operations.[5] For example, if two distinct type tests on a single object load the same metadata (e.g., the object's class pointer), the optimizer may find one of the loads to be redundant.

4.5 Experience

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's 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.

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.[6] For this reason, the present optimizer limits code motion to provably effect-free, non-throwing operations. This limits redundancy elimination (including redundant load/store optimizations) to full, but not partial, redundancies. Better redundancy elimination may require dynamic information to justify the extra analysis effort and code expansion for specific code paths.

With the notable exception of the synchronization-driven kills of cached loads described in Section 4.3, the potentially multithreaded nature of Java programs has not significantly affected the implementation of the Marmot optimizer, for two reasons. First, many per-method optimizations model only the values of

[5]The high level operations are not lowered immediately on conversion because some optimizations (e.g., synchronization removal, cast check elimination) rely on the ability to recognize, rewrite, or remove such operations. Once these have been lowered, they are no longer available as atomic operations.

[6]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.

Figure 4: The Marmot code generation phase.

local variables and locally-allocated storage, which are by nature unaffected by threading or synchronization. Second, most of the inter-module analyses are flow-insensitive[7] and thus model all possible statement interleavings at no additional cost.

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.[8] This transformation-specific phi-operator maintenance is often the most difficult implementation task (and sometimes the largest compilation-time cost) in Marmot optimizations.

We are considering modifying the JIR to view SSA graphs as sparse use-definition annotations on top of a conventional statement-based representation, rather than as the primary representation. Doing so would allow us most of the benefits of SSA form (minus transparent lifetime extension) without the need to keep the SSA graph up to date at all times. Wolfe's parallelizing FORTRAN compiler [Wol96] uses a similar approach.

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 extra care in copy propagation, but does not affect most analyses, and has the benefit of simplifying the phi elimination phase.

The local type propagation algorithm is dependent upon IIA for estimates of the runtime types of method formals, call returns, array contents and storage locations. In the presence of polymorphic datatypes (e.g., containers of `java.lang.Objects`), these estimates are imprecise (e.g., such containers always appear to hold all instantiated subclasses of `java.lang.Object`). We are experimenting with context-sensitive inter-module analyses to provide better type estimates in the face of such polymorphism.

5 Code Generation

The Marmot code generation phase converts JIR programs to x86 assembly code. Figure 4 shows the steps of the conversion. JIR programs are first converted to a low level intermediate representation called MIR (machine intermediate representation). MIR is described in Section 5.1 and the conversion process is described in Section 5.2.

Next, it performs optimizations that clean up the converted code, including copy propagation, constant propagation, and dead-code elimination. In addition, redundant comparisons of registers with zero (to set the condition code) are eliminated. Most of these unnecessary comparisons arise from the translation of JIR conditional statements, which take only boolean variables and not comparisons as arguments.

After cleanup, register allocation is performed. Jumps to control instructions and branches to jumps are then eliminated. This optimization is deferred until after register allocation because register allocation often

[7]Some amount of flow sensitivity is encoded by the SSA representation, reducing the precision loss due to flow-insensitive analysis.

[8]Using SSA form 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.

eliminates moves and creates basic blocks containing only jumps. Following that, peephole optimizations are performed and code layout is chosen. The code layout phase arranges code so that loop control instructions are placed at the bottom of loops, avoiding unconditional jumps. Finally, assembly code is emitted, along with tables for exception handling and garbage collection. Because of complications caused by Java's precise exception semantics, Marmot does not currently schedule instructions.

5.1 Low Level Intermediate Representation

MIR is a conventional low level intermediate representation. It shares the control-flow graph and basic block representations of JIR, which enables reuse of algorithms that operate on the CFG. The instruction set of MIR is a two-address instruction set based on the instruction set of the x86 processor family (the current target architecture of Marmot). 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.

Each operand in MIR is annotated with *representation information*. This is a simplified version of type information that is used to identify pointers that must be tracked by the garbage collector.

5.2 Conversion to MIR

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

Marmot implements all metadata, 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 metaclass layout has been determined, MIR conversion is able to statically construct the required metadata instances for all class and array types.

Marmot converts methods to MIR procedures using syntax-directed translation [ASU86]. Most three-address JIR statements map to two or more two-address MIR instructions. The coalescing phase of the register allocator removes unnecessary moves and temporaries created by this translation.

The following list describes interesting cases in the translation of JIR statements to MIR instructions.

- Runtime type operations: JIR provides several operations that use runtime type information: `checkcast`, `instanceof`, and `checkarraystore`. Most of these operations are lowered during high level optimization (Section 4.4). The remaining operations are replaced by calls to Java functions that implement the operations.

- Exception handlers: JIR exception handling blocks may have two kinds of incoming edges: normal and exception. Normal edges only represent transfer of control, while exception edges also bind a variable. The conversion to MIR splits every target of an exception edge into two blocks. The first block contains the code corresponding to the original block. The second block binds the exception variable and jumps to the exception block. The conversion redirects all normal predecessors of the original JIR block to the first block, and all exception predecessors to the second block.

- Switch statements: Marmot converts dense switches to jump tables and small or sparse switches to a chain of conditional branches.

- 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 in which 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.

5.3 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[9]. 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.

Register allocation is particularly important on the x86 architecture because there are fewer than eight general-purpose registers available. Performance problems are disproportionately attributable to register spilling.

Precise 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.

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

6 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. Marmot emits code for them that rivals hand-coded assembly code.

6.1 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 6.3), 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.

6.2 Runtime Type Operations

Cast checks, array store checks, and `instanceof` operations all need to test type inclusion. To check whether one class is a subclass of another class, we use a scheme similar to that used in the DEC SRC Modula-3 system [VHK97, page 147]. Each node in the class hierarchy tree is numbered using a pre-order traversal. For each node, Marmot records the index of the node and the largest index max of a child of the node. A node n is a subclass of another node m iff $m.index \leq n.index \leq m.max$. This can be implemented in several machine instructions via an unsigned comparison of the expressions $(n.index - m.index)$ and $(m.max - m.index)$ (the *width* of the class). Marmot stores the index and the width in the vtable for each class. If a class involved in a runtime type operation is known at compile-time, Marmot generates a specialized version of the operation that uses the compile-time constants instead of fetching values at runtime.

To check whether a class implements an interface, Marmot stores a list of all interfaces that a class implements in the vtable for the class. At runtime, the list is scanned for the interface.

6.3 Interfaces

Marmot implements interface dispatch via a per-class data structure called an interface table, or *itable*. The vtable of the class 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.[10] All new methods declared in I are placed after all the itables for the direct superinterfaces of I.

[10] 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.

6.4 Exceptions

Marmot uses a program-counter-based exception handling mechanism. Memory containing Marmot-generated machine code is divided into ranges, each of which is associated with a list of exception handlers. There is no runtime cost for the expected case in which no exception is thrown. Should an exception occur, the runtime system finds the range containing the throwing program point and finds the appropriate handler in the list.

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.

6.5 Threads and Synchronization

In Marmot, 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.

Bacon et al. [BKMS98] describe a lightweight implementation of monitors for Java, based on the fact that contention for locks in Java programs is rare. The most common cases are a thread locking an object that is unlocked and a thread locking an object that it has already locked several times. For the first case, both implementations execute only a single machine-level synchronization instruction per `monitorenter/monitorexit` pair. For the second case, both implementations execute only a few instructions, none of which are machine-level synchronization instructions.

The implementations differ in allocation of monitor objects. Bacon et al. store information in a bit-field of a header word that every object contains. If an uncommon case occurs, the bit-field is overwritten by an index into a table of monitor objects. Marmot allocates a monitor object on the heap the first time an object is locked. This cost is amortized over the number of synchronizations per object, which Bacon et al. have shown is usually more than 20.

6.6 Garbage Collection

Marmot offers a choice of three garbage collection schemes: a conservative collector, a copying collector, and a generational copying collector. The conservative collector uses a mark-and-sweep technique. The copying collector is a semi-space collector using a Cheney scan.

The generational collector is a simple two generation collector that has a nursery and an older generation. Small objects are allocated in the nursery, while large objects are pretenured into the older generation. When the nursery fills, it is collected and all live objects are copied to the older generation.

The collector has a write barrier that tracks pointers from the older generation to the nursery. Those pointers are treated as roots when the nursery is collected. There are two write barrier implementations available: a card-marking write barrier and a sequential store buffer (SSB). When the SSB is used with multithreaded programs, the buffer is broken into chunks that are allocated on demand to individual threads. This avoids the need for a synchronization operation when an entry is stored into the buffer.

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 the array contains pointers. The copying and generational collectors use additional vtable fields to locate pointers in objects.

For the copying and generational collectors, Marmot generates tables that allow the collectors to find all pointers on the call stack. Every call site is associated with an entry that describes the callee-save registers and the stack frame locations that contain pointers that are live across the call.

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	113	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++.

Marmot uses the generational collector as the default collector for the system, with a sequential store buffer for the write barrier. The generational collector generally has a smaller memory footprint and a shorter average pause time than the other collectors.

6.7 Native Code

Marmot uses the same alignment and calling conventions as native x86 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 explicitly with the garbage collector. Before executing a blocking system call, a thread must put itself into a heap-safe state to allow the collector to run during the call. Any pointers into the heap must be saved for the duration of the heap-safe state in a location visible to the collector, not used until the state has been exited, and then restored afterward. Marmot per-thread state includes fields that native code can use for this purpose.

Native code also interacts with high level optimization, in that native methods can invoke Java methods and can read and write Java data structures. Marmot supports optimization of code containing native calls via explicit annotations in an external file.

7 Libraries

Marmot uses a set of libraries written from specifications of the Java 1.1 class libraries [CL98a, CL98b]. The `java.lang` (the core language classes), `java.util` (utility classes), `java.io` (input/output), and `java.awt` (graphics) packages are mostly complete. A majority of classes in the `java.net` (network communications), `java.text` (international date/text formatting support), and `java.applet` (browser interface) packages are also implemented.

The `java.lang`, `java.io`, `java.net`, and `java.awt` packages all provide an interface to the underlying system. All the logic of the classes in these packages has been implemented in Java; native code is used only to call the necessary system functions. Native code 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 51K lines of Java code (25K lines implementing `java.lang.awt`), 11.5K lines of C++ code (4.5K lines for garbage collectors), 3K lines of C++ header files, and 2K lines of assembly code.

8 Performance Measurements

In this section, we examine the performance of Marmot-compiled code relative to C++ and to other Java implementations. We also analyze the costs of safety checks, and the benefits of the synchronization and stack allocation optimizations. Finally, we compare three different garbage collection algorithms for Marmot.

8.1 Marmot Compared to a C++ Compiler

To understand how the performance of tuned applications written in Java compares to that of tuned applications written in C++, we examined the performance of a set of benchmarks, described in Figure 5, that have both C++ and Java variants. The benchmarks were transliterated from C++ to Java by the IMPACT/NET project [HGmWH96, HCJ+97]. To better understand the remaining performance differences between C++ and Java versions, we then modified and reexamined some of the Java benchmarks to remove transliteration artifacts and to work around limitations of the current Marmot optimizations.

We compiled the benchmark programs using Marmot and Microsoft Visual C++ version 6.0. We also benchmarked three other Java systems, each representative of a different category:

- Just-in-Time compilers: Microsoft Java VM (MS JVM), version 5.00.3168, 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 (IBM HPJ), version IVJH3001(I).

These static compilers, including Marmot, do not implement dynamic class loading. Because the Microsoft Java VM supports dynamic class loading, it does not perform whole program optimizations such as IIA that would be invalidated if the class hierarchy were extended at runtime.

To measure overall performance, we executed the benchmarks in each environment and measured the execution time on an otherwise unloaded, dual processor Pentium II-300 Mhz PC running Windows NT 4.0 SP3 in 512MB of memory. The running times are "UTime" averaged over several runs of each program, with loops around some of the short-running programs to minimize measurement noise due to the 10-15 ms granularity of the clock. The standard deviations in the running times are nominal.

Figure 6 shows the speed of the unmodified Java programs and the corresponding C++ versions. The C++ versions are 1.03 to 1.76 times the speed of the Marmot versions with a median of 1.28.[11]

Figure 7 shows the speed of these programs with array bounds checking disabled. Array bounds checking costs do not account for the majority of the C++ speed advantage.

We then profiled several programs for which the C++ version was more than 10% faster than the Marmot version and made the following small changes to the Java source code:

- impcmp, which compares two files, contained two virtual function calls in the inner loop of the Marmot version that are macros in the C++ version. Two static fields were declared as InputStreams, but were only ever assigned BufferedInputStreams (a subclass of InputStream). The flow-insensitive IIA was not strong enough to rebind the virtual calls. We redeclared the static fields as BufferedInputStreams, which eliminated the virtual calls.
- impcompress contained register spills in the inner loop of the Marmot version, but not in the C++ version. There were three reasons for the spills. First, the Java code used instance fields and virtual functions, whereas the C++ version used static fields. We changed the Java version to use static fields and functions like the C++ version, which eliminated the need for a register to hold the `this` pointer.

[11] The original C++ version of the impsieve benchmark was 30% slower than the original Java version compiled by Marmot because Marmot automatically inlines several tiny methods that the C++ compiler leaves out of line. In Figure 6, we report timings of a faster C++ variant in which those methods have been inlined.

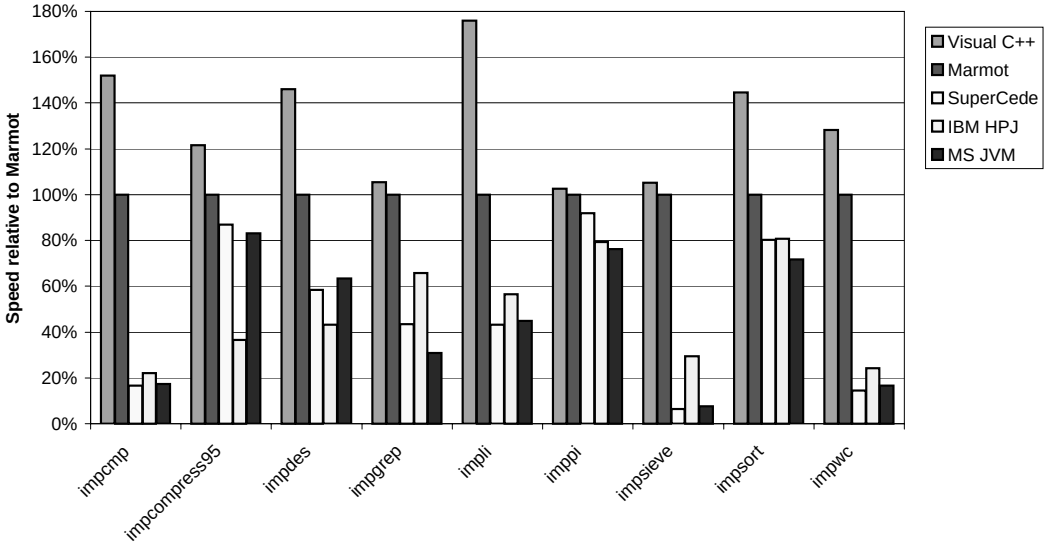

Figure 6: Relative speed of compiled code on benchmarks having both C++ and Java variants (normalized: Marmot = 100%).

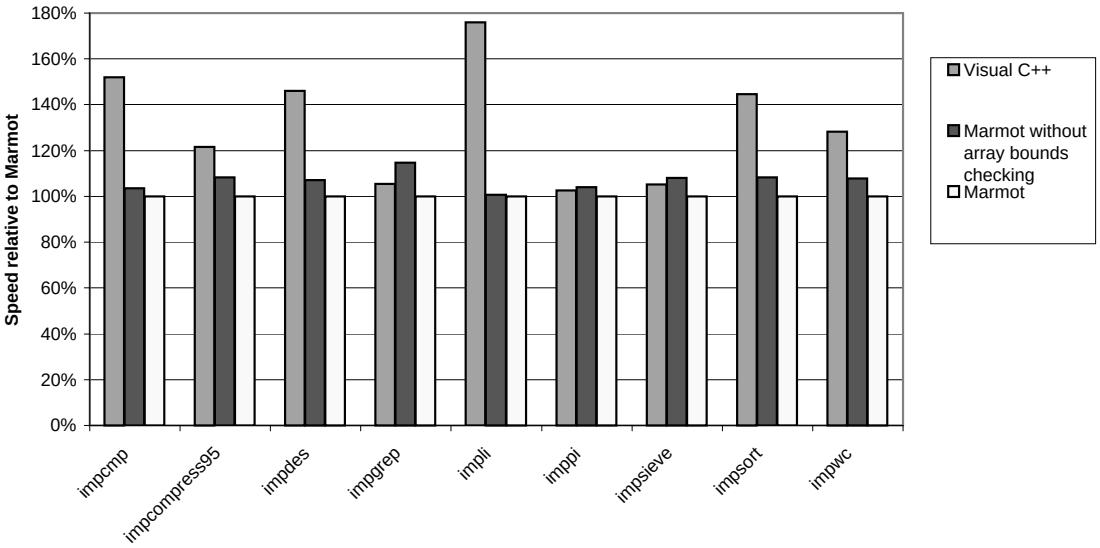

Figure 7: Relative speed of compiled code on benchmarks having both C++ and Java variants, with and without array bounds checking enabled for Java (normalized: Marmot = 100%).

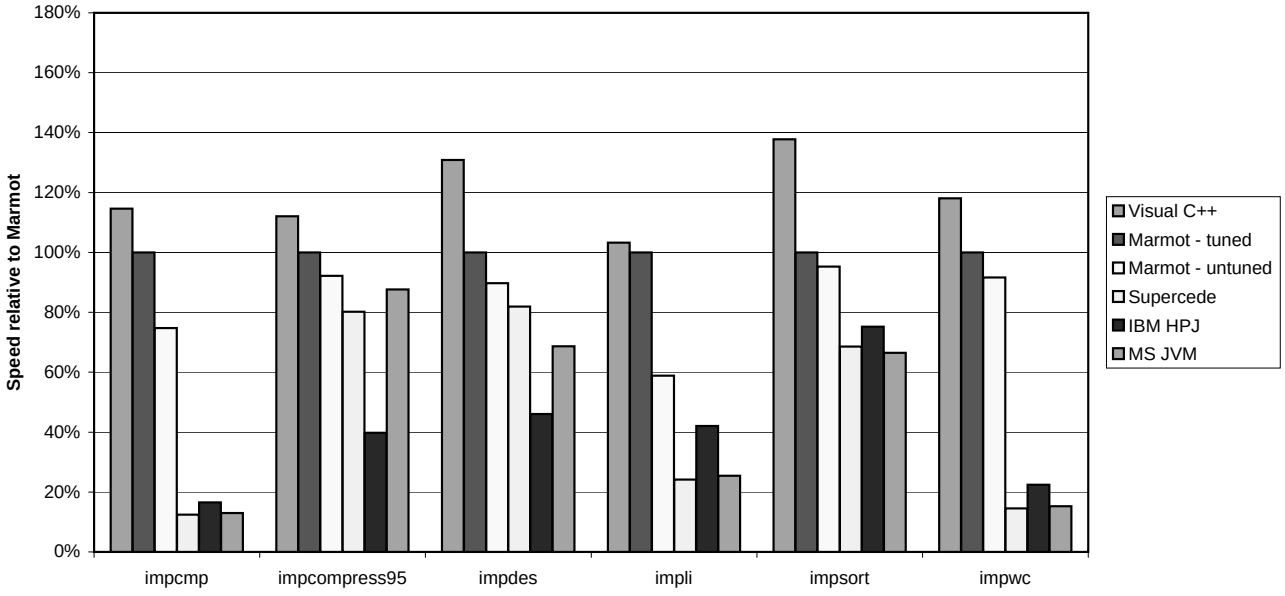

Figure 8: Relative speed of benchmarks after tuning (normalized: Marmot - Tuned = 100%).

Second, the Java auto-increment operator `++` caused additional register pressure when it was used in an array subscript expression of the form `a[i++]`. The variable `i` must appear to be updated before the array subscript expression is evaluated, in case an exception occurs during the subscripting. This makes the new value of `i` live while the old value of `i` is still live, increasing register pressure by one register. Because Marmot currently lacks support for the automatic code motion needed to obviate this extra register pressure, we manually moved the index increment after the array subscript expression. Third, Marmot currently lacks frame pointer omission, which the C++ compiler uses to free another register.

- impdes allocates 2-element arrays in an inner loop of the Java version. Because the original C++ program contained no such allocation, we modified the Java source to reuse a single temporary array instead of repeatedly allocating and discarding a new one. The Java version also executes array bounds checks in its inner loop, so we added a conditional test before the loop that allowed the compiler to eliminate some, but not all, of them. The innermost loop contains multiple references to several different constant arrays. The C++ compiler allocates and initializes these arrays statically and never enregisters their base addresses. Marmot currently allocates these arrays on the heap and stores pointers to them in static fields, causing extra loads and creating extra register pressure to hold the heap addresses.

- impli is a LISP interpreter that has a private garbage collector, that interacts badly with the Marmot collector. We eliminated the private garbage collector from the Java version and relied on the Marmot garbage collector instead.

- impsort, a merge sort, had array bounds checks in its innermost loops. We added conditional tests before the loops that allowed the optimizer to eliminate some of these checks. Marmot currently lacks strength reduction on pointer arithmetic that the C++ compiler provides.

- impwc, a word counting program, has several static fields that are better declared as local variables, allowing Marmot to eliminate more array bounds checks and to improve register allocation. Here, too, the C++ compiler performs strength reduction on pointer arithmetic.

Name	LOC	Description
marmot	88K	Marmot compiling _213_javac
jlex100	14K	JLex generating a lexer for sample.lex, run 100 times.
javacup	8760	JavaCup generating a Java parser
SVD	1359	Singular-Value Decomposition (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

Figure 9: Small-to-medium Java benchmark programs.

Name	LOC	Description
_201_compress	927	Compression program compressing and decompressing files
_202_jess	11K	Java Expert Shell System solving set of puzzles
_209_db	1028	An in-memory database program performing a sequence of operations
_213_javac	unavailable	Java bytecode compiler
_222_mpegaudio	unavailable	MPEG audio program decompressing audio files in MPEG Layer-3 format
_228_jack	unavailable	Java parser generator generating its own parser

Figure 10: SPEC JVM Client98 benchmarks.

Name	Lines of code	Description
Numeric Sort	340	Sorts an array of integers
String Sort	578	Sorts an array of strings
Bit-field Operations	411	Tests a variety of functions that manipulate bit-fields
FP Emulation	1540	Implements floating-point operations in software
Fourier	365	Computes coefficients for series approximations of waveforms
Assignment	485	Assigns tasks using an operations research algorithm
IDEA Encryption	687	Encrypts and decrypts in-memory data.
Huffman Compression	632	Compresses and decompresses in-memory data
Neural Net	760	Simulates back-propagation in a neural net
LU decomposition	513	Solves linear equations

Figure 11: jBYTEmark 0.9: a Java transcription of the BYTE magazine BYTEmark benchmarks [Hal98].

Figure 8 shows the performance of the benchmarks after tuning. The C++ versions are 1.03 to 1.38 times the speed of the Marmot versions with a median of 1.13.

The performance of several benchmarks could be improved through better compiler optimization. For impdes, an analysis that determined that the constant arrays could be allocated and initialized statically would reduce register pressure. For impsort and impwc, techniques for strength reduction of pointer arithmetic in the presence of garbage collection, such as those of Diwan et al. [DMH92], might improve performance.

The results of this section suggest that production-quality compilers for Java can produce code competitive with that produced by production-quality C++ compilers.

8.2 Marmot Compared to Other Java Systems

We also compared the performance of Marmot and the other Java systems on a set of Java benchmarks (described in Figures 9 to 11). Figure 12 shows the relative performance of the four Java systems on these benchmarks. The results indicate that the well-known compilation techniques used by Marmot produce executables whose performance is comparable or superior to those produced by other Java systems.

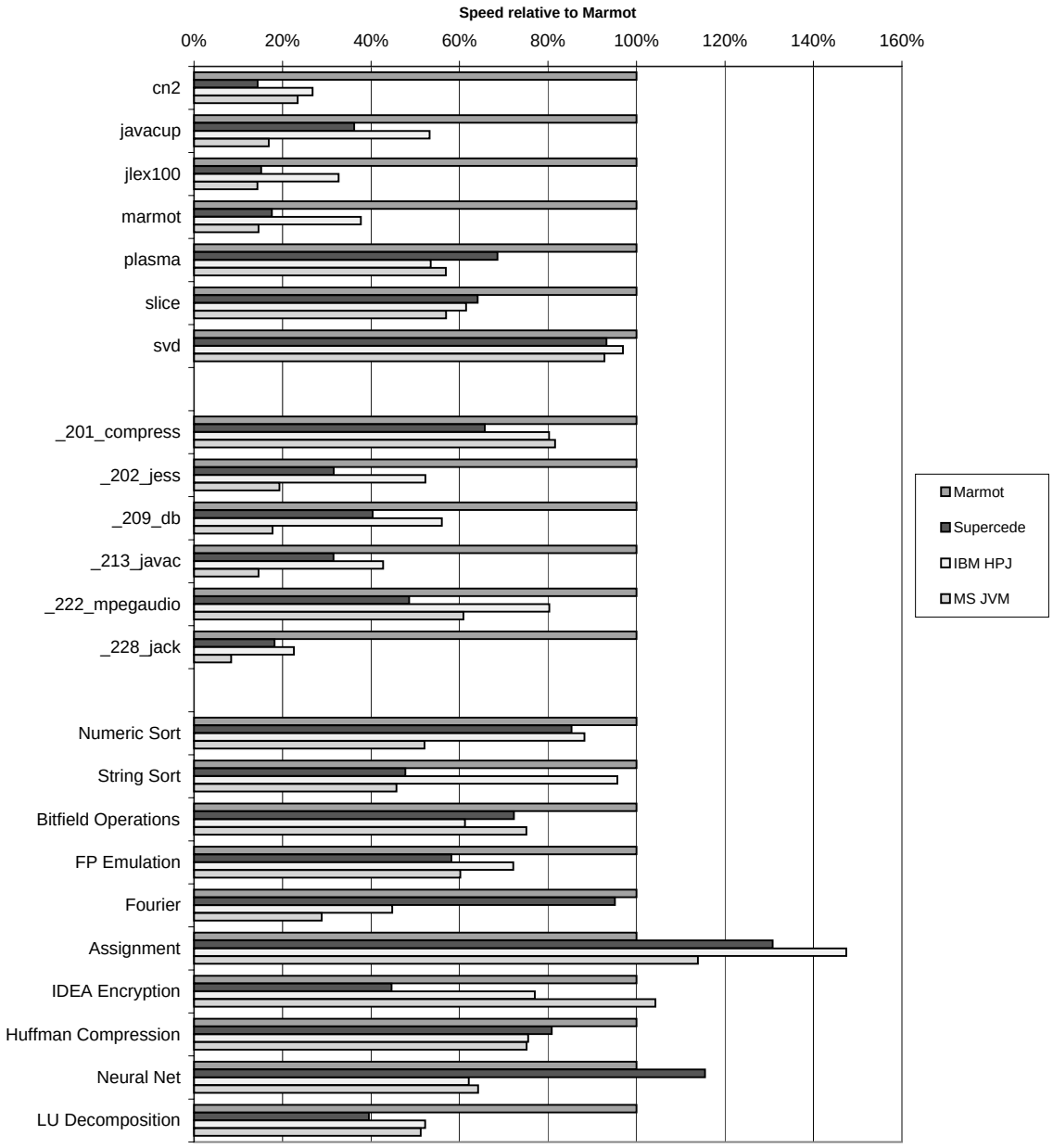

Figure 12: Relative speed of benchmarks (normalized: Marmot = 100%).

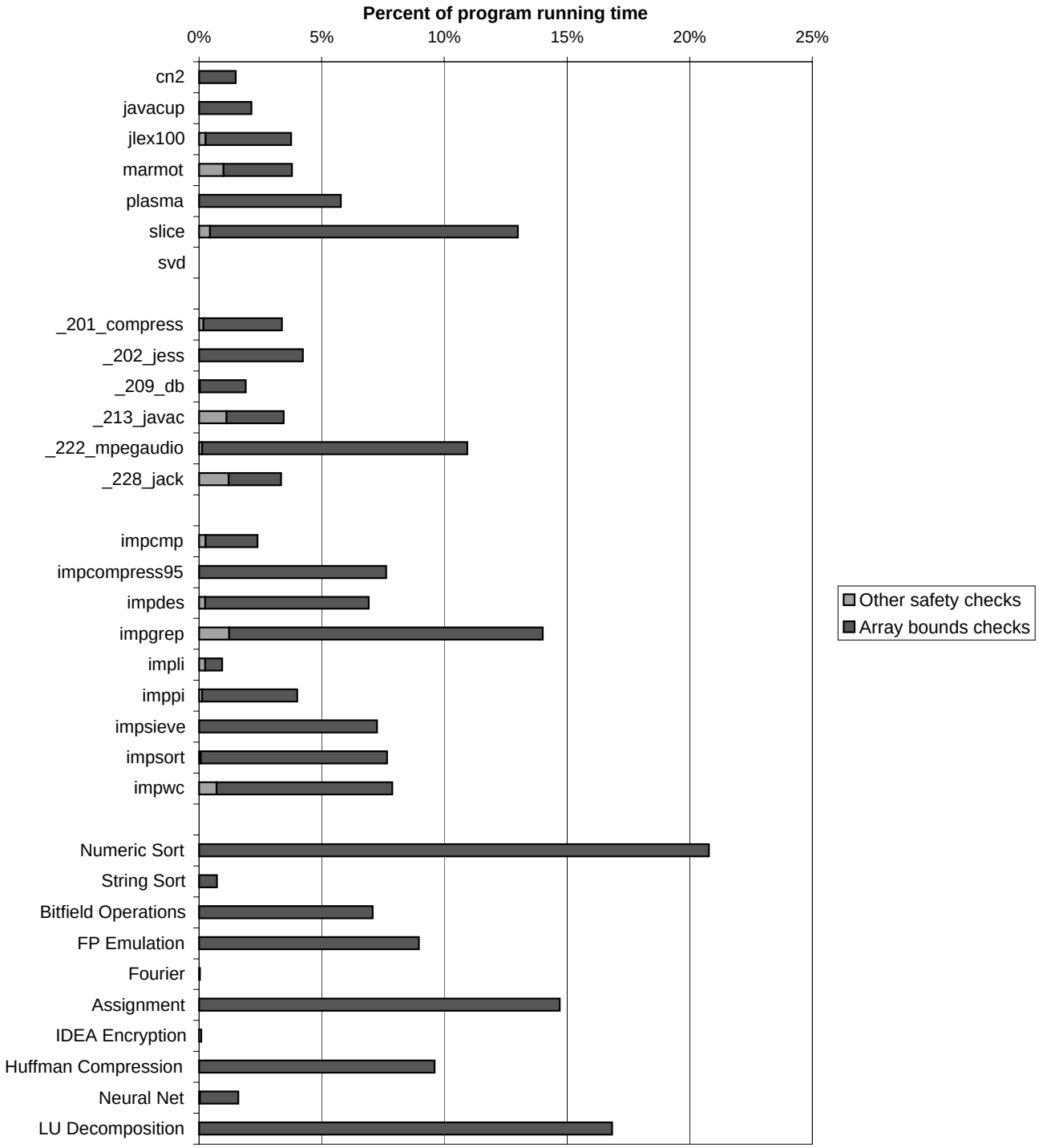

Figure 13: The cost of safety checks in programs compiled by Marmot. The costs are relative to program execution times when the programs are compiled with all safety checks enabled.

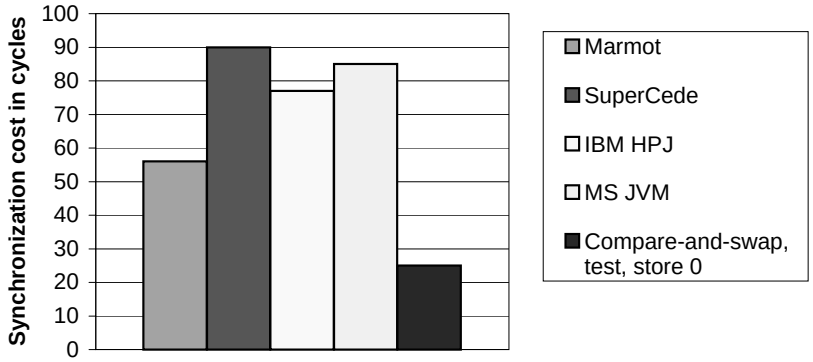

Figure 14: Relative speed of invoking an empty synchronized method.

8.3 The Cost of Safety Checks

Java programs execute several kinds of run time safety checks: array bounds checks, array store checks, cast checks, and null pointer checks. Marmot allows us to selectively disable particular run time checks to determine their cost.[12] Figure 13 shows the costs of the various safety checks as percentages of the execution times of the programs with all checks enabled. For all programs, the combined cost of array store checks, null pointer checks, and dynamic cast checks is insignificant. The median combined cost of these checks is 0.1%, with a maximum of 1.2%. For most programs, the total cost of safety checks is small. The median cost of safety checks is 4.1%, with 80% of the programs having a safety check cost of less than 10%. A few array-intensive programs have array bounds check costs that exceed 10%.

8.4 Synchronization Elimination

The synchronization elimination optimization removes synchronization operations from programs that can be proved at compile time to be single threaded. The magnitude of the benefit depends on the cost of synchronization. Bacon et al. [BKMS98] have shown that the most common synchronization case is an unlocked object being locked by a thread. Figure 14 compares the average cost of a virtual call to an empty synchronized method for each of the Java systems that we have measured and also includes as a lower bound the cost of the core synchronization code sequence: a compare-and-swap synchronizing instruction, a test that the lock had been acquired, and a store of 0 to release the lock. Marmot has primitive synchronization costs that are at least competitive with those of the other Java systems and are within about a factor of 2 of the lower bound. This shows that the costs of synchronization primitives are reasonable and that Marmot synchronization costs are not an artifact of a poor implementation.

The effect on performance of disabling the synchronization elimination optimization is shown in Figure 15. The effect varies widely and is dependent on the particular program. At one extreme are single threaded programs such as impwc and impcmp, which run 3 to 4 times faster with the optimization than without it. These programs spend almost all their time in inner loops that call a synchronized library function to read a byte from a buffered file. At the other extreme are computation-intensive programs such as _222_mpegaudio and SVD, which execute almost no synchronization and for which the optimization has no effect. The optimization also has no effect on Plasma and Slice, which are actually multithreaded.

In general, eliminating synchronization is more important for the larger programs (SPEC JVM 98 and the small-to-medium sized benchmarks) than for the smaller programs (the IMPACT benchmarks and the jBYTEmark benchmarks). For the larger programs, disabling the optimization reduces the speed of the programs by a median value of 30%. For the smaller programs, disabling the optimization reduces the speed

[12] None of the safety checks fail during the execution of any of the benchmarks, so eliminating the checks does not change the behavior of the programs on the benchmark input.

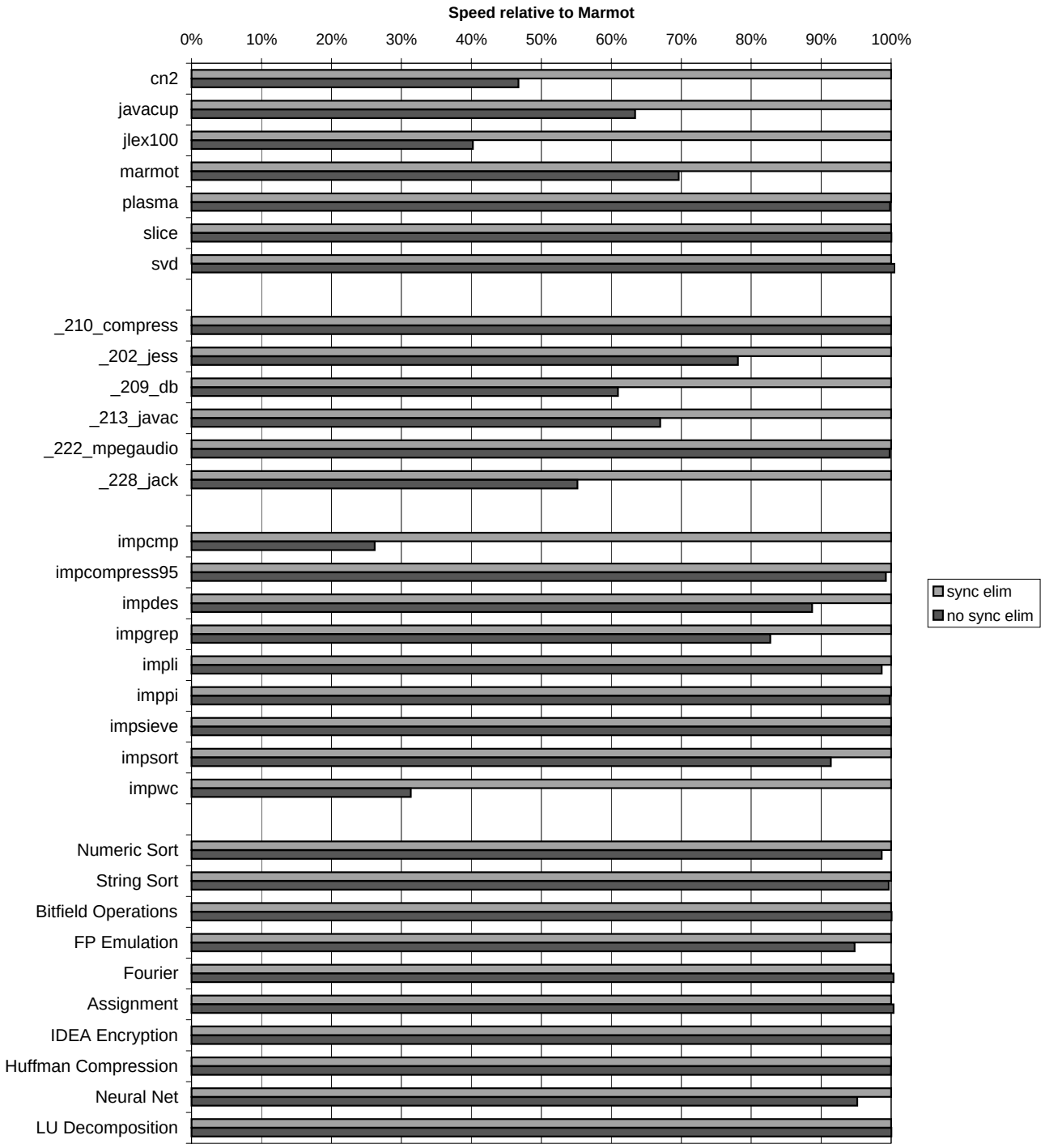

Figure 15: The effect on performance of disabling the synchronization elimination optimization (normalized: no sync elim = 100%).

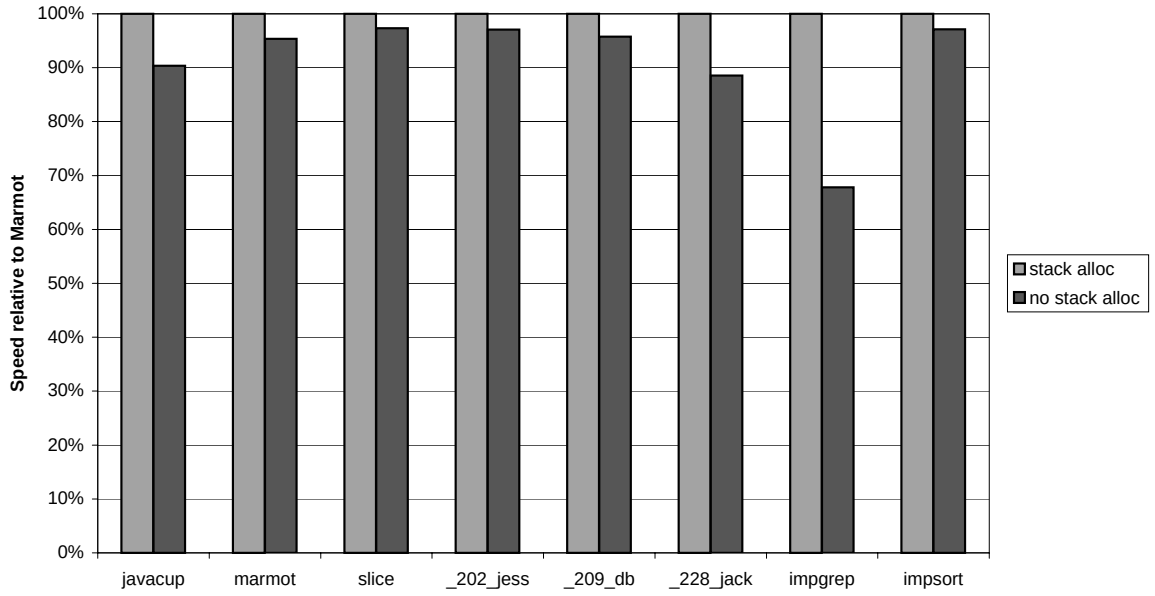

Figure 16: The effect on performance of disabling the stack allocation optimization, for selected benchmarks (normalized: stack alloc = 100%).

of the programs by a median value of 2%. This indicates that the larger programs make more extensive use of the Java libraries, which are heavily synchronized.

Even when little or no lock contention occurs, larger programs often spend substantially more time synchronizing than performing safety checks. Because the Marmot implementation of synchronization is at least competitive with that of other systems, it may also be important to reduce synchronization cost in multithreaded programs.

8.5 Effect of Stack Allocation

The stack allocation optimization reduced program execution time for some non-trivial benchmarks by as much as 11%. Figure 16 shows the effect of disabling the optimization on the performance of these benchmarks, which include two of the small-to-medium sized benchmarks (Marmot and java_cup) and three of the SPEC Client JVM98 programs (_202_jess, _209_db, and _228_jack). The effect on the remaining benchmarks is negligible and is not shown.

8.6 Comparison of Garbage Collectors

For benchmarks that do significant amounts of allocation, we compare the effect on application speed of Marmot's three garbage collectors: a conservative collector, a semi-space copying collector, and a generational collector.

The collectors are configured in the following manner. For the copying and conservative collectors, a collection occurs after every 32 MBytes of data allocation. For the generational collector, the nursery is 2 MBytes in size. Objects larger than 1K are pretenured in the older generation. The older generation is collected after 32 MBytes of data has been copied from the nursery or pretenured.

Figure 17 lists benchmarks that allocate significant amounts of memory and the amounts that they allocate. It excludes benchmarks, such as impcompress, that only allocate large data structures that persist for the entire benchmark run.

Program	Allocation (MBytes)
cn2	56
javacup	29
jlex100	157
marmot	1,353
_201_compress	157
_202_jess	261
_209_db	31
_213_javac	204
_228_jack	113
impli	95

Figure 17: Benchmarks that allocate significant amounts of memory.

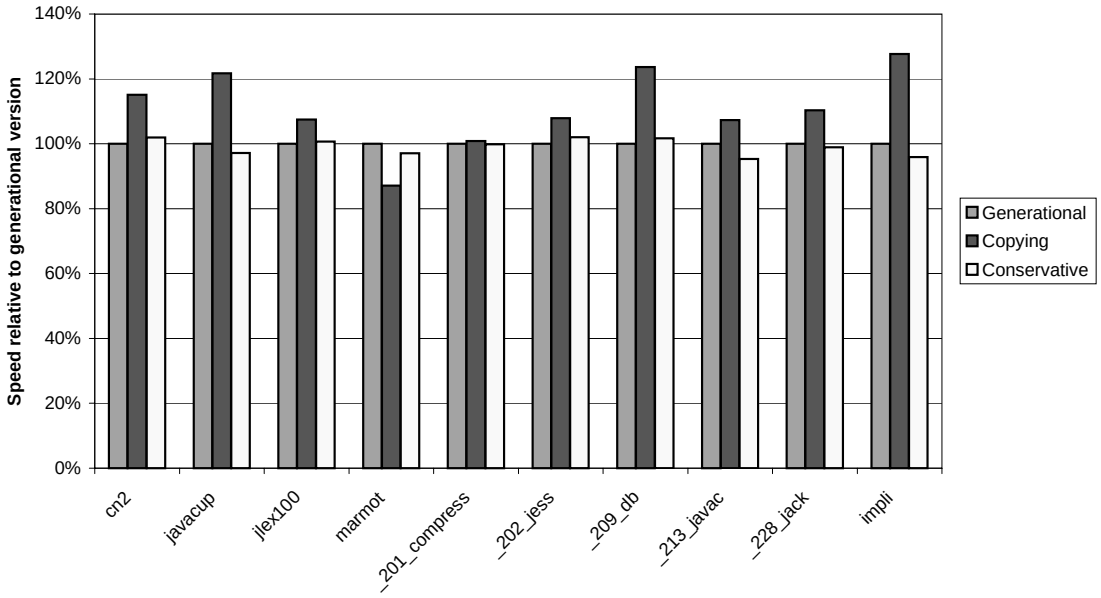

Figure 18: The effect of different garbage collectors on performance, for selected benchmarks (normalized: generational = 100%).

Figure 18 compares performance of the different benchmarks with the various garbage collectors. The benchmarks are run without safety checks. For Marmot, which allocates more memory than any of the other benchmarks, the version that uses the generational garbage collector is fastest. For the rest of the programs, the copying collector is generally the fastest, followed by the generational collector and then the conservative collector. With the current settings, the cost of the generational write barrier exceeds any possible reduction in garbage collection time. The cost of garbage collection using the copying collector is already too low, in the range of 0-3%.

For _213_javac, the only other program besides Marmot for which the cost of copying collection is high, the garbage collection times for the copying collector and the generational collector are about the same. The survival rate of objects allocated in the nursery is quite high (over 20%), which offsets the reduction in collections of the older generation. The additional cost of the write barrier makes the generational collector version slower.

9 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, while the `j2s` from UCSB [KH97] and the distinct `j2s` from University of Colorado [MDG97] both use the SUIF system. The IMPACT NET compiler [HGmWH96, 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 precise garbage collector tuned for Cecil programs; the other systems retrofit conservative garbage collectors into their runtime systems since the systems were not designed for precise 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, and include their own runtime systems.

10 Conclusion

We have described the implementation of Marmot: a native-code compiler, runtime system, and library for Java, and evaluated the performance of a set of Marmot-compiled benchmarks. Because Marmot is intended primarily as a high quality research platform, we initially chose to concentrate on the extension of known, successful imperative and object-oriented language implementation techniques to Java. Similarly, we focused more on ease of implementation and modification than on compilation speed, compiler storage usage, debugging support, library completeness, or other requirements of production systems. The remainder of this section summarizes what we have learned from implementing the Marmot system and from examining the performance of programs compiled by Marmot.

We found Java bytecode to be an inconvenient input language, in that it obscures much of the information present in the Java source code (e.g., type information and the structure of high level operations such as `try-finally`). To generate good code, Marmot is forced to reconstruct missing types and recognize and optimize bytecode idioms.

We were able to apply a large number of conventional scalar and object-oriented optimizations, most of which required extensions to support new Java language features. The features inducing the most significant changes were the precise exception model, which (in the absence of interprocedural effect analyses) severely limited the use of code motion in optimizations, and the multithreaded storage model, which required modifications to storage analyses. Compact modeling of precise exceptions in our SSA-based representation required additional effort. Current limitations on code motion also hinder instruction scheduling.

Overall, Marmot's techniques worked well, yielding application performance at least competitive to that of other Java systems, and approaching that of C++. This suggests that a production-quality compiler for Java could produce code competitive with that produced by production-quality compilers for C++. Marmot optimizations reduced the cost of safety checks to a quite modest level: a median of 4.1% for our benchmarks. Even with an efficient lock implementation, simple synchronization elimination reduces execution time of our

larger benchmarks by a median of 30%. Storage management added significant runtime costs, both inside and outside the garbage collector. Stack allocation reduced program execution time by as much as 11%.

Acknowledgements

We thank David Gay for implementing stack allocation for Marmot and Jacob Rehof for improving the Marmot type elaboration algorithms. We thank Cheng-Hsueh Hsieh and Wen-mei Hwu of the IMPACT team for graciously 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.

[CL97] Keith D. Cooper and John Lu. Register promotion in C programs. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, 1997.

[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.

[DMH92] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992. SIGPLAN, ACM Press.

[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.

[Fre98] Stephen N. Freund. The costs and benefits of java bytecode subroutines. In *Formal Underpinnings of Java Workshop at OOPSLA, 1998.*, 1998.

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

[GH99] Etienne Gagnon and Laurie Hendren. Intra-procedural inference of static types for Java bytecode. Technical Report 1, McGill University, 1999.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[GS99] David Gay and Bjarne Steensgaard. Stack allocating objects in Java. In preparation, 1999.

[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.

[Hal98] Tom R. Halfhill. Benchmarking Java. *BYTE Magazine*, May 1998.

[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.

[HGmWH96] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen mei W. 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.

[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.

[LCK+98] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Lu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 26–37, June 1998.

[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.

[VHK97] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 142–157, October 1997. Published as Proceedings OOPSLA '97, ACM SIGPLAN Notices, volume 32, number 10.

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

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[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.