Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections

Yuan Zhang[1], Vugranam C. Sreedhar[2], Weirong Zhu[3]**, Vivek Sarkar[4], and Guang R. Gao[1]

[1] University of Delaware, Newark, DE, {zhangy,ggao}@capsl.udel.edu
[2] IBM T.J.Watson Research Center, Hawthorne, NY, vugranam@us.ibm.com
[3] Microsoft Corporation, Seattle, WA, weirong.zhu@microsoft.com
[4] Rice University, Houston, TX, vsarkar@rice.edu

Abstract. In this paper we propose a lock assignment technique to simplify the mutual exclusion enforcement in multithreaded programs. Programmers are allowed to annotate the regions of code that are expected to be mutually exclusive as `critical` sections, without using explicit locks. The compiler then automatically infers an assignment of the minimum number of locks to critical sections by solving the Minimum Lock Assignment (MLA) problem so as to enforce mutual exclusion without any loss of concurrency. We show that the MLA problem is NP-hard. We have proposed a heuristic to solve the MLA problem, and tested the optimality of the heuristic with the Integer Linear Programming (ILP) solver. We have also tested the efficiency of the heuristic using scientific applications, from which we obtain up to 30% performance gain with respect to the programs in which all critical sections are controlled by a single lock.

1 Introduction

Given that the processors in current and future computer systems are becoming multi- or many-core by default, it is important to address the performance and productivity issues in multithreaded programming. One of the major performance and productivity issues in multithreaded programming arises from enforcing the mutual exclusion (mutex for short) using lock/unlock operations. Programmers explicitly assign lock variables to control mutex regions, and the lock variables are acquired by the executing thread before the mutex region is executed, and are released after the execution of the mutex region completes. Explicitly managing multiple locks is error prone since it is easy for programmers to introduce data races and create deadlocks. Alternatively, programmers can use a single lock to control all mutex regions to avoid deadlocks and data races. However, they lose concurrency among mutex regions by unnecessarily serializing them.

** The author participated this work when he was a graduate student in the University of Delaware

In this paper, we propose a lock assignment technique to simplify the enforcement of mutual exclusion in multithreaded programs. We allow the programmers to annotate regions of code that are expected to be executed mutually exclusively as `critical` sections, without managing explicit locks. The compiler then automatically infers an assignment of multiple compiler-managed locks to critical sections (possibly multiple locks for one critical section) to preserve the mutual exclusion and also exploit the concurrency among critical sections.

A naive lock assignment approach associates one lock to each shared memory location, and the lock set of a critical section is the set of locks assigned to memory locations it accesses. This approach, however, may use more locks than necessary, and introduce excessive overhead on lock acquisition and release. To control the locking overhead, we would use the minimum number of locks which is necessary to preserve the mutual exclusion and fully exploit the concurrency between critical sections. We formulate this lock assignment task as the **Minimum Lock Assignment** (MLA) problem:

Problem 1 (Minimum Lock Assignment). Given a multithreaded program with a set of critical sections, find the minimum number of distinct locks that are needed for controlling the critical sections such that

(a) Two critical sections are assigned disjoint sets of locks if (1) they are concurrent and (2) they do not access any common location, or if they access a common location then none of them writes to the common location.

(b) Two critical sections are assigned at least one common lock if (1) they are concurrent and (2) they access some common location and at least one of them writes to the common location.

Note that a critical section can be assigned a set of locks. The semantics of a lock set follows the strict two-phase locking policy [1].

The solution of the MLA problem consists of two main phases: the analysis phase and the lock assignment phase. In the analysis phase, the compiler reads the multithreaded program and statically determines whether a pair of critical sections are interfering. Two critical sections are interfering if they are concurrent and they access some common shared memory location(s), with at least one of them writes to the common location(s). In the lock assignment phase the compiler calculates the minimum number of locks to control critical sections according to the analysis result, and assigns one or more locks to each critical section. Besides, the runtime system guarantees that an execution is deadlock free by acquiring and releasing locks in a predetermined order. The analysis phase is solved by concurrency analysis, data set analysis and pointer analysis. However, due to the space limitation, in this paper we simply assume the analysis result is already calculated, and we only focus on the lock assignment phase. Readers can refer to [2, 1] for more details on the analysis phase and deadlock avoidance options. In the following discussion we refer to the MLA problem as the lock assignment phase exclusively, without any further clarification.

The rest of the paper is organized as follows. In Section 2 we introduce the concurrency graph as the main data structure to solve the MLA problem. In

Section 3 we prove that MLA problem is related to the graph coloring problem and it is NP-hard. We then present a heuristic to solve the MLA problem. We also formulate the MLA problem as an Integer Linear Programming (ILP) problem. In Section 4 we evaluate our heuristic by comparing it with optimal solutions produced by the commercial ILP solver CPLEX. In 300 randomly generated testing cases we observe that our MLA heuristic is optimal for 83.3% of them. We also test the performance of our heuristic using a 10-way Sunfire machine on a set of Splash2 [3] benchmarks, and obtain up to 30.17% performance speedup with respect to programs in which all critical sections are controlled by a single lock. Related work is presented in Section 5, and finally we conclude in Section 6.

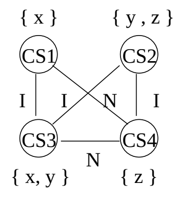
2 Concurrency Graph and Critical Sections

In this section, we introduce the *concurrency graph* to model the potential concurrency and interference among critical sections in a multithreaded program.

2.1 Concurrency Graph

```
cobegin
   critical { x ++; } // CS1
   critical { y ++; z ++; } // CS2
   ||
   critical { x ++; y ++; } // CS3
   ||
   critical {z ++; } // CS4
coend
```

(a)



(b)

Fig. 1. (a) Example program (b) Concurrency graph

Definition 1. A **Concurrency Graph** is an undirected graph $G = (V, E)$, in which: a vertex $v \in V$ denotes a textual critical section, and there is an edge $(u, v) \in E$ if instances of critical sections u and v may be concurrent.

In the above definition, if two instances of the critical section u are concurrent, we do not introduce a self-loop on u, since we will assign at least one lock to each critical section, and the mutual exclusion of u with respect to itself is self preserved. As an example, Figure 1(b) illustrates the concurrency graph for the program shown in Figure 1(a). The set of shared memory locations that are accessed within critical sections are also listed within curly braces in Figure 1(b).

Two concurrent critical sections are said to be **non-interfering** if either they do not access a common location or if they access a common location then none of them writes to the common location. Two concurrent critical sections are **interfering** if they access some common location and at least one of them writes to the common location. We extend the concurrency graph defined in Definition 1 by labeling an edge (u, v) with label I when critical sections u and v are interfering, and with label N when u and v are non-interfering.

Note that a general concurrency graph may be a forest of connected graphs, and we analyze each connected component independently. In the following discussion, we simply assume that a concurrency graph G is a connected graph.

2.2 Non-interfering Concurrency Graphs

Consider a class of multithreaded programs P_n whose corresponding concurrency graph contains only non-interfering edges. Since all incident edges of a critical section are non-interfering, it cannot share any lock with its neighbors. This implies that whenever two critical sections are connected (concurrent), they require different locks. We can now rephrase the MLA problem (Problem 1) for non-interfering concurrency graphs as follows:

Problem 2. Given a program with a set V_n of non-interfering critical sections, find the minimum number of locks that can be assigned to critical sections such that if two different critical sections in V_n are concurrent then they get different locks.

The above problem is equivalent to the classical graph coloring problem — color the vertices (critical sections) of a graph using the minimum number of colors (locks) such that no two adjacent (concurrent) vertices (critical sections) are given the same color (lock). The MLA problem for this special class of programs is NP-hard[5].

2.3 Interfering Concurrency Graphs

Consider a class of programs P_i, for which the concurrency graph contains only interfering edges. In this case, two critical sections are either concurrent and interfering, or are not concurrent (not connected). If they are concurrent and interfering, they should share at least one common lock to preserve the mutual exclusion, which implies that they must be serialized. If they are not concurrent, they are already serialized. Therefore, in this interfering special case, there is no inherent concurrency, so we can use a single lock to control all critical sections without introducing any performance penalty.

2.4 Concurrency Graph Partition

In general cases, a concurrency graph contains both non-interfering edges and interfering edges. Given a concurrency graph $G = (V, E)$, let E_n denote the set

[5] For certain classes of graphs, such as the interval graphs, the graph coloring problem can be solved in polynomial time. However, the general concurrency graphs are not necessarily interval graphs.

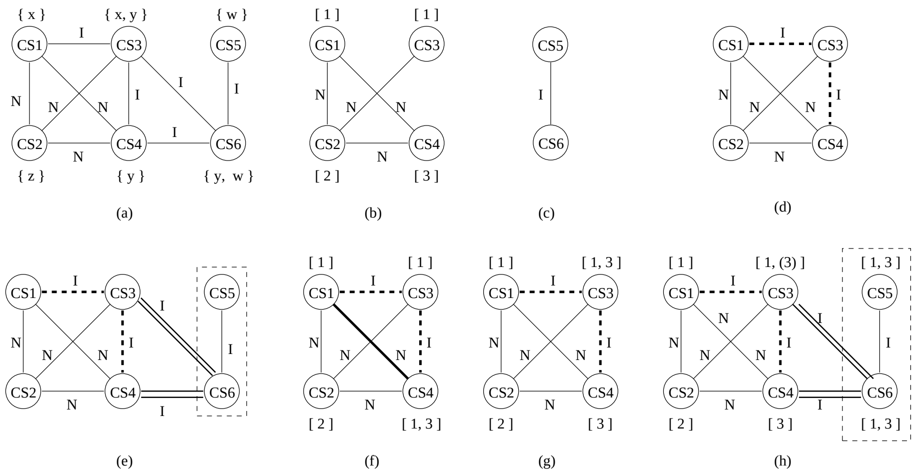


Fig. 2. (a) A general concurrency graph (b) The non-interfering subgraph G_n (c) The interfering subgraph G_i (d) The SNIG G_n^s (e) The crossing edges (double lines), serializing interfering edges (dotted lines), and the interfering subgraph (in dotted box) (f) A un-safe borrowing from CS_3 to CS_4 (g) A safe borrowing from CS_4 to CS_3 (h) Final lock assignment result

of non-interfering edges and E_i denote the set of interfering edges in G, such that $E = E_n \cup E_i$ and $E_n \cap E_i = \emptyset$. Let $G_n = (V_n, E_n)$ be the *non-interfering subgraph* induced by E_n, where $V_n \subseteq V$ such that a vertex $v_n \in V_n$ has at least one non-interfering edge incident on it. Figure 2(b) illustrates the non-interfering subgraph of Figure 2(a). Let $G_i = (V_i, E'_i)$ be the *interfering subgraph* induced by vertices V_i, where $V_i = V - V_n$ and $E'_i \subseteq E_i$ is a set of interfering edges (u_i, v_i) such that $u_i, v_i \in V_i$. Figure 2(c) illustrates the interfering subgraph for Figure 2(a). Finally, let $E''_i = E_i - E'_i$ be a set of interfering edges that are not in G_i. Some of interfering edges in E''_i connect vertices of the non-interfering subgraph, for example, edges (CS_1, CS_3) and (CS_3, CS_4), as illustrated as bold dashed lines in Figure 2(d). We call such interfering edges that occur inside a non-interfering subgraph as *serializing* interfering edges E_s, because they could "serialize" the inherent concurrency that exists within non-interfering subgraph. The remaining interfering edges $E_{ci} = E''_i - E_s$ are *crossing edges* between vertices in G_n and G_i. In the example shown in Figure 2(a), $E_{ci} = \{(CS_3, CS_6), (CS_4, CS_6)\}$, illustrated as double solid lines in (e). Besides the non-interfering subgraph G_n and the interfering subgraph G_i, we introduce the notion of the *serializing non-interference graph* (SNIG) as the non-interfering subgraph with serializing edges, $G_n^s = (V_n, E_n \cup E_s)$. Figure 2(d) illustrates an example of SNIG. SNIGs have some interesting properties that will influence the lock assignment.

2.5 Serializing Non-Interference Graph

Let us consider a class of concurrency graphs called *Serializing Non-Interfering Graphs* (SNIGs). A SNIG consists of only non-interfering edges and serializing interfering edges (as defined in the previous section). Serializing interfering edges constrain the inherent concurrency in a non-interfering concurrency graph. They also constrain the minimum number of locks required to color a SNIG.

The following observation states that sometimes it is impossible to color a SNIG if a vertex can be assigned only one color.

Observation 1. It is impossible to color an arbitrary SNIG with the following conflicting constraints:

1. Each vertex gets only one color,

2. If vertices u and v are connected by a non-interfering edge then they are given two different colors,

3. If two vertices u and v are connected by a serializing interfering edge then they are given the same color.

Consider the SNIG in Figure 3. Assume we satisfy all above constraints, then all critical sections get the same lock, because they are connected by serializing interfering edges (CS_1, CS_3), (CS_3, CS_4) and (CS_4, CS_2). However, the constraint (2) requires that CS_1 and CS_2 are given two different colors, a contradiction. Therefore Figure 3 cannot satisfy all three constraints.

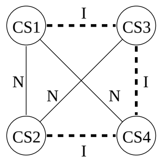


Fig. 3. Example SNIG for Observation 1

There are two ways to handle the above impossibility: relax constraint (1) in the above observation, or relax constraint (2). By relaxing constraint (1) we are allowed to assign multiple colors to each vertex. By relaxing constraint (2) we will reduce the concurrency. Constraint (3) must be satisfied since otherwise the mutual exclusion will be violated. In the MLA solution we will take the approach of assigning multiple locks so as to maximize the concurrency.

Let $C(x)$ be the set of colors that are assigned to a vertex u, the coloring problem on SNIG is stated as the following:

Problem 3. Given a SNIG $G_n^s = (V_n, E_n \cup E_s)$ find the minimum number of colors to color G_n^s such that:

(a) If two vertices u and v are connected by a non-interfering edge then $C(u) \cap C(v) = \emptyset$ and

(b) If two vertices u and v are connected by a serializing edge then $C(u) \cap C(v) \neq \emptyset$.

Let G be an arbitrary concurrency graph, and let G_n^s be the SNIG of G. We will show in Section 3.2 that the minimum number of locks required by G equals the minimum number of locks required by G_n^s.

3 Minimum Lock Assignment Solution

The MLA problem for arbitrary concurrency graphs is NP-hard because one special case - MLA problem for non-interfering concurrency graph - is NP-hard. In this section we present a heuristic approach for solving MLA. We also formulate the MLA problem as an Integer Linear Programming (ILP) problem, and in Section 4 we will use this ILP formulation to quantitatively evaluate our heuristic.

3.1 A Naive Solution

Assume all shared memory locations that a critical section accesses can be statically identified by compiler analysis, then a simple solution to the MLA problem is to assign a distinct lock to each shared memory location, and the lock set of a critical section is the set of locks assigned to memory locations it accesses. However, this approach may use more locks than necessary, and introduce more overhead of lock acquisition and release. We say the number of locks required in this simple solution, i.e., the total number of memory locations accessed in a program, denoted as $|M|$, is the *upper bound* (UB) of the optimal MLA solution.

3.2 MLA Heuristic

Our MLA heuristic consists of three main steps (see Figure 4):

Step 1: Assign locks to non-interfering subgraph G_n using graph coloring heuristic (Line 6).

Step 2: Ensure that the serializing interfering edges in SNIG are correctly handled (Line 7).

Step 3: Finally propagate the locks to the interfering subgraph G_i (Line 8).

The first step is straightforward. We use a heuristic graph coloring algorithm [4] to color G_n, and one possible solution for our example is shown in Figure 2(b).

Next, we must ensure that critical sections connected by serializing interfering edges in SNIGs are correctly serialized. The details of this step are given by the function HandleSerializingEdges in Figure 4. In Figure 2(d), CS_1, CS_3 and CS_4 are in G_n and each of them has obtained a lock from the graph coloring. Interfering critical sections CS_1 and CS_3 are automatically serialized by sharing lock 1, but CS_3 and CS_4 are not. A straightforward method to solve this is let one of them "borrow" the lock from the other. For a serializing interfering edge (u, v), we say vertex u borrows the lock from v, denoted as $\mathit{borrow}(u \leftarrow v)$, if u adds v's lock to its lock set, $\mathit{Lock}(u) = \mathit{Lock}(u) \cup \mathit{Lock}(v)$. Denote the set of locks from u's non-interfering neighbors as $NIN(u)$, $NIN(u) = \bigcup_{(u,w)\in G_n} \mathit{Lock}(w)$. Before the

```
LockAssignment($G$)
1.  Initialize $Lock(u)$ for all $u \in V$ as empty
2.  Partition the graph $G$
3.  if $G_n = \phi$
4.      assign a global lock to each critical section
5.  else
6.      $HLB$ = GraphColoring($G_n$)
7.      HandleSerializingEdges($G_n^s$)
8.      LockPropagation($E_{ci}, G_i$)
9.  end if
10. if $HLB > |M|$ then
11.     for each $v \in V$
12.         $Lock(v) = \bigcup_{i \in LS(v)} Lock(i)$
13.     end for
14. end if

HandleSerializingEdges ($G_n^s$)
15. for each serializing interfering edges $(u, v)$
16.     if $Lock(u) \cap Lock(v) = \emptyset$
17.         if $borrow(u \leftarrow v)$ is safe
18.             $Lock(u) = Lock(u) \cup Lock(v)$
19.         else if $borrow(v \leftarrow u)$ is safe
20.             $Lock(v) = Lock(v) \cup Lock(u)$
21.         else
22.             HLB = HLB + 1
23.             add a new lock to $u$ and $v$'s lock sets
24.         end if
25.     end if
26. end for

LockPropagation($E_{ci}, G_i$)
27. for each $(v_n, v_i) \in E_{ci}$
28.     $sequence$ = BreadthFirstSearch($G_i$, $v_i$)
29.     Arbitrarily pick one lock $l$ from $v_n$'s lock set
30.     for each $v$ in $sequence$
31.         $Lock(v) = Lock(v) \cup \{l\}$
32.     end for
33. end for
```

Fig. 4. Lock Assignment Heuristic

borrowing, u has a disjoint set of locks with all its non-interfering neighbors, i.e., $Lock(u) \cap NIN(u) = \emptyset$. This implies that the concurrency between u and its non-interfering neighbors is maximized. After the borrowing, we also require u not share any lock with its non-interfering neighbors. This is satisfied if $Lock(v) \cap NIN(u) = \emptyset$, that is, none of u's non-interfering neighbors has u's borrowed lock from v. In this case we say the borrowing is "safe", which means it does not reduce concurrency among non-interfering critical sections.

In our example in Figure 2, in order to enforce the mutual exclusion between CS_3 and CS_4, we first let CS_4 borrow the lock from CS_3, then $Lock(CS_4) = \{1, 3\}$. This is shown in Figure 2(f). However, this borrowing is not safe, because one of CS_4's non-interfering neighbor CS_1 would share lock 1 with it. Then we try the alternative way. We let CS_3 borrow the lock from CS_4. This is illustrated in Figure 2(g). This borrowing is safe because $Lock(CS_4) \cap NIN(CS_3) = \emptyset$, where $NIN(CS_3) = \{2\}$. Note that if neither borrowing is safe, we will introduce a new lock and add it to both end vertices' lock sets. The procedure of lock borrowing is summarized in Figure 4.

The first two steps together color the SNIG G_n^s. Finally, in function Lock-Propagation, we propagate the SNIG lock assignment result to the interfering subgraphs G_i . The interfering subgraph G_i is connected to the non-interfering subgraph G_n through a set of crossing edges (v_n, v_i), where $v_n \in G_n$, and $v_i \in G_i$. Each (v_n, v_i) is an interfering edge, that means v_i should share at least one of v_n's lock obtained from the graph coloring. We say v_n "propagate" a lock to v_i. If v_i has more than one incident crossing edges, then it should inherit locks from all its neighbors in G_n. Subsequently, v_i propagates its lock set to its neighbors in G_i. This propagation continues until every vertex in G_i inherits locks from its neighbors. This procedure can be simply implemented as a set of breath-first searches, with each v_i at a crossing edge as the source vertex. The algorithm is shown in Figure 4. One propagation result of our example is shown in Figure 2(h). An important property of this lock propagation is that it does not introduce any new lock, therefore the number of locks required to color G_i cannot exceed the number of locks required to color the SNIG G_n^s.

The final lock assignment result is shown in Figure 2(h). We refer to the number of locks required to color G as the Heuristic Lock Bound (HLB). We have mentioned in the naive solution that the upper bound UB of the required locks is the number of shared memory locations accessed in the concurrency graph G. In some cases HLB might exceed UB, and we need to choose the smaller one from HLB and UB for lock assignment. The MLA heuristic algorithm is summarized in Figure 4.

The following theorems show that our MLA heuristic can preserve the mutual exclusion between critical sections without any loss of concurrency. They also show that lock assignment on an arbitrary concurrency graph G is optimal if the lock assignment on SNIG of G is optimal. Detailed proofs can be found in [5].

Theorem 1. *When the algorithm LockAssignment (G) terminates, any pair of interfering critical sections in G share at least one common lock.*

Theorem 2. *When the algorithm LockAssignment (G) terminates, any pair of non-interfering critical sections do not share any lock.*

Theorem 3. *Lock assignment on a concurrency graph G is optimal if and only if the lock assignment on its SNIG G_n^s is optimal.*

The concurrency graph partitioning runs in $\mathcal{O}(V + E)$ time, and the graph coloring runs in $\mathcal{O}(V^2)$ time. At the worst case, the time complexity of HandleSerializingEdgess and LockPropagation are $\mathcal{O}(E * V)$ and $\mathcal{O}(E^2 + V * E)$, respectively. Therefore, at the worst case the total time complexity of LockAssignment is $\mathcal{O}(E^2 + V * E)$.

3.3 ILP Formulation

In this section, we formulate the MLA problem as an ILP problem. Given a concurrency graph $G = (V, E)$, we introduce 0-1 variables $f_{u,i}$ to indicate whether lock i is assigned to node u in G, $1 \le u \le |V|$, and $1 \le i \le |M|$, where M is the set of shared memory locations that are accessed in all critical sections. Recall that the number of locks given by an optimal solution cannot exceed $|M|$. Since each critical section must be assigned at least one lock, we have the following constraint:

$$f_{u,1} + f_{u,2} + \cdots + f_{u,|M|} \ge 1 \quad \text{for all } u \in G \tag{1}$$

We use 0-1 variables l_i to indicate whether lock i is assigned to any critical section, $l_i = f_{1,i} \vee f_{2,i} \vee \cdots \vee f_{|V|,i}$. This condition is represented by the following constraints:

$$f_{1,i} + \cdots + f_{|V|,i} \ge l_i \tag{2}$$
$$f_{1,i} + \cdots + f_{|V|,i} \le |V| \times l_i \tag{3}$$

Next we derive conditions that ensure the lock assignment is correct and maximizes the parallelism. Recall that a lock assignment solution is correct if interfering critical sections u and v share some lock, and parallelism is maximized if non-interfering critical sections are assigned two disjoint sets of locks. Let 0-1 variable $s_{u,v,i}$ indicate whether u and v share lock i, then $s_{u,v,i} = f_{u,i} \wedge f_{v,i}$. This condition is imposed by the following constraints:

$$f_{u,i} + f_{v,i} \ge 2 \times s_{u,v,i} \tag{4}$$
$$f_{u,i} + f_{v,i} \le 2 \times s_{u,v,i} + 1 \tag{5}$$

We use 0-1 variable $s_{u,v}$ to indicate whether u and v share any lock. Then $s_{u,v} = s_{u,v,1} \vee \cdots \vee s_{u,v,|M|}$. The following two constraints represent this condition:

$$s_{u,v,1} + \cdots + s_{u,v,|M|} \ge s_{u,v} \tag{6}$$
$$s_{u,v,1} + \cdots + s_{u,v,|M|} \le |M| \times s_{u,v} \tag{7}$$

Then

$$s_{u,v} = 1 \qquad \text{for interfering edge } (u, v) \tag{8}$$
$$s_{u,v} = 0 \quad \text{for non-interfering edge } (u, v) \tag{9}$$

The total number of locks used is:

$$N = l_1 + \cdots + l_{|M|} \tag{10}$$

Therefore, the MLA problem is to minimize N subject to inequalities (1) to (9).

4 Experimental Results

In this section, we present two sets of experiments to evaluate our lock assignment algorithm. In the first set of experiments, we compare the results produced by our MLA heuristic with the optimal solutions based on the ILP formulation on a set of 300 random concurrency graphs. In the second set of experiment we evaluate the effectiveness of the MLA heuristic using Splash2 [3] benchmarks.

4.1 Precision Evaluation

	Avg	Min	Max
Vertices (V)	8.63	2	16
Edges(E)	16.73	1	53
Edge Density E/V^2	0.19	0.09	0.28
Non-interfering edges (E_n)	3.37	0	20
E_n/E	0.22	0	1
Serializing interfering edges	2.85	0	27

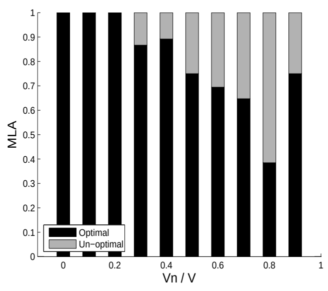
Table 1. Features of random concurrency graphs

To study the precision of our MLA heuristic we implemented our ILP formulation in the commercial ILP solver CPLEX, and tested the heuristic and the ILP formulation on a set of 300 randomly generated concurrency graphs with characteristics shown in Table 1. We limited our random concurrency graphs to contain at most 16 nodes due to time constraints in the ILP solver. It shows that our heuristic solution is optimal for 83.3% of tested graphs. For the remaining 16.7% of graphs our heuristic assigns more locks than the optimal solutions, and in the worst case at most two more locks than optimal solutions are assigned.

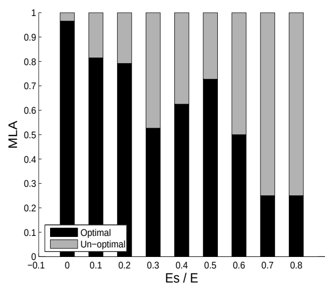
We also evaluated the influence of non-interfering subgraph G_n and serializing interfering edges E_s for lock assignment. For this purpose, we listed the precision of the MLA heuristic with the increase of the relative size of non-interfering subgraph, given by V_n/V, and with the increase of the relative number of serializing interfering edges, given by E_s/E, in Figure 5(a) and (b), respectively. As an example, Figure 5(a) shows that our MLA heuristic gives optimal solutions to about 70% of test cases that have $V_n/V = 0.6$ and sub-optimal solutions (i.e., assign extra locks) for the remaining 30%. Figure 5(a) and (b) illustrate that the precision of our heuristic depends on the non-interfering subgraph size and the relative number of serializing interfering edges.

4.2 Performance Study on Sun-Fire

Next we study the performance of the MLA heuristic using a set of Splash2 [3] benchmarks listed in Table 2. Splash2 benchmarks call the Pthreads li-



(a) MLA Vn / V



(b) MLA Es / E

Fig. 5. Precision of the MLA heuristic

Application	Barnes	Cholesky	Ocean-cont	Radiosity	Water-nsq
Description	N-body	Matrix factoring	Hydro-dynamics	3-D rendering	Water molecules
Problem size	262144 bodies	tk29.O B8 C256	514 × 514	largeroom batch	512 molecules
CSs	6	7	4	37	9
CS time (1 proc)	6.29%	32.37%	0.11%	9.93%	11.54%
Lines of code in CS (avg/max)	17.17 / 68	10.86 / 37	1.75 / 3	12.79 / 85	2.89 / 6
Funcs in CS	1	1	0	10	1
Locks assigned	3	4	4	8	7
Locks for each CS (max)	1	1	1	4	1

Table 2. Benchmarks and lock assignment results

brary [6], and mutual exclusion is enforced by *pthread_mutex_lock(<lock_var>)* and *pthread_mutex_unlock(<lock_var>)* functions with explicit lock variables. For the purpose of our performance study, we manually transformed each lock/unlock region into a critical section. We constructed the concurrency graph for each benchmark manually, and applied the MLA algorithm to calculate the lock assignment. The number of locks assigned to each benchmark is shown in Table 2.

We then ran the set of benchmarks on Sunfire 10-processor 750MHz machine, and collected two sets of data for each benchmark to evaluate our heuristics: (1) T_s: the execution time of the benchmark when all critical sections are controlled by a single lock, and (2) T_{MLA}: the execution time of the benchmark with lock assignment using our MLA heuristic. Figure 6 shows the performance improvement of our lock assignment with respect to the single global lock, i.e., $(T_s - T_{MLA})/T_s$, running on different number of threads. Cholesky and Radiosity

[6] The original Splash2 benchmarks utilize the Argonne National Laboratories (ANL) parmacs macros for parallel constructs. We have re-configured them to call the Pthreads library.

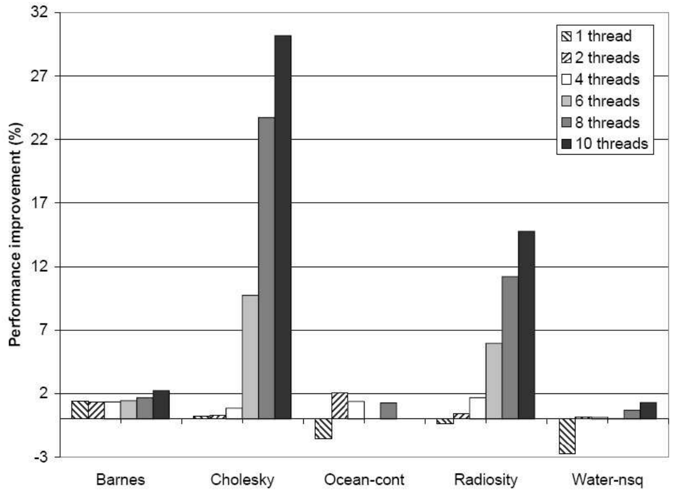


Fig. 6. Performance improvement with respect to single lock

have shown a performance improvement of 30.17% and 14.76%, respectively, due to the decrease of lock contention and serialization. On the other hand, Barnes, Ocean-cont and Water-nsq show a much lower performance improvement for two main reasons. First, the amount of time spent on critical sections is a small portion of the total execution time. For instance, as shown in Table 2, for Ocean-cont, the time spent on critical sections takes only 0.11% of the total execution time. Second, in Barnes and Water-nsq data is often organized as arrays or complicated user-defined data structures, and is accessed in a dynamic pattern that cannot be predicted during the compilation time. When we constructed the concurrency graphs we conservatively treated such arrays and user-defined data structures as scalar units. This conservative approach may introduce "spurious" interference among critical sections, which results in unnecessary serialization. The unnecessary serialization will then increase lock contention among threads during the execution time. Some more sophisticated analysis techniques such as shape analysis [6], or dynamic conflict resolving techniques such as transactional memory and synchronization state buffer (SSB) [7] are needed to exploit further concurrency among critical sections in these benchmarks.

5 Related Work

Recently there has been some work on compiler based lock inference technique. Emmi et.al. [8] propose a lock allocation problem that takes a multithreaded program annotated with `atomic` sections and infers a lock assignment to `atomic` sections to preserve its atomicity and deadlock freedom. They formulate the lock allocation problem as an ILP problem which minimizes the conflict cost between atomic sections and minimizes the number of locks. No heuristic solution is presented in their work. Our lock assignment differs from lock allocation in the following two aspects. First, our lock assignment problem maximizes the parallelism among critical sections using the minimum number of locks, while the lock allocation problem uses the minimum number of locks to minimize the conflict cost, a metric that is not clearly related with the parallelism. Second, we present both the heuristic solution and the ILP formulation for lock assignment problem. We use the ILP formulation to evaluate the optimality of the lock assignment heuristic. We also use scientific applications to evaluate the lock assignment heuristic and present performance improvement.

Hicks et.al. [9] has proposed a lock inference techniques for atomic sections, which first determines a set of shared memory locations in the program, then uses a "mutex inference" algorithm to infer a set of locks for each atomic section to preserve its atomicity. The basic idea of their mutex inference algorithm is to find the dependence relation among shared memory locations, and partition the shared memory locations into sets according to this dependence relation. Locks are then assigned to each memory location set. Since the mutex inference algorithm is not optimization based, it may infer more locks than our lock assignment algorithm.

Autolocker [10] takes the programs annotated with pessimistic atomic sections and a programmer controlled lock assignment, and infers a compiler controlled lock assignment that is free of deadlocks and data races.

Vaziri et.al. [11] proposed a data-centric synchronization approach for writing concurrent programs using atomic sets, which are a set of shared memory locations that have "similar" data consistency properties. Accesses to fields in an atomic set are assumed to take place atomically in "units of work". Taken a program with annotated atomic sets, the compiler infers units of work automatically and translates them into synchronized blocks. Our work complements Vaziri et.al.'s work in that we can analyze and determine the atomic sets and units of work using concurrency analysis and lock assignment algorithm.

Some other optimization techniques on locks have been reported. Diniz and Rinard [12] present data lock coarsening and computation lock coarsening techniques to reduce the overhead of fine-grain locks in Java programs. Choi et.al. [13] and Aldrich et.al. [14] remove unnecessary synchronization from Java programs.

6 Conclusions

In this paper we proposed a lock assignment technique to simplify the mutual exclusion in multithreaded programs. It takes the programs annotated with `critical` sections and finds the minimum number of locks needed to enforce mutual exclusion among interfering critical sections without any loss of concurrency. Experimental results are very encouraging and show that our method can be used to improve the performance of multithreaded programs with mutual exclusion by exploiting concurrency among multiple critical sections. An extension of this work to support read/write locks is a subject for future work.

References

1. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
2. V. Sreedhar, Y. Zhang, and G. Gao. A new framework for analysis and optimization of shared memory parallel programs. Technical Report CAPSL-TM-063, University of Delaware, Newark, DE, 2005.
3. The Stanford FLASH Prjoect. Stanford parallel applications for shared memory (SPLASH) benchmark. In http://www-flash.stanford.edu/apps/SPLASH/.
4. P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, 1992.
5. Y. Zhang, V. Sreedhar, W. Zhu, V. Sarkar, and G. Gao. Optimized lock assignment and allocation: A method for exploiting concurrency among critical sections. Technical Report CAPSL-TM-065-revised, University of Delaware, Newark, DE, 2007.
6. Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, 1996.
7. Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 35–45, 2007.
8. Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–296, 2007.
9. M. Hicks, J. Foster, and P. Pratikakis. Lock inference for atomic sections. In *TRANSACT'06: Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.
10. Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, 2006.
11. M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06*, pages 334–345. ACM, 2006.
12. P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *LCPC'96*, pages 284–299, 1996.
13. Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.
14. J. Aldrich, E. Sirer, C. Chambers, and S. Eggers. Comprehensive synchronization elimination for java. *Science of Computer Programming*, 47(2-3):91–120, 2003.