Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs

Rajkishore Barik and Vivek Sarkar
Department of Computer Science, Rice University
Email: {rajbarik,vsarkar}@rice.edu

***Abstract*—Load elimination is a classical compiler transformation that is increasing in importance for multi-core and many-core architectures. The effect of the transformation is to replace a memory access, such as a read of an object field or an array element, by a read of a compiler-generated temporary that can be allocated in faster and more energy-efficient storage structures such as registers and local memories (scratchpads). Unfortunately, current just-in-time and dynamic compilers perform load elimination only in limited situations. In particular, they usually make worst-case assumptions about *potential side effects* arising from *parallel constructs* and *method calls*. These two constraints interact with each other since parallel constructs are usually translated to low-level runtime library calls.**

In this paper, we introduce an interprocedural load elimination algorithm suitable for use in dynamic optimization of parallel programs. The main contributions of the paper include: a) an algorithm for load elimination in the presence of three core parallel constructs – *async*, *finish*, and *isolated*, b) efficient side-effect analysis for method calls, c) extended side-effect analysis for parallel constructs using an Isolation Consistency memory model, and d) performance results to study the impact of load elimination on a set of standard benchmarks using an implementation of the algorithm in Jikes RVM for optimizing programs written in a subset of the X10 v1.5 language. Our performance results show decreases in dynamic counts for getfield operations of up to 99.99%, and performance improvements of up to 1.76× on 1 core, and 1.39× on 16 cores, when comparing the algorithm in this paper with the load elimination algorithm available in Jikes RVM.

***Keywords*-Load elimination; scalar replacement; parallel program; dynamic compilation; dynamic optimization; memory model.**

I. Introduction

The computer industry is at a major inflection point in its hardware roadmap. Unlike previous generations of hardware evolution, the shift towards multicore and manycore computing will have a profound impact on software — not only will future applications need to be deployed with sufficient parallelism for manycore processors, but the parallelism must also be energy-efficient. For decades, caches have helped bridge the memory wall for programs with high spatial and temporal locality. Unfortunately, caches come with an energy cost that limit their use as on-chip memory in future manycore processors. It is therefore desirable for programs to use more energy-efficient storage structures such as registers and local memories (scratchpads) instead of caches, as far as possible.

Load elimination [1]–[4] is a classical compiler transformation that is increasing in importance for multi-core and many-core architectures. The effect of the transformation is to replace a memory access, such as a read of an object field or an array element, by a read of a compiler-generated temporary that can be allocated in registers or local memories. This transformation has also been referred to as *scalar replacement* in past work [5]. Unfortunately, current just-in-time and dynamic compilers perform load elimination only in limited situations. Specifically, there are two major challenges that need to be overcome to make load elimination more broadly applicable in dynamic compilation. First load elimination must be performed *interprocedurally* i.e., must be extended to take into account the side effects of method calls. Second, load elimination must be *parallelism-aware* i.e., must be extended to take parallel constructs into account. The two challenges interact with each other since parallel constructs are usually translated to low-level runtime library calls which need special treatment by an interprocedural optimizer.

The main contributions of this paper include:

- support for load elimination in the presence of three core parallel constructs – *async*, *finish*, and *isolated*.
- efficient side-effect analysis for method calls.
- extended side-effect analysis for parallel constructs using an Isolation Consistency memory model that establishes the legality of our load elimination transformation. constructs.
- performance results to study the impact of load elimination on a set of standard benchmarks using an implementation of the algorithm in Jikes RVM [6] for optimizing programs written in a subset of the X10 v1.5 language [7]. Our performance results show decreases in dynamic counts for getfield operations of up to 99.99%, and performance improvements of up to 1.76× on 1 core, and 1.39× on 16 cores, when comparing the algorithm in this paper with an earlier load elimination algorithm.

These contributions have been designed and implemented in the context of the Jikes RVM dynamic optimizing compiler.

The rest of the paper is organized as follows. Section II describes the parallel execution model assumed for this work, which is derived from the X10 v1.5 language [7]. Section III describes our approach to side effect analysis of

parallel constructs and method calls, and introduces our load elimination algorithm. Section IV presents our experimental results, Section V discusses related work, and Section VI contains our conclusions.

II. Parallel Execution Model

The full X10 language contains multiple parallel constructs that were introduced for high productivity [7]. In Section II-A, we summarize three core parallel constructs in the X10 v1.5 subset that are supported by the work presented in this paper: *async*, *finish*, and *isolated*. These constructs can be used to support higher level constructs such as *foreach* and *ateach*. For this subset, there is no significant difference between v0.41 of the language specification summarized in [7], and v1.5 of the language used in our work. Both versions are based on Java and include the core constructs studied in this paper. However, as advocated in [8], we use the *isolated* keyword instead of *atomic* to make explicit the fact that the construct supports weak isolation rather than strong atomicity. Since the X10 v1.5 specification does not include a complete memory model definition, we also summarize in Section II-B the *Isolation Consistency* memory model assumed in this paper.

A. X10 Subset

In this section, we summarize three key X10 constructs — async, finish, and isolated. For simplicity, we will restrict our attention to *single-place parallel programs* in this paper. In a single-place X10 program, all activities execute in the same place, and have uniform read and write access to all shared data, as in multithreaded Java programs where all threads operate on a single shared heap. However, our approach is applicable to multi-place parallel programs as well — the only extension required is to ensure that *BadPlaceException*'s [7] are properly handled by the compiler analysis. An important safety result in X10 is that any program written with async, finish, and isolated can never deadlock. We now briefly describe the three constructs below.

1) async ⟨stmt⟩: Async is the X10 construct for spawning a new asynchronous task or *activity*. The statement, *async* ⟨*stmt*⟩, causes the parent activity to create a new child activity to execute ⟨*stmt*⟩. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to the statement following the async.

2) finish ⟨stmt⟩: The X10 statement, *finish* ⟨*stmt*⟩, causes the parent activity to execute ⟨*stmt*⟩ and then wait till all sub-activities created within ⟨*stmt*⟩ have terminated globally. There is an implicit *finish* statement surrounding the main program in an X10 application. If async is viewed as a fork construct, then finish can be viewed as a join construct. However, the async-finish model is more general than the fork-join model [7].

Consider the following X10 code example in which the main program activity use the *async* statement to create a child activity that executes the *for-i* loop to compute `oddSum.val` while it proceeds in parallel to execute the *for-j* to compute `evenSum`[1]. The main activity uses the `finish` construct to ensure that `oddSum.val` is fully computed before printing the total sum obtained by adding `oddSum.val` and `evenSum`:

```
finish {
  // Compute oddSum in child activity
  async for (int i=1; i<=n; i+=2) oddSum.val+=f(i);
  // Compute evenSum in parent activity
  for (int j=2; j<=n; j+=2) evenSum+=f(j);
} // finish
System.out.println("Sum = " + (oddSum.val+evenSum));
```

As discussed in the next section, the Isolation Consistency memory model is weak enough to allow `oddSum.val` to be allocated to a register during the execution of the entire *for-i* loop in this example.

3) isolated ⟨stmt⟩, isolated ⟨method-decl⟩: The isolated construct is our renaming of X10's *atomic* construct. As stated in [7], an atomic block in X10 is intended to be "executed by an activity as if in a single step during which all other concurrent activities in the same place are suspended". This definition implies a *strong atomicity* semantics for the atomic construct. However, all X10 implementations that we are aware of (including the one used in this paper) use a single lock per place to enforce mutual exclusion of atomic blocks. This approach supports weak atomicity, since no mutual exclusion guarantees are enforced between computations within and outside an atomic block. As advocated in [8], we use the *isolated* keyword instead of *atomic* to make explicit the fact that the construct supports weak isolation rather than strong atomicity.

B. Isolation Consistency Memory Model

There is a range of memory models that have been studied in the literature including Sequential Consistency (SC) [9], the Java Memory Model (JMM) [10], the OpenMP memory model [11], and Location Consistency (LC) [12]. It is well known that all these models yield the same semantics for data-race-free programs, but may exhibit different semantics for parallel programs with races. A major research challenge lies in dealing with the common case when a compiler (especially a dynamic compiler) does not know for sure that the input parallel program is data-race-free. To address this case, we define a weak memory model, *Isolation Consistency* (IC), for which the load elimination optimizations described in this paper are guaranteed to be correct even in the presence of data races. They will also be correct for any data-race-free parallel program with a stronger memory model, but the optimizations may not be correct for parallel programs with data races that must obey a stronger memory model.

[1]Function f is assumed to be a pure function of its input i, and to involve sufficient computation granularity to ensure that the async overhead is insignificant in these examples, for n=10000.

```
1:  A a = new A ();
2:  a.f = ...;
3:  async { ... }
4:  ... = a.f;
    // Can reuse a.f from Stmt 2
```

(a) Case 1

```
1:  final A a = new A ();
2:  a.f = ...;
3:  async { while(...) a.f = F(a.f); }
4:  ... = a.f;
    // Can reuse a.f from Stmt 2
```

(b) Case 2

```
1:  final A a = new A();
2:  a.f = ...;
3:  finish async { a.f = 2; ... }
4:  ... = a.f;
    // Can reuse a.f from Stmt 3
```

(c) Case 3

```
1:  final A a = new A();
2:  a.f = ...
3:  async { isolated if (...) a.x++; }
4:  ... = a.f;
    // Can reuse a.f from Stmt 2
```

(d) Case 4

Figure 1. Four parallel code fragments that demonstrate load elimination opportunities in the presence of parallel constructs

The definition of *Isolation Consistency* builds on the operational definition of Location Consistency in [12] in which an *abstract interpreter* models the state of each shared location as a *partially ordered multiset* (pomset) of write operations. In any execution that satisfies the LC model, the result returned by a read operation R must belong to the *value set* of the location *i.e.,* it must have been written by a write operation that is a "most recent predecessor write" with respect to R in the pomset or a write operation that is unrelated to R in the pomset. However, the LC model also placed the restriction that the abstract interpreter executes each instruction in a thread in its original order, thereby ensuring that causality is not violated. The reader is referred to [12] for more details on the LC model.

In Isolation Consistency (IC), we assume that only the control and data dependences within a thread need to be preserved in the abstract interpreter. Thus the abstract interpreter is allowed to execute instructions out-of-order within a thread so long as intra-thread dependences are not violated. These intra-thread dependences are defined using a *weak atomicity* model [13] which ensures the correct ordering of load and store operations from multiple threads when they occur in isolated sections. For load and store operations that occur outside an isolated section, the only inter-thread ordering constraints arise from the "happens-before" relationships enforced by the finish construct.

Consider the four example parallel code fragments shown in Figure 1. Cases 1 and 2 demonstrate the potential for load elimination across async constructs, Case 3 across a finish construct, and Case 4 across isolated constructs. In each case, we want to know if the load of `a.f` in statement 4 can be eliminated by substituting the value of a prior store operation. (The ... notation represents computations that do not contain accesses to any instances of field `f`.) The IC model permits load elimination in all four cases, but that is not the case for the SC and JMM models. Cases 1 and 4 have no data races, but for the non-Isolation Consistency memory models, the onus is on the compiler to establish that there are no data races in those cases.

Case 1 appears to be an easy case because the async body is assumed to not perform any access to field `f`. Both the JMM and IC models permit load elimination of the `a.f` getfield operation in statement 4 by using the value stored in statement 2. However, an additional *delay set analysis* [14] is necessary for the SC model to ensure that there is no other access to field f elsewhere in the program that could contribute to a cycle and result in an execution that is potentially inconsistent with the SC model. Delay set analysis is a time-consuming whole program analysis that will be impractical for use in a dynamic optimizing compiler.

In Case 2, there is a potential data race between the conditional store of `a.f` in statement 3 and the load in statement 4. With the Isolation Consistency model, the compiler can conclude that the value stored in `a.f` in statement 2 will always be part of the value set for the load in statement 4, therefore making it legal to perform a load elimination accordingly. The SC and JMM models will not permit load elimination in this case, but the OpenMP [11] model will.

Case 3 demonstrates the scope of eliminating loads across *finish* boundaries. In this case, the load in statement 4 may not be eliminated with respect to statement 2. The `finish` scope in statement 3 demarcates the completion of the execution of the async body in statement 3 and hence is visible to the rest of the program.

Case 4 shows the effect of load elimination in the presence of *isolated* constructs. The load in statement 4 cannot be eliminated in the SC and JMM models due to the `isolated` construct. However, if we can analyze the side effect of the isolated construct, we should able to eliminate the load in statement 4. In this case,the async only updates field `a.x`. Hence, eliminating the load of `a.f` in statement 4 is safe in the Isolation Consistency model.

III. Load Elimination in the presence of Parallel Constructs: Async, Finish, Isolated

As mentioned earlier, current just-in-time and dynamic compilers perform load elimination only in limited situations. In this paper, we use the FKS load elimination

algorithm by Fink, Knobe, and Sarkar [2] as a baseline for comparison. This algorithm is based on Array SSA form, and has been implemented in the Jikes RVM dynamic optimizing compiler. Like many other optimization algorithms for dynamic compilers, the FKS algorithm conservatively assumes that each procedure call may contain a def and a use for every heap variable accessed in the program. As is well known from past work on static interprocedural analysis, including the seminal paper by Banning [15], this level of imprecision can be improved by analyzing the body of the called procedure and inserting appropriate defs and uses for only the field accesses that may occur in the called procedure (and the procedures that it calls). This simple technique of computing *side-effects* by analyzing the called procedure gets complicated in the presence of *parallel constructs*, and is challenging to perform in a dynamic compilation environment due to its overhead. In this section, we describe how we extend the FKS algorithm to incorporate side effects from both method calls and parallel constructs. There is a natural interplay between both, since parallel constructs are usually translated to low-level runtime library calls in the intermediate representation level at which load elimination is performed. Section III-A discusses a simple flow-insensitive side-effect analysis algorithm suitable for analyzing method calls in a dynamic compilation environment. Section III-B describes how the side-effect analysis algorithm is extended for the async, finish, and isolated parallel constructs. Section III-C presents the overall algorithm for load elimination. Finally, Section III-D discusses two compiler transformations that create more opportunities for load elimination and improved register allocation.

A. Side-Effect Analysis of Method Calls

Consider the code fragment shown in Figure 2. The load statement on line 8 cannot be eliminated by intraprocedural load elimination, due to the lack of knowledge of the effects of the method call `setNothing()` in line 7. In contrast, a load elimination algorithm based on interprocedural side-effect analysis can determine that the method call `setNothing()` does not have any side-effects, thereby realizing the opportunity for eliminating the load in line 8. We also observe that the load in line 10 cannot be eliminated by total redundancy elimination, but is a good candidate for partial redundancy elimination (PRE). The implementation described in this paper currently does not support PRE because of the complexity of combining PRE with Java's and X10's precise exception semantics, and leaves it as a topic for future work.

In this section, we first summarize the *heap array* representation introduced in [2] for field accesses in strongly-typed languages like Java. We then describe our proposed approach for computing interprocedural side effects using the heap array representation.

```
1: class foo {
2:   int f;
3:   void setField  (int n) { this.f = n; }
4:   void setNothing () {}
5:   void bar (foo a) {
6:     a.f = 4;
7:     a.setNothing ();
8:     ... = a.f; // Can we eliminate this load?
9:     if (C) a.setField (3);
10:    ... = a.f; // Can we eliminate this load?
11:  }
12:}
```

Figure 2. Example: Interprocedural side-effect information can enable the load in line 8 to be removed. The load in line 10 cannot be fully removed when condition C is statically unknown.

1) Heap Array Representation: As described in [2], accesses to object fields and array elements[2] in the program can be represented using hypothetical *heap arrays* that are compile-time abstractions of the runtime heap. Each object field x in the program is abstracted by a distinct heap array, H^x. H^x represents all the instances of field x in the heap. A `GETFIELD` of $a.x$ is represented as an use of element $H^x[a]$, and a `PUTFIELD` of $b.x$ is represented as a def of element $H^x[b]$. The use of heap arrays ensures that field x is considered to be the same across instances of two different static types T_1 and T_2, if (say) T_1 is a subtype of T_2. We say that $H^x[a]$ and $H^x[b]$ are *definitely same* (*DS*) if valuenum(a) = valuenum(b), where valuenum(a) represents the value number associated with a after the global value numbering pass.

2) Method Level Side-effect: As discussed earlier, the goal of interprocedural side-effect analysis is to determine for each call site, a safe approximation of the side effects that the method involved at that call site may have. This recursively includes any side effects of the methods called from that site. We capture this using the generalized flow-insensitive side-effect formulation proposed by Banning [15]. The generalized flow-insensitive side-effects of a method are represented using *GMOD* and *GREF* sets. *GMOD*(m) denotes the set of heap arrays whose value *may* be modified either directly or indirectly, as a result of the invocation of method m. Similarly, *GREF*(m) denotes the set of heap arrays whose value *may* be inspected or referenced either directly or indirectly, as a result of the invocation of method m. In Banning's formulation, *MOD* and *REF* sets are defined for specific call sites and were computed using both the parameter bindings at the call site and the *GMOD* of the callee. Since our analysis uses the heap array representation for modeling side effects, we do not need to pay special attention to parameter bindings.

In general, determining the target of a method call can be complicated in the presence of virtual methods calls and dynamic class loading. However, since X10 does not share Java's dynamic class loading semantics, we can separate the

[2]As mentioned earlier, this paper focuses on load elimination for field accesses only. Extensions for array accesses is a subject for future work.

```
1: void main() {
2:   p.x = ...
3:   s.w = ...
4:   finish { //finish_main
5:     async { //async_main
6:       p.x = ...
7:       isolated { ... = q.y; . . . q.y = ...; }
8:       ... = p.x
9:     }
10:    foo()
11:  }
12:  ... = p.x
13:  ... = s.w
14:}
```

```
15: void foo() {
16:   async bar() //async_foo
17:   isolated { ... = q.y; . . . q.y = ...; }
18:   ... = s.w
19: }
20: void bar() {
21:   r.z = ...
22:   ... = r.z
23: }
```

Figure 3. Example X10 program for side effect analysis in the presence of parallel constructs.

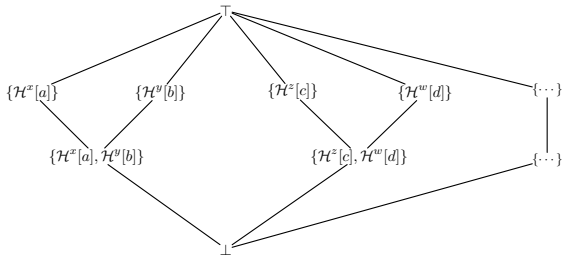

Figure 4. Lattice for heap array GMOD and GREF sets

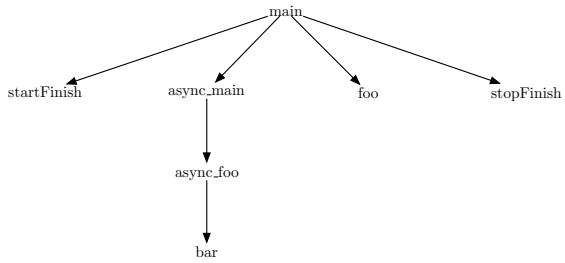

Figure 5. Call Graph for Example Program in Figure 3

X10 classes from the Java classes and assume that it is safe to pre-load X10 classes. Specifically, we determine the target of a call to an X10 method as follows. First, we check if the method call has been resolved and has exactly one target. Second, we check if the method can be resolved using the existing set of classes loaded in the VM. Third, we trigger loading of the X10 class if necessary to resolve the target. Finally, for virtual calls, we use whatever type information we have available for the `this` parameter to try and resolve the call to a single target. If the above steps do not yield a single unique target, we conservatively propagate $\perp$ as summaries for the given method. Merging side effects from multiple targets is a subject for future work. Currently, we limit our attention to X10 classes only, and conservatively propagate $\perp$ for all methods in Java classes.

The complete lattice for heap array *GMOD* and *GREF* sets is shown in Figure 4, with lattice ordering defined by the subset relationship.

B. Extended Side-effect Analysis for Parallel Constructs

Consider the example program shown in Figure 3, which will serve as a running example to demonstrate side-effect analysis of parallel constructs in X10 programs. The example program has a `main` method that invokes two asyncs (one in line 5 and another in line 16 via the call to `foo`) and awaits for their termination using the finish construct that spans lines 4-11. Both the asyncs use *isolated* constructs to perform read-modify-write operations on the shared object field $q.y$. The call graph for the example program is shown in Figure 5. Using the method level side-effect analysis described in Section III-A, *GMOD*(bar) and *GREF*(bar) can be computed as $\{H^z[r]\}$.

We describe our proposed side-effect analysis for *finish* constructs, methods with *escaping async's*, and *isolated* constructs in subsections III-B1 – III-B3 respectively, and then present the complete interprocedural side-effect analysis algorithm in Section III-B4 and Figure 6.

1) Side Effects for Finish Scopes: Finish scopes in X10 impose the constraint that any async created within its scope must be completed before the statement after the finish scope is executed. Compiler optimizations such as code motion must pay attention to finish scope boundaries as it may be incorrect in general, to perform code motion into the body of the finish scope or out of the finish scope without knowing the effect of the finish scope. Hence, we introduce $FMOD(f)$ and $FREF(f)$ to represent the set of heap arrays modified and referenced within a finish scope f respectively. The *GMOD* and *GREF* sets for any method invoked within a finish scope f, either directly or indirectly, is propagated to the finish scope by unifying them with the $FMOD(f)$ and $FREF(f)$ sets respectively. Each dynamic instance of an X10 statement has a unique *Immediately Enclosing Finish* (IEF) instance [16]. In our static analysis, we define $IEF(s)$ to be the closest enclosing finish scope for statement s in the same method. $IEF(s)$ is undefined if s does not have an enclosing finish statement in the same method.

Consider the method `main` in Example 3. The finish scope encompasses the side effects of all the meth-

ods and asyncs invoked within it. Ignoring the *isolated* constructs on line 7 and 17 (which will be discussed later), the *FMOD*(finish_main) can be computed as $\{H^x[p], H^z[r]\}$. Similarly, *FREF*(finish_main) is computed as $\{H^x[p], H^z[r], H^w[s]\}$.

2) Side Effects for Methods with Escaping Asyncs: X10 permits methods with *escaping asyncs* i.e., asyncs that have no enclosing finish scopes in the same method. We define an *async-escaping* method as: 1) a method which contains an *async* invocation that is not enclosed in a *finish* scope, or 2) a method which invokes another *async-escaping* method that is not wrapped in a finish scope. The *GMOD* and *GREF* sets for async-escaping methods are propagated to their enclosing finish scopes as their termination is guaranteed only at the end of the enclosing finish scopes. We introduce escaping *EMOD* and *EREF* sets along with *GMOD* and *GREF* to handle async-escaping methods. Async-escaping methods continue to be async-escaping in the call graph chain until an *IEF* is encountered. Note that the interprocedural side effects of non-escaping async's will be collected by normal side-effect analysis of methods.

The method `foo` in Example 3 invokes an async on line 16 that is not wrapped in a finish scope and is an async-escaping method. The *EMOD*(foo) and *EREF*(foo) are computed as $\{H^z[r]\}$.

3) Side Effects for Isolated Blocks: The *isolated* synchronization primitive enforces mutual exclusion among async's. To enforce the Isolation Consistency memory model described in Section II-B, we introduce *IMOD* and *IREF* sets that represent all the heap arrays modified and referenced respectively across all isolated blocks in the program. Note that, this is an overly conservative approximation as some of the *isolated* blocks may never execute in parallel with other *isolated* blocks due to a "happens-before" relationship. Further refinement of *IMOD* and *IREF* sets using *May-Happen-in-Parallel* (MHP) information [17] is a subject for future work.

The isolated blocks on lines 8 and 21 modify and reference heap array $H^y[q]$. Hence, $IMOD = IREF = \{H^y[q]\}$.

4) Overall Side-Effect Analysis Algorithm: The overall side-effect analysis algorithm in the presence of finish, async, and isolated constructs is presented in Figure 6. This algorithm is designed to be performed on the Java code produced by the X10 compiler, which (for X10 v1.5) translates each *async* construct to a *runAsync* call in the Java-based X10 runtime, which in turn calls the *runX10Task* method in an inner class that contains the body of the async. Further, every *finish* scope is translated into a pair of *startFinish()* and *stopFinish()* runtime calls.

For statements/methods executed in isolated blocks, we unify the *IMOD* and *IREF* sets using the meet operator $\bigvee_i$. $\bigvee_i$ is a conditional meet operation which is performed only if the current statement/method call is in an isolated block. Note that the X10 language does not permit any usage

Require: Method m and its *IR*.
Ensure: Compute side-effect summaries for m and its called procedures. Gather *IMOD* and *IREF* for isolated blocks. Return $GMOD(m)$ and $GREF(m)$.

1: Initialize information for method m; $GREF(m) = \top$ and $GMOD(m) = \top$
2: $inProgress(m) = true$
3: **for all** instruction I in *IR* **do**
4: **case** I:
5: **GETFIELD/GETSTATIC** $a.f$: resolve the target of the field access $a.f$
6: $GREF(m) = GREF(m) \bigvee \{H^f[a]\}$
7: $IREF = IREF \bigvee_i \{H^f[a]\}$
8: **PUTFIELD/PUTSTATIC** $a.f$: resolve the target of the field access $a.f$
9: $GMOD(m) = GMOD(m) \bigvee \{H^f[a]\}$
10: $IMOD = IMOD \bigvee_i \{H^f[a]\}$
11: **CALL** $p()$: resolve the target of the method access p
12: **if** the target of p is unknown or has several targets **then**
13: $GREF(m) = \bot$ and $GMOD(m) = \bot$
14: $EREF(m) = \bot$ and $EMOD(m) = \bot$
15: **else if** the target of p is *startFinish* **then**
16: $FMOD(IEF(I)) = \top$ and $FREF(IEF(I)) = \top$
17: **else if** the target method is *stopFinish* **then**
18: $GMOD(m) = GMOD(m) \bigvee FMOD(IEF(I))$
19: $GREF(m) = GREF(m) \bigvee FREF(IEF(I))$
20: **else if** the target method is *runAsync* **then**
21: Determine the target *runX10Task*, t
22: Obtain $GMOD(t)$ and $GREF(t)$ by invoking $ParallelSideEffectAnalysis(t)$ {recursive call}
23: **if** $IEF(I)$ is *undefined* **then**
24: $EMOD(m) = EMOD(m) \bigvee GMOD(t) \bigvee EMOD(t)$
25: $EREF(m) = EREF(m) \bigvee GREF(t) \bigvee EREF(t)$
26: **else**
27: $FMOD(IEF(I)) = FMOD(IEF(I)) \bigvee GMOD(t) \bigvee EMOD(t)$
28: $FREF(IEF(I)) = FREF(IEF(I)) \bigvee GREF(t) \bigvee EREF(t)$
29: **end if**
30: **else if** $inProgress(p)$ is set OR $GMOD(p)$ and $GREF(p)$ sets are available OR recursively invoke $ParallelSideEffectAnalysis(p)$ for p **then**
31: $GMOD(m) = GMOD(m) \bigvee GMOD(p)$
32: $GREF(m) = GREF(m) \bigvee GREF(p)$
33: $IMOD = IMOD \bigvee_i GMOD(p)$
34: $IREF = IREF \bigvee_i GREF(p)$
35: **if** $IEF(I)$ is *undefined* **then**
36: $EMOD(m) = EMOD(m) \bigvee EMOD(p)$
37: $EREF(m) = EREF(m) \bigvee EREF(p)$
38: **else**
39: $FMOD(IEF(I)) = FMOD(IEF(I)) \bigvee EMOD(p)$
40: $FREF(IEF(I)) = FREF(IEF(I)) \bigvee EREF(p)$
41: **end if**
42: **end if**
43: **esac**
44: **end for**
45: $inProgress(m) = false$
46: return $GMOD(m)$ and $GREF(m)$

Figure 6. *ParallelSideEffectAnalysis*(m): Side-effect analysis in the presence of X10 parallel constructs for method m

$GMOD(\text{bar}) = GREF(\text{bar}) = \{H^z[r]\}$
$EMOD(\text{bar}) = EREF(\text{bar}) = \top$
$GMOD(\text{async\_foo}) = GREF(\text{async\_foo}) = \top$
$EMOD(\text{async\_foo}) = EREF(\text{async\_foo}) = \{H^z[r]\}$
$GMOD(\text{foo}) = \top$ and $GREF(\text{foo}) = \{H^w[s]\}$
$EMOD(\text{foo}) = EREF(\text{foo}) = \{H^z[r]\}$
$FMOD(\text{startFinish}) = FREF(\text{startFinish}) = \top$
$GMOD(\text{async\_main}) = GREF(\text{async\_main}) = \{H^x[p]\}$
$EMOD(\text{async\_main}) = EREF(\text{async\_main}) = \top$
$FMOD(\text{stopFinish}) = \{H^x[p], H^z[r]\}$
$FREF(\text{stopFinish}) = \{H^x[p], H^z[r], H^w[s]\}$
$GMOD(\text{main}) = GREF(\text{main}) = \{H^x[p], H^z[r], H^w[s]\}$
$EMOD(\text{main}) = EREF(\text{main}) = \top$
$IMOD = IREF = \{H^y[q]\}$

Table I
SIDE-EFFECT RESULTS FOR EXAMPLE PROGRAM SHOWN IN FIGURE 3

of `async` or `finish` constructs in the body of isolated sections [7].

The algorithm presented in Figure 6 walks over the *IR* in a flow-insensitive manner and considers different cases for each instruction *I*. For example, if *I* represents a field access, then the access is unified with the method's *GREF* or *GMOD* set as shown in lines 6 and 9. At a *stopFinish* function call, *FMOD* and *FREF* are merged into the *GMOD* and *GREF* sets for the current caller *m* (as shown in lines 18-19). For the *runAsync* function call, we determine the side effects of the target *runX10Task* method and unify them in the caller's enclosing finish scope's side effects as shown in lines 23-29. If the *runAsync* method call was not enclosed in a finish scope, the side effect sets of *runX10Task* are unified with the *EMOD* and *EREF* for the caller (this is shown in lines 24-25). Lines 31-41 account for normal method calls (not related to parallel constructs).

For the example program shown in Figure 3 and its corresponding call graph in Figure 5, the final side-effect sets are shown in Table I.

C. Extended Load Elimination Algorithm

Once side-effects for methods and parallel constructs are computed, we need to incorporate them into the load elimination algorithm that obeys the Isolation Consistency memory model. Figure 7 contains the complete load elimination algorithm in the presence of parallel constructs. Steps 2-17 determine the type of method call based on the parallel constructs and inserts appropriate pseudo-def and pseudo-use instructions for their *GMOD* and *GREF* sets. Each entry into the *isolated* block is annotated with pseudo-defs to fields in *IMOD*. This prohibits any load reuse in the *isolated* block for fields that may be modified in any isolated scope. Each exit of an isolated construct is annotated with pseudo-uses of fields in *IREF*. This permits loads to be eliminated in and after the isolated block exit. *startFinish* and *runAsync* method calls are handled by side-effect analysis and act as a no-op for the load elimination algorithm. At *stopFinish*,

Require: Method *m* and its *IR*
Ensure: Transformed *IR* after interprocedural load elimination.
1: Compute side-effect summary information by invoking *ParallelSideEffectAnalysis*(*m*).
2: **for all** instruction *I* in *IR* **do**
3: **case** *I*:
4: **isolatedenter**: Insert pseudo-defs for each field in *IMOD* at *I*.
5: **isolatedexit**: Insert pseudo-uses for each field in *IREF* at *I*.
6: **startFinish**: no-op.
7: **stopFinish**: Insert pseudo-defs for each field in $FMOD(IEF(I))$; Insert pseudo-refs for each field in $FREF(IEF(I))$.
8: **runAsync**: no-op.
9: **CALL p**:
10: **if** target of *p* is an *isolated* method **then**
11: Insert pseudo-defs for each field in *IMOD* before *I*.
12: Insert pseudo-uses for each fields in *IREF* after *I*.
13: **else**
14: Insert pseudo-defs and pseudo-uses for each field in $GMOD(p)$ and $GREF(p)$ respectively at *I*.
15: **end if**
16: **esac**
17: **end for**
18: Construct extended array ssa form for each heap operand access including the pseudo-def and pseudo-use accesses introduced above.
19: Perform global value numbering to compute *definitely-same* (*DS*) and *definitely-different* (*DD*) relations.
20: Perform data flow analysis to propagate uses to defs: Create data flow equations for ϕ, $d\phi$, and $u\phi$ nodes. Iterate over the data flow equations until a fixed point is reached.
21: Perform load elimination: For a load of a heap operand, if the value number of the associated heap operand is *available*, then replace the load instruction.

Figure 7. Interprocedural load elimination algorithm in the presence of parallel constructs of X10

pseudo-def and pseudo-use instructions are added for *FMOD* and *FREF* finish summary sets of the current finish scope. Other normal method calls insert pseudo-def and uses for *GMOD* and *GREF* summary sets if the target of the method call is not an isolated method. Otherwise, pseudo-defs for fields in *IMOD* and pseudo-uses for fields in *IREF* are inserted before and after the method call.

Steps 18-19 in Algorithm 7 first construct an extended array ssa form representation of the *IR* over which a global value numbering is performed to compute object accesses that may be *definitely-same* (*DS*) or *definitely-different* (*DD*). In Step 20, a data flow analysis is performed that propagates uses of heap operands to their definition points. Finally, actual load elimination is performed by replacing the memory load operation by a compiler generated temporary in cases where the load is already fully available. The steps 18-21 are described in details in [2].

D. Additional Transformations

We perform two additional compiler transformations that create more opportunities for load elimination and improved register allocation:

1) *Loop-invariant getfield code motion pre-pass*: In general, a loop-invariant `getfield` operation cannot be moved out of a loop since it may throw a `NullPointerException`. To address this case,

we perform the standard transformation of replacing a while loop by a zero-trip test and a repeat-until loop so as to enable loop-invariant code motion of `getfield` operations while still preserving exception semantics. This transformation is performed as a pre-pass to load elimination. We use the side-effect analysis described in Section III-B for method calls inside the loop to determine if a `getfield` operation is loop-invariant.

2) *Live-range splitting post-pass*: a potential negative impact of load elimination is that increasing the size of live ranges can lead to increased register pressure. This in turn may cause a performance degradation if the register allocator does not perform live-range splitting. Since the Linear Scan register allocator in Jikes RVM currently does not split live ranges, we introduce a live-range splitting pass after load elimination that only splits live-ranges of the scalar temporaries introduced by our optimizations. The live-ranges of these scalars are split around all call instructions and loop entry-exit regions. This creates smaller scalar live-ranges for which spilling and register assignment decisions can be made separately. However, in some cases, this benefit can be undone by the register allocator if it decides to coalesce the live ranges back before allocation.

IV. Experimental Results

We present an experimental evaluation of the load elimination algorithm introduced this paper for a set of programs written in the subset of X10 consisting of the async, finish and isolated parallel constructs. The performance results were obtained using Jikes RVM 3.0.0 [6] on a 16-core system that has four 2.40GHz quad-core Intel Xeon processors running Red Hat Linux (RHEL 5). The system has 30GB of memory.

For our experimental evaluation, we use the *production* configuration of Jikes RVM with the following options:`-X:aos:initial_compiler=opt -X:irc:O0`. By default, Jikes RVM does not enable SSA based HIR optimizations like load elimination at optimization level O0. We modified Jikes RVM to enable the SSA and load elimination phases at O0. However, since the focus of this paper is on optimizing application classes, the boot image was built with load elimination turned off and the same boot image was used for all execution runs reported in this paper. The *ParallelSideEffectAnalysis* procedure presented in Figure 6 was implemented as an HIR optimization pass in the `OptimizationPlanner`, and the new load elimination algorithm from Figure 7 was implemented as an extension to the existing load elimination algorithm in Jikes RVM based on the FKS algorithm [2].

All results were obtained using the `-Xmx2000M` JVM option to limit the heap size to 2GB, thereby ensuring that the memory requirement for our experiments was well below the available memory on the 16-core Intel Xeon SMP. The `PLOS_FRAC` variable in `Plan.java` was set to $0.4f$ for all runs, to ensure that the Large Object Size (LOS) was large enough to accommodate all benchmarks. The main program was extended with a five-iteration loop within the same Java process for all JVM runs, and the best of the five times was reported in each case. This approach was chosen to reduce the impact of dynamic compilation time and other virtual machine services in the performance comparisons.

For our experiments, we used the five largest X10 programs that we could find — three Section 3 Java Grande Forum (JGF) benchmarks (Moldyn, RayTracer, Montecarlo) and two NAS Parallel (NPB) benchmarks (CG and MG). All JGF benchmarks were run with the largest data size available. Sizes "A" and "W" were used for CG and MG respectively, to ensure completion in a reasonable amount of time. For all runs in this paper, we set the `NUMBER_OF_LOCAL_PLACES` runtime option for X10 to 1 to obtain a single-place configuration, and also set `INIT_THREADS_PER_PLACE` to the number of worker threads (k) used in the evaluation. All executions used the work-sharing X10 v1.5 runtime scheduling system described in [18].

The five X10 benchmarks listed above use `finish` and `async` constructs, but not `isolated`. To evaluate our optimization in the presence of `isolated` constructs, we created a hybrid X10+Java version of SPECjbb2000 benchmark that uses the `async`, `finish` and `isolated` constructs from X10, but also uses the `CyclicBarrier.await()` construct from Java (which was modeled as an unknown method call in our analysis).

Experimental results are reported for the following cases: 1) *1-thread NOLOADELIM* – Baseline measurement with *no load elimination* and a single worker thread; 2) *k-thread FKS LOADELIM* – use of the *FKS load elimination* algorithm [2] with no side effect analysis and k worker threads; 3) *k-thread FKS+TRANS LOADELIM* – – use of the *FKS load elimination* algorithm [2] with the two *transformation* passes from Section III-D but no side effect analysis, and k worker threads; 4) *k-thread PAR LOADELIM* – – use of the extended *parallel load elimination algorithm* from this paper (Figure 7) with side effect analysis and k worker threads; 5) *k-thread PAR+TRANS LOADELIM* – – use of the extended *parallel load elimination algorithm* from this paper combined with the two *transformation* passes from Section III-D and k worker threads. In this study, the results for 2), 3), 4), and 5) were restricted to elimination of getfield operations only. Extension of these results for array-load operations is a subject for future work.

Table II reports the compile-time results for various passes. The total compilation time for *PAR+TRANS LOADELIM* is on average 1.38× slower than *FKS+TRANS LOADELIM* and ranges from 1.22× (for SPECjbb2000) to 1.47×

Benchmark	NO LOADELIM	FKS+TRANS LOADELIM			PAR+TRANS LOADELIM			
	Total Comp. time in ms	ssa+loadelim time in ms	trans time in ms	Total Comp. time in ms	sideeffect time in ms	ssa+loadelim time in ms	trans time in ms	Total Comp time in ms
CG-A	461	277	75	811	102	398	84	1137
MG-W	574	336	98	989	131	442	110	1348
Moldyn-B	263	194	35	493	76	255	47	673
Raytracer-B	275	157	35	468	77	246	44	670
Montecarlo-B	273	156	35	469	90	253	44	692
SPECjbb2000	4336	1099	232	5625	580	1153	329	6867

Table II
COMPILATION TIMES IN MILLISECONDS OF VARIOUS JIKES RVM PASSES FOR NPB BENCHMARKS (CG, MG), JGF BENCHMARKS (MOLDYN, RAYTRACER, MONTECARLO) AND SPECJBB2000 BENCHMARK

(for Montecarlo). This modest increase in compile-time establishes that the interprocedural load elimination algorithm introduced in this paper is practical for use in dynamic compilation.

Table III reports the dynamic number of GETFIELD operations for different load elimination algorithms. We observe that *PAR+TRANS LOADELIM* reduces the dynamic GETFIELD counts for all benchmarks in the range of 31.93% for Raytracer and 99.99% in CG compared to *FKS LOADELIM* (shown in column 8). With respect to total GETFIELD operations (column 2), *PAR+TRANS LOADELIM* reduces the dynamic counts in the range of 47.38% for Montecarlo and 99.99% in CG (shown in column 9). For the CG benchmark, the dominant method in terms of execution time is `step0`. In the absence of our side effect analysis, load elimination for this function was limited due to the presence of a function call inside the inner loop.

Figure 8 presents the relative performance improvements of the three parallel Section 3 Java Grande benchmarks and the two Nas Parallel benchmarks[3] with respect to the *1-thread NO LOADELIM* case. For the 1-thread case, we observe an average of 1.29× performance improvement of *PAR+TRANS LOADELIM* in comparison to the *FKS LOADELIM* case, with a best-case 1.76× improvement (for Moldyn). While comparing with *FKS+TRANS LOADELIM*, *PAR+TRANS LOADELIM* yields an average improvement of 1.20× with best-case 1.32× improvement (for Moldyn).

For the 16-thread case, the parallel interprocedural load elimination techniques presented in this paper including the two optimizations (*PAR+TRANS LOADELIM* Thread=16) resulted in a 1.15× improvement over the FKS intraprocedural approach without optimizations, on average. For the MolDyn benchmark, we achieved a maximum of 1.39× improvement. When we compare against FKS with optimizations, on average *PAR+TRANS LOADELIM* Thread=16 resulted in a 1.11× improvement with best-case 1.20× improvement for Moldyn. Three of the five benchmarks (CG, MolDyn, Montecarlo) show measurable speedup with the use of *PAR LOADELIM*, whereas for the remaining two (MG, Raytracer) there was no measurable speedup. Using live-range splitting as part of *PAR+TRANS LOADELIM*, we can see that both MG and Raytracer do not degrade performance. We believe that a live-range splitting based register allocator could further improve the performance results reported in this paper. Figure 9 shows the speedup details for MolDyn and CG as the number of workers (k) increases for *PAR+TRANS LOADELIM*.

[3]For SPECjbb2000, we haven't as yet obtained a measurable difference in runtime due to the reduction in dynamic getfields shown in Table III, because of the inability of the X10 v1.5 work-sharing runtime to work efficiently with `await()` calls from Java. In the future, we plan to extend our load elimination algorithm to support *phasers* [16] which can be used as a replacement for the `await()` calls.

V. RELATED WORK

A. Side-Effect Analysis

Interprocedural analysis for side effects of procedure calls has been widely studied in the literature [15], [19]–[21]. Banning [15] formalized the notion of side effects using both the flow-insensitive sets *e.g., MOD*, *REF*, *USE* and the flow-sensitive *e.g., DEF* set. They provide a data flow technique that operates over the call graph to determine side-effects. Later on, Cooper et al. [21] improved the efficiency of the analysis for formal parameters using binding-multi graph. Subsequently, Landi et al. [22] proposed a side-effect analysis for C programs that uses pointers. They introduced the conditional MOD sets *i.e., CMOD* and *PMOD* based on pointer-induced aliasing.

Clausen [23] developed an interprocedural side effect analysis for Java byte-codes and show its effectiveness in optimizing Java programs *e.g.,* dead-code elimination, loop-invariant code motion, constant propagation, common subexpression elimination in the presence of virtual methods. Side-effects in their analysis are specified in terms of field variables that a method and its callee may modify. Most recently, Le et al. [4] used SPARK - the interprocedural analysis component of SOOT compiler infrastructure to computer side effect information for Java programs. These side-effects are then fed into Jikes RVM for performing interprocedural optimizations. This approach facilitates computation of more precise side-effect information using various existing points-to analysis algorithms in SOOT and it assumes the whole program be presented to SOOT infrastructure for

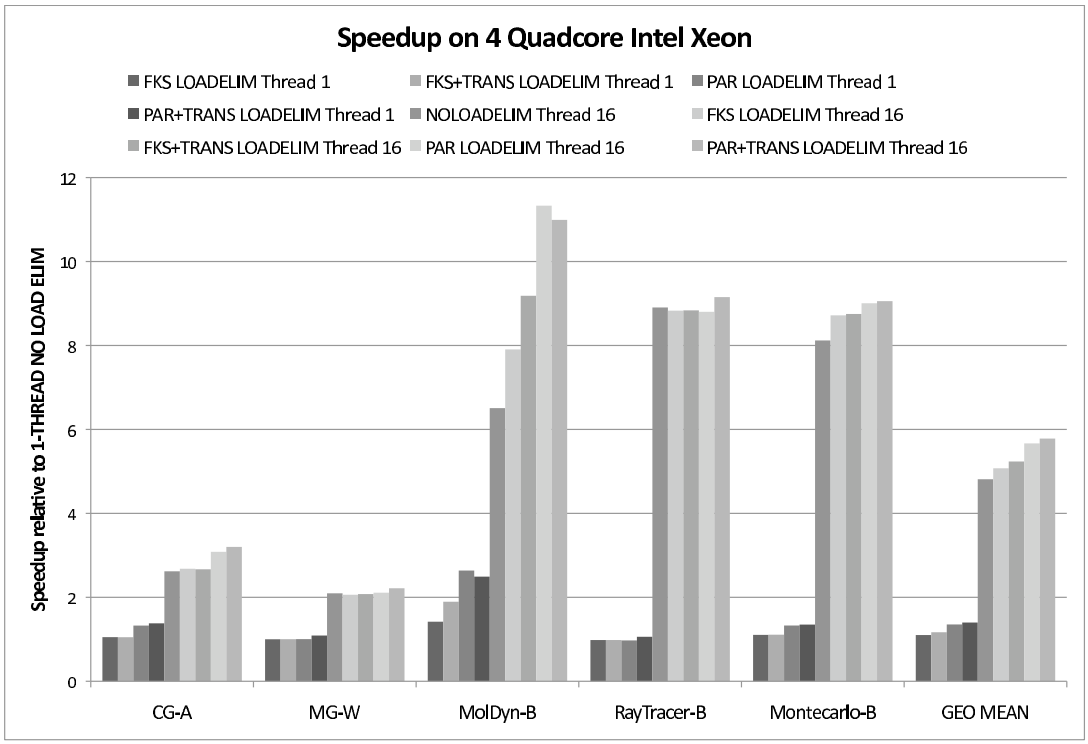

Figure 8. Performance improvement for NPB benchmarks (CG, MG) and JGF Benchmarks (Moldyn, Raytracer, Montecarlo) using the load elimination algorithm presented in Figure 7. The improvement is shown relative to the 1-thread *NO LOADELIM* case.

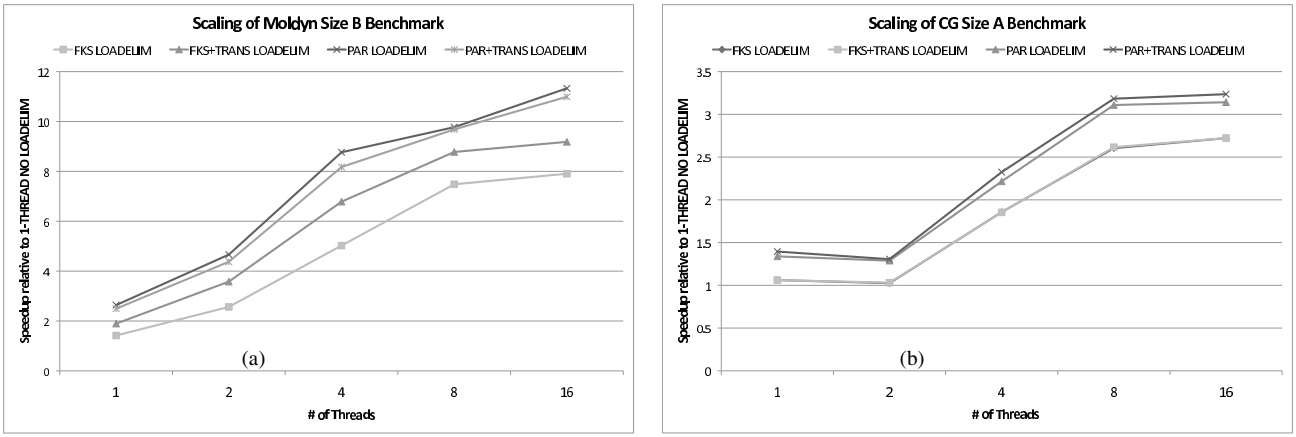

Figure 9. Scaling of JGF Section 3 MolDyn Size B Benchmark and NPB CG Size A Benchmark using the load elimination algorithm introduced in this paper. The speedup is shown relative to the 1-thread *NO LOADELIM* case.

Benchmark	# getfield (original)	# getfield after FKS load elim.	# getfield after FKS+TRANS load elim.	# getfield after PAR load elim.	# getfield after PAR+TRANS load elim.	improvement relative to FKS+TRANS (%age)	improvement relative to FKS (%age)	improvement relative to original (%age)
CG-A	3.89E09	3.10E09	3.03E09	2.34E09	3.92E05	99.99 %	99.99 %	99.99 %
MG-W	1.41E04	1.15E04	1.13E04	7.96E03	6.71E03	40.58 %	41.72 %	52.55 %
MolDyn-B	1.19E10	7.91E09	5.82E09	4.91E09	3.11E09	46.49 %	60.62 %	73.89 %
Raytracer-B	3.08E10	2.02E10	2.02E10	1.67E10	1.38E10	31.82 %	31.93 %	55.25 %
Montecarlo-B	1.75E09	1.54E09	1.48E09	1.15E09	9.19E08	37.95 %	40.47 %	47.38 %
SPECjbb2000	1.19E09	1.025E09	8.95E08	6.65E08	5.78E+08	35.44 %	43.19 %	51.56 %

Table III
DYNAMIC COUNTS OF GETFIELD OPERATIONS FOR NPB BENCHMARKS (CG, MG), JGF BENCHMARKS (MOLDYN, RAYTRACER, MONTECARLO) AND SPECJBB2000 BENCHMARK

analysis. We take a different approach. We compute fast flow-insensitive and field-insensitive side-effect summary information as an interprocedural pass in Jikes RVM. Additionally, we compute side-effect summaries in presence of parallel constructs of X10 that obeys Isolation Consistency memory model.

B. Memory Model

Starting from sequential consistency memory model introduced by Lamport [9], there has been several work in memory models including [10], [12], [24]. The location consistency memory model described in [12] models every memory location as a partially ordered multiset of write and synchronization operations. They show that location consistency memory model is strictly weaker than existing memory models, but is still equivalent to stronger models for parallel programs having no data races. Our work on Isolation Consistency memory model is based on the location consistency weakest memory model [12] and *weak atomicity* memory model [13] used in transactional memory systems. In weak atomicity, atomic sections are executed atomically only with respect to the other atomic sections and not other non-atomic section codes.

C. Load Elimination

Scalar Replacement [25], [26] is well-studied in the context of optimizing array references in scientific programs. Early on, scalar replacement algorithms were based on data dependence analysis and were applied in loop nest to improve register reuse. Recently, Bodik et al. [1] and Lo et al. [27] focused on redundant memory load operations and presented partial redundancy elimination (PRE) based solutions to them.

For Java programs, Fink, Knobe and Sarkar [2] presented an unified framework to analyze memory load operations for both array-element and object-field references. Their algorithm detects fully redundant memory operations using an extended Array SSA form representation for array-element memory operations and global value numbering technique to disambiguate the similarity of object references. Later on, Praun et al. [3] presented a PRE based interprocedural load elimination algorithm that takes into account Java's concurrency features and exceptions. The concurrency based side-effects were obtained using their conflict analysis [28] and obeyed SC memory model.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduced an interprocedural load elimination algorithm for dynamic optimization of parallel programs. The algorithm has been implemented in Jikes RVM for optimizing a subset of X10 parallel programs. The main contributions of the paper include: a) side-effect analysis of method calls, b) support for load elimination in the presence of three core parallel constructs – async, finish, and isolated, c) an Isolation Consistency memory model that establishes the legality of our load elimination transformation for parallel constructs, and d) performance results to study the impact of load elimination on a set of standard X10 parallel programs. Our performance results show decreases in dynamic counts for getfield operations of up to 99.99%, and performance improvements of up to 1.76× on 1 core, and 1.39× on 16 cores, when comparing the algorithm in this paper with the load elimination algorithm available in Jikes RVM.

Possible directions for future work include improving the precision of analyzing *isolated* blocks, extending the *ParallelSideEffectAnalysis* procedure with the *Never-Execute-In-Parallel* analysis presented in [17], and implementing our techniques for array accesses that go beyond simple field accesses.

ACKNOWLEDGMENTS

We would like to thank all members of the X10 team at IBM and the Habanero team at Rice for valuable discussions and feedback related to this work, especially Igor Peshansky for discussion of the semantics of Java final variables and John Mellor-Crummey for memory model discussions. We are also thankful to all X10 team members for their contributions to the X10 software used in this paper. We gratefully acknowledge support from an IBM Open Collaborative Faculty Award. This work was supported in part by the National Science Foundation under the HECURA program,

award number CCF-0833166. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. Finally, we would like to thank the anonymous reviewers for their comments and suggestions, which helped improve the experimental evaluations and the overall presentation of the paper.

REFERENCES

[1] R. Bodík et al., "Load-reuse analysis: design and evaluation," *SIGPLAN Not.*, vol. 34, no. 5, pp. 64–76, 1999.

[2] S. J. Fink et al., "Unified analysis of array and object references in strongly typed languages," in *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*. London, UK: Springer-Verlag, 2000, pp. 155–174.

[3] C. von Praun et al., "Load elimination in the presence of side effects, concurrency and precise exceptions," in *LCPC '03: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, 2003.

[4] A. Le et al., "Using inter-procedural side-effect information in JIT optimizations," in *14th International Conference on Compiler Construction (CC). LNCS*. Springer Verlag, 2005, pp. 287–304.

[5] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[6] "Jikes RVM," http://jikesrvm.org/.

[7] P. Charles et al., "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA 2005 Onward! Track*, 2005.

[8] J. R. Larus and R. Rajwar, *Transactional Memory*. Morgan & Claypool, 2006.

[9] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, vol. C-28, no. 9, pp. 690–691, September 1979.

[10] J. Manson et al., "The Java memory model," in *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2005, pp. 378–391.

[11] J. P. Hoeflinger and B. R. de Supinski, "The OpenMP Memory Model."

[12] G. R. Gao and V. Sarkar, "Location consistency-a new memory model and cache consistency protocol," *IEEE Trans. Comput.*, vol. 49, no. 8, pp. 798–813, 2000.

[13] M. Martin et al., "Subtleties of Transactional Memory Atomicity Semantics," *IEEE Comput. Archit. Lett.*, vol. 5, no. 2, p. 17, 2006.

[14] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988.

[15] J. Banning, "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables," *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pp. 29–41, January 1979.

[16] J. Shirako et al., "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 277–288.

[17] S. Agarwal et al., "May-happen-in-parallel analysis of X10 programs," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 2007, pp. 183–193.

[18] R. Barik et al., "Experiences with an SMP Implementation for X10 based on the Java Concurrency Utilities (Extended Abstract)," in *Proceedings of the 2006 Workshop on Programming Models for Ubiquitous Parallelism, co-located with PACT 2006, September 2006, Seattle, Washington*, 2006.

[19] T. C. Spillman, "Exposing side-effects in a PL/I optimizing compiler," in *Proceedings of the IFIP Congress 1971*, 1971, pp. 376–381.

[20] F. E. Allen, "Interprocedural data flow analysis," in *Proceedings of the IFIP Congress 1974*, 1974, pp. 398–402.

[21] K. D. Cooper and K. Kennedy, "Fast interprocedural alias analysis," in *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1989, pp. 49–59.

[22] W. Landi, B. G. Ryder, and S. Zhang, "Interprocedural modification side effect analysis with pointer aliasing," *SIGPLAN Not.*, vol. 28, no. 6, pp. 56–67, 1993.

[23] L. R. Clausen, "A Java bytecode optimizer using side-effect analysis," *ACM Workshop on Java for Science and Engineering Computation*, June 1997.

[24] S. Adve et al., "Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems," *Proceedings of the IEEE*, vol. 87, no. 3, pp. 445–455, March 1999.

[25] D. Callahan et al., "Improving register allocation for subscripted variables," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York*, pp. 53–65, June 1990.

[26] S. Carr and K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Software—Practice and Experience*, no. 1, pp. 51–77, January 1994.

[27] R. Lo et al., "Register promotion by sparse partial redundancy elimination of loads and stores," *SIGPLAN Not.*, vol. 33, no. 5, pp. 26–37, 1998.

[28] C. von Praun and T. R. Gross, "Static conflict analysis for multi-threaded object-oriented programs," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2003, pp. 115–128.