A Formal Model and Soundness

We formalize the weak-memory fragment used by SuperCollider's load/store instrumentation to prove that our core detectors produce no false positives. As in the implementation, we model only shared locations and exclude thread-private addresses.

A.1 Execution Model

Let T be the set of threads and A the set of shared addresses. While the full PTX memory model is axiomatic and non-multi-copy-atomic (NMCA), we reason about execution traces using a simplified operational proxy: a single global memory $M : A \to Val$ together with per-thread views $V_t : A \to Val$ that may differ from M because weak loads can be served from caches that have not yet observed writes by other threads. This model is multi-copy-atomic (MCA): a store to a is immediately reflected in $M(a)$, so any subsequent strong load from any thread returns the same value. The actual PTX model is NMCA and admits additional behaviors our model cannot represent. Our model is, however, a conservative proxy. Because SuperCollider's detection is always from a *single* thread's perspective—comparing that thread's own weak load against its own subsequent `.sys.strong` load—the NMCA freedom of other threads does not affect our soundness argument.

A memory access is a tuple $op = (t, a, k, \sigma)$, where $t \in T$ is the issuing thread, $a \in A$ is the address, $k \in \{R, W\}$ is the kind, and $\sigma \in \{weak, strong\}$ indicates whether the access is an ordinary weak access or a strong access. In the PTX model, an operation is strong if it carries a qualifier such as `.relaxed`, `.acquire`, `.release`, or `.volatile`. Our instrumentation uses `.strong` duplicate accesses to read directly from $M(a)$ without introducing new inter-thread causality edges.

Weak loads return $V_t(a)$. Strong loads return $M(a)$. A weak store by thread t to address a updates both $M(a)$ and $V_t(a)$; stores by other threads update $M(a)$ but may leave $V_t(a)$ unchanged. $V_t(a)$ is refreshed to $M(a)$ when t executes a strong operation on a, or when t participates in a synchronization operation. A `nanosleep` instruction yields execution but is expressly not a synchronization operation: it introduces no causality edges and does not refresh any thread's view.

A.2 Causality Order

The PTX memory model relies on a **Causality Order** ($\to_{cause}$) rather than a standard happens-before relation. The causality order $\to_{cause}$ on an execution trace E is a transitive relation built upon:

(1) **(Program order)** If op_1 and op_2 are executed by the same thread and op_1 precedes op_2 in E, then op_1 precedes op_2 in program order (po).
(2) **(Synchronization order)** Fences, barriers, and acquire/release observation sequences establish synchronizes-with (sw) edges between threads.
(3) **(Base Causality)** The transitive closure of po and sw defines base causality, which is further extended into $\to_{cause}$.

Because the SuperCollider instrumentation inserts only `nanosleep` and redundant `.strong` (non-synchronizing) loads, it introduces no new inter-thread $\to_{cause}$ edges.

A.3 Conflicting Accesses and Data Races

Two operations $op_1 = (t_1, a, k_1, \sigma_1)$ and $op_2 = (t_2, a, k_2, \sigma_2)$ are *conflicting* if:

(a) $t_1 \neq t_2$ (different threads)
(b) they access the same address a
(c) at least one is a write (W)

Definition A.1 (Data Race). According to the PTX specification, a trace E contains a data race if there exist conflicting operations op_1, op_2 in E such that they are not related in causality order ($op_1 \nrightarrow_{cause} op_2$ and $op_2 \nrightarrow_{cause} op_1$) and they are not morally strong. Because the PTX memory model requires both operations to be strong and to specify inclusive scopes to be considered morally strong, the presence of at least one `.weak` operation in our targeted pairs means they trivially fail the criteria for moral strength, regardless of scope.

A.4 Soundness

A race detector is *sound* if every reported race corresponds to a genuine data race in E. We prove soundness for SuperCollider's core detection mechanisms.

Theorem A.2 (Clobbered Read Soundness). *If the clobbered-read instrumentation for a weak load by thread t at address a reports an error, then E contains a data race on a.*

Proof. Thread t executes: (1) $op_1 = \texttt{load\_weak}(a) \rightarrow v_1$, (2) `nanosleep`, (3) $\texttt{load\_strong}(a) \rightarrow v_2$, with $v_1 \neq v_2$. Since the strong load returns $M(a)$, we have $M(a) = v_2$ at step (3). We split into cases based on whether any write to a by a thread $t' \neq t$ occurs between steps (1) and (3) in E (i.e., after op_1 but before the strong load), or whether the causally relevant write precedes step (1), leaving t's view stale.

Case 1 (Intervening write). Some thread $t' \neq t$ writes a between steps (1) and (3); let op_w be one such write. The pair (op_1, op_w) is conflicting: different threads, same address, op_w is a write, and op_1 is weak (so the pair is not morally strong). They are causally unordered: t executes no synchronization between steps (1) and (3), so no $\rightarrow_{cause}$ path leads from op_1 to op_w; and op_w is later in E than op_1, so $op_w \nrightarrow_{cause} op_1$.

Case 2 (Stale view). No thread writes a between steps (1) and (3). Then $M(a)$ already held value v_2 at step (1), yet $V_t(a) = v_1 \neq v_2$: the weak load observed a stale value. Since t's own stores keep $V_t(a)$ in sync with $M(a)$, this implies some thread $t' \neq t$ previously wrote v_2 to a without that write propagating to t's view. Let op_w be the most recent write to a by any $t' \neq t$ prior to step (1). The pair (op_1, op_w) is conflicting and not morally strong. Since op_w precedes op_1 in E, we have $op_1 \nrightarrow_{cause} op_w$.

Since $M(a) = v_2$ from op_w through step (1) (no write by $t' \neq t$ after op_w by choice, and no store by t since $V_t(a) = v_1 \neq v_2 = M(a)$ shows t could not have stored to a after op_w), any synchronization by t in that interval would refresh $V_t(a)$ to $M(a) = v_2$. But $V_t(a) = v_1 \neq v_2$, so no such synchronization occurred. Any causality path from op_w (by $t' \neq t$) to op_1 (by t) requires an inter-thread *sw* edge into t, corresponding to exactly such a synchronization. Since none exists, $op_w \nrightarrow_{cause} op_1$.

In both cases, E contains a conflicting, non-morally-strong, causality-unordered pair: a genuine data race. □

Theorem A.3 (Lost Update Soundness). *If the lost-update instrumentation for a weak store by thread t at address a reports an error, then E contains a data race on a.*

Proof. Thread t executes: (1) $op_1 = \texttt{store\_weak}(a, v_1)$, (2) `nanosleep`, (3) $\texttt{load\_strong}(a) \rightarrow v_2$, with $v_1 \neq v_2$. Since the weak store sets $M(a) = v_1$ at step (1), yet $M(a) = v_2 \neq v_1$ at step (3), some thread $t' \neq t$ must have written to a between steps (1) and (3); call that write op_w.

The pair (op_1, op_w) is conflicting: different threads, same address, both writes. op_1 is weak, so the pair is not morally strong. Since t performs no synchronization between steps (1) and (3), no *sw* edge out of t exists in that interval, so no causality path leads from op_1 to op_w: $op_1 \nrightarrow_{cause} op_w$. Since op_w occurs after op_1 in E and causality respects execution order, $op_w \nrightarrow_{cause} op_1$. Therefore E contains a genuine data race. □

Theorem A.4 (Intra-Warp Soundness). *If the intra-warp check detects that two or more active lanes in a warp target the same shared address a for a weak store, then E contains a data race on a.*

Proof. Let t_1 and t_2 be distinct active threads in the same warp, both about to execute `store_weak`$(a, \cdot)$. The two operations are conflicting: different threads, same shared address, both are writes, and both are weak (thus not morally strong). Because $t_1 \neq t_2$, program order (*po*) does not relate them. Furthermore, despite executing in the same instruction, SIMT execution does not establish memory synchronization. Since no explicit synchronization intervenes, the writes are unordered by $\rightarrow_{cause}$, constituting a genuine data race. □

Theorem A.5 (Asynchronous Copy Soundness). *If the instrumentation for an asynchronous copy (*`cp.async`*) by thread t from global address a_g to shared address a_s reports an error, or if a subsequent load from a_s by thread t reports a clobbered-read error triggered by the clobbering phase, then E contains a data race on a_g or a_s.*

Proof. The instrumentation for `cp.async` has two phases. The *emulation phase* detects inter-thread races: (1) `load_weak`$(a_g) \rightarrow v_0$, (2) `store_weak`(a_s, v_0), (3) `nanosleep`, (4) `load_strong`$(a_g) \rightarrow v_1$, (5) `load_strong`$(a_s) \rightarrow v_2$; an error is reported if $v_1 \neq v_0$ or $v_2 \neq v_0$. The *clobbering phase* detects intra-thread missing synchronization: a sentinel value (`0xbadbeef`) is written to a_s, the actual `cp.async` is issued, and t's subsequent weak load from a_s is covered by standard clobbered-read instrumentation.

Case 1 (Inter-thread race on source, $v_1 \neq v_0$). By the same argument as Theorem A.2, the mismatch between the weak load at step (1) and the strong load at step (4) establishes a conflicting, causally unordered write by some $t' \neq t$ to a_g: a data race on a_g.

Case 2 (Inter-thread race on destination, $v_2 \neq v_0$). The weak store at step (2) set $M(a_s) = v_0$, yet $M(a_s) = v_2 \neq v_0$ at step (5). By the same argument as Theorem A.3, some thread $t' \neq t$ wrote to a_s between steps (2) and (5). The pair consisting of the step (2) store and that write is conflicting, not morally strong, and causally unordered (t performs no synchronization between those steps): a data race on a_s.

Case 3 (Intra-thread missing synchronization). The emulation phase passes, but the clobbered-read check on t's subsequent weak load from a_s reports an error. By Theorem A.2, this error implies a data race on a_s.

In all cases, the reported error corresponds to a genuine data race in E. □

Corollary A.6 (No False Positives). *Within this model, SuperCollider's core weak-memory detectors produce no false positives: clobbered read, lost update, intra-warp lost update, and asynchronous copy each report only when a genuine data race exists.*

Proof. Immediate from Theorems A.2–A.5, each of which establishes that a reported error implies the existence of a genuine data race in E. □

Lemma A.7 (Semantics Preservation). *The core weak-load and weak-store instrumentation is semantics-preserving for race-free executions: it does not alter the observable memory state or control flow of a valid program.*

Proof. The instrumentation introduces only three elements: a `nanosleep` instruction, a duplicate `.strong` load, and a comparison branch. The `nanosleep` instruction perturbs warp scheduling but has no architectural side effects on memory state. The duplicate `.strong` loads are strictly read-only and utilize fresh temporary registers, leaving the original program's register state intact. Finally, as established by the preceding Corollary, in a race-free execution the comparison will never fail, meaning the error-reporting control flow path is strictly unreachable. Therefore, the original program semantics are perfectly preserved. □

B Supplemental Results

Table 5. Sensitivity study using the HeCBench autohecbench script, which runs 146 applications. The *Total* column shows the total number of races discovered for each configuration across all runs. A race reported in the 1/6 column was only reported in one of the six runs. For each configuration, the #Apps column shows the number of apps in the HeCBench suite where races were detected.

Configuration	**HeCBench Races Discovered**							
#rdelay,#wdelay	1/6	2/6	3/6	4/6	5/6	6/6	Total	#Apps
1,1	0	0	0	0	0	35	35	12
1,10	0	0	0	0	0	35	35	12
1,25	0	0	0	0	0	35	35	12
1,50	0	0	0	0	0	35	35	12
1,250	0	0	0	0	0	35	35	12
1,1000	0	0	0	0	0	35	35	12
1,5000	1	1	0	0	0	35	37	12
10,1	0	0	0	0	0	35	35	12
10,10	0	0	0	0	0	35	35	12
10,25	0	0	0	0	0	35	35	12
10,50	0	0	0	0	0	35	35	12
10,250	0	0	0	0	0	35	35	12
10,1000	0	0	0	0	0	35	35	12
10,5000	2	1	0	0	0	35	38	13
25,1	0	0	0	0	0	35	35	12
25,10	0	0	0	0	1	34	35	12
25,25	0	0	0	0	0	35	35	12
25,50	0	0	0	0	0	35	35	12
25,250	0	0	0	0	0	35	35	12
25,1000	0	0	0	0	0	35	35	12
25,5000	1	1	0	0	0	35	37	13
50,1	0	0	0	0	0	35	35	12
50,10	0	0	0	0	0	35	35	12
50,25	0	0	0	0	0	35	35	12
50,50	0	0	0	0	1	34	35	12
50,250	0	0	0	0	0	35	35	12
50,1000	0	0	0	0	0	35	35	12
50,5000	1	1	0	0	0	35	37	12
250,1	1	0	0	0	0	35	36	12
250,10	0	0	0	0	1	34	35	12
250,25	0	0	0	0	0	35	35	12
250,50	0	0	0	0	0	35	35	12
250,250	0	0	0	0	0	35	35	12
250,1000	0	0	0	0	0	35	35	12
250,5000	1	0	0	0	0	35	36	13
1000,1	0	0	0	0	0	35	35	12
1000,10	0	0	0	0	0	35	35	12
1000,25	0	0	0	0	0	35	35	12
1000,50	0	0	0	0	0	35	35	12
1000,250	0	0	0	0	0	35	35	12
1000,1000	0	0	0	0	0	35	35	12
1000,5000	2	2	0	0	0	35	39	13
5000,1	2	2	0	1	0	43	48	16
5000,10	2	0	1	0	1	43	47	16
5000,25	0	2	1	1	0	42	46	16
5000,50	1	1	0	2	2	42	48	16
5000,250	4	0	1	2	0	42	49	16
5000,1000	2	0	2	1	0	42	47	16
5000,5000	1	0	1	1	3	41	47	16

Table 6. Races detected in each benchmark suite.

Suite	Benchmarks
Gunrock	color, dawn, kcore, mst, sssp
CUTLASS	ctest_profiler_spgemm, ctest_profiler_conv2d, ctest_profiler_gemm
CUDA Samples	Mandelbrot, MarchingCubes, newdelete, segmentationTreeThrust, simpleAttributes
HeCBench (autohecbench)	adv, aop, asta, atomicCAS, bh, dxtc2, expdist, fsm, hausdorff, knn, linearprobing, multinomial, p4, tissue, tqs, tridiagonal
HeCBench standalone	btree, d3q19-bgk, f16atomic, fpdc, lebesgue, michalewicz, pns, qtclustering, scatter, simpleSpmv, word2vec, wyllie

Received 2025-11-13; accepted 2026-04-03