A Case for an SC-Preserving Compiler

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, Satish Naranasamy

Presented by
Akshay Gopalakrishnan

McGill University

July 9, 2021

Introduction

- Relaxed memory models.
- Program transformations for high performance.
- Many of them are SC-Preserving.
- Non-SC preserving transformations can be broadly attributed to reorderings.
- A speculation based language construct to perform load-store reorderings when SC-Preserving.
- Much of the performance is retained.

Memory Consistency Model - Sequential Consistency

Memory consistency model describes what values a read can have in a concurrent execution.

Sequential consistency guarantees that the read values of any concurrent execution of a program can always be justified by an execution of the same program in a uniprocessor (interleaving semantics).

The need for Relaxed Memory Model

Sequential consistency is:

- Too strict which prevents much of the performance H/W provides.
- Too strict to do common compiler optimizations responsible for a lot of performance benefits (eg: Common Sub-expression elimination)

Relaxed memory models:

- Describe pretty well what freedom hardware provides for reads (Store/Load buffers, Speculation, (Non)Multicopy Atomicity, etc).
- Allows a large class of compiler optimizations responsible for performance (high level languages).
- Giving low level constructs to provide fine grained concurrency (eg: Store-Load fence, MFence, Lwsync, etc).

Example of Non-SC Preserving Transformation

Original		Transformed	Concurrent Context

`L1: t = X*2;`
`L2: u = Y;`
`L3: v = X*2;`
	$\Rightarrow$	`L1: t = X*2;`
		`L2: u = Y;`
		`M3: v = t;`
			`N1: X = 1;`
			`N2: Y = 1;`

(a)		(b)	(c)

A Counter view

The authors note in their empirical study of LLVM compiler optimization passes:

- Much of the performance in concurrent programs due to program transformations are already SC-Preserving.
- A majority of Non-SC transformations responsible for much of the performance can be attributed to eager load/store transformations.
- These transformations can be summarized as reordering loads and stores throughout the program (thread-local).
- Part of these Non-SC transformations can be enabled using a simple speculation-based program constructs.

List of Major SC-Preserving Transformations

a) redundant load:	`t=X; u=X;`	$\Rightarrow$	`t=X; u=t;`
b) forwarded load:	`X=t; u=X;`	$\Rightarrow$	`X=t; u=t;`
c) dead store:	`X=t; X=u;`	$\Rightarrow$	`X=u;`
d) redundant store:	`t=X; X=t;`	$\Rightarrow$	`t=X;`

Figure 3: SC-preserving transformations

Categorizing Reorderings

Reordering of memory accesses can be classified into four parts:

- Load-Load
- Load-Store
- Store-Load
- Store-Store

Such categories also are fences in certain hardware and software level memory models.

- Allowing Eager load optimization is equivalent to eliminating Load-Load and Load-Store constraint.
- Allowing Eager store optimization is equivalent to elimination Store-Load and Store-Store constraint.

Example Being Optimized: SC-transformation

```
float Distance(
    float* x, float*y, int n){
  float sum = 0;
  int i=0;

  for(i=0; i<n; i++){
     sum += (x[i]-y[i])
           *(x[i]-y[i]);
     // Note: x[i] is *(x+i*4)
     //  and  y[i] is *(y+i*4)
  }

  return sqrt(sum);
}
```

(a)

```
float Distance(
    float* x, float* y, int n){
  register float sum = 0;
  register px = x;
  register py = y;
  register rn = n;

  for(; rn-->0; px+=4,py+=4){
    sum += (*px-*py)
          *(*px-*py);
  }

  return sqrt(sum);
}
```

(b)

Example Being Optimized: Non-SC Transformation

```
float Distance(
    float* x, float* y, int n){
  register float sum = 0;
  register px = x;
  register py = y;
  register rn = n;

  for(; rn-->0; px+=4,py+=4){
    sum += (*px-*py)
          *(*px-*py);
  }

  return sqrt(sum);
}
```

(b)

```
float Distance(
    float* x, float* y, int n){
  register float sum = 0;
  register px = x;
  register py = y;
  register rn = n;

  for(; rn-->0; px+=4,py+=4){
    register t = (*px-*py);
    sum += t*t;
  }

  return sqrt(sum);
}
```

(c)

Idea of Speculation

- Three base instructions.
- Monitor load (*m.load*).
- Monitor store (*m.store*).
- Interference check (*i.check*).

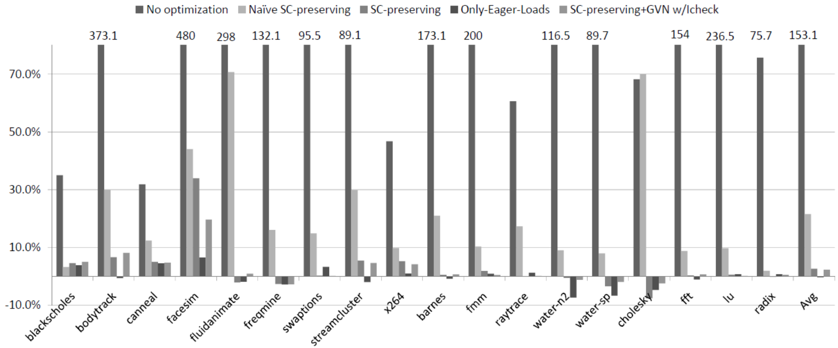
Basic idea of Interference Check algorithm

```
DOM                 DOM'
ORIG                ORIG'
CONTINUE   ⇒        i.chk monitoredAccesses, rcvr
                    jump cont
              rcvr: RECOVER
              cont: CONTINUE'
```

Evaluation strategy of performance

- First no optimization.
- Then only SC-preserving.
- Then SC-Preserving with speculative checks.
- Then even non-SC opt.

Results



No optimization
Naïve SC-preserving
SC-preserving
Only-Eager-Loads
SC-preserving+GVN w/Icheck
373.1
480
298
132.1
95.5
89.1
173.1
200
116.5
89.7
154
236.5
75.7
153.1
70.0%
50.0%
30.0%
10.0%
-10.0%
blackscholes
bodytrack
canneal
facesim
fluidanimate
freqmine
swaptions
streamcluster
x264
barnes
fmm
raytrace
water-n2
water-sp
cholesky
fft
lu
radix
Avg

Thank you

Questions?