Interprocedural Strength Reduction of Critical Sections in Explicitly-Parallel Programs

Rajkishore Barik, Rice University, Houston, Texas
Jisheng Zhao, Rice University, Houston, Texas
Vivek Sarkar, Rice University, Houston, Texas

Technical Report TR 13-05
Department of Computer Science
Rice University
July 2013

Abstract

In this paper, we introduce novel compiler optimization techniques to reduce the number of operations performed in critical sections that occur in explicitly-parallel programs. Specifically, we focus on three code transformations: 1) *Partial Strength Reduction (PSR) of critical sections* to replace critical sections by non-critical sections on certain control flow paths; 2) *Critical Load Elimination (CLE)* to replace memory accesses within a critical section by accesses to scalar temporaries that contain values loaded outside the critical section; and 3) *Non-critical Code Motion (NCM)* to hoist thread-local computations out of critical sections. The effectiveness of the first two transformations is further increased by interprocedural analysis.

The effectiveness of our techniques has been demonstrated for critical section constructs from three different explicitly-parallel programming models — the `isolated` construct in Habanero Java (*HJ*), the `synchronized` construct in standard Java, and transactions in the Java-based Deuce software transactional memory system. We used two SMP platforms (a 16-core Intel Xeon SMP and a 32-Core IBM Power7 SMP) to evaluate our optimizations on 17 explicitly-parallel benchmark programs that span all three models. Our results show that the optimizations introduced in this paper can deliver measurable performance improvements that increase in magnitude when the program is run with a larger number of processor cores. These results underscore the importance of optimizing critical sections, and the fact that the benefits from such optimizations will continue to increase with increasing numbers of cores in future many-core processors.

Index Terms

Critical sections; transactions; partial strength reduction; critical load elimination; non-critical code motion; interprocedural optimization.

I. Introduction

It is expected that future computer systems will comprise of massively multicore processors with hundreds of cores per chip. Compiling programs for concurrency, scalability, locality, and energy efficiency on these systems is a major challenge. This paper focuses on compiler techniques to address some of the scalability limitations of applications when executing on many-core systems. According to Amdahl's law, the speedup of parallel applications is limited by the amount of time spent in the sequential part of the applications. A major source of these sequential bottlenecks in parallel applications can be found in critical sections which are logically executed by at most one thread at a time. These critical sections are most commonly found in explicitly-parallel programs which can be non-deterministic in general. In contrast, automatically parallelized programs are usually deterministic and use critical sections sparingly (often as a surrogate for more efficient mechanisms, *e.g.,* when a critical section is used to implement a parallel reduction).

As the number of threads increases, the contention in critical sections increases. It is thus important for both performance and scalability of multi-threaded applications that critical sections be optimized heavily. To this end, recent work [40], [41] has proposed combined software and hardware-based approaches to accelerate critical sections for multi-core architectures. The focus of our paper is on compiler-based approaches to move operations out of critical sections whenever possible, akin to the attention paid to innermost loops by classical optimizing compilers for sequential programs. We refer to such code transformations as "strength reduction" because the overall cost of an operation is reduced when it is performed outside a critical section than within it.

As an example, consider memory load operations within a critical section. Such a load operation may cause performance problems in two ways if the load results in a cache miss. The overhead of a cache miss can be expensive, not only because it hampers the performance of a single thread, but also because it delays other threads from entering the critical section. It is thus desirable to move cache misses out of critical sections, to whatever extent possible. In addition, moving loads out of critical sections can reduce cache consistency overheads.

Well-known redundancy elimination techniques to help achieve this goal include *scalar replacement for load elimination* [10], [12], [21], [43], [27], *redundant memory operation* analysis [16], and *register promotion* [32]. For explicitly-parallel programs, most compilers make conservative assumptions while performing load elimination around critical sections. For example, the load elimination algorithm implemented in the Jikes RVM 3.1 dynamic optimizing compiler [1] conservatively assumes that the boundaries of synchronized blocks kill all objects, thereby prohibiting hoisting of loads on shared objects outside of synchronized blocks. A more recent load elimination algorithm [6] eliminates load operations in the context of `async`, `finish` and `isolated`/`atomic` constructs found in the Habanero-Java (*HJ*) [13] and X10 [14] languages. For critical sections (`isolated` blocks), this algorithm proposed a flow-insensitive and context-insensitive approach that unifies side-effects for all isolated blocks in the entire application. This approach only allows a restricted class of load operations to be hoisted out of critical sections since neither does it not take into account the May-Happen-in-Parallel (*MHP*) relationship among critical sections, nor does it eliminate loads from critical methods that are not "inlinable" (not amenable to inline expansion).

There are other opportunities for strength reduction of critical sections that go beyond load elimination. Sometimes, a critical section may contain *non-critical* operations that are included in the critical section for programmer convenience. Hoisting these computations out of critical sections may be tedious for the programmer, but could yield important performance and scalability benefits when performed by the compiler. Further, analogous to partial redundancy elimination in sequential programs, it may be possible to replace critical sections by non-critical code blocks on certain control flow paths.

Historically, optimizing compilers have paid special attention to innermost loops in sequential loops because improvements in those code regions lead to the larger impact on overall performance. Likewise, it is now becoming essential for an optimizing compiler for parallel programs to pay special attention to critical sections since doing so can potentially reduce the critical path length of the parallel program's computation graph. To that end, this paper introduces three code transformations for strength reduction of critical sections. The effectiveness of these transformations is further increased by interprocedural analysis of parallel programs.

The motivation for optimizing critical sections also arises from recent trends in parallel programming models. The key idea behind transactional memory systems [17], [25] is to enable programmers to think of the semantics of transactions as being akin to that of critical sections, while using a combination of compiler and runtime techniques to support speculative execution of transactions in parallel. However, as shown in our experimental results, hoisting operations out of transactions can improve the performance of transactional memory systems as well. There have also been proposals for "isolated-by-default" semantics, for example, in the yield-based approach in [46]. In such cases, the number of critical sections naturally increases since every statement in the program is required to belong to some critical section.

In this paper, we focus on compiler optimization techniques for strength reduction of critical sections. We make the following novel contributions in the context of both *HJ* and *Java* explicitly-parallel programs:

- An interprocedural *Partial Strength Reduction (PSR)* transformation to replace critical sections by non-critical sections on certain control flow paths (Section IV-C). We are unaware of any prior work that performs this transformation on critical sections. Past work related to partial redundancy elimination of synchronization operations (*e.g.,* [11]) used sequential code as a starting point, and focused on optimizing dependence-based synchronization across loop iterations resulting from automatic parallelization. As a result, their approach is not applicable to optimization of non-deterministic explicitly-parallel programs. Further, the scope of their techniques was limited to loop nests (relying on inline expansions of procedure calls in loop bodies), unlike PSR which is global and interprocedural in scope.
- An MHP-aware interprocedural *Critical Load Elimination (CLE)* transformation to replace memory accesses within a critical section by accesses to scalar temporaries that contain values loaded outside the critical section (Section IV-D). Additionally, this transformation promotes the scalar temporaries to method parameters, when possible, so as to enable further optimization of the memory accesses within a calling context, e.g., hoisting load operations out of a loop containing the call site. In contrast, prior work on scalar replacement in explicitly-parallel programs (*e.g.,* [6]) did not use MHP information for load elimination in critical sections, and did not explore parameter promotion extensions.
- A *Non-critical Code Motion (NCM)* transformation to hoist thread-local computations out of critical sections in explicitly-parallel programs (Section IV-E). In contrast, the code motion algorithms in past work (*e.g.,* [36]) were performed with respect to compiler-inserted dependence-based synchronizations in deterministic auto-parallelized programs. The compiler analyses needed for explicitly-parallel programs are fundamentally different.
- An experimental evaluation consisting of 17 explicitly-parallel benchmarks written in Habanero-Java, Java threads, and the Java-based Deuce [25] Software Transactional Memory (STM) system. Our results for the 5 *HJ* benchmarks show performance improvements of up to $1.16\times$ with a geometric mean speedup of $1.09\times$ on the Xeon SMP and $1.11\times$ with a geometric mean speedup of $1.07\times$ on the Power7 SMP due to the three transformations introduced in this paper, when using 16 cores (Xeon) and 32 cores (Power7). Our results for the 4 *Java* benchmarks due to the three transformations show performance improvements of up to $1.08\times$ with a geometric mean speedup of $1.06\times$ on the Xeon SMP using 16 cores and $1.10\times$ with a geometric mean speedup of $1.06\times$ on the Power7 SMP using 32 cores. Using the Deuce STM on the remaining 8 STAMP benchmarks, we observed performance improvements of up to $1.68\times$ when using the 16-core Xeon SMP and a geometric mean speedup of $1.21\times$, again due to using these three transformations[1]. These results underscore the importance of optimizing critical sections and the fact that the benefits from such optimizations will continue to increase with increased numbers of cores in future many-core processors.

The rest of this paper is organized as follows. Section II summarizes the explicitly-parallel programming models studied in this work. Section III motivates our compiler optimizations via three examples, one for each transformation. Section IV discusses the three compiler transformations in detail. Section V presents the experimental results summarized above. Related work is discussed in Section VI, and Section VII presents our conclusions.

II. Programming Model

In this section, we briefly summarize the subsets of the three explicitly-parallel programming models that are the target of our compiler optimizations. Our compiler optimizations are sound in the sense that programs that use features outside these subsets will be correctly transformed, though the presence of additional constructs unknown to our optimizations (*e.g.,* futures, phasers) could lead to conservativeness in the May-Happen-in-Parallel analysis performed by our optimizations. Also, though

[1]The Deuce STM appears to currently be unable to execute the STAMP benchmarks correctly on the Power7 SMP, perhaps due to its weak memory model.

the three models studied in this paper are based on Java, our approach can also be used to optimize critical sections in C-based and other explicitly-parallel programming models.

For Java threads, our compiler focused on the subset consisting of Thread `start()` and `join()` methods and the `synchronized` construct. For the Deuce STM system [25], transactions are specified as calls to methods that are annotated as `@Atomic` and instead of using `synchronized`. For the Habanero-Java (*HJ*) programming model [13], our compiler focused on the subset consisting of `async`, `finish`, and `isolated`.

A quick summary of the aforementioned HJ constructs is as follows. The *async S* statement causes the parent task to create a new child task to execute S asynchronously. The *finish S* termination statement causes the parent task to execute S and then wait until all parallel tasks created within S have completed, including transitively spawned tasks. The *isolated S* statement guarantees that each instance of S will be performed in mutual exclusion with respect to all other potentially parallel interfering instances of isolated statements. Finally, an unordered parallel iteration over a set of points p within a range of values R is captured by the syntax *forall (point p : R)* (which also includes an implicit `finish`).

We assume the Isolation Consistency (*IC*) memory model proposed in [6] for *HJ* programs. In the *IC* model, out-of-order execution within a thread is allowed as long as intra-thread dependences are not violated. The intra-thread dependences are defined via a weak-isolation model that ensures the correct ordering of load and store operations from multiple threads when they occur in critical sections or are ordered by other happens-before dependences. There is no ordering guarantee between loads and stores outside isolated regions (thereby, avoiding strong isolation). Moreover, the program transformations described in this paper using the *IC* model is guaranteed to produce the same result as that of a stronger memory model like Sequential Consistency (SC) [26] for data-race-free programs. For *Java* programs, we assume the standard Java Memory Model (JMM) semantics [33].

III. Examples

Figure 1 contains three code examples that illustrate the three optimizations for critical sections introduced in this paper. Example 1) was extracted[2] from the `jcilk` benchmark [28] written in *Java*. This examples illustrates one case of the Partial Strength Reduction (PSR) optimization. In the original code, each call to `getSum()` results in a critical section since the method is declared as `synchronized`. After transformation, a non-synchronized clone of `getSum()` is generated called `getSum_()`, and a call to `getSum_()` is inserted in method `compute1()`. However, `compute2()` still calls the original synchronized `getSum()` method.

Example 2) was extracted from an *HJ* version of the TSP benchmark [43]. Past algorithms such as [6] unify side-effects for all isolated blocks in the entire application. As a result, they can eliminate loads to arrays `A` and `B` from the implicit `this` object in both methods `bar1` and `bar2`, but cannot eliminate the memory load operation of `O.sum`. The Critical Load Elimination (CLE) optimization introduced in this paper discovers from May-Happen-in-Parallel (*MHP*) analysis that *MHP(bar1, bar2)* is *false*, which in turn enables it to hoist the load of `O.sum` from both the `isolated` and `forall` regions. It also promotes loads of arrays `A` and `B` to arguments of methods `bar1_` and `bar2_` respectively, which further enables the hoisting of the load operation in the while-loop of the caller method `bar`.

Example 3) was extracted from method `processLines` of `Order.java` in the specJBB-2000 *Java* benchmark. With careful analysis, the Non-critical Code Motion (NCM) optimization discovers that the update to `global_array` is the only non-local computation in synchronized method `far()`. We apply a non-local slicing algorithm to separate non-local computations in the critical section. In this example, we first apply two preprocessing transformations: 1) distribute the for-loop; and 2) scalar expand `a`. Once these transformations are performed, we can now determine that the object creation is local. The transformed code shown to the right shrinks the critical section to apply only to the non-local computations. In this example, since the update to `global_array` is control dependent on the `if` statement and `for-loop`, we have to include them in the critical section. There may be situations where the `if` condition may be proved to be local, in which case only the `global_array` update needs to be performed in the critical section.

IV. Compiler Transformations for Critical Sections

In this section, we first present our overall optimization framework. We then introduce the common Parallel Intermediate Representation (*PIR*) for the three different explicitly-parallel programming models used in this paper. Using this representation, we describe our three compiler transformation algorithms to optimize critical sections. The transformations are applied in the order in which they are presented in this section, since each creates opportunities for the next.

A. Overall Framework

Figure 2 depicts the basic flow of the compilation process. Our compiler framework accepts either *HJ* or *Java* programs as input source code and translates them into a Parallel Intermediate Representation (PIR) which represents the `async`/`finish`/`isolated` parallel constructs of *HJ* and `start`/`join`/`synchronized` constructs at the *Java IR* level. We

[2]For simplicity, class and method names were renamed, and other code details elided, for all code extractions discussed in this section.

Example 1) Partial Strength Reduction (PSR):

```
1  class Foo {
2    int f_sum=0;
3    synchronized void compute1(Goo g) {
4      f_sum+=g.getSum(); // critical section
5    }
6    void compute2(Goo g) {
7      ...=g.getSum(); // critical section
8    } }
9  class Goo {
10   int g_sum=0;
11   synchronized int getSum() {
12     return g_sum;
13   }
14   synchronized void apply(Foo f) {
15     f.compute1(this);
16 } }
```

$\Longrightarrow$

```
1  class Foo {
2    int f_sum=0;
3    synchronized void compute1(Goo g) {
4      f_sum+=g.getSum_(); // non-critical
5    }
6    void compute2(Goo g) {
7      ...=g.getSum(); // critical section
8    } }
9  class Goo {
10   int g_sum=0;
11   synchronized int getSum() {
12     return g_sum; }
13   int getSum_() {
14     return g_sum; }
15   synchronized void apply(Foo f) {
16     f.compute1(this); } }
```

Example 2) Critical Load Elimination (CLE):

```
1  class Foo {
2    int sum = 10, f_sum = 10;
3    int[] A, B;
4    void bar1 (Foo O) {
5    forall (point i:[0..N-1]) {
6      A[i] = ...; // S1
7      isolated O.sum+=A[i]; // S2
8     } }
9    void bar2 (Foo O) {
10     forall (point i:[0..N-1]) {
11       B[i] = ...; // S3
12       isolated O.f_sum+=O.sum*B[i]; // S4
13   } }
14   void bar(Foo O) {
15     while (...) {
16       bar1(O);
17       bar2(O);
18 } } }
```

$\Longrightarrow$

```
1  class Foo {
2    int sum = 10, f_sum = 10;
3    int[] A, B;
4    void bar1_(Foo O, int[] A_) {
5      forall (point i:[0..N-1]) {
6        A_[i] = ...;
7        isolated O.sum+=A_[i];
8      } }
9    void bar2_(Foo O, int[] B_) {
10     int O_sum = O.sum;
11     forall (point i:[0..N-1]) {
12       B_[i] = ...;
13       isolated O.f_sum+=O_sum*B_[i];
14   } }
15   void bar(Foo O) {
16     int[] A_ = this.A; int[] B_ = this.B;
17     while (...) {
18       bar1_(O, A_); bar2_(O, B_);
19     } } }
```

Example 3) Non-critical Code Motion (NCM):

```
1  ...
2  class Foo { int x; } ;
3  ...
4  Foo[] global_array;
5  int count = 0;
6  ...
7  class Goo {
8    float o_sum;
9    public synchronized void far() {
10     float l_sum = 0;
11     for (int i=0; i<num; i++) {
12       Foo a = new Foo();
13       if (...) {
14         l_sum += a.x;
15         global_array[count++] = a;
16     } }
17       o_sum = l_sum * ...;
18   } }
```

$\Longrightarrow$

```
1  ...
2  class Foo { int x; };
3  ...
4  Foo[] global_array;
5  int count = 0;
6  ...
7  class Goo {
8    float o_sum; int num;
9    public void far_(Foo[] global_array) {
10     float l_sum = 0;
11     Foo[] a_ = new Foo[num];
12     for(int i=0; i<num; i++) a_[i] = new Foo();
13     synchronized {
14       for (int i=0; i<num; i++) {
15         if (...) {
16           l_sum += a_[i].x;
17           global_array[count++] = a_[i];
18       } }
19     o_sum = l_sum * ...;
20     } }
```

Fig. 1. Examples that illustrate opportunities for strength reduction in critical sections: 1) partial strength reduction; 2) critical load elimination; 3) non-critical code motion.

also extend the *Java IR* to model `atomic` regions of transactional programs similar to *Java* constructs. The *PIR* representation is suitable for several parallel program optimizations as it is enriched with structures that capture program structure, happen-before relation, and mutual exclusion relation[3].

The interprocedural May-Happen-in-Parallel analysis described in Section IV-D and the side-effect analysis are performed at the PIR-level. Our optimization framework is driven by a whole-program alias analysis that leverages interprocedural information to disambiguate memory references.

The transformations described next in this section are applied in the following order. First, we apply Partial strength reduction (PSR) as described in Section IV-C to eliminate `atomic`/`synchronized`/`isolated` blocks on certain control flow paths. We then apply the Critical Load Elimination (CLE) transformation (Section IV-D) that replaces load operations by scalar temporaries (even interprocedurally), when legal to do so. Finally, the Non-critical Code Motion (NCM) transformation (Section IV-E) moves non-critical code out of critical regions to reduce the amount of time spent in critical sections. CLE is best performed after PSR, since PSR will reduce the number of critical sections that CLE should focus on. CLE is best performed prior to NCM so as to create more opportunities for code motion out of critical sections.

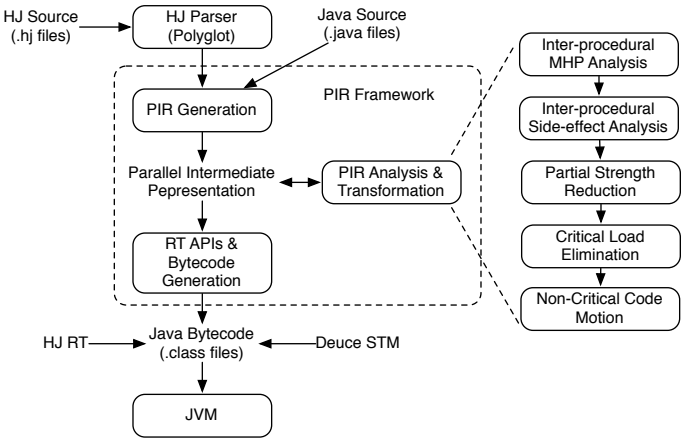

Fig. 2. Compilation Framework.

B. Parallel Intermediate Representation (PIR)

The *PIR* is designed to be a common intermediate language for the explicitly-parallel programs studied in this paper. For every method, the *PIR* consists of three key data structures: 1) a Region Structure Tree (*RST*); 2) a set of Region Control Flow Graphs (*RCFG*); and 3) a set of Region Dictionaries (*RD*). The *RST* represents the region nesting structure of the method being compiled, analogous to the Loop Structure Tree (LST) introduced in [39].

For *HJ*, the single-entry regions considered in this work include `FINISH`, `ASYNC`, `ISOLATED`, and loop regions. Note that `forall` constructs are expanded to for-loops with `ASYNC` regions enclosed in a common `FINISH`. For *Java*, a thread `start` method call is represented by an `ASYNC` region with its body consisting of a call to the `run` method. A `join` method is represented by a special finish region node, denoted as `L_FINISH`, that captures both the thread creation and join operations. (Note that `L_FINISH` is a local finish operation that only joins threads from the outermost level; transitively created threads will require individual `L_FINISH` nodes.) A `synchronized` block or method is represented using an `ISOLATED` region that keeps track of the lock object in the region dictionary. Two special empty regions `START` and `END` are added to designate the start and end of a method. Currently, other parallel constructs in *Java* and *HJ*, such as barriers, volatile variables, futures, and phasers are ignored during *MHP* analysis. This means that our *MHP* analysis may conservatively return a *true* value in some cases where *false* would be a more precise answer, but its results will always be sound.

The RST and RCFG for the methods `Foo.bar1` and `Foo.bar2` in Example 2) of Figure 1 are shown in Figure 3.

C. Partial Strength Reduction of Critical Operation (PSR)

There has been a lot of past work on optimizing multithreaded *Java* programs that focused on eliminating `synchronized` operations from methods and regions based on escape analysis results. In this paper, we identified scenarios where a `synchronized` or `isolated` region is partially redundant; *i.e.,* it is redundant on some path of execution, but not necessarily on all paths. Figure 4 presents an algorithm to "strength reduce" critical sections via a form of partial redundancy elimination in which critical sections are replaced by non-critical sections on certain control flow paths. It requires a *lock-set analysis*

[3]For Java code, the PIR translator treats synchronized methods and synchronized regions as `HJ isolated` constructs with specific lock sets.

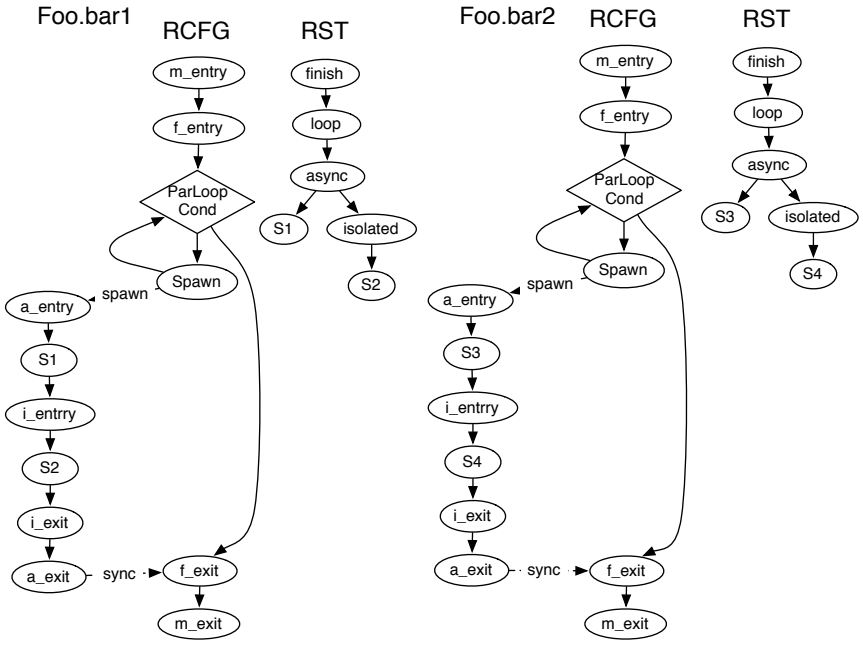

Fig. 3. RST and RCFG for Example program in Figure 1 (Case 2)

that statically computes the set of locks that are held during the execution of a *PIR* statement [20]. $pred(s)$ denotes the control-flow predecessor of s in the program. For an isolated procedure P, if the isolated object is already locked at every call site of P, then the synchronized region for P is fully redundant and thus is replaced with non-synchronized version (Lines 7-13). If the isolated object is only redundant on certain control flow path, we create a specialized version of P by removing the synchronization operation (Lines 15-20). Similarly, for isolated code blocks, we replace their enclosing procedures with non-isolated regions (Lines 22-25).

Applying this algorithm to the code fragment in the left column of Figure 1 (Example 1) yields the code to the right column.

```
1  function RemoveRedundantSync ()
     Input  : Call graph CG, and PIR for each method;
     Output: Modified CG and PIRs
2    Perform lock-set analysis [20];
3    for each procedure, P, in CG in reverse topological sort order do
4        for each isolated region, R, in PIR(P) do
5            if R is procedure P itself then
6                c := static class type of P;
7                callsites := all callsites of P;
8                all_redundant := true;
9                for each call site, s, in callsites do
10                   if c ∉ lock-set(pred(s)) then
11                       all_redundant := false;
12               if all_redundant == true then
13                   Modify P's signature to remove isolated or synchronized operation;
14               else
15                   Create a specialized version of P, called P' after removing isolated or synchronized operation;
16                   c := static class type of P;
17                   for each call site, s, in callsites do
18                       if c ∈ lock-set(pred(s)) then
19                           Update the call site s to invoke P' instead of P;
20                   Update CG;
21           else
22               c := static class of the synchronization object of region R;
23               if c ∈ lock-set(pred(R)) then
24                   Create a specialized version of P that contains R, called P', after removing the isolated or synchronized operation for region R;
25                   Update all callsites of P to call P' instead of P;
```

Fig. 4. Perform *partially redundant critical operation removal* transformation.

D. Interprocedural Critical Load Elimination (CLE) using Scalar Replacement

Object Model: Both objects and arrays are represented using a compile-time abstraction called heap-arrays [21]. An access to field x of object a is represented as access to $\mathcal{H}^x[a]$. Equality among heap arrays is performed using aliasing information, *i.e.,* $\mathcal{H}^x[a]$ and $\mathcal{H}^x[b]$ are not equal if a and b are not aliased to each other.

May-Happen-in-Parallel Analysis: Since *RST*s are akin to Program Structure Trees (*PST*s), the intraprocedural *MHP* information can be computed using the algorithm for X10 and *HJ* in [3] at the *PIR*-level. To extend this algorithm to interprocedural level for both *Java* and *HJ*, we require the following analysis results. First, we require *lock-set analysis* results. We refine the *MHP*(s_1, s_2) relation between two statements s_1 and s_2 to additionally check for the property that *lock-set*$(s_1) \cap$ *lock-set*$(s_2) \neq \phi$.

Second, the extension makes the *MHP* algorithm interprocedural. We use the following two observations to improve intraprocedural *MHP* information. One important observation is that, if a call site s in procedure p executes in parallel with the `END` region node of p in *RST* using intraprocedural *MHP* information, then any procedure reachable from the called procedure in s can execute in parallel with p. Additionally, they will also execute in parallel with transitive callers of p until the innermost nested `FINISH` region (for *Java* programs, until `L_FINISH` region). The second observation deals with *happens-before* relationship between statements, *i.e.,* if procedures f and g are invoked from a common caller p with the call-sites in p being the only invocations for f & g, and intraprocedural *MHP* is $false$ for their call sites in p, then any procedure reachable from f in call graph that is wrapped in ASYNC region which is preceded by a FINISH region can never execute in parallel with any procedure reachable from g. Using these two observations, we devise an algorithm that conservatively extends intraprocedural *MHP* to interprocedural.

Since *RST*s are akin to Program Structure Trees (*PST*s), the intraprocedural *MHP* information at *PIR* level can be computed using the algorithm in [3]. An extension is necessary to handle the `synchronized` constructs of *Java*. We conservatively compute the lock objects for `ISOLATED` regions statically using the following manner: In *Java*, locks are obtained in a scoped manner – all possible locks held during a statement execution can be statically computed easily by walking over the call graph and extending them with alias analysis results. We define *lock-set*(s) that consists of the set of objects that are locked while executing any statement s. The *MHP* algorithm of [3] can then be modified in the following way. We refer to this refined intraprocedural MHP information as *IMHP*.

Definition 4.1: IMHP(s_1, s_2): For a pair of statements s_1 and s_2, check if they are nested in isolated blocks. In case both s_1 and s_2 are nested in isolated blocks, *MHP*(s_1, s_2) is computed by first determining *lock-set*$(s_1) \cap$ *lock-set*$(s_2) \neq \phi$ and then combining it with the *MHP* information from [3]. If both s_1 and s_2 are not nested in isolated blocks, we obtain *MHP* using [3] as is.

1) Procedure-level MHP (PMHP): We are interested in computing *MHP* information at procedure-level. There are three scenarios in which a procedure f may execute in parallel with another procedure g. We capture this formally in the following definition.

Definition 4.2: PMHP$(f, g) = true$, if

1) f itself or any procedure in the a call-chain originating from f invokes g in an `async` scope but does not enclose it in a preceding `finish` call. (We term this as *escaping MHP* or *EMHP*.)
2) f and g are invoked by the same caller with call sites s_f and s_g respectively and *IMHP*(s_f, s_g) is $true$.
3) transitive closure of (1) and (2) implies *PMHP*$(f, g) = true$.

For computing escaping MHP i.e., *EMHP*, we need to know if the call-site is a children of `ASYNC` or `FINISH` region in the *RST*. This information is then propagated in the call-graph nodes in a bottom-up fashion. We define this recursively as follows:

$$\begin{aligned} \textit{LMHP}(f) = \{&g, \text{such that } \exists \text{ call-site} \\ &s_g \text{ that invokes } g, \text{ and} \\ &\textit{IMHP}(s_g, END) = true.\} \\ \textit{GMHP}(f) = &\bigcup_{g \in \textit{LMHP}(f)} \textit{GMHP}(g) \cup \textit{CALLCHAIN}(g) \end{aligned}$$

CALLCHAIN(g) denotes the set of procedures that are reachable from g in the call graph. Now we define *PMHP*$(f, g) = true$, if $g \in$ *GMHP*(f).

2) Final MHP:

Definition 4.3: Given any two statements, s_1 contained in procedure f_1 and s_2 contained in procedure f_2, *MHP*$(s_1, s_2) = true$, if

- f_1 and f_2 are same procedure and *IMHP*$(s_1, s_2) = true$.

- *PMHP*$(f_1, f_2) = true$.

Parallel Side-effect Analysis: MOD and *REF* side-effects for parallel constructs and procedures are computed using the algorithm described in [6].

Critical load elimination: Under our memory model assumptions, a load from a memory location inside a critical block can be eliminated if, and only if, the same memory location is not concurrently modified by another thread in another isolated block. Formally,

Definition 4.4: Let I denote the set of all isolated blocks of a program. A memory load operation of $\mathcal{H}^x[a]$ in statement s inside $i \in I$ can be eliminated if $\nexists$ another statement s' in another isolated block $j \in I$, such that *MHP*$(s, s') = true$ and $\mathcal{H}^x[a] \in$ *MOD*(s').

Argument Promotion: Frequently executed memory load operations that are hoisted out of isolated procedures should be promoted to arguments to obtain maximum benefits out of load elimination. In most optimizing compilers, the arguments to a procedure are held in machine registers during the execution of the procedure entirely. This enforces pre-coloring of values before register allocation. In reality, registers are scarce and have to be used efficiently not to promote every possible eliminated load to arguments. We decide a set of *hot memory references* and choose these references for argument promotion. The hot memory references are determined either statically or by profiling all the memory operations in isolated procedures. Another important factor to keep in mind during argument promotion is that the underlying target architecture may dictate the number of arguments that can be passed in registers.

Algorithm: Our interprocedural scalar replacement for load elimination algorithm is presented in Figure 5. Applying this algorithm to the example program on the left of Figure 1 (Example 2) yields the transformed code on the right.

E. Non-critical Code Motion (NCM)

In many scenarios, it is both non-trivial and error-prone for a programmer to manually hoist computations that are local out of a critical section. Sometimes, they can only be identified after preprocessing compiler transformations are applied, e.g., in Figure 1(Example 3), both loop distribution and scalar expansion are applied before identifying the critical computations. Therefore, it is more effective and productive for this task of hoisting local computations out of critical sections to be performed by the compiler rather than the programmer.

In order to design an algorithm to determine local computations of a critical section, we describe a program slicing based algorithm [45]. Our algorithm takes the program dependence graph (*PDG*) as input. The *PDG* is made single rooted and represents both data and control dependences among *PIR* statements. Any cycle in the *PDG* is reduced apriori to a single macro node. We start with a set of *async-escaping* statements in critical regions and build a closure of the *PIR* statements from a given critical section that are either data- or control- dependent on the async-escaping statements. The *async-escaping* statements are obtained from well-known escape analysis for *Java* [38], [15] and *HJ* [4]. This set comprises the `non-local` set. Any remaining computations from the critical section are hoisted out of the critical section. The complete algorithm is presented in Figure 6. Applying the algorithm from Figure 6 to the code fragment on the left of Figure 1(Example 3) yields the transformed code on the right.

V. Experimental Results

In this section, we present an experimental evaluation of our critical section optimizations implemented using the framework described in Section IV-A which is implemented on top of the Soot program analysis and optimization framework [42].

A. Experimental Setup

Our experimental evaluation was conducted on 17 explicitly-parallel benchmarks written in Habanero-Java, Java threads, and the Java-based Deuce [25] Software Transactional Memory (STM) system. Among the five *HJ* benchmarks, `Health` was ported from the BOTS benchmark suite [19], `KMeans` & `Montecarlo` were ported from the DPJ benchmark suite [8], and `Tsp` & `Elevator` were obtained from the authors of [43]. These *HJ* benchmarks use `finish`, `async`, and `isolated` constructs. The four *Java* benchmarks include `SpecJBB 2000`, `JCilk` [28], `Xalan` [7], and `PMD` [7]. For the `JCilk` evaluation, we used the Fibonacci JCilk program with input size 36 since it exercises many of the JCilk runtime interfaces. Finally, we also used eight benchmarks from the JSTAMP suite [2] — `Bayes`, `Intruder`, `Genome`, `Vacation`, `Yada`, `SSCA2`, `Labyrinth3D`, and `MatrixMul`. The STM runtime used in our evaluation is the Deuce software transactional memory subsystem [25].

While additional benchmarks could certainly be used, Table I and Figure 7 indicate that the 17 benchmarks show sufficient variation in characteristics to provide a comprehensive evaluation. Of course, we would not expect the benchmarks in which critical sections do not occur in hot paths to benefit from our optimizations.

All results were obtained on two platforms: 1) a 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30GB of memory running Red Hat Linux (RHEL 5) and Sun JDK 1.6 (64-bit version); 2) a 32-core (quad-socket, 8 cores per socket) 3.55GHz Power7 with 256 GB main memory running Red Hat Linux (RHEL 6.1) with SMT turned off and IBM

```
1  function CriticalLoadElimination ()
     Input  : Callgraph (CG) and Parallel intermediate representation (PIR) for each procedure of CG
     Output: Modified PIR and CG
2    Perform Interprocedural May-Happen-in-Parallel (MHP) analysis (Section IV-D);
3    Perform Side-Effect Analysis [6];
4    list := set of isolated region nodes and procedures of the whole program;
5    for each procedure, P, in CG in reverse topological sort order do
6        for each isolated region, R, in PIR(P) do
7            mods := scalars := φ;
8            for each isolated region, R', in list do
9                if MHP(R, R') == true then
10                   mods := mods ∪ MOD(R');
11           for H^x[a] in REF(R) − mods do
12               scalars := scalars ∪ H^x[a];
13           if scalars != φ then
14               if R is procedure P itself then
15                   promotes = SelectScalars(scalars, P);
16                   if promotes == scalars then
17                       PromoteScalars (scalars, P);
18                   else
19                       Create a new wrapper non-isolated procedure P' that internally calls P;
20                       Updates call sites of P to invoke P' instead of P;
21                       Update CG;
22                       PromoteScalars (P, scalars);
23                       PromoteScalars (P', promotes);
24               else
25                   Save the memory load for H^x[a] in scalar temporary H_x_a outside R;
26                   Replace memory loads to H^x[a] in R by H_x_a;
27   return;
28   function PromoteScalars ()
     Input  : Procedure P; List of heap-arrays that need to be promoted to scalars scalars
     Output: New procedure P'
29     P' := clone of P with original argument list;
30     list_temp := φ;
31     callsites := set of callsites for P;
32     for each call site, s, in callsites do
33         for H^x[a] in scalars do
34             Save the memory load for H^x[a] in scalar temporary H_x_a before s;
35             list_temp := list_temp ∪ H_x_a;
36         Modify s to call P' with appended actual arguments from list_temp;
37     Add the scalars of list_temp as formal arguments to P';
38     for H^x[a] in scalars do
39         Replace memory loads to H^x[a] in P' by H_x_a;
40     Remove redundant arguments from P';
41     Update CG;
42     return P';
43     function SelectScalars ()
         Input  : List of heap-arrays scalars; Procedure P
         Output: list of heap-arrays from scalars that may benefit from argument promotion
44       return Select most frequently executed heap-arrays from scalars based on profiling information and architectural consideration;
```

Fig. 5. Algorithm for *interprocedural scalar replacement for critical load elimination*

JDK 1.6 (64-bit version). Since Deuce STM does not run on Power7 system, we do not report STM performance on Power7 SMP. All *HJ* and *Java* results were obtained using the `-Xmx6G` JVM option to limit the heap size to 6GB. The JSTAMP results were obtained using the option `-Xmx10G` to be able to run largest input sizes and to reduce GC impacts. For all runs, the execution times use the methodology described in [23], *i.e.,* we report the average runtime performance of 30-runs within a single VM invocation. Additionally, our results are obtained using the -server option, which eliminates the need for warm-up runs.

For our evaluation, we choose the hot memory references for CLE transformation based on static frequency estimates as in register allocators (a memory reference inside a loop-nest is estimated as 10^d, where d denotes loop depth). We select the top k memory operations in a critical section based on these estimates and promote them to arguments. k is dictated by the architecture and is set to 6 for our X86 evaluation and 10 for PowerPC evaluation.

One important factor to note is that the load elimination algorithm in Section IV-D can have a potentially negative impact due to the increase in the size of live ranges which may then lead to increased register pressure. This, in turn, may cause a performance degradation if the register allocator does not perform live-range splitting. To mitigate this problem, we introduce

```
1  function CodeMotionOpt ()
     Input  : Program dependence graph PDG, Call graph CG, and PIR for each method; S=Set of PDG nodes that are in critical sections and contain object accesses which
              are async-escaping;
     Output: Modified PIR for each critical method with local computations, moved out of critical regions
2    for each critical region, R do
3        Apply preprocessing transformations such as loop distribution, loop unswitching, and scalar expansion;
4        for each node, n, in PDG do
5            visited[n] = false;
6        L = φ;
7        for each node, n, in S do
8            ComputeNonLocalSlice(n, visited, L);
9        Transform the PIR to move code not in L to outside of R;
10   return Modified PIR
11   function ComputeNonLocalSlice(n, visited, slice)
       Input  : Node n, visited array, slice obtained so far
       Output: Updated slice
12     if node n is not visited then
13         visited[n] = true;
14         slice = slice ∪ {n};
15         for each node m that n is directly dependent upon do
16             ComputeNonLocalSlice(m, visited, slice);
```

Fig. 6. Perform *non-critical code motion* opt.

a compiler pass that splits live ranges at critical section boundaries by introducing new local variables. Although this does not guarantee the absence of spills in a critical section, it increases the opportunity for the register allocator to assign variables in critical sections to registers.

B. Experimental Results

Reduction in load operations: Column 2 in Table I reports the static number of critical sections in the given benchmark. Columns 3 and 4 show the effect of critical load elimination transformation (from Section IV-D) by comparing the dynamic number of load operations in the program for all critical sections before and after the optimizations are performed. Column 5 reports the total number of dynamic load operations (within and outside critical sections) in a benchmark without optimization. For our optimizations to have a real impact on execution time, we would need to see a reduction from column 3 (unoptimized count of loads in critical sections) to column 4 (optimized count of loads in critical sections) and also column 3 be a significant fraction of the total count in column 5. Column 6 shows that `SpecJBB`, `Elevator`, and all the `STAMP` benchmarks contain a large fraction of loads in critical sections. The `STAMP` benchmarks' large load fraction is not surprising since they target transactional systems. Column 7 reports the percentage reduction in load operations in critical sections by our optimizations. Columns 8, 9 and 10 describe the specific optimization from sections IV-C (PSR), IV-D (CLE), and IV-E (NCM) that have been applied by our compiler to each benchmark.

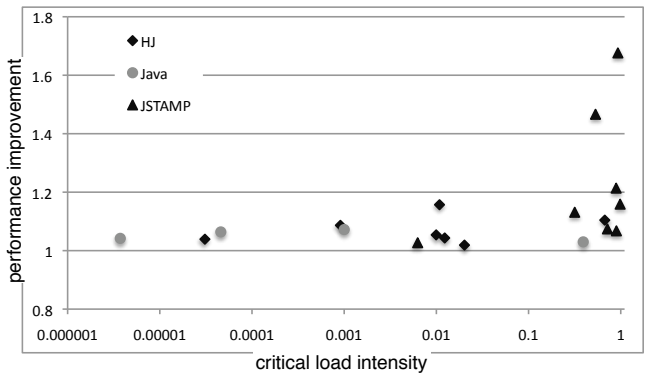

Fig. 7. Critical Load Intensity (X-axis) vs. performance improvements (Y-axis) for 16 thread case on Xeon SMP.

Analysis of experimental results: Figure 7 shows the relation between *critical load intensity* (*i.e.,* the ratio of dynamic load operations in critical section compared to all load operations performed in the program) vs. the performance improvements achieved by applying our optimizations on all 17 benchmarks (on the Xeon platform). Most of the JSTAMP benchmarks (barring `SSCA2`) have high critical load intensity due to their large critical sections, and thus, they exhibit high performance improvements (a maximum improvement of 1.68×). In contrast, most of the `HJ` and `Java` benchmarks have lower *critical*

load intensity (except `Elevator` and `SpecJBB`) due to smaller critical section sizes. However, our optimizations still yielded respectable performance improvements of up to 1.16× for the HJ benchmarks and up to 1.08× for the Java benchmarks.

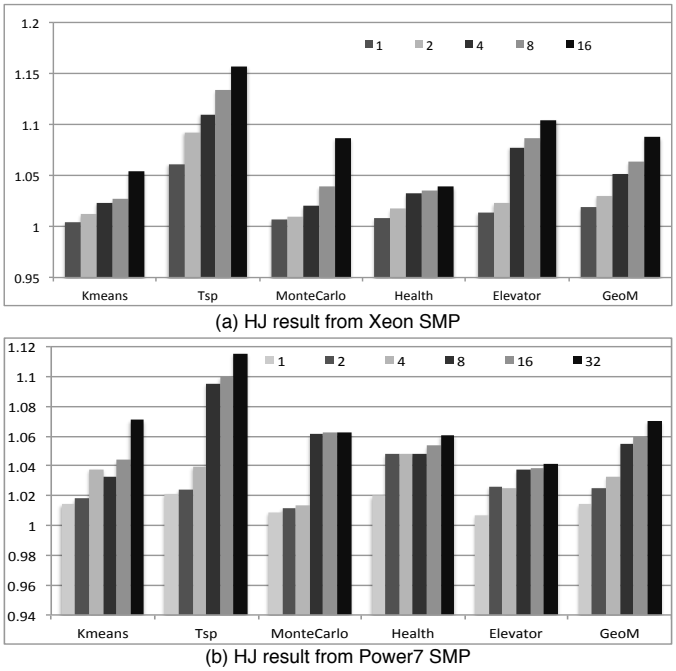

Fig. 8. Performance Improvements for *HJ* benchmarks using different number of threads on Xeon and Power7 SMPs with our optimizations rel. to their unoptimized versions using the same number of threads.

Runtime impact on HJ benchmarks: Figure 8 presents the relative performance improvement for *HJ* benchmarks using 1-16 threads on Xeon SMP and using 1-32 threads on Power7 SMP with respect to their unoptimized versions using the same number of threads. For 16-thread Xeon SMP (see Figure 8 (a)), the compiler transformations described in this paper resulted in a maximum speedup of 1.16× and a geometric mean speedup of 1.09× over no optimization. On 32-thread Power7 SMP (see Figure 8 (b)), the transformations achieve a maximum speedup of 1.11× and a geometric mean speedup of 1.07× over no optimization. `Tsp`, and `Elevator` benchmarks show significant speedup with our techniques primarily because of PSR transformation (for `Tsp`) and the reduction in the dynamic number of load operations after optimizations (in Table I), respectively. For single thread case, our optimizations yield lesser performance improvements (maximum speedup of 1.06× and geometric mean speedup of 1.01×) due to no contention for critical sections. Note that, the best improvement due to our optimization is always observed for 16-cores on Xeon SMP and for 32-cores on Power7 SMP.

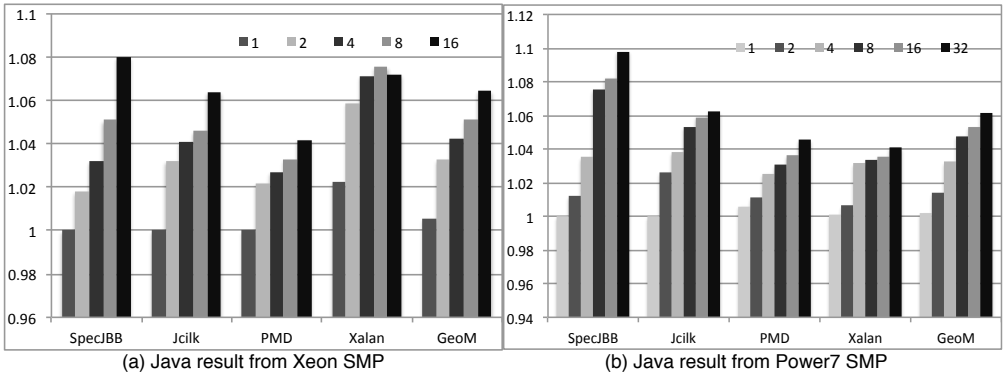

Fig. 9. Speedups for *Java* benchmarks using different number of threads on Xeon and Power7 SMPs with our optimizations relative to their unoptimized versions using the same number of threads.

Runtime impact on Java benchmarks: Figure 9 presents the relative performance improvement for *Java* benchmarks using

1-16 threads on Xeon and 1-32 threads on Power7 with respect to their unoptimized versions using the same number of threads. For 16-thread Xeon (see Figure 9 (a)), we observe a maximum speedup of $1.08\times$ and a geometric mean speedup of $1.06\times$ using our optimizations over no optimization. On 32-thread Power7 SMP (see Figure 9 (b)), we observe a maximum speedup of $1.10\times$ and a geometric mean speedup of $1.06\times$ using our optimizations over no optimization. The `SpecJBB` and `JCilk` benchmarks show large benefits due to our PSR transformation and the reduction in the dynamic number of load operations after optimizations (in Table I).

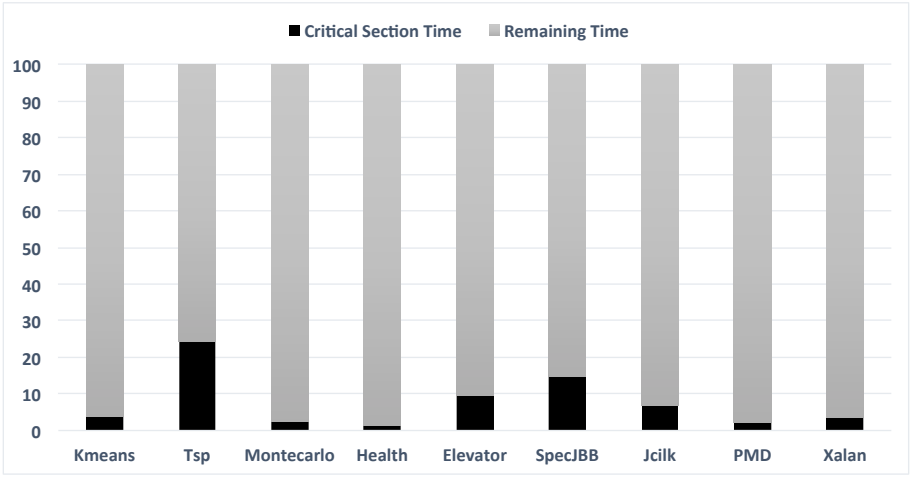

Fig. 10. Percentage of time spent in critical section compared to total execution time using single thread for *Java* and *HJ* benchmarks on Xeon SMP.

Analysis of *Java* and *HJ* results: The performance improvements in the *Java* and *HJ* benchmarks depend on the opportunities that remain after performance tuning by the benchmark developers. Some workloads have been developed by domain experts (e.g., PMD) and offer few opportunities for additional optimizations as a result. To understand this better, we collected the percentage of time spent in critical section for *Java* and *HJ* benchmarks for single-threaded execution. This is shown in Figure 10. The benchmarks such as `Tsp`, `Elevator`, `SpecJBB`, and `JCilk` spend relatively higher fraction of time in critical section with 24.3%, 9.7%, 14.8%, and 6.9% respectively. As a result, these benchmarks yield better performance improvements using our optimizations.

Our experimental results also show performance improvements for single thread case in *HJ* and *Java* benchmarks. This is because our code motion transformations not only hoist memory load operations out of critical sections, they enable other optimization opportunities for these hoisted operations in the contained non-critical code regions. For example:

```
1  for(i=0;i<N;i++) {
2   isolated { //critical section
3     = a.x;
4   }
5  }
```

In the above code, once the memory load of $a.x$ is hoisted out of the critical section by our optimization, it can further be hoisted out of the outer for-loop. This may result in performance improvement for sequential execution.

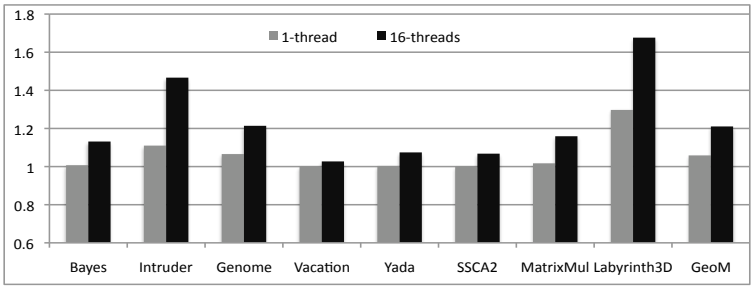

Fig. 11. Speedups for JSTAMP benchmarks using Deuce STM runtime for single thread and 16 threads cases on Xeon SMP with our optimizations relative to their unoptimized versions using the same number of threads.

Runtime impact on STM benchmarks: Figure 11 shows the speedups obtained from running JSTAMP benchmarks using Deuce STM runtime before and after applying our optimizations. For these applications, the performance improvement

Benchmark	static # of critical sections	(A): Loads in crit. sections w/o opts	(B): Loads in crit. sections w/ opts	(C): Total loads	A/C %age	B/A %age	PSR	CLE	NCM
KMeans	1	4.11E+09	2.72E+09	4.13E+11	9.95E-01	6.62E+01		×	
Tsp	6	3.98E+06	3.18E+06	3.68E+08	1.08E+00	7.99E+01	×	×	×
MonteCarlo	1	1.81E+06	1.20E+06	1.99E+09	9.10E-02	6.63E+01		×	
Health	1	3.06E+04	0	9.92E+08	3.08E-03	0.00E+00		×	
Elevator	8	2.12E+09	1.88E+09	3.16E+09	6.71E+01	8.87E+01		×	
SpecJBB	52	4.78E+09	4.62E+09	1.22E+10	3.92E+01	9.67E+01	×	×	×
JCilk	76	2.78E+04	2.78E+04	6.06E+08	4.59E-03	1.00E+02	×		
PMD	2	3.32E+04	8.89E+03	8.91E+09	3.73E-04	2.68E+01		×	×
Xalan	44	9.36E+06	3.12E+06	9.35E+09	1.00E-01	3.33E+01		×	×
Bayes	10	5.32E+09	5.02E+09	1.68E+10	3.17E+01	9.44E+01		×	
Intruder	3	8.36E+08	7.11E+08	1.57E+09	5.32E+01	8.50E+01		×	×
Vacation	3	2.22E+09	2.19E+09	2.49E+09	8.92E+01	9.86E+01		×	
SSCA2	3	8.90E+07	8.59E+07	1.42E+10	6.27E-01	9.65E+01		×	
Genome	5	1.47E+09	1.46E+09	2.05E+09	7.17E+01	9.93E+01		×	×
Yada	6	1.75E+10	1.74E+10	1.95E+10	8.97E+01	9.94E+01		×	
MatrixMul	1	3.38E+07	3.36E+07	3.43E+07	9.85E+01	9.94E+01		×	
Labyrinth3D	3	3.11E+08	1.69E+08	3.34E+08	9.31E+01	5.43E+01		×	×

TABLE I
STATISTICS OF BENCHMARKS' FEATURE, DYNAMIC LOADS AND TRANSFORMATIONS APPLIED.

mainly comes from critical load elimination (Section IV-D) and non-critical code motion (Section IV-E) transformations. These optimizations reduce the contention within the code regions that run speculatively. `Vacation`, `SSCA2`, and `Yada` did not show obvious improvements since there is not much scope for critical load elimination transformation. For 16-thread case, we observe a maximum speedup of $1.68\times$ and a geometric mean speedup of $1.21\times$ over no optimization. For single thread case, a maximum speedup of $1.29\times$ and a geometric mean speedup of $1.05\times$ were observed. `Labyrinth3D` benchmark shows maximum speedup of $1.68\times$ for 16-thread case as both CLE and NCM optimizations are applicable. The large performance improvements in STM benchmarks is due to the fact that STM systems incur multiple overheads such as logging memory accesses. By reducing the number of memory operations within a transaction, the overhead of logging also reduced.

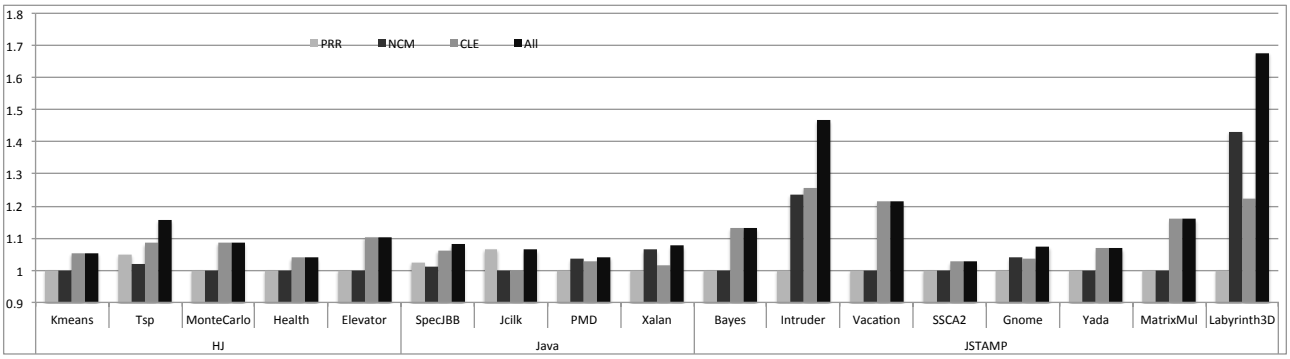

Fig. 12. Break down of the 3 transformations using 16-threads on the Xeon SMP for all benchmarks.

Impact of each optimization separately: The breakdown of the effect of individual optimization is presented in Figure 12. It compares the speedup obtained from 16-thread case on Intel Xeon SMP by applying our 3 transformation separately and also together. As shown in Table I, most of the benchmarks benefit from `CLE`. `PSR` contributes performance improvement by eliminating the redundant lock operations for both `HJ` and `Java` benchmarks for `Tsp`, `SpecJBB` and `JCilk` benchmarks. For `HJ` and `Java` benchmarks, `NCM` moves arithmetic operations and non-critical memory copy operations outside of the critical sections for `Tsp`, `SpecJBB`, `PMD`, and `Xalan`. For `Xalan`, `NCM` moves non-critical exception processing code outside of the critical section, this brings 7% performance improvement. For STM benchmarks, `NCM` brings significant improvement, due to moving the object allocation operations outside of critical sections.

Comparison with prior work [6]: Since the algorithm in Barik-Sarkar [6] targeted only *HJ* benchmarks, in Figure 13 we compare our approach to theirs for *HJ* benchmarks using 16-threads and 32-threads. Their approach unified side-effects for all isolated blocks of a benchmark and, thus, is expected to underperform compared to our approach. For `Tsp`, we benefit from both load elimination and argument promotion transformations described in Section IV-D.

The above results establish the fact that our optimizations reduce the amount of time spent in critical sections in most scenarios, and the difference in performance is noticeable for larger number of threads where there is contention.

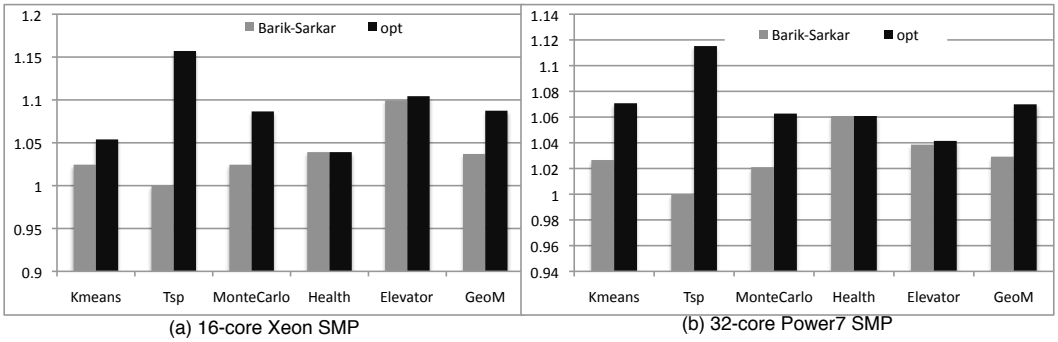

Fig. 13. Comparison of Barik-Sarkar [6] vs. our optimizations using 16-threads on Xeon SMP and using 32-threads on Power7 SMP for *HJ* benchmarks.

VI. Related Work

Optimization of explicitly parallel programs has been studied by several researchers [43], [6], [24], [29]. Among these optimizations, *Scalar Replacement* [10], [12], [9], [31] is a classic program optimization which replaces memory load operations by scalar temporaries that subsequently get allocated in registers or scratchpads for better performance and power. Praun et al. [43] 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 a conflict analysis [44] and the transformation obeyed the Sequential Consistency (SC) memory model. Most recently, Barik and Sarkar [6] described an interprocedural scalar replacement for load elimination for *HJ* and X10 programs. They describe scalar replacement across both method calls and parallel constructs (such as `async`, `finish`, and `isolated or atomic`), using the isolation consistency memory model. Our work focuses primarily on critical sections and proposes three compiler transformation algorithms that attempt to reduce the amount of time spent in a critical section. In one of the transformations, we extend [6] to make their load elimination algorithm MHP-aware [18], [35], [3], [30] for better precision and also handle procedures that are critical, but not inlinable via argument promotion.

The safety of compiler optimizations in the presence of concurrency constructs is strongly tied to the underlying memory model. There has been a lot of past work that illustrates the benefits of different memory models for certain classes of optimizations [34], [22]. Recently, Joisha et al. [24] presented a framework that showed how to reuse the classic program optimizations (*i.e.,* optimizing sequential program) to optimize the parallel program. Our work focuses primarily on memory load operations and redundant computations within critical sections for explicitly parallel programs using their respective memory models which are weaker than SC.

There has been a large body of work on removing unnecessary synchronization operations from *Java* programs [5], [38], [15]. The outcome of our technique is not the same as synchronization removal proposed in past works since our approach eliminates *partially* redundant synchronization operations. Compared to the lock indepenedent code motion algorithm in [37], our non-critical code motion algorithm does not require a concurrent-SSA form representation and is both interprocedural and MHP-aware.

Recent work by [11], [36] focus on the placement of synchronization operations in the context of automatic parallelization of C/C++ programs. Their goal is to uncover more opportunities for automatic parallelization of sequential programs (esp. loops with carried dependencies) and then schedule the synchronization operations to preserve data dependences. The techniques described in our paper are for explicit task-parallel languages and take advantage of interprocedural information as opposed to dependence-based synchronization across loop iterations described in the above prior works.

VII. Conclusions and Future Work

In this paper, we present compiler optimizations for eliminating and hoisting redundant operations out of critical regions of explicitly parallel programs in order to improve performance and scalability. Specifically, we describe three transformations with their accompanying legality conditions: 1) *partial strength reduction of critical operations* to replace critical sections by non-critical code blocks on certain control flow paths; 2) *critical load elimination* to replace memory accesses within critical sections by scalar temporaries; and 3) *non-critical code motion* to identify and hoist local computations out of critical sections; Using *HJ* and *Java* runtimes, our preliminary experimental results for these benchmarks show performance improvements of up to $1.16\times$ and $1.08\times$ on Intel Xeon SMP by using 16-Cores respectively. On the Power7 SMP, we observe performance improvements of up to $1.11\times$ and $1.10\times$ for *HJ* and *Java* runtimes using 32-Cores respectively. Furthermore, by using a software transactional memory subsystem, we observe a much better performance improvement with a maximum speedup of $1.68\times$ and average speedup of $1.21\times$ for JSTAMP benchmarks. We are confident that the optimizations described in this paper would achieve better performance and scalability for upcoming many-core processors with hundreds of cores per chip. Possible

directions for future work include extending our optimizations to handle more advanced synchronization constructs of *HJ* and *Java* such as barriers, futures, and phasers.

ACKNOWLEDGMENT

We would like to thank all members of the Habanero team at Rice for valuable discussions and feedback related to this work. We gratefully acknowledge Christoph von Praun for providing us the multi-threaded Java version of the *Tsp* and *Elevator* benchmarks. 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. This research was supported in part by the National Science Foundation through grants CCF-0926127 and CCF-0964520.

REFERENCES

[1] Jikes RVM. http://jikesrvm.org/.
[2] JSTAMP. http://demsky.eecs.uci.edu/software.php.
[3] S. Agarwal et al. May-happen-in-parallel analysis of X10 programs. In *PPoPP'07*.
[4] S. Agarwal et al. Static detection of place locality and elimination of runtime checks. In *APLAS'08*.
[5] J. Aldrich et al. Static analyses for eliminating unnecessary synchronization from Java programs. In *SAS'99*.
[6] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT'09*.
[7] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*.
[8] R. L. Bocchino Jr et al. A type and effect system for deterministic parallel Java. In *OOPSLA'09*.
[9] R. Bodík et al. Load-reuse analysis: design and evaluation. *SIGPLAN Not.*, 34(5), 1999.
[10] D. Callahan et al. Improving Register Allocation for Subscripted Variables. In *PLDI'90*.
[11] S. Campanoni et al. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In *CGO'12*.
[12] S. Carr and K. Kennedy. Scalar Replacement in the Presence of Conditional Control Flow. In *SPE'94*.
[13] V. Cavé et al. Habanero-java: the new adventures of old X10. In *PPPJ'11*.
[14] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*.
[15] J. Choi et al. Escape analysis for java. In *OOPSLA'99*.
[16] K. D. Cooper and Li Xu. An efficient static analysis algorithm to detect redundant memory operations. *SIGPLAN Not.*, 2003.
[17] Dave Dice et al. Transactional locking II. In *DISC'06*.
[18] E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV'91*.
[19] A. Duran et al. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP'09*.
[20] T. Elmas et al. Goldilocks: a race and transaction-aware java runtime. In *PLDI'07*.
[21] S. J. Fink et al. Unified Analysis of Array and Object References in Strongly Typed Languages. In *SAS'00*.
[22] G. R. Gao and V. Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Trans. Comput.*, 2000.
[23] A. Georges et al. Statistically rigorous Java performance evaluation. In *OOPSLA'07*.
[24] P. G. Joisha et al. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL*, 2011.
[25] G. Korland et al. Noninvasive concurrency with java stm. In *MULTIPROG'10*.
[26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), September 1979.
[27] A. Le et al. Using inter-procedural side-effect information in JIT optimizations. In *CC'05*.
[28] I-Ting A. Lee. The JCilk multithreaded language. Master's thesis, MIT, Dept of Electrical Engineering and Computer Science, August 2005.
[29] J. Lee et al. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *LCPC'97*.
[30] J. K. Lee et al. Efficient may happen in parallel analysis for async-finish parallelism. In *SAS'12*.
[31] R. Lo et al. Register promotion by sparse partial redundancy elimination of loads and stores. *SIGPLAN Not.*, 1998.
[32] J. Lu and K. D. Cooper. Register promotion in C programs. In *PLDI'97*.
[33] Jeremy Manson et al. The Java memory model. In *POPL '05*.
[34] D. Marino et al. A case for an sc-preserving compiler. In *PLDI'11*.
[35] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *FSE'98*.
[36] A. Nicolau et al. Synchronization optimizations for efficient execution on multi-cores. In *ICS'09*.
[37] D. Novillo et al. Optimizing mutual exclusion synchronization in explicitly parallel programs. In *LCR'00*.
[38] E. Ruf. Effective synchronization removal for java. In *PLDI'00*.
[39] V. Sarkar. Automatic selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM J. Res. Dev.*, 41(3), 1997.
[40] M. A. Suleman et al. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS'09*.
[41] M. A. Suleman et al. Data marshaling for multi-core architectures. In *ISCA'10*.
[42] R. Vallée-Rai et al. Soot - a Java bytecode optimization framework. In *CASCON'99*.
[43] C. von Praun et al. Load elimination in the presence of side effects, concurrency and precise exceptions. In *LCPC'03*.
[44] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI '03*.
[45] M. Weiser. Program slicing. In *ICSE'81*.
[46] Jaeheon Yi et al. Cooperative reasoning for preemptive execution. In *PPoPP'11*.