Optimizing Fence Placement in TSO

Lucas Van Mol, Einar de Gruyl, Alexandru Dumitriu
Nicolae Rusnac and Sara Kooistra

Supervisors: Dennis Sprokholt, Konstantin Sidorov,
Emir Demirović, Soham Chakraborty

June 2024

Abstract

The correctness of concurrent software systems is critical, particularly in safety-critical applications where synchronization errors can lead to catastrophic failures. This paper addresses the challenge of optimizing fence placement in the Total Store Order (TSO) memory model to ensure program correctness while minimizing performance overhead. We propose an Adaptive Large Neighborhood Search (ALNS) approach with metaheuristics to identify optimal fence placements. Our method is evaluated on a toy language against the ILP formulation by Alglave et al. [4], demonstrating improvements in execution time and confined primal integrals (CPI) [6] across various benchmarks. Different heuristics and their effectiveness within the ALNS framework are discussed. Finally, we suggest some research directions that could improve this approach further. Our source code, including benchmark programs used in this paper, are made available online.[1]

1 Introduction

The Therac-25 is a computer-controlled radiation therapy machine that was involved in at least six accidents in which patients were accidentally poisoned by a massive overdose of radiation due to a race condition in the program [12]. This case is one of many examples where the correctness of a software system plays a crucial role in the safety of a system.

To address this challenge, concurrent code reasoning with memory models that support analysis and verification can be formalized. Recent research in program verification has been successful by using SMT solvers to handle circular dependencies, but ensuring correct synchronization remains a major concern. [9]

So-called fences can be placed to enforce specific ordering of memory operations. To optimize fence placements in software execution models, and reduce unnecessary delays while ensuring correctness, this project aims to investigate techniques such as ILP optimization, enhanced with metaheuristics such as large neighbourhood search. [16] By enhancing the efficiency and reliability of critical software systems, we can mitigate risks, and decrease computation times. Our main question is: How can we optimize fence placement in TSO memory models to reduce unnecessary delays while ensuring program correctness? To answer this our work aims to address the following subquestions:

RQ1: How can we effectively use ILP techniques with ALNS metaheuristics to optimize fence placement?

RQ2: How does our approach compare with Alglave et al.'s ILP formulation [4] for fence placement regarding metrics such as execution time of critical cycle finding and fence placement, in addition to the confined primal integrals (CPI)[6] of the different approaches?

Our approach introduces novelty by addressing fence placement optimization within weak memory models such as TSO. We initially leverage ILP fence placement algorithms on a toy language and compare it to metaheuristics introduced by adaptive large neighbourhood search (ALNS). This novel approach aims to increase efficiency whilst maintaining correctness, a critical factor in high-stakes domains.

This paper uncovers a few key heuristics that supports ALNS as a novel approach to fence

[1] `https://github.com/idmp-fences/toylang`

optimization that improves upon existing work. Notably, a greedy 'hot-edge' approach to initialization of partial solutions proves to be very effective on some real-world benchmarks and improves the measured CPI when compared to ILP solving.

2 Background

In parallel programming, it is crucial to understand the foundational concepts for addressing the complexities and challenges associated with concurrent systems. This section delves into weak memory models, the significance of critical cycles in maintaining consistency, Large Neighbourhood Search as a heuristic for optimization and presents some existing approaches for fence synthesis.

2.1 Weak Memory Models

In the world of computer architecture and parallel programming, memory models are highly important in defining the behavior of concurrent systems. A memory model sets the rules that govern the visibility and ordering of memory operations among different threads or processors in a multiprocessor system. Weak memory models, in particular, introduce complexities and challenges due to their relaxed consistency semantics compared to stronger models like sequential consistency [2].

Weak memory models differ from the intuitive notion of sequential consistency, where the execution of a program appears as though operations occur in a single, global order. Instead, weak memory models allow for a broader range of possible behaviors, enabling optimizations such as out-of-order execution to enhance performance. However, this flexibility comes at the cost of increased complexity, up to doubly exponential, in reasoning about program correctness and understanding the semantics of concurrent execution [8].

One characteristic of weak memory models is the relaxation of the ordering constraints on memory operations. In contrast to sequentially consistent models, weak models permit the reordering of memory operations across threads or processors, leading to potential inconsistencies in the observed execution order. This reordering can result in data races, where multiple threads concurrently access and modify shared memory locations without proper synchronization, leading to nondeterministic outcomes [2].

Total Store Ordering (TSO) is a weak memory model commonly found in computer architectures like x86 and ARM. TSO allows write operations to be seen by other threads in a different order than they were issued by the storing thread [17]. On the other hand, read operations are strictly ordered, ensuring that a load sees the effects of all preceding stores in the program order. This difference in the treatment of stores and loads can make reasoning about TSO-based systems difficult.

TSO's relaxed consistency semantics can cause unexpected outcomes, such as out-of-order visibility of stores and memory operation reordering across threads. This requires careful synchronization and memory ordering annotations in concurrent programs targeting TSO architectures. One way to solve this issue is by placing memory fences. A *fence* is a type of CPU instruction that enforces a specific ordering of memory operations depending on the fence type. X86 has a single fence type `MFENCE`, whilst Power and ARM, with their comparatively weaker memory models, have multiple fence types. Despite these complexities, TSO is still widely used in modern processor designs because it strikes a balance between performance and programmability [13]. It is, therefore, important to understand TSO's semantics and especially how it can be enhanced with the optimal use of memory fences that can ensure a balance between correctness and performance in terms of runtime [4].

2.2 Critical Cycles

In the context of memory models, a *cycle* refers to a sequence of memory operations that forms a closed loop of dependencies. A *minimal cycle* is a cycle where removing an individual dependency results in the cycle being broken. On the other hand, a *critical cycle* (CC) is a minimal cycle that, if allowed to execute without proper synchronization, would violate sequential consistency [4].

Critical cycles are especially useful in weak memory models like TSO because they represent scenarios where the relaxed memory ordering can result in unexpected outcomes. By identifying and analyzing these critical cycles, we can determine where in the program to place fences to enforce sequential consistency [4].

To detect critical cycles in a program we can utilize Abstract Event Graphs (AEGs). These AEGs represent the collection of all possible candidate execution of a program by representing all events and relationships between them. However, unlike concrete executions, AEGs do not contain specific values associated with events and are thereby providing an over-approximation of all possible program executions [4].

2.3 Large Neighbourhood Search

Large Neighbourhood Search (LNS) [16] is an advanced heuristic technique used in solving constrained optimization problems that are otherwise challenging using traditional methods. LNS functions by iteratively destroying and repairing parts of a solution, exploring a larger portion of the search space compared to standard local search methods. The process involves using heuristic methods for the removal and addition of the solution components, to potentially identify more optimal configurations.

This technique has been documented by Ropke and Pisinger[16] in their study of the pickup delivery problem with time windows. Their paper demonstrates that LNS can be used to identify new optimal solutions in problems with complex constraints and large search spaces.

Adaptive Large Neighbourhood Search (ALNS) is a metaheuristic framework within LNS. It employs an adaptive mechanism where different heuristics are weighted according to their effectiveness in improving the solution over time. Heuristics that are involved in producing better solutions are used more frequently compared to others. This approach maintains a level of exploratory diversity while also maintaining efficiency in the search process.

3 Related Work

This section reviews key advancements in memory models and software verification. It examines axiomatic models, which define memory behavior through constraints, and existing work on fence synthesis.

3.1 Axiomatic Models

Memory models can roughly be grouped into two: operational models and axiomatic models. Operational models describe the behaviour of systems focusing on the execution of programs. Axiomatic models, on the other hand, constrain various relations on memory accesses to distinguish allowed behaviour from forbidden behaviour. [5]

Axiomatic models usually consist of three stages. The *instruction semantics* maps each instruction to mathematical objects. The *candidate executions* describe the communications that might happen between different threads of the program. Third, the *constraint specification* shows which executions are valid and which are not [5].

We now take a deeper look at the candidate executions. Candidate executions are graphs: a node models the effect of instruction, and an edge forms relations over events [3]. A candidate execution is obtained by first generating an *event structure* [4].

An event structure is a set of memory events $\mathbb{E}$ together with the program order relation PO. The PO includes the address (addr), data and control dependency relations (ctrl) [3]. An event is a read-from- or write-to memory consisting of an identifier, direction, memory address and a value [4].

By adding execution witnesses, representing the communication between the threads, an event structure can be completed into a candidate execution. The execution witness in the candidate execution consists of three relations:

- co: orders writes to the same memory location
- rf: links a write w to a read r such that r reads the value written by w
- fr: a read r is in fr with a write w if the write w' that r reads from hits the memory before w.

An example of an event structure and candidate execution can be seen in Figure 1. The abstract event graph (AEG) summarizes all the candidate executions of an event structure by over-approximating co rf and fr relations as competing edges (cmp).

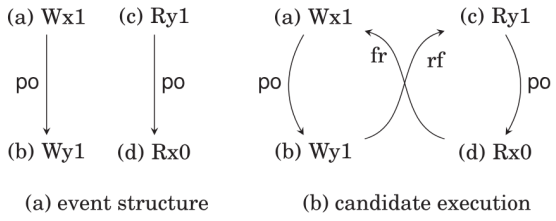


Figure 1: Event structure and candidate execution [4]

3.2 Fence Synthesis

In order to forbid weak memory behaviours arising as a result of critical cycles in the AEG, we need a way to compute which types of fences to place at which locations in the program. Moreover, we wish for this placement to be *optimal*.

FenSying is a related method for fence synthesis described in [18] which takes all buggy traces of a program as an input and uses a SAT solver to compute the minimal set of fences required to invalidate the buggy traces by enforcing the weakest memory orders possible. This method, while effective in some cases, works with each buggy trace independently and requires the conjunction of SAT queries for all buggy traces, which can become computationally expensive for larger programs with numerous buggy traces.

The approach of Alglave et al. [4] describes an ILP formulation of this problem that leverages the previously presented overapproximation technique, such that its solution gives a set of fences to insert to forbid the critical cycles in the AEG. This approach improved upon existing tools, such as `trencher`[7], which needs to construct an exponential ILP compared to the linear size of this formulation. We choose Alglave et al.'s approach as a baseline for this research, since it has a more scalable and holistic approach than FenSying. It is compared to ALNS, on which section 4.3 goes into further detail.

4 Methods

4.1 AEG Construction

For the purposes of this paper, we introduce a toy language, aptly named *toy*. A more detailed example of its syntax can be found at Appendix A. Each *toy* program is converted into an abstract event graph, representing an overapproximation of every possible execution of a program. The AEG is constructed in several steps. We first consider only the global variables in the initial section of the *toy* program. Then, each thread is added to AEG as a separate directed graph, whose paths represent the possible execution of instructions within that thread. Finally, we join each thread graph by their (undirected) competing edges, completing the AEG.

Global variables The initial statements of a *toy* program set up the global variables that will be used in the AEG. These globals are then used to filter events in each thread to only include global variables. Variables local to a thread are excluded from the AEG as they cannot be part of competing edges, and therefore will not be present in any critical cycle.

Assign & modify statements Each thread in *toy* is comprised of a list of statements. There are three simple statements `assign`, `modify` and `fence`, which are added to the AEG as shown in Table 1. A thread containing only these instructions would result in a simple path from the first node to the last node. Since we only consider full fences under TSO, fences are implemented as a special type of edge, breaking up the transitivity of *po* edges.

If statements The `if` statements introduce branching, and include boolean expressions in their conditions. An example of an `if` statement being converted into an AEG is shown in Figure 2. The result is a directed acyclic graph that covers all branches containing global variables.

While statements The `while` statements introduce backward jumps and thus cyclicity. Figure 3 shows how `while` loops are represented in the AEG.

Competing edges Each thread now represents all possible executions of that thread as po edges between (global) events. We subsequently create an undirected competing edge *cmp* between each global write from one thread to a global read or write from another thread to the same address. This step causes the disconnected graphs to become the whole abstract event

Statement	Example	Resulting AEG structure
`assign/modify` numeric	`x = 4;`	$\cdots \xrightarrow{\text{po}} Wx \xrightarrow{\text{po}} \cdots$
`assign/modify` literal	`x = y;`	$\cdots \xrightarrow{\text{po}} Ry \xrightarrow{\text{po}} Wx \xrightarrow{\text{po}} \cdots$
`fence`	`Fence(WR);`	$\cdots \xrightarrow{\text{fence}} \cdots$

Table 1: AEG construction from simple *toy* statements

```
let x: u32 = 0;
let y: u32 = 0;
thread t1 {
    let a: u32 = 0;
    if (x == y) {
        a = y;
    } else {
        y = a;
    }
    x = a;
}
```

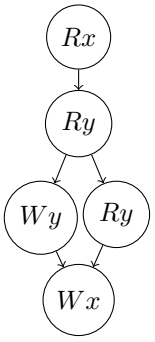


Figure 2: The AEG nodes and program order edges created for an `if` block.

```
let x: u32 = 0;
let y: u32 = 0;
thread t1 {
    let a: u32 = 0;
    while (x == 0) {
        y = a;
    }
    x = a;
}
```

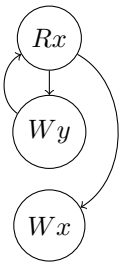


Figure 3: The AEG nodes and program order edges created for a `while` block.

graph, representing the over-approximation of candidate executions of the program.

4.2 Critical cycle detection

A critical cycle is defined in Alglave et al. [4] as a cycle in the AEG that satisfies the following properties:

- (CS1) the cycle contains at least one delay, for TSO this means a poWR edge;
- (CS2) per thread, there are at most two accesses, the accesses are adjacent in the cycle, and the accesses are to different memory locations; and
- (CS3) for a memory location l, there are at most three accesses to l along the cycle, the accesses are adjacent in the cycle, and the accesses are from different threads

Critical cycle detection is done by looping over the AEG's nodes, performing a DFS on each and looking for cycles that satisfy these conditions[2]. Once we have finished a DFS on one node, we can then exclude subsequent critical cycles that include it, as we have already found every critical cycle passing through that node. The algorithm is detailed in Algorithm 1.

In the function `CanAddNode` we make sure that adding a node would not violate (CS2) or (CS3), taking special care to handle the cyclical nature of the path. The function `FormsCycle` checks if the path is cyclical, and if it contains a delay as per (CS1). The function `Neighbors` finds the neighbors of a node, including neighbors connected through transitive po edges.

4.2.1 Potential fence placement detection

Suppose the graph in Figure 2 is a subgraph of the AEG that contains a critical cycle involving the nodes Rx and Wx, creating an RW delay. From the subgraph, the two nodes are connected through a transitive po edge as there exists a path through po edges connecting the two.

When considering fence placements, we must consider all the possible code execution paths. In this example, it would suffice to place a fence on the outgoing edge of Rx. It's also possible

[2]The original paper by Alglave et al. [4] claims to use Tarjan's algorithm [19] to detect critical cycles. However, we suspect this is dubious as Tarjan's algorithm finds a graph's strongly connected components by partitioning the graph's vertices. This would suggest each event can only be part of a unique critical cycle, which is untrue. The code in their musketeer tool seems to support our suspicions that this section of their paper is lacking clarification.

Algorithm 1 Critical cycle detection

```
Require: Abstract event graph AEG
Ensure: 'cycles' contains all the critical cycles of AEG
 1: explored ← ∅
 2: cycles ← ∅
 3: for node ∈ AEG.nodes do
 4:     cycles ← cycles ∪ DFS(node)
 5:     explored ← explored ∪ {node}
 6:
 7: function DFS(node)
 8:     stack ← [[node]]
 9:     discovered ← ∅
10:     cycles ← ∅
11:     while stack is not empty do
12:         path ← stack.pop()
13:         curr ← path.last()
14:         for succ ∈ Neighbors(curr) do
15:             if succ ∉ explored then
16:                 if CanAddNode(path, succ) then
17:                     stack.push(path + [succ])
18:                 else if FormsCycle(path) then
19:                     cycles ← cycles ∪ {path}
20:     return cycles
```

to place two fences, one on each branch, say on both incoming edges of Wx, in order to break the cycle.

In order to correctly model this behavior, we take the Cartesian product of all simple program order paths for each delay in the critical cycle. As such, there may be multiple critical cycles with the same nodes, but a different list of potential fence placements, representing every possible execution path, each of which must be broken. An alternative to this approach is discussed in section 6.

Considering that we measure optimality in fewest fences placed, one may be inclined to think that placing a fence on the common edge is always optimal, but this may not always be the case - for example when a separate critical cycle is already enforcing a fence on one of the branches.

4.3 ILP Formulation & Implementation

In this section, we outline our approach to Integer Linear Programming (ILP) for optimal fence placement, closely following the methodology described by Alglave et al. [4]. Our implementation aims to ensure the elimination of critical cycles in Total Store Order (TSO) memory models while minimizing the number of fences, thereby maintaining program consistency and optimizing performance.

Objective Function: The primary goal of our ILP is to minimize the total number of fences. This is achieved by defining a binary decision variable for each potential fence placement and summing these variables in the objective function.

$$\text{Minimize} \quad \sum_{f \in F} f \tag{1}$$

where $F = \{f_1, ..., f_n\}$ represent all decision variables indicating whether a fence is placed.

Constraints: To ensure that all critical cycles are broken, the ILP includes constraints that enforce at least one fence placement within each critical cycle. A critical cycle, as defined in Alglave et al.'s work [4], is a sequence of memory operations forming a closed loop of dependencies that would violate sequential consistency if not properly synchronized.

For each critical cycle C, we enforce the following constraint:

$$\sum_{e \in C} f_e \geq 1 \tag{2}$$

where f_e represents the fence decision variable for edge e in the critical cycle C and in the possible fence placement set. Note that a single edge e in the critical cycle may have multiple possible fence placements, as described in the previous section.

4.4 ALNS

In this section, we outline our approach to using Adaptive Large Neighbourhood Search (ALNS) for the optimal fence placement. During the design of an ALNS implementation, there are design choices to be made: the initial state, destroy operators, repair operators, operator selection scheme and acceptance criteria.

Intial state: There is a trade-off between the time spend generating an initial state and its quality. Keeping this in mind we designed the following initial state operators:

- **ALNS-ILP**: Generate the initial state using an ILP solver until a solution is found.
- **ALNS-HOT**: Generate an initial state by placing a fence on an unbroken cycle's hottest edge until they are all broken. In this context, an edge's 'hot'-ness is defined by the number of critical cycles that contain it.

Destroy operators:

- **random_30**: Randomly remove 30% of the used fences.
- **random_10**: Randomly remove 10% of the used fences.
- **hot_fences**: Destroy the top 10% of the fences, sorted in decreasing order by the number of cycles they are part of.
- **cold_fences**: Destroy the bottom 10% of the fences, sorted in decreasing order by the number of cycles they are part of.
- **biggest_cycle**: Destroy all the fences in the cycle containing the most fences. If a cycle contains multiple fences, likely, it is not targeted to be broken by the repair heuristic. Using this approach might lead to unexplored areas in the search space.

Repair operators:

- **ilp_full**: Use the ILP solver to repair the solution to (local) optimality. Note that only unbroken cycles need to be passed to the solver.
- **ilp_partial**: Use the ILP solver to repair the solution, but stop at the first solution.
- **most_cycles**: Greedily repair the solution by prioritizing adding fences on the 'hottest' edges. This process is repeated until no more cycles are left unbroken. The intuition behind this heuristic is that in many cases following the greedy approach is likely to minimize the number of used fences.
- **hot_fences**: Follows a similar idea to the most_cycles heuristic, but uses a probabilistic rather than a deterministic approach, giving more weight to 'hotter' edges.
- **in_degrees**: Greedily repair the solution by prioritizing adding fences on edges that have a higher number of incoming edges. The idea is that the higher the number of incoming edges, the likelier it is that it is part of multiple cycles.
- **unbroken**: Repair by randomly placing fences on unbroken cycles until all cycles are repaired.

Operator selection scheme:

- **Roulette wheel**: We use the roulette wheel selection scheme as described by Pisinger and Ropke [15] with weights $\omega_{1,2,3,4} = [3, 2, 1, 0.5]$ and decay parameter $\lambda = 0.8$. The probability ϕ for an operator j to be chosen for the next iteration is

$$\phi_j = \frac{\rho_j}{\sum_k \rho_k} \tag{3}$$

where an operator's weight vector ρ_j is updated at each iteration it used, such that

$$\rho_j = \lambda \rho_j + (1 - \lambda)\omega_i \tag{4}$$

The choice of which of the four ω to choose depends on whether the new solution is the global best, is better than the current one, is accepted or is rejected.

The chosen parameter values were found to give good results for our purposes, and future work could look into more thoroughly tuning these values.

Acceptance criteria:

- **Hill Climbing**: Only solutions with a better objective value are accepted.

4.5 Confined Primal Integral

In evaluating optimization solvers, the confined primal integral (CPI) emerges as a robust performance measure that balances speed and solution quality. The CPI, introduced by Berthold and Csizmadia [6], is particularly well-suited for comparing heuristic, local, and global solvers. The CPI is defined through the primal gap, which measures the relative difference between a solution and the best-known solution over time. Unlike traditional performance measures that may disproportionately favor global solvers over heuristics by extending time limits, the CPI incorporates an exponential decay in its calculation. This characteristic ensures that the measure emphasizes the early stages of the solution process, aligning more closely with practical user expectations in optimization tasks and making it a fairer comparison metric for solvers that operate under different paradigms.

5 Results

The primary objective of these experiments was to assess the impact of Adaptive Large Neighborhood Search (ALNS) on the performance of solving the fence placement problem. Specifically, we aimed to evaluate the optimality and efficiency of the solutions generated by ALNS compared to those obtained from a baseline Integer Linear Programming (ILP) approach.

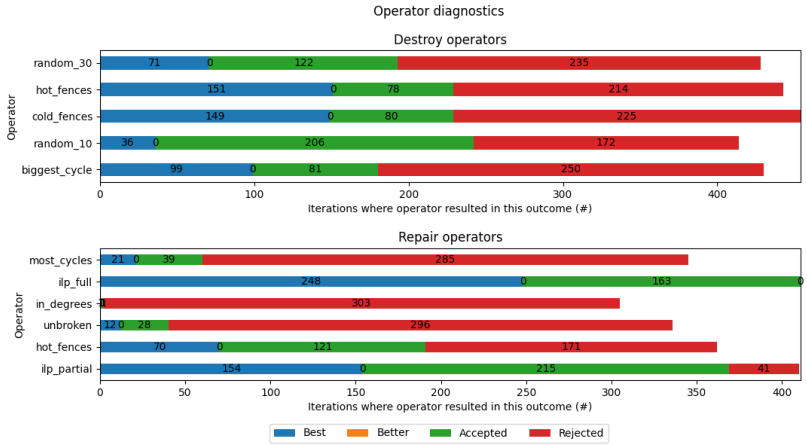


Figure 4: Operator diagnostics

5.1 Benchmarks

The experiments were conducted on 64 bit Windows 10, with an AMD Ryzen 5 5600H (6-core, 3.30 GHz, 16 MB L3 cache), with 16 GB RAM. A series of benchmark programs have been specifically selected for their relevance to the fence placement problem. The first program is Lamport's Bakery Algorithm [11] on 2, 3, 4 and 5 threads. This classical mutual exclusion algorithm represents common inter-thread communication and is heavily dependent on fence placement in order to function correctly in TSO. The second program is a generalization of Peterson's Algorithm for more than two threads [14, 10] on 2, 3 and 4 threads. This is another mutual exclusion algorithm that heavily depends on fence placement in order for it to function in TSO. Additionally, we generated various random programs to stress test our implementation on various, unpredictable programs, and to provide a diverse set of scenarios to thoroughly test the effectiveness of our algorithms. We use Gurobi as the solving technology for ILP solving.

5.2 Critical cycle detection

Table 2 summarizes the time taken to enumerate the critical cycles of the Peterson and Lamport programs. Currently, attempting to generate the critical cycles for lamport-6 and peterson-5 raised out-of-memory issues on our machine, and could potentially take a significant amount of time. More information on the random programs is also available in Appendix B.

Benchmark	Exec time (s)	Cycles
lamport-2	0.01 (±0.00)	83
lamport-3	0.11 (±0.01)	5161
lamport-4	5.10 (±0.06)	213420
lamport-5	356.0 (±9.0)	9406476
peterson-2	0.01 (±0.00)	5
peterson-3	0.02 (±0.00)	637
peterson-4	2.98 (±0.07)	128218

Table 2: Execution time for critical cycle detection.

5.3 ALNS operators

Figure 4 describes the performance of the repair and destroy operators by running them on randomly generated programs.

From these results, we gather that the destroy operators all do a similar job of driving the solution towards a better neighborhood. In addition, they all are relatively fast to execute, so we decide to use all of them in the ALNS experiments that follow.

In terms of repair operators, we decide to only use the full ILP repair operator. It proves to be the best-performing repair operator, striking the best balance between execution time and solution quality. It also synergizes well with the 'hot-edges' initialization heuristic, as the ILP operator will only have to repair relatively small instances, because only a small subset of cycles will be unbroken after the solution is partially

destroyed by a destroy operator.

5.4 Performance on random programs

Figure 5 shows the performance of ALNS on randomly generated programs. The random programs are generated with 2 threads, 2 global variables, 10 statements and a max-depth of 4. Even with these shared parameters, we still observe that the number of cycles per program can vary wildly.

The time to solution graph shows that ALNS finds the optimal solution in a similar time to the ILP, usually within the same order of magnitude and occasionally being almost an order of magnitude faster. The CPI graph shows that ALNS consistently outperforms ILP when it comes to finding decent solutions quickly. This can largely be attributed to the method of generating the initial solution, which is to place fences on 'hot' edges first, and seems to synergize well with ILP as a repair operator.

5.5 Performance on Peterson & Lamport programs

Figure 6 shows the execution time for Lamport and Peterson programs with a varying amount of threads for each. ILP and ALNS-ILP always find the solution in one iteration, except for peterson-2 and -3, suggesting that these more realistic problems are, in general, easier than randomly generated ones. The results shown by ALNS-HOT seem to confirm these suspicions. In this flavor of ALNS, the initial solution is generated by greedily placing fences on the 'hottest' edges first until all cycles are broken, which is much faster than the ILP initialization. An edge's 'hotness' is defined as the number of critical cycles it is involved in. This simple heuristic is found to immediately generate optimal solutions for all tested Peterson & Lamport programs, except in the case of lamport-2, where extra search is necessary.

6 Conclusions

This paper presents a novel approach to optimizing fence placement in the TSO memory model with Adaptive Large Neighborhood Search (ALNS). Our method demonstrates significant improvements in execution time on Lamport and Peterson benchmarks. On randomly generated programs, we see similar execution times, but better confined primal integrals (CPI) compared to the traditional ILP formulation by Alglave et al. [4]. Additionally, we note that the detection of critical cycles themselves is also a non-trivial problem, taking up a significant portion of time in the fence placement pipeline. To follow are certain areas of improvement we have identified that could guide further research.

ALNS improvements. Whilst the `alns` Python library used provides an excellent user interface, the implementation of the ALNS algorithm along with destroy/repair operators in a compiled language like C or Rust could help boost performance and provide a level playing field to ILP solvers such as Gurobi which is implemented in C.

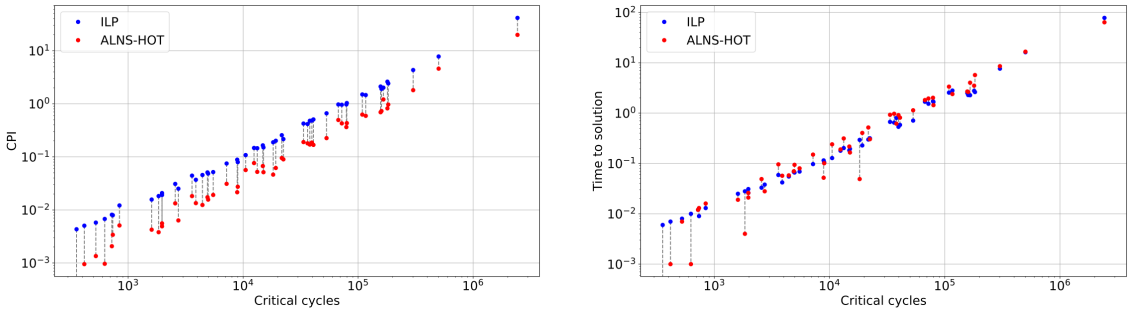
This improvement could also be combined with more sophisticated repair and destroy operators, the design of which could guided by additional research into the structure of programs for which assigning fences to 'hot' edges does not immediately lead to an optimal solution.

ILP improvements. One significant difference with our ILP implementation that is not mentioned in Alglave et al.'s previous work [4] is the way we treat branching paths when enumerating potential fence placements of a critical cycle, as discussed in section 4.2.1.

A better solution than taking the Cartesian product of all simple program order paths between two nodes of a critical cycle could be to write constraints as a collection of or/and relationships. For example, suppose a delay occurs over an edge that then branches into two because of an `if` statement. There could be a fence placed either on the common edge e_{com} *OR* on the if branch e_{if} *AND* the else branch e_{else}. If this was the only delay in the critical cycle, it could be expressed in the ILP as:

$$\begin{aligned} e_{\text{com}} + e_{\text{if}} + e_{\text{else}} &\geq 1 \\ e_{\text{if}} + e_{\text{else}} &\geq 2(1 - e_{\text{com}}) \\ e_{\text{com}}, e_{\text{if}}, e_{\text{else}} &\in \{0, 1\} \end{aligned}$$

Critical cycle detection. As it stands, when Gurobi is used as a solver, the ILP/ALNS algorithms are relatively fast compared to the actual



(a) CPI per random program

(b) Time to optimal solution per random program

Figure 5: CPI and time to optimal solution comparison between ALNS and ILP on randomly generated *toy* programs (lower is better).

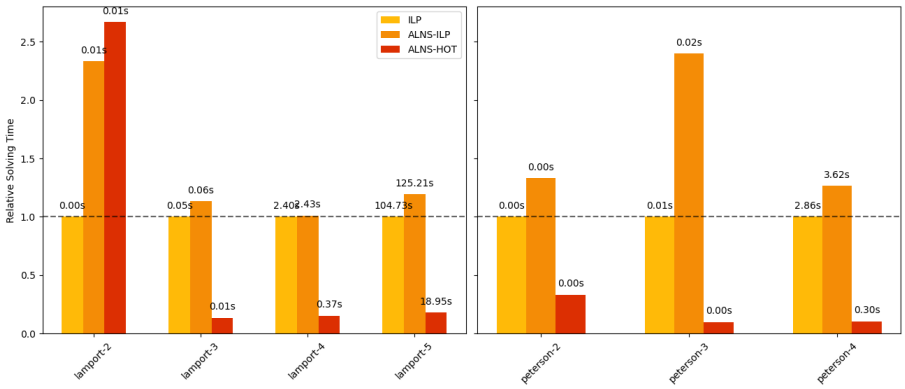


Figure 6: Time to optimal solution relative to the ILP solving time for fence placement of different programs. All algorithms use Gurobi as a solver backend. For fairness, the time stated for ILP does not include the time taken to verify optimality.

detection of critical cycles in the case of Lamport and Peterson. Whilst they are generally in the same order of magnitude in terms of execution time, as demonstrated in Table 2 and Figure 6, it could be worthwhile to look into improving the performance of critical cycle detection in order to improve the efficiency of the whole pipeline.

Interestingly, critical cycle detection in random programs generally takes less time than solving it with the ILP, as shown in Appendix B. We loosely speculate that ILP solving could be faster on Lamport and Peterson programs thanks to some sort of inherent structure.

Evaluation improvements. Expanding the evaluation to include a wider variety of real-world applications will help validate the robustness and generalizability of our approach. As an example, Alglave et al. [4] evaluate their approach on systems like `memcached`. It may be possible that such examples have different structures, or wildly different scales (for example, containing only very few critical cycles in practice).

Finally, evaluating this approach for different architectures could also be fruitful for real-world applications - notably with the increasing use of ARM chips amongst computer manufacturers[1] (with a memory model much weaker than TSO) or even applying this approach the memory models of GPUs and TPUs, with their meteoric rise in popularity thanks to recent trends in artificial intelligence.

References

[1] Pcs are about to be flooded with new chip competition — digital trends.

[2] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. *ACM SIGARCH Computer Architecture News*, 19(3):234–243, April 1991.

[3] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Transactions on Programming Languages and Systems*, 43(2):1–54, June 2021.

[4] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Transactions on Programming Languages and Systems*, 39(2):1–38, May 2017.

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2):1–74, July 2014.

[6] Timo Berthold and Zsolt Csizmadia. The confined primal integral: a measure to benchmark heuristic minlp solvers against global minlp solvers. *Mathematical Programming*, 188(2):523–537, 2021.

[7] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7792 LNCS:533–553, 2013.

[8] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, July 1995.

[9] Fei He, Zhihang Sun, and Hongyu Fan. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21. ACM, June 2021.

[10] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Elsevier, June 2012.

[11] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[12] Nancy G. Leveson and Clark S. Turner. An investigation of the therac-25 accidents. *Computer (Long Beach Calif.)*, 26(7):18–41, July 1993.

[13] Abdul Naeem, Axel Jantsch, and Zhonghai Lu. Architecture support and comparison of three memory consistency models in noc based systems. In *2012 15th Euromicro Conference on Digital System Design.* IEEE, September 2012.

[14] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[15] David Pisinger and Stefan Ropke. Large neighborhood search. *Handbook of metaheuristics*, pages 99–127, 2019.

[16] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, November 2006.

[17] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[18] Sanjana Singh, Divyanjali Sharma, Ishita Jaju, and Subodh Sharma. Fence synthesis under the c11 memory model. In Ahmed Bouajjani, Lukáš Holík, and Zhilin Wu, editors, *Automated Technology for Verification and Analysis*, pages 83–99, Cham, 2022. Springer International Publishing.

[19] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

A peterson-2.toy

```
let x: u32 = 0;
let y: u32 = 0;
let turn: u32 = 0;

thread t1 {
    x = 1;
    turn = 0;
    while (!(y == 0) && (turn == 0)) { }
    x = 0;
}

thread t2 {
    y = 1;
    turn = 1;
    while (!(x == 0) && (turn == 1)) { }
    y = 0;
}

final {
}
```

B Execution time & metadata for the randomly generated programs used as benchmarks

Program ID	AEG nodes	AEG edges	Cycles	Cycle Detection (s)	ILP (s)	Min fences
5	40	260	356	0.014 ± 0.001	0.007	8
49	36	250	416	0.017 ± 0.004	0.008	9
41	35	206	524	0.017 ± 0.005	0.009	9
50	46	271	626	0.014 ± 0.001	0.011	6
27	53	332	722	0.020 ± 0.004	0.012	6
17	40	348	736	0.018 ± 0.001	0.011	9
10	57	382	841	0.022 ± 0.001	0.015	15
35	48	343	1602	0.023 ± 0.002	0.026	7
46	52	412	1842	0.025 ± 0.004	0.029	6
45	63	587	1972	0.050 ± 0.004	0.031	8
36	53	580	1972	0.032 ± 0.002	0.027	12
26	45	302	2568	0.022 ± 0.002	0.039	8
47	40	303	2735	0.023 ± 0.002	0.038	9
24	54	368	3596	0.025 ± 0.004	0.060	12
42	45	405	3883	0.034 ± 0.004	0.050	12
14	49	512	4430	0.041 ± 0.002	0.063	11
8	51	512	4900	0.043 ± 0.007	0.070	14
18	68	784	4964	0.085 ± 0.006	0.068	15
4	58	662	5494	0.068 ± 0.007	0.070	13
48	58	623	7187	0.065 ± 0.005	0.099	15

Program ID	AEG nodes	AEG edges	Cycles	Cycle Detection (s)	ILP (s)	Min fences
29	71	869	8886	0.084 ± 0.006	0.137	14
19	61	793	8986	0.090 ± 0.004	0.123	13
12	63	660	10524	0.054 ± 0.005	0.157	11
20	60	445	12447	0.049 ± 0.006	0.263	10
23	64	725	13273	0.101 ± 0.006	0.206	16
28	65	618	14844	0.070 ± 0.004	0.222	15
9	65	833	15031	0.087 ± 0.004	0.241	13
16	65	723	18253	0.115 ± 0.009	0.304	13
33	79	1161	19239	0.205 ± 0.014	0.275	17
39	78	1189	21704	0.199 ± 0.015	0.396	16
6	71	1044	22395	0.193 ± 0.010	0.322	15
44	73	1028	33502	0.243 ± 0.007	0.680	16
31	75	1074	36370	0.195 ± 0.010	0.657	14
15	81	1186	38028	0.248 ± 0.011	0.814	17
43	90	1367	39639	0.319 ± 0.016	0.749	20
13	91	1421	40880	0.299 ± 0.011	0.867	18
1	82	1033	53161	0.199 ± 0.006	0.974	19
37	96	1346	67301	0.345 ± 0.008	1.704	21
32	71	819	72174	0.160 ± 0.006	1.557	20
0	89	1600	79066	0.406 ± 0.014	1.745	23
40	93	1471	79787	0.426 ± 0.029	1.694	22
11	90	1506	108791	0.393 ± 0.010	2.588	25
38	86	1258	116578	0.365 ± 0.016	2.845	21
2	84	1280	156907	0.447 ± 0.016	4.536	21
25	95	1794	159683	0.672 ± 0.004	3.684	23
34	97	1440	165924	0.546 ± 0.015	3.538	22
21	83	1318	179817	0.435 ± 0.012	4.630	22
22	111	2122	182942	1.199 ± 0.021	4.418	26
7	110	2049	301200	1.442 ± 0.015	7.736	29
3	106	1655	501640	1.228 ± 0.021	16.385	26
30	125	2673	2432676	7.004 ± 0.021	79.071	35