

Load Elimination in the Presence of Side Effects, Concurrency and Precise Exceptions

Christoph von Praun, Florian Schneider, and Thomas R. Gross

Laboratory for Software Technology
ETH Zürich
8092 Zürich, Switzerland

Abstract. Partial redundancy elimination can reduce the number of loads corresponding to field and array accesses in Java programs. The reuse of values loaded from memory at subsequent occurrences of load expressions must be done with care: Precise exceptions and the potential of side effects through method invocations and concurrent modifications in multi-threaded programs must be considered.

This work focuses on the effect of concurrency on the load optimization. Unlike previous approaches, our system determines accurate information about side effects and concurrency through a whole-program analysis. Partial redundancy elimination is extended to exploit this information and to broaden the optimization scope.

There are three main results: (1) Load elimination is effective even in the most conservative variant without side effect and concurrency analysis (avg. dynamic reduction of loads 23.4%, max. 55.6%). (2) Accurate side effect information can significantly increase the number of optimized expressions (avg. dynamic reduction of loads 28.6%, max. 66.1%). (3) Information about concurrency can make the optimization independent of the memory model, enables aggressive optimization across synchronization statements, and can improve the number of optimization opportunities compared to an uninformed optimizer that is guided by a (weak) memory model (avg. dynamic reduction of loads 28.5%, max. 70.3%).

1 Introduction

A common storage model for object-oriented programs is to allocate objects explicitly and access them indirectly through references. While this model is convenient for the programmer and the memory management (garbage collection), indirect memory accesses may become a performance bottleneck during program execution. Object and array access is done through access expressions.

Objects can refer to other objects, resulting in a sequence of indirect loads, so called *path expressions* (e.g., `o.f1.f2.f3`). The evaluation of a path expression results in a *pointer traversal*, which is a common runtime phenomenon in object-oriented programs. Current processor architectures and memory subsystems are not designed to handle pointer traversals at peak rates, and hence pointer traversals may cause a performance bottleneck.

Standard optimization techniques such as common subexpression elimination (CSE) and partial redundancy elimination (PRE) can be applied to reduce the number of (indirect) loads that are evaluated at runtime. The technique is also known as *register promotion* because non-stack variables are 'promoted' to faster

register memory during a certain period of the program execution. However, the elimination and motion of loads must be done with special care to account for constraints imposed by the programming language:

Aliasing The assignment to a reference field may invalidate a loaded value in the assigning method.

Side effects Method calls may modify objects and hence invalidate loaded values in the caller.

Precise exception semantics The evaluation of an indirect load may raise an exception. Hence code motion that involves indirect loads must not lead to untimely exceptions.

Concurrency In multi-threaded programs, objects may be modified concurrently. The elimination of a load is only admissible if the visibility of concurrent updates, as prescribed in the thread and memory model of the language, is not violated.

Java programs may be affected by all of these aspects. Previous work [6] has employed a simple type-based alias analysis to account for some of these impediments to load elimination but situations that could not be resolved through simple alias information are handled in the most conservative manner. This paper improves upon previous studies by employing a detailed side effect and concurrency analysis. We classify redundant load expressions according to their interaction with aliasing, side effects, exception handling and concurrency, and quantify the consequences of the individual aspects on the effectiveness of PRE.

In the context parallel programs, Lee and Padua [9] define a program transformation as *correct*, if “*the set of possible observable behaviors of a transformed program is a subset of the possible observable behaviors of the original program*”. The possible observable behaviors of a parallel program (and consequently the permissible program transformations) are determined by the memory model. A restrictive memory model like SC would defeat a number of standard reordering optimizations (due to the potential of data races) [12].

One way to handle the problem is to conceive the memory model as weak as possible, allow a large number of behaviors, and hence designate standard transformations that are known from the optimization of sequential programs as “correct” in the context of the parallel program. The design of the revised Java memory model (we refer to this model as JMM) follows this strategy [11, Appendix A].

Our technique pursues a different strategy that is independent of the memory model, i.e., its correctness does not rely on a consistency weaker than SC. Concurrency analysis determines a conservative set of variables and access sites with *access conflicts*. A conflict is found, if the analysis cannot determine start/join or monitor-style synchronization for read/write accesses to the same data from different threads. The number of conflicting variables and access sites is typically moderate and such sites are exempted from the optimization. For the remaining accesses, the synchronization strategy is known (determined by the concurrency analysis) and a number of aggressive optimizations are possible that would not be performed if the optimizer only followed the minimal constraints of the memory model.

The main difference between (1) the conventional application of standard optimizations constrained by the memory model and (2) our approach based

```

... = s1.f;
<call to synchronized method>
... = s1.f;

```

Fig. 1. Program fragment illustrating uninvolvement synchronization

on concurrency analysis is as follows: Approach (1) applies optimization to all loads disregarding the potential of sharing or access conflict. Access to volatile variables or synchronization kill the availability of previous loads. Approach (2) is conservative about loads of variables with access conflicts and in addition puts fewer constraints on the availability of load expressions. Hence, approach (2) enables a number of optimizations that are rejected by approach (1). Consider the program in Figure 1.

Approach (1) would abstain from optimizing the second load expression due to the intervening synchronization (kill). Approach (2) can determine that the object referenced by `s1` is not conflicting (it is thread-local or accesses are protected by enclosing monitor synchronization); hence the synchronization that occurs in the call is uninvolvement in protecting variable `s1.f` and hence the second load can be optimized.

2 Example

Figure 2 reviews several simple control-flow graphs that illustrate a classification of load-redundancies. We assume that local variables follow static single assignment constraints and that – unless explicitly specified – there are no side effects on the involved objects. The classification applies to occurrences of expressions and is not exclusive, i.e., a single expression occurrence could fall into several categories. Partial and full redundancies can be affected by updates due to aliasing, side effects, or concurrency:

- (a) **Loss of redundancy due to aliasing:** The expression in block (4) is fully redundant wrt. the syntactically equivalent expression in block (1). However, the aliasing of local variables `p` and `o` in combination with the update in block (3) invalidates this redundancy.
- (b) **Loss of redundancy due to side effect:** Assume that the call in block (3) has a potential side effect on the object referenced by `o`. Hence, the redundancy of the load in block (4) wrt. the load in block (1) is lost due to a side effect since the call occurs on some control-flow path between the redundant expressions.
- (c) **Loss of redundancy due to precise exceptions:** The load in block (4) is partially redundant wrt. the load in block (2). This redundancy could be avoided by hoisting the expression from block (4) to the end of block (3). Such code motion would however violate precise exception semantics: The evaluation of the access expression `o.f` may throw a `NullPointerException`. Assume that code motion is performed (`o.f` is hoisted above the assignment to variable `a`) and the hoisted expression throws an exception at runtime; then the update to `a` would not be performed and hence would not be visible in the handler. Thus, code motion to avoid partial redundancies must not bypass updates that should be visible in some handler. The occurrence of

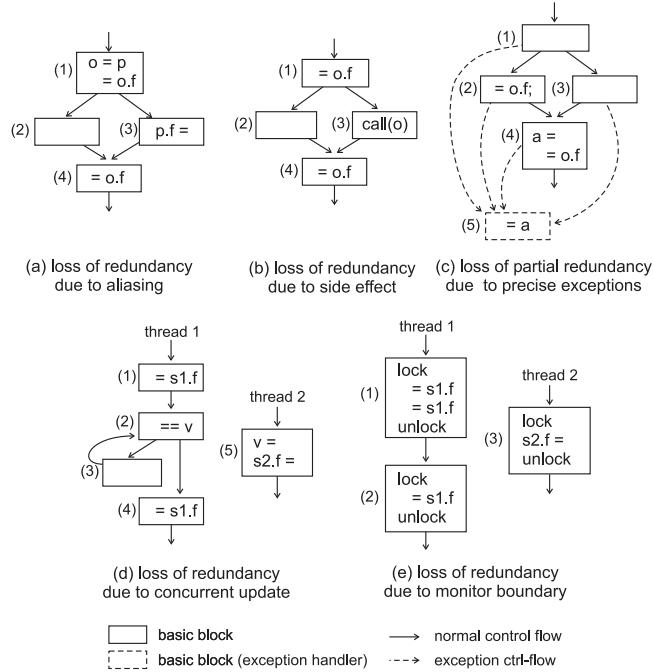


Fig. 2. Possible losses of full and partial redundancies

`o.f` in block (4) is hence a partial redundancy that is lost due to precise exceptions and must not be optimized.

Redundancies like the load in block (4) can be optimized in combination with speculative code motion and compensation code that is executed in the case of an exception [5]. This approach is further discussed in Section 5.

- (d) **Loss of redundancy due to concurrent update:** Let `v` be a volatile variable. The load in block (4) is fully redundant wrt. the load in block (1). There is however an intervening load of a volatile variable that enforces that updates of thread 2 become available to thread 1. This includes an update of field `f` on the object referenced by `s1` and `s2`, and hence the elimination of the load expression in block (4) should not be performed. The redundancy in block (4) is lost due to a concurrent update, because its elimination might lead to the phenomenon that the update of thread 2 might not become visible to thread 1.
- (e) **Loss of redundancy due monitor boundary:** There is a full redundancy of the second load in block (1) and the load in block (2) wrt. the first load in block (1). Accesses to the object referenced by `s1` and `s2` are guarded through locks, hence there is no access conflict. The elimination of loads must nevertheless be done with special care: The second load in block (1) can be optimized. The load in block (2) however must not be eliminated due to the potential update done in thread 2.

In cases (a), (b), (d) and (e) a full or partial redundancy can be lost. In case (c), it is the opportunity to optimize a partial redundancy that is lost. In cases (d) and (e), redundancy is lost irrespective of the memory model, i.e., an optimization would not be permissible in SC and weaker models.

3 PRE of Path Expressions

PRE is a well-known technique to reduce the number of load expressions, and Chow et al. [2] describe a practical approach (SSAPRE) to use PRE with an SSA intermediate representation. SSAPRE was originally developed in the context of translating C/C++ programs. Java programs must obey precise exception semantics, and the language prescribes a thread model, so a compiler that wants to eliminate (some) path expressions for this language must handle the situations illustrated in Section 2. We describe here an algorithm that builds on SSAPRE of [2] to remove load operations. This algorithm requires additional whole-program analyses that provide information about aliasing, side effects and concurrency.

The outline of the algorithm is as follows:

1. Transform the program such that covert redundancies are revealed (3.1).
2. Compute alias information (Section 3.2).
3. Determine side effects at all call sites (Section 3.3).
4. Compute escape information and conflicting fields (Section 3.4).
5. Perform partial redundancy elimination (Section 3.6).

3.1 Program Transformations

The detection of redundancies through the compiler is enhanced by two program transformations: First, *method inlining* allows the intraprocedural algorithm to operate to some degree across method boundaries. Second, *loop peeling* allows to hoist loop invariant expressions and can significantly reduce the dynamic frequency of loads inside loops.

3.2 Alias Analysis

The alias analysis is based on global value numbering, where value sets are associated (1) with local variables and parameters, (2) with fields, and (3) with array variables. A value number is created at an allocation site and flows into the value set of the variable holding the reference to the allocated object. Value numbers are propagated in a flow-insensitive, interprocedural manner along a variable-type analysis (VTA) that is described in detail in [14].

Value numbers approximate may-alias information: Two reference variables may refer to the same object at runtime if the intersection of their value number sets is not empty.

3.3 Side Effect Analysis

The purpose of the side effect analysis is to determine at a specific call site if the callee updates objects that are referenced in available load expressions in the caller. If a side effect is determined, the call site kills the availability of the respective load expression.

The update effects and the aliasing relationships introduced by the callee are encoded in a *method summary*, which is computed separately for each method in a bottom up traversal of the call graph. A summary contains abstractions of the objects that are accessed or allocated in the dynamic scope of a method and the reference relationships among those objects induced through field variables. Updates are specified per object and do not differentiate individual fields. The concept and computation of method summaries is described in more detail in Ruf [13]; the method also accounts for recursion.

A method summary encodes updates in a generic form and this information has to be adapted to individual contexts where the method is called. At a call site, the actual parameters and the objects reachable through those are unified with the corresponding formal parameters of the summary. Caller-side aliasing is approximated through the may-alias information computed in Section 3.2. At this point, the embedded method summary provides a conservative approximation about the objects that are modified in the callee at a specific call site.

3.4 Concurrency Analysis

Concurrency analysis determines for a specific access expression if the accessed object is *shared* and if there are potential *access conflicts* on field variables. A variable is subject to a conflict if there are two accesses that are not ordered through enclosing monitor synchronization and at least one access is an update. In our model, we limit the optimization to loads that access data without conflicts.

In addition to access conflicts, certain statements may necessitate a reload of a variable even if the variable is not subject to an access conflict. Such statements are called *killing statements* because they kill the availability of preceding load expressions.

First, two methods for determining the absence of access conflicts are discussed, then the notion of killing statements and their computation is defined.

Stack-Locality In a simple approximation, concurrent access can be excluded for those objects that remain confined in the scope of their allocating method and hence are not made available to other threads. Such abstract objects are called *stack-local*. Stack-locality can be computed similar to update effects (Section 3.3): Instead of abstract objects that are updated, sets of objects that *escape* from the stack are noted in the method summary.

Stack locality is however a very conservative approximation to the property of ‘having no conflicting access’: First, the definition applies to whole objects instead of individual field variables. Second, there is typically a significant number of objects that escape the stack and are nevertheless not shared or subject to conflicting access [16].

Object Use Analysis The object use analysis determines the set of field variables that are regarded as conflicting. A context-sensitive symbolic program execution is used to track accesses to fields in different object and locking contexts. Synchronization patterns like “init then shared read”, thread start/join, and monitors can be recognized and allow to determine the absence of conflicts for a large number of variables and statements. A detailed description of the object use analysis is presented in [16].

Given the set of fields on which potential conflicts arise, the load elimination refrains from optimizing loads of such fields through reference variables that are stack-escaping. If arrays are subject to conflicts accesses to escaping arrays are not optimized.

Kill Analysis Kill information specifies if a statement necessitates to reload previously loaded values from memory. Our load optimization is intraprocedural and only targets objects that have no conflicts. For shared variables that are protected by a monitor, a reload is necessary if the scope of the protecting monitor is temporarily quit (... other threads could enter the monitor and update the loaded variable). In Java, there are two cases in which a monitor can be temporarily left at the method scope: First, at the boundary of a block monitor; second, at a call site of `Object::wait` or callers of it. Access ordering on shared variables can also be guaranteed by thread start and join; hence calls to `Thread::start` and `Thread::join` are also considered as killing. The kill analysis determines such killing statements in a single pass over the caller hierarchy of the program.

3.5 Exceptions

The potential of exceptions narrows the flexibility of code motion when eliminating partial redundancies. *Precise exception semantics* in Java demand the following behavior in the case of an exception:

- All updates prior to the excepting statement must appear to have taken place.
- Updates that follow the excepting statement must not appear to have taken place.
- Program transformation must not change the order of thrown exceptions.

To account for these semantics, a simple program traversal identifies *potentially excepting instructions* (PEI) [5]: explicit raise of exception, indirect loads, memory allocation, synchronization, type check, and calls. We assume that PEIs are the only source of exceptions (*synchronous exceptions*) and do not account for Java’s asynchronous exceptions that are raised at very severe error conditions (e.g., machine error, lack of memory) and often hinder the further execution of a thread or program.

Information about PEIs is used to restrain load elimination, such that indirect load expressions are not hoisted above PEIs or assignments to local variables, if the current method defines an exception handler.

3.6 PRE

This section elaborates on the modifications to the SSAPRE algorithm [2] to account for side effects, concurrency and precise exception semantics. The algorithm is driven by a worklist of expressions and optimizes one expression at a time; compound expressions (e.g., access path expressions) are split and handled in the appropriate order.

SSAPRE performs the following steps:

- Initialize worklist with candidate expressions.
- Insert availability barriers.
- While worklist not empty do:
 1. Φ -Insertion
 2. Rename
 3. DownSafety
 4. WillBeAvail
 5. Finalize
 6. CodeMotion
 7. If there are new expressions then:
 - Add new expressions to worklist.
 - Determine availability barriers.

The detailed description of each of these steps can be found in [2]. We describe the changes to the original algorithm in the following paragraphs. The steps (5), (6) and (7) remain unchanged.

Candidate Expressions The PRE implementation described here optimizes three types of expressions:

- Arithmetic expressions
- Scalar loads (static field accesses)
- Indirect loads (non-static field and array accesses)

Optimization candidates are determined during the collection phase, where expressions that cannot be optimized due to an access conflict are filtered out. If a program is single-threaded, there are no further constraints. For multi-threaded programs, the results of the escape and concurrency analysis inhibit the optimization of a load expression if

- the base object is globally visible (for direct loads, this is always true, for indirect loads, all stack-escaping objects are assumed to be globally visible), and
- there is a conflict on the accessed field.

Accesses to volatile variables are also excluded from the optimization.

Availability Barriers A second pass over the program determines statements that kill the availability of an expression, i.e., call sites with side effects or potentially aliased stores (may-defs). There are two cases that render an access expression invalid: (1) An update of the base reference variable; (2) An update of the accessed field variable.

Updates of the callee that are visible in the caller are determined from the method summary information. (Section 3.3). If a side effect is detected, the affected expression must be invalidated and a so called *kill-occurrence* of that expression is inserted at the call-site; this is done for both cases of invalidation, (1) and (2). Similarly, *kill-occurrences* are inserted at stores that are potentially aliased to the base reference variable of an expression (case (1)).

The availability of load expression is also killed by a store through the same reference variable to the same field (must-def, case(2)). In this case however, a so called *left-occurrence* [10] is inserted. Since such a store is a definition, a *left-occurrence* makes an expression available, so that subsequent loads of the same variable are redundant wrt. this store.

For PEIs and other statements with side effects that are potentially visible in a local exception handler or outside the current method, an *exception-side-effect occurrence* is inserted; such occurrences do not specify a potential update, but only serve to prohibit optimization.

Φ -Insertion Φ -nodes for an expression are inserted at the iterated dominance frontier (IDF) of each real occurrence. Additionally, Φ s are inserted at the IDF of left occurrences and kill occurrences since these may change the value of an expression.

Rename The *rename* step builds the *factored redundancy graph* [2] (FRG) and assigns version numbers to each occurrence of an expression. This is done in a pre-order pass over the dominator tree. Left occurrences always get a new version number, whereas kill occurrences simply invalidate the current version. PEIs have no impact on the version numbers. Multiple expression occurrences with the same version number indicate redundancy.

Exception Safety To account for the restrictions of Java’s exception semantics, the computation of safe insertion points for expressions needs to be extended. In the original algorithm, *down-safety* is a sufficient criterion for the insertion of E at a position S in the program. Here, we add the concept of *exception-safety*, which needs to be satisfied in addition to *down-safety* for expressions E that contain PEIs (e.g., indirect loads, division).

A position T in a program is *exception-safe* with respect to an expression occurrence E iff there is no critical statement on any path from T to E . Critical statements are:

- PEIs
- Stores to escaping objects
- Stores to local variables that are visible inside a local exception handler.

The *exception-safe* flag is initialized along the *rename*-step. For each critical statement, the immediately dominating Φ -node is determined and marked as *not*

Table 1. Benchmarks, * = multi-threaded benchmark

Name	Description	LOC
moldyn*	Molecular dynamics simulation (Java Grande Forum)	864
montecarlo*	Monte Carlo simulation (Java Grande Forum)	3132
mtrt*	Multi-threaded raytracer (JVM98)	3821
tsp*	Traveling Salesmen Problem	705
db	Memory resident database (JVM98)	1087
compress	Modified Lempel-Ziv compression (JVM98)	952
jess	Java Expert Shell System (JVM98)	10441

exception-safe. In a second step, the *exception-safe* flag is propagated upward beginning at the Φ -nodes that are initially *not exception-safe*. If an operand of such a Φ is defined by another Φ we mark the defining Φ as *not exception-safe*. This is done recursively until no more Φ s are reached.

For example, the insertion point at the end of block (3) in Figure 2(c) is not exception-safe with respect to the expression occurrences in block (4) because there is an assignment with side effect that is visible inside the exception handler. In the scenario of Figure 2(b), the occurrence of `o.f` in block (4) is fully redundant. In that case a thrown exception does not impose any restrictions because there is no insertion of `o.f` on a new path.

WillBeAvail This step determines at which points an expression will be made available by inserting code. There are two modifications related to precise exception semantics in this step: First, for partially available Φ -nodes marked as *not exception safe* the *willBeAvail* flag is reset so that there will be no code motion that violates the exception semantics. Second, expressions that are partially available at the beginning of an exception handler must be invalidated. Hence the *willBeAvail* flag of partially available Φ -nodes at the beginning of an exception handler is reset.

4 Evaluation

We implemented the modified SSAPRE in a Java-X86 way-ahead compiler and report here on its effectiveness. The runtime system is based on GNU libgcj version 2.96 [4]. The numbers we present in the static and dynamic assessment refer to the overall program including library classes, but excluding native code. The effect of native code for aliasing and object access is modeled explicitly in the compiler.

The efficiency of the optimization has been evaluated for several single- and multi-threaded benchmarks, including programs from the SPEC JVM98 [15] and the Java Grande Forum [7] suite (Table 1).

The benchmarks have been compiled in four variants:

- (A) **No side effect and no concurrency analysis:** Every method call is assumed to have side effects and all loads are optimized according to the JMM, i.e., all synchronization barriers invalidate the availability of loads. Alias information is used to disambiguate the value of reference variables. This is

Table 2. Static number of optimized load expression occurrences. Multi-threaded benchmarks are marked with *

program	(A)	(B)	(C)		(D)
	#exprs	#exprs (B/A)	#exprs (C/A)	(C/B)	#exprs (D/C)
moldyn*	665	727 ₁ 109.3%	248 ₁ 37.3% ₁ 34.1%		785 ₁ 316.5%
montecarlo*	234	301 ₁ 128.6%	334 ₁ 142.7% ₁ 111.0%		349 ₁ 104.5%
mtrt*	311	597 ₁ 192.0%	630 ₁ 202.6% ₁ 105.5%		656 ₁ 104.1%
tsp*	245	297 ₁ 121.2%	313 ₁ 127.8% ₁ 105.4%		324 ₁ 103.5%
compress	296	375 ₁ 126.7%	434 ₁ 146.6% ₁ 115.7%		434 ₁ 100.0%
db	281	346 ₁ 123.1%	495 ₁ 176.2% ₁ 143.1%		495 ₁ 100.0%
jess	669	807 ₁ 120.6%	1232 ₁ 184.2% ₁ 152.7%		1232 ₁ 100.0%
average		₁ 131.7%	₁ 145.3% ₁ 109.6%		₁ 132.7%

the most conservative configuration of our algorithm and could, e.g., be implemented in an optimizing JIT compiler. This configuration resembles the analysis in [6].

- (B) **Side effect but no concurrency analysis:** Side effects are determined at method calls, and all loads are optimized according to the JMM.
- (C) **Side effect and concurrency analysis:** Precise information about side effects and concurrency guide the optimization. Loads of variables that are not conflicting are aggressively optimized across synchronization statements; loads of conflicting variables are not optimized. The resulting optimization is correct with respect to an SC memory model.
- (D) **Upper limit for concurrency analysis:** In case (C), some optimizations might be defeated because conservatism in the conflict analysis may classify too many variables as conflicting. The artificial configuration (D) constitutes an upper bound on the optimization potential that could be achieved by a “ideal synchronization analysis”: All variables are assumed to be free of conflicts and there are no synchronization barriers. The optimization might be incorrect in this configuration and hence we present only static counters (Table 2).

4.1 Number of Optimized Expressions

The comparison of column (A) and (B) in Table 2 shows the improvement due to the side effect analysis: For `mtrt`, `montecarlo`, and `compress` the analysis is most effective and many calls to short methods that have only local effects can be identified. The increase of optimization opportunities for PRE ranges between 9.3% for `moldyn` and 92.0% for `mtrt`, with an average of 31.7%.

The concurrency analysis (Table 2, column (C)) increases the number of optimization opportunities for all but one benchmarks in comparison to the version that optimizes according to the JMM (Table 2 column (B)). For the single-threaded programs the improvement is generally a bit higher (15.7% to 52.7% for `compress` resp. `jess`) because there are no conflicting accesses that prevent load elimination. For the multi-threaded benchmarks the improvement is between 5.4% (`tsp`) and 11% (`montecarlo`). The average improvement due to concurrency analysis is 9.6%. This average value is reduced through a loss of

Table 3. Dynamic count of loads in millions of memory accesses

program	orig	(A)		(B)		(C)	
	#ops	#ops	(A/orig)	#ops	(B/orig)	#ops	(C/orig) (C/B)
moldyn*	1651.3	973.4 ₁	58.9%	972.6 ₁	58.9%	1409.5 ₁	85.4% ₁ 144.9%
montecarlo*	478.6	212.6 ₁	44.4%	162.5 ₁	33.9%	142.3 ₁	29.7% ₁ 87.6%
mtrt*	366.9	364.7 ₁	99.4%	333.4 ₁	90.9%	333.4 ₁	90.9% ₁ 100.0%
tsp*	899.0	671.5 ₁	74.7%	671.5 ₁	74.7%	674.3 ₁	75.0% ₁ 100.4%
compress	2423.5	1901.4 ₁	78.5%	1712.5 ₁	70.7%	1692.8 ₁	69.9% ₁ 98.8%
db	446.6	393.6 ₁	88.1%	393.6 ₁	88.1%	300.5 ₁	67.3% ₁ 76.3%
jess	323.5	299.1 ₁	92.4%	267.1 ₁	82.6%	265.8 ₁	82.2% ₁ 99.5%
average			76.6%		71.4%		71.5% ₁ 101.2%

optimization opportunities in `moldyn` in variant (C): A cycle in the heap shape abstraction leads to unnecessary conservatism in the conflict analysis such that a number of actually thread-local object are classified as conflicting. A simple change in the source code of the program could improve the precision of the analysis such that variant (C) would provide 9% more optimization opportunities than (B).

Column (D) shows the theoretical upper bound for a perfect concurrency analysis. Apart from `moldyn`, the results of variant (C) for the multi-threaded benchmarks are within 5% of that upper bound. For the single-threaded benchmarks the concurrency analysis provides already “perfect” information, hence there is no further improvement.

Note that the concurrency analysis is only effective in combination with the side effect analysis: If method calls kill all available expressions, then not much is gained through the reduction of synchronization barriers provided by the concurrency analysis. Overall, the combined side effect and concurrency analysis increases the number of optimized occurrences for all programs but `moldyn`, ranging from 27.8% (`tsp`) up to 102.6% (`mtrt`), with an overall average of 45.3%.

4.2 Dynamic Count of Load Operations

Table 3 specifies the dynamic number of loads in different variants of the benchmarks. The base variant (A) is already quite successful in reducing the number of loads (avg. 23.4%, max. 55.6%). Side effect information improves the reduction further (avg. 28.6%, max. 66.1%). Most successful is again variant (C) which achieves a reduction from 9.1% for `mtrt` up to 70.3% for `montecarlo` (avg. 28.5%).

For single-threaded benchmarks, the concurrency analysis determines the absence of concurrency and hence creates additional optimization opportunities compared to variant (B). The benefit is most pronounced for `db` where variant (C) executes only 76.3% (last column of Table 3) of the loads of the variant without concurrency information (B).

For the benchmarks `mtrt`, `compress` and `jess`, Table 2 shows an increase in the number of optimized expressions among variant (B) and (C). This effect is not manifested at runtime, i.e., the last column in Table 3 specifies no reduction in the number of dynamic loads. There are two reasons for this behavior: First, the execution frequency of optimized expressions can vary greatly, i.e., few optimized expressions can contribute to a majority of the dynamic savings. Second, variant

(C) might optimize other expressions than variant (B) and consequently the overall dynamic effect can be different.

Similarly for `tsp`: Side effect analysis produces only an insignificant reduction and concurrency analysis even results in a slight increase of dynamic load operations. In this benchmark, the concurrency analysis classifies a frequently accessed variable as conflicting and hence the configuration (C) is conservative about corresponding loads. Configuration (B) in contrast allows to optimize the respective load expressions, resulting in an overall dynamic benefit over configuration (C).

`modly` is again hampered by the loss of optimization opportunities due to spurious conflict reports. The dynamic situation is worse than the static situation because exactly those fields that are falsely assumed to be conflicting (hence are not optimized in variant (C)) are most frequently accessed.

The speedup of execution time compared to the unoptimized version ranges between 0% and 12% (avg. 8%).

5 Related Work

There are numerous contributions in the field of program analysis and optimization for Java and related programming languages. We discuss only a selection of approaches that are closely related to our work.

Side Effect Analysis Modification side effect analysis for C programs has been done by Landi, Ryder, and Zhang [8]. The analysis computes a precise set of modified abstract storage locations for call sites and indirect store operations. It is difficult to compare this work with ours: On the one hand, Java has a more uniform storage model, on the other hand, polymorphism and loose type information in object-oriented languages necessitate conservatism when approximating side effects at compile-time.

Clausen [3] developed an interprocedural side effect analysis for Java bytecodes and demonstrates its utility for various optimizations like dead code removal and common subexpression elimination. Side effects are specified by field variables that a method and its callees might modify. The computational complexity of this analysis is lower than ours, however the analysis is not sensitive to the heap-context of a method call and hence is less precise. In contrast to Clausen's work, our analysis does not distinguish individual fields, but merely specifies updates for specific abstract objects (including field information would however be a straightforward extension).

Optimization in the Presence of Precise Exceptions Optimizing Java program in the presence of exceptions has been studied by Gupta, Choi, and Hind [5]. Since common Java programs contain many PEIs (about 40%), many optimization opportunities are prohibited by the dependencies created through Java's precise exception model. However [5] observes that the visibility of updates in exception handlers can be limited to a few variables that are live. Hence liveness information helps to significantly reduce the number of dependencies and enables reordering transformations. Reordering of PEIs remains however critical as, in case an exception occurs, the order of thrown exceptions must not

be altered. For preserving the correct order, compensation code is introduced that triggers the correct exception that would have been thrown by the unoptimized code. [5] is orthogonal to our work, because liveness analysis could also be used to enable optimization that our algorithm neglects due to conservatism assumptions. Gupta, Choi and Hind use the relaxed dependency model to enable loop transformations with aggressive code motion; Our focus is on PRE for load elimination where code motion is only required to handle partial redundancies. Hence the overall limitations of precise exception semantics in our work is not as pronounced as in [5].

Load Elimination Lo et al. [10] have developed an algorithm for eliminating direct and indirect load operations in C programs, promoting memory that is accessed through these operations to registers. The algorithm is based on their previous work on SSAPRE [2], which we also use as a foundation. The authors employ aggressive code motion, relying on speculative execution and hardware support to mask potential exceptions; this is possible, because the C programming language does not require precise exception semantics. In addition to eliminating loads, the authors have defined SSU (static single use) form which allows to eliminate stores through the dual of the SSAPRE algorithm. Bodik, Gupta, and Soffa also explore PRE-based load elimination in [1] and consider, besides syntactical information, also value-number and symbolic information to capture equivalent loads. The focus of our study is on object-oriented programs (Section 4). Compared to the C routines that have been investigated in [10, 1], the size of methods is usually smaller and the call-interaction among methods is more vivid. Hence our work emphasizes the modeling of side effects through procedure interaction and concurrency.

PRE has also been used by Hosking et al. [6] to eliminate access path expressions in Java; program transformation is done at the level of bytecodes. Similar to our evaluation, the authors achieve a clear reduction of load operations (both static and dynamic counts). Contrary to our work, the authors do not use whole-program information to determine inter-procedural side effects. Moreover, our evaluation clarifies the impact of precise exceptions, interprocedural side effects and concurrency on the effectiveness of load elimination.

Lee and Padua [9] adopt standard reordering transformations to parallel programs and describe caveats and limitations. Similar to our approach, a particular program analysis and IR are used to determine the interaction of threads on shared data (concurrent static single assignment form, CSSA) and to conclude on restrictions of the optimization. Their algorithms handle programs with structured parallelism (SPMD) and hence allow for a precise analysis and selective optimization of accesses to variables with access conflicts. Our approach addresses general programs with unstructured parallelism but is more conservative in the treatment of loads from conflicting variables.

6 Conclusions

There are three main results: (1) Load elimination is effective even in the most conservative variant without side effect and concurrency analysis (avg. dynamic reduction of loads 23.4%, max. 55.6%). (2) Accurate side effect information

can significantly increase the number of optimized expressions (avg. dynamic reduction of loads 28.6%, max. 66.1%). (3) Information about concurrency can make the optimization independent of the memory model, enables aggressive optimization across synchronization statements, and can improve the number of optimization opportunities compared to an uninformed optimizer that is guided by a (weak) memory model (avg. dynamic reduction of loads 28.5%, max. 70.3%).

7 Acknowledgements

We thank Matteo Corti for his contributions to our compiler infrastructure and the anonymous referees for their useful comments. This research was supported, in part, by the NCCR “Mobile Information and Communication Systems”, a research program of the Swiss National Science Foundation, and by a gift from the Microprocessor Research Lab (MRL) of Intel Corporation.

References

- [1] R. Bodik, R. Gupta, and M. Soffa. Load-reuse analysis: Design and evaluation. In *Proc. PLDI'99*, pages 64–76, 1999. 403
- [2] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. Partial redundancy elimination in SSA form. *ACM TOPLAS*, 21(3):627–676, May 1999. 394, 397, 398, 403
- [3] L.R. Clausen. A Java bytecode optimizer using side-effect analysis. In *ACM Workshop on Java for Science and Engineering Computation*, June 1997. 402
- [4] GNU Software. gcj - The GNU compiler for the Java programming language. <http://gcc.gnu.org/java>, 2000. 399
- [5] M. Gupta, J. Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In *Proc. ECOOP'00*, pages 422–446, June 2000. LNCS 1850. 393, 396, 402, 403
- [6] A. Hosking, N. Nystrom, D. Whitlock, Q. Cutts, and A. Diwan. Partial redundancy elimination for access path expressions. *Software Practice and Experience*, 31(6):577–600, May 2001. 391, 400, 403
- [7] Java Grande Forum. Multi-threaded benchmark suite. <http://www.epcc.ed.ac.uk/javagrande/>, 1999. 399
- [8] W. Landi, B.G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. *ACM SIGPLAN Notices*, 28(6):56–67, 1993. 402
- [9] J. Lee, D. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *Proc. PPOPP'99*, pages 1–12, May 1999. 391, 403
- [10] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proc. PLDI'98*, pages 26–37, 1998. 398, 403
- [11] J. Manson and B. Pugh. JSR-133: Java Memory Model and Thread Specification. <http://www.cs.umd.edu/~pugh/java/memoryModel>, 2003. 391
- [12] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In D. Padua, editor, *Proc. ICPP'90*, pages 105–113, Aug. 1990. 391
- [13] E. Ruf. Effective synchronization removal for Java. In *Proc. PLDI'00*, pages 208–218, June 2000. 395
- [14] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *OOPSLA'00*, pages 264–280, Oct. 2000. 394

- [15] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. <http://www.spec.org/osg/jvm98>, 1996. 399
- [16] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proc. PLDI'03*, pages 115–128, June 2003. 395, 396