SPECULATIVE BARRIERS WITH TRANSACTIONAL MEMORY

M. PEDRERO, R. QUISLANT, E. GUTIÉRREZ, E. L. ZAPATA, O. PLATA

DEPT. COMPUTER ARCHITECTURE

UNIVERSIDAD DE MÁLAGA

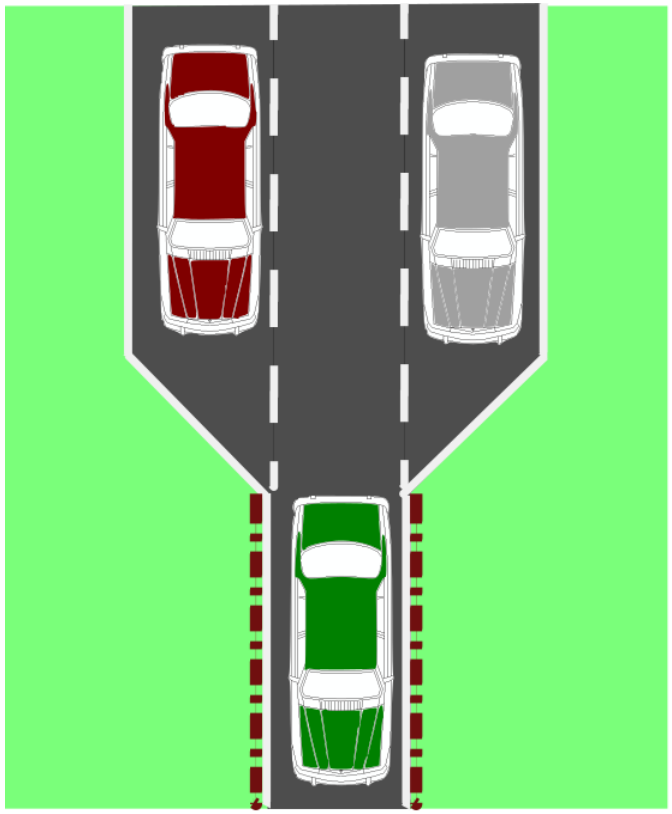
UNIVERSIDAD DE MÁLAGA

CONTENTS

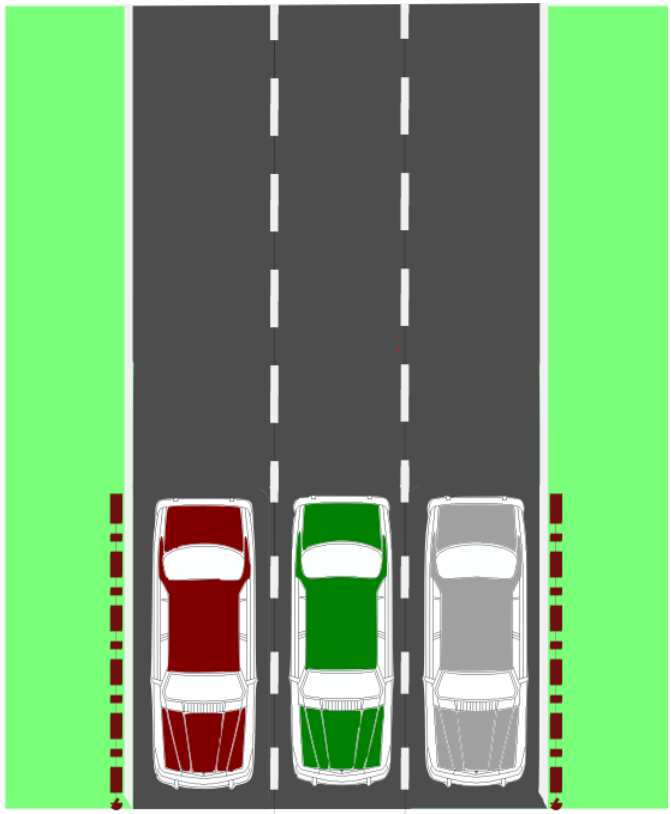
- **Background**
- Introduction
- Speculative Barriers (SBs)
  - API
  - SBs and non-transactional codes
- Experimental Evaluation
  - Methodology
  - Results
- Related Work
- Conclusions

BACKGROUND

LOCKS

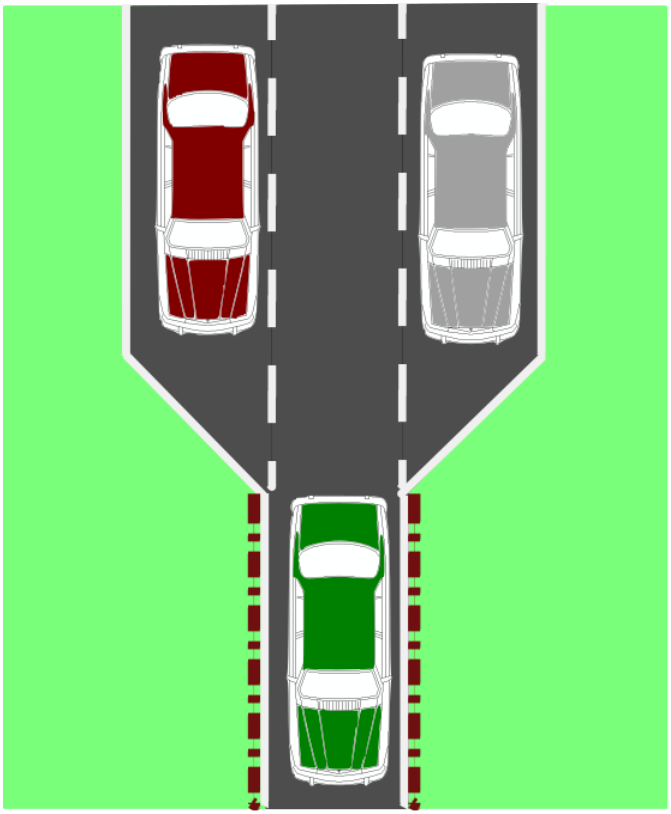


TRANSACTIONS

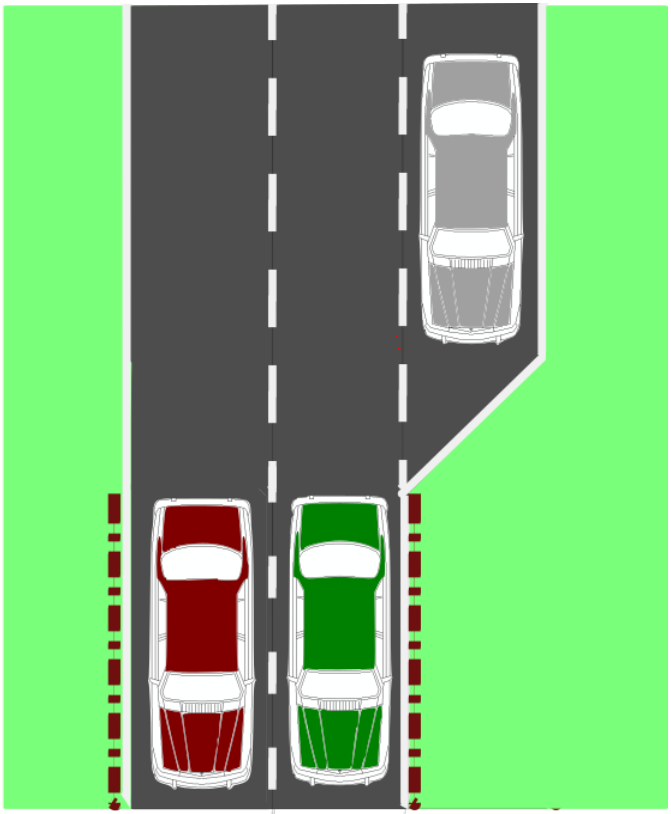


BACKGROUND

LOCKS

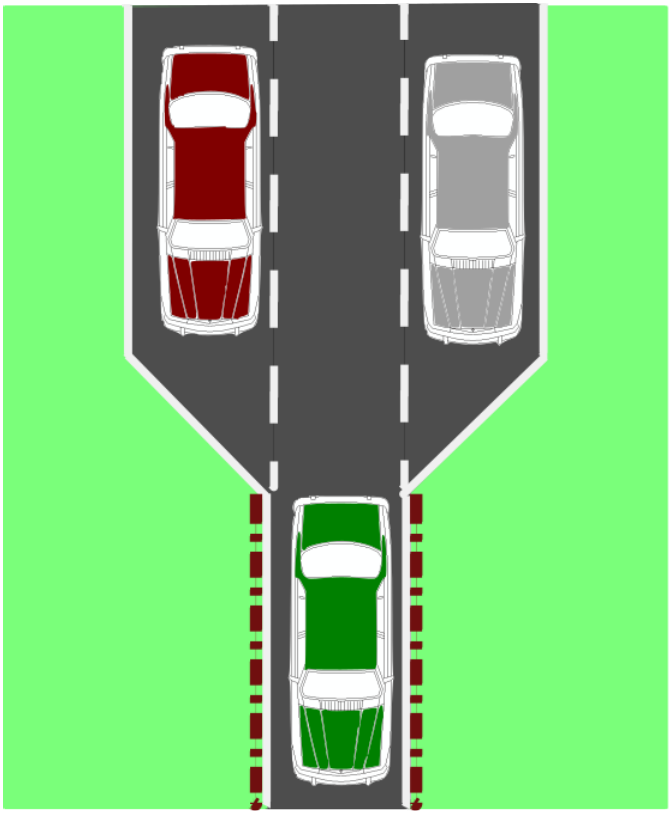


TRANSACTIONS

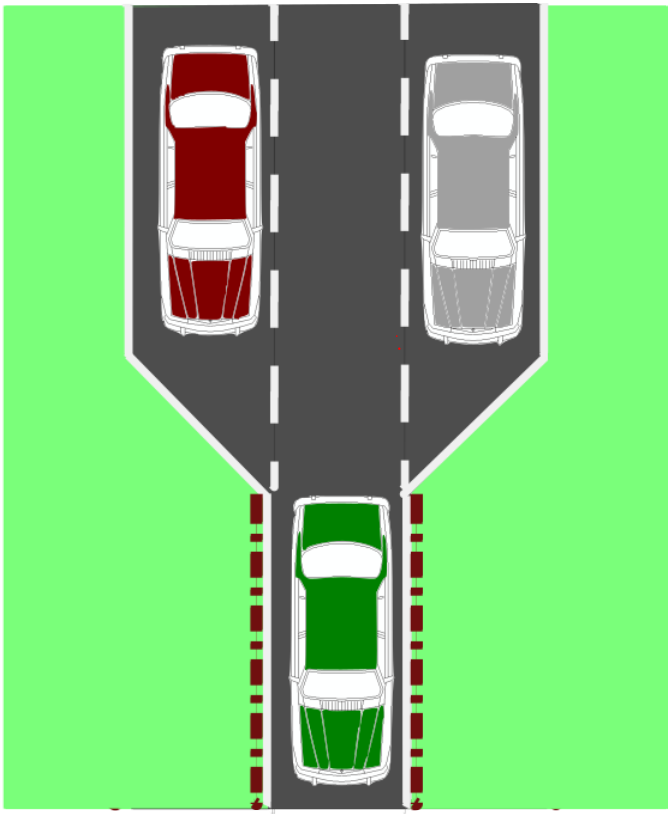


BACKGROUND

LOCKS



TRANSACTIONS



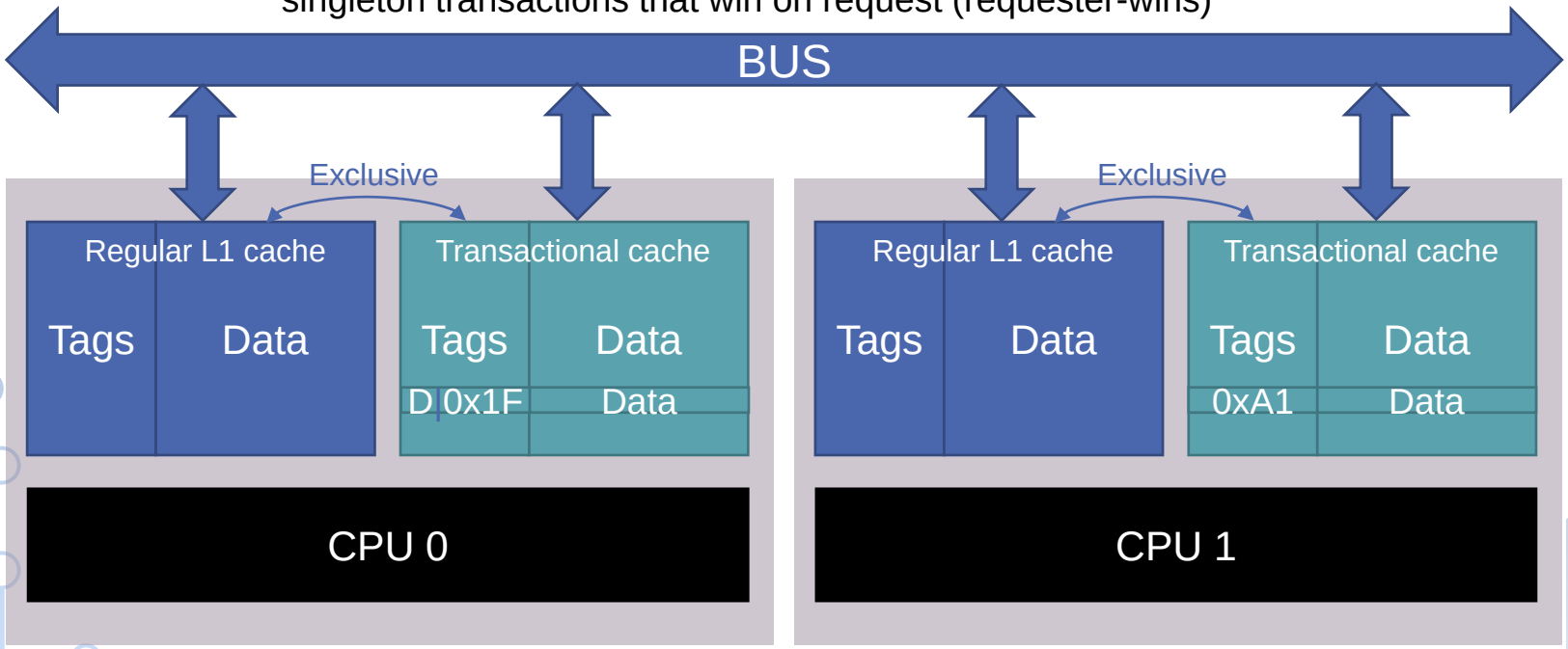
BACKGROUND

- **Hardware Transactional Memory** [Herlihy'93]
  - ISA extensions:
    - Load (exclusive) transactional, store transactional, abort, validate, commit
    - Explicit HTM: each load/store is to be annotated
  - Per-core small private transactional fully-associative cache:
    - Aborts/commits in one-cycle
    - Small/short transactions
  - Coherence protocol modification:
    - In charge of detecting conflicts among transactions
  - **Pros & Cons:**
    - Performance: fewer accesses
    - Forward progress not guaranteed

```
ST(&counter, LTX(&counter) + 1);
if (COMMIT()) {                    Transaction
  success++;
  backoff = BACKOFF_MIN;
} else {
  wait = random()%(1<<backoff);
  while (wait--);
  if (backoff < BACKOFF_MAX)
    backoff++;
}
```

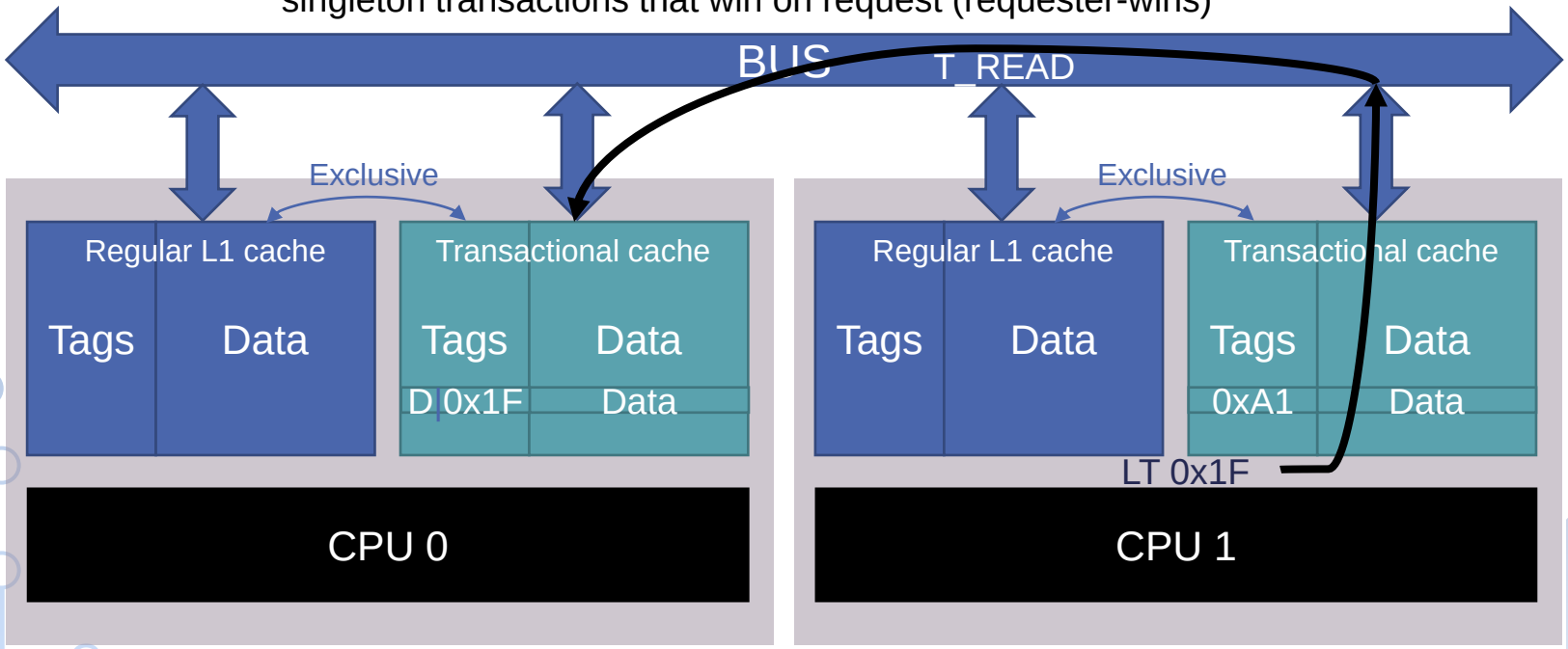
BACKGROUND

- **Hardware Transactional Memory** [Herlihy'93]
  - Conflict resolution policy: Requester aborts
  - Herlihy's approach does not define behavior on interaction with non-transactional code
    - Strong isolation [Blundell'06]: non-transactional instructions are treated as singleton transactions that win on request (requester-wins)



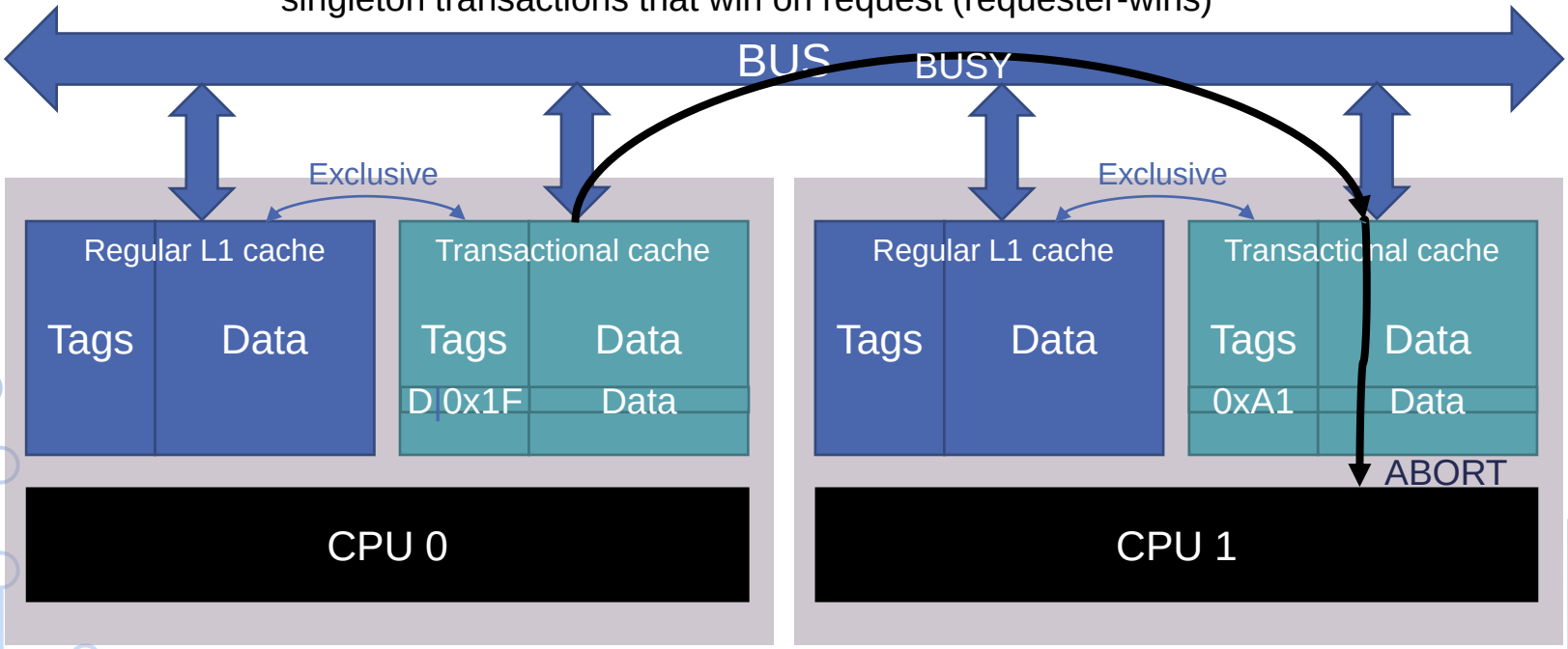
BACKGROUND

- **Hardware Transactional Memory** [Herlihy'93]
  - Conflict resolution policy: Requester aborts
  - Herlihy's approach does not define behavior on interaction with non-transactional code
    - Strong isolation [Blundell'06]: non-transactional instructions are treated as singleton transactions that win on request (requester-wins)



BACKGROUND

- **Hardware Transactional Memory** [Herlihy'93]
  - Conflict resolution policy: Requester aborts
  - Herlihy's approach does not define behavior on interaction with non-transactional code
    - Strong isolation [Blundell'06]: non-transactional instructions are treated as singleton transactions that win on request (requester-wins)



BACKGROUND

- **Research following Herlihy's seminal paper:**
  - Focused on supporting larger/longer transactions:
    - Transactional data set (read/write set) is limited to cache size
    - Transactions do not survive interrupts
  - Proposals:
    - UTM [Ananian'05]: Transactions nearly as large as virtual memory; transaction information in memory log (surviving context switches and migrations); nested transactions are flattened
    - VTM [Rajwar'05]: Overflowed transactions allocate entries in a virtual memory data structure; page faults are managed by suspending-resuming the transaction
    - LogTM-SE [Yen'07]: Software undo-log; signatures for fast/unbound conflict detection; escape actions; closed-nesting with partial aborts and open-nesting
    - OneTM [Blundell'07]: Allows for one overflowed transaction to continue, while others stall

BACKGROUND

- **Commercial HTM extensions:**
  - Back to Herlihy's **best-effort** approach
  - **Intel TSX** (Transactional Synchronization eXtensions), Core Haswell, 2013:
    - HLE: Hardware Lock Elision
      - Prefixes to certain instructions (e.g., XCHG) to elide locks and execute transactions instead
      - Enables backwards compatibility and portability
    - RTM: Restricted Transactional Memory
      - New instructions (`xbegin`, `xend`, `xabort`): Implicit HTM
      - L1D cache holds transactional stores (~22KB); L3 keeps track of transactional loads (~4MB) [Nakaike'15]
      - Cache coherence protocol implements a requester-wins conflict resolution policy
      - Flat nesting
      - Transaction **suspend/resume** (escape actions) is **not supported**
        - Future extensions (Sapphire Rapids) to include Suspend Load Address Tracking (TSXLDTRK)

BACKGROUND

- **Commercial HTM extensions:**
  - **Intel TSX Transaction API:** wrappers implementing retry and fallback path

```
#define XBEGIN()
  int retries = 0;
  do {
    retries++;
    if (retries > MAX_RETRIES) {
      acquire_lock(g_fallback_lock);
      break;
    }
  } while (_xbegin() != _XBEGIN_STARTED)

#define XEND()
  if (retries <= MAX_RETRIES) {
    if (g_fallback_lock) _xabort(LOCK_TAKEN);
    _xend();
  } else
    release_lock(g_fallback_lock);
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
    move eax, _XBEGIN_STARTED
    xbegin _txnL1
_txnL1: move status, eax
  }
  return status;
}
```

Intrinsic

BACKGROUND

- **Commercial HTM extensions:**
  - **IBM Power8/9**, 2014/2017:
    - New instructions (`tbegin`, `tend`, `tabort`, `tsuspend`, `tresume`): Implicit HTM
    - L2 TMCAM (Content Addressable Memory) holds transactional loads and stores (8KB) [Nakaike'15]
    - Flat nesting
    - Transaction **suspend/resume** (escape actions) **are supported**
      - Debugging
      - Thread-Level Speculation: extracting parallelism from a sequential application
        - One thread expanded per loop iteration
        - Loop iteration enclosed within a transaction to detect inter-loop dependencies
        - Escaped loop on a control variable before calling `tend`
      - **Speculative Barriers**
  - **ARM TME (Transactional Memory Extension)**, ARMv9, 2021

BACKGROUND

- **Commercial HTM extensions:**
  - **Power8 Transaction API:** wrappers implementing retry and fallback path

```
#define TBEGIN()
  int retries = 0;
  do {
    retries++;
    if (retries > MAX_RETRIES) {
      acquire_lock(g_fallback_lock);
      break;
    }
  } while (!_tbegin())

#define TEND()
  if (retries <= MAX_RETRIES) {
    if (g_fallback_lock) _xabort(LOCK_TAKEN);
    _tend();
  } else
    release_lock(g_fallback_lock);
```

Returns true if the transaction was successfully started; and false if not, or on abort

UNIVERSIDAD DE MÁLAGA

CONTENTS

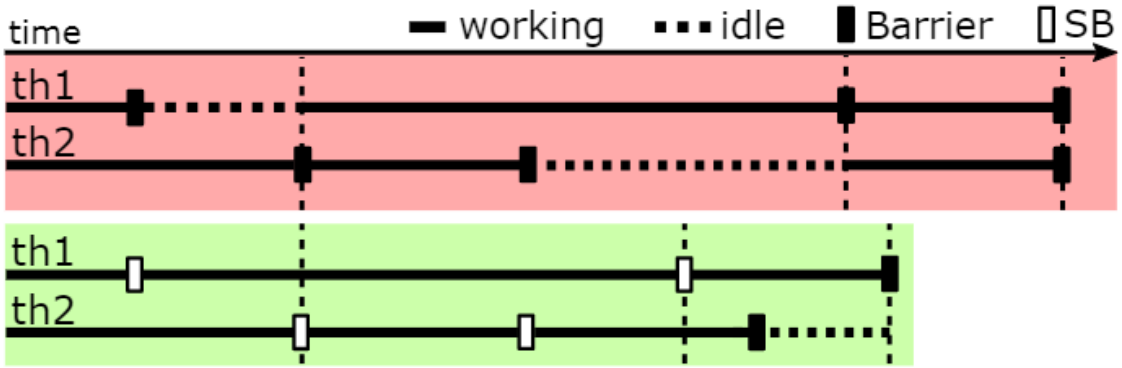
- Background
- **Introduction**
- Speculative Barriers (SBs)
  - API
  - SBs and non-transactional codes
- Experimental Evaluation
  - Methodology
  - Results
- Related Work
- Conclusions

INTRODUCTION

- Speculative Barriers (SB)
  - A thread reaching a barrier elides it and executes transactionally from there on
  - The transaction opened by a thread after crossing a barrier (SB transaction) is committed as soon as all threads cross the barrier
  - Escape actions are used inside transactions to perform control actions and prevent aborts
  - A partial order is implemented among transactions before and after a barrier
  - We provide an API for SBs including wrappers for beginning/committing a transaction, and a barrier substitute
    - IBM Power8 HTM extensions are used for the implementation (tsuspend-tresume)

INTRODUCTION

- **Motivation**
  - *Target*: barrier-intensive applications that exhibits poor performance due to:
    - Load imbalance (idle threads)
    - Contention (many threads, one barrier lock)
  - *Solution:* Harnessing imbalance to get ahead with work and eliding the barrier to avoid contention

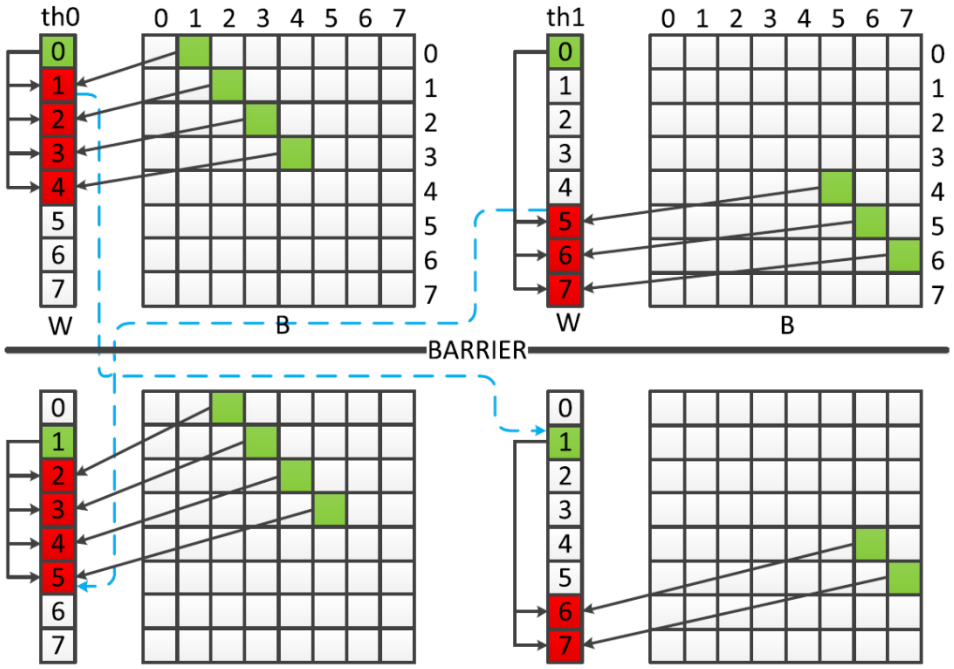


Standard barriers (top) versus SBs (bottom).

INTRODUCTION

- **Motivation:** Recurrence

```
chunk = N/#TH;
start = tid*chunk;
stop = MIN(start+chunk,N);
for (t=0; t ≤ N-2; t++) {
  for (k=start; k<stop; k++)
  {
    if (k < (n-t-1))
      w[t+k+1]+=
          b[k][t+k+1]*w[t];
  }
  barrier();
}
```



UNIVERSIDAD DE MÁLAGA

CONTENTS

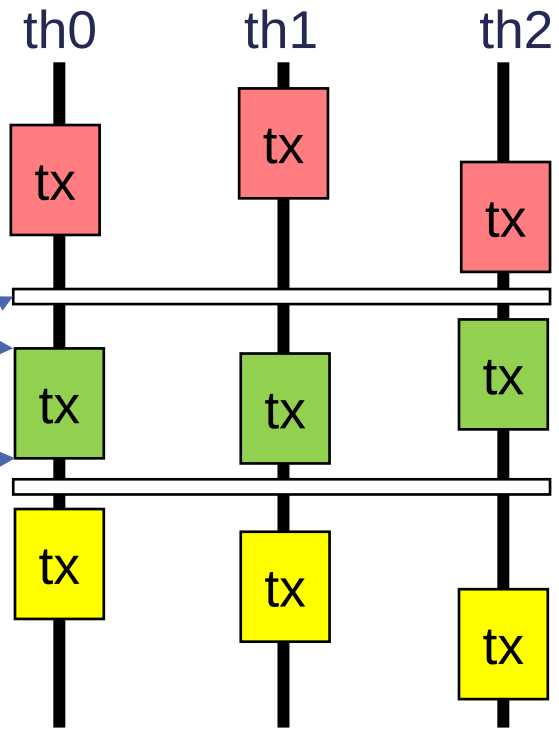
- Background
- Introduction
- **Speculative Barriers (SBs)**
  - **API**
  - **SBs and non-transactional codes**
- Experimental Evaluation
  - Methodology
  - Results
- Related Work
- Conclusions

SPECULATIVE BARRIERS

- **SB implementation:**
  - SBs first intended for codes with transactions
  - We use begin/commit as well as barrier wrappers to implement SBs

```
chunk = N/#TH;
start = tid*chunk;
stop = MIN(start+chunk,N);
for (t=0; t ≤ N-2; t++) {
  for (k=start; k<stop; k++) {
    TX_START();
    if (k < (n-t-1))
      w[t+k+1]+=
          b[k][t+k+1]*w[t];
    TX_STOP();
  }
  SPEC_BARRIER();
}
LAST_BARRIER();
```

Recurrence



SPECULATIVE BARRIERS

- **SB implementation:** SB API
  - Global variables and transaction descriptor

```
global int sbBarrier = #TH; //Barrier counter
global int sbOrder = 1; //Barrier global order
thread local tx {
  int order = 1;  //Local order
  int retries = 0;
  bool spec = 0;  //Speculation flag
  int specMax = MAX_SPEC;   //Adaptive
  int specLevel = MAX_SPEC; //speculation level
}
```

SPECULATIVE BARRIERS

- **SB implementation:** SB API
  - TX_START() wrapper

```
chunk = N/#TH;
start = tid*chunk;
stop = MIN(start+chunk,N);
for (t=0; t ≤ N-2; t++) {
  for (k=start; k<stop; k++) {
    TX_START();
    if (k < (n-t-1))
      w[t+k+1]+=
          b[k][t+k+1]*w[t];
    TX_STOP();
  }
  SPEC_BARRIER();
}
LAST_BARRIER();
```

Recurrence

```
#define TX_START()
  if (tx.spec == 0) { //Not executing an SB
    TBEGIN(); //Normal transaction begin
  }//If in SB mode => do nothing
```

```
#define TBEGIN()
  int retries = 0;
  do {
    retries++;
    if (retries > MAX_RETRIES) {
      acquire_lock(g_fallback_lock);
      break;
    }
  } while (!_tbegin())
```

tx.retries

Transaction API

SPECULATIVE BARRIERS

- **SB implementation:** SB API
  - TX_STOP() wrapper

```
chunk = N/#TH;
start = tid*chunk;
stop = MIN(start+chunk,N);
for (t=0; t ≤ N-2; t++) {
  for (k=start; k<stop; k++) {
    TX_START();
    if (k < (n-t-1))
      w[t+k+1]+=
          b[k][t+k+1]*w[t];
    TX_STOP();
  }
  SPEC_BARRIER();
}
LAST_BARRIER();
```

Recurrence

```
#define TX_STOP()
  if (tx.spec == 0) { //Not in SB mode
    TEND(); //Normal transaction end
    resetDescriptor(tx);
  } else { //In SB mode
    _tsuspend();
    if (tx.order <= sbOrder) {
      //1. All crossed the barrier
      _tresume();
      _tend(); //Commit SB
      resetDescriptor(tx);
    } else {
      //2. Some threads didn't cross
      _tresume();
      tx.specLevel--;
      if (tx.specLevel == 0) {
        _tsuspend();
        while (tx.order > sbOrder) ;
        _tresume();
        _tend(); //Commit SB
        resetDescriptor(tx);
      }
    }
  }
```

Busy wait

SPECULATIVE BARRIERS

- **SB implementation:** SB API
  - SPEC_BARRIER() wrapper

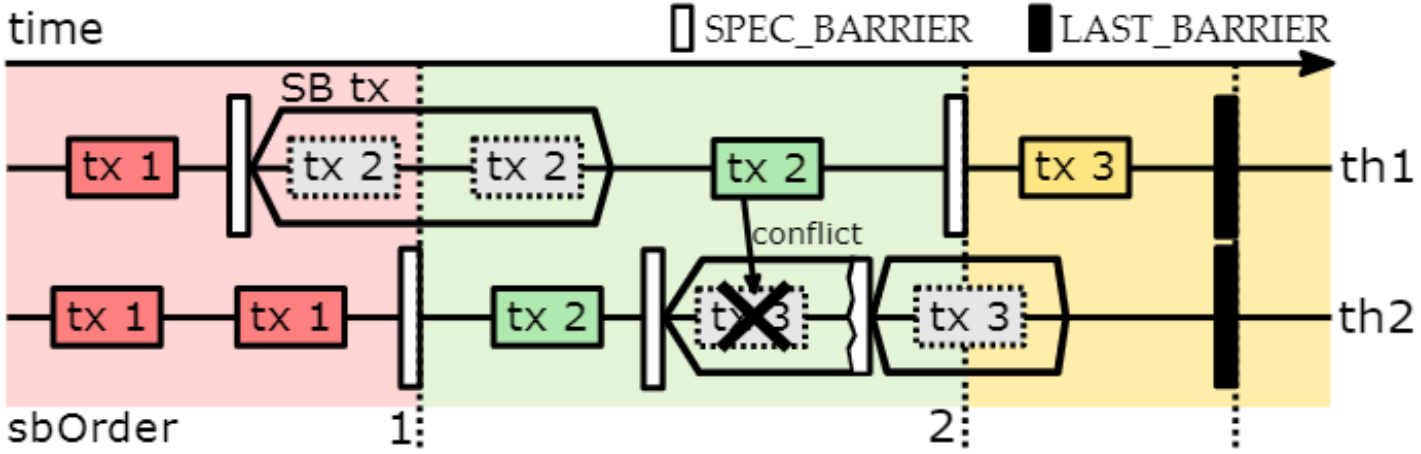
```
chunk = N/#TH;
start = tid*chunk;
stop = MIN(start+chunk,N);
for (t=0; t ≤ N-2; t++) {
  for (k=start; k<stop; k++) {
    TX_START();
    if (k < (n-t-1))
      w[t+k+1]+=
          b[k][t+k+1]*w[t];
    TX_STOP();
  }
  SPEC_BARRIER();
}
LAST_BARRIER();
```

Recurrence

```
#define SPEC_BARRIER()
  if (tx.spec == 1) {
    _tsuspend();
    while (tx.order > sbOrder) ;
    _tresume()
    _tend(); //Commit SB
    resetDescriptor(tx);
  }
  tx.order++; //Local order
  if (_addAndFetch(sbBarrier,-1) == 0) {
    sbBarrier = #TH; //Barrier reset
    _addAndFetch(sbOrder,1); //Global order
  } else { //Not the last thread to cross
    do {
      tx.retries++;
      if (tx.order <= sbOrder) {
        resetDescriptor(tx);
        break;
      }
      tx.spec = 1;
      decrementMaxSpecLevel();
    } while (!_tbegin());
  }
```

SPECULATIVE BARRIERS

- **SB implementation:** SB API
  - LAST_BARRIER() wrapper
    - Busy waits until speculative barrier threads
    - It is similar to SPEC_BARRIER() but it does not begin new speculative transactions



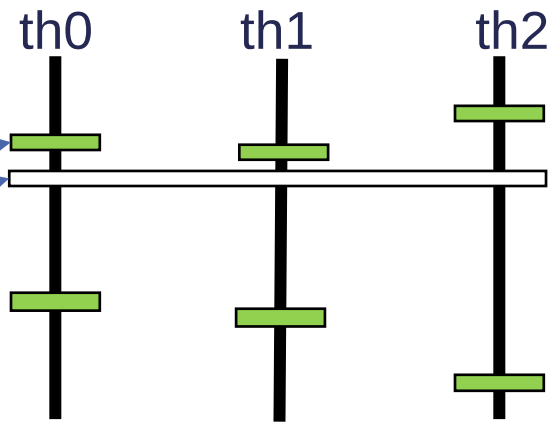
SB scenario.

SPECULATIVE BARRIERS

- **SBs and non-transactional code**
  - CHECK_SPEC() wrapper
    - Transactional codes offer SB check points naturally on TX_STOP()
    - In non-transactional codes, SB transactions may run for long (e.g., from barrier to barrier) → CHECK_SPEC() solves the problem
  - Implementation: like else clause from TX_STOP()
  - It can be used with transactional codes as well
    - If transactions are far away each other, CHECK_SPEC() provides additional SB check points

```
for (t=0; t ≤ N-2; t++) {
  for (k=start; k<stop; k++) {
    if (k < (n-t-1))
      w[t+k+1]+=
          b[k][t+k+1]*w[t];
    CHECK_SPEC();
  }
  SPEC_BARRIER();
}
```

Recurrence



UNIVERSIDAD DE MÁLAGA

CONTENTS

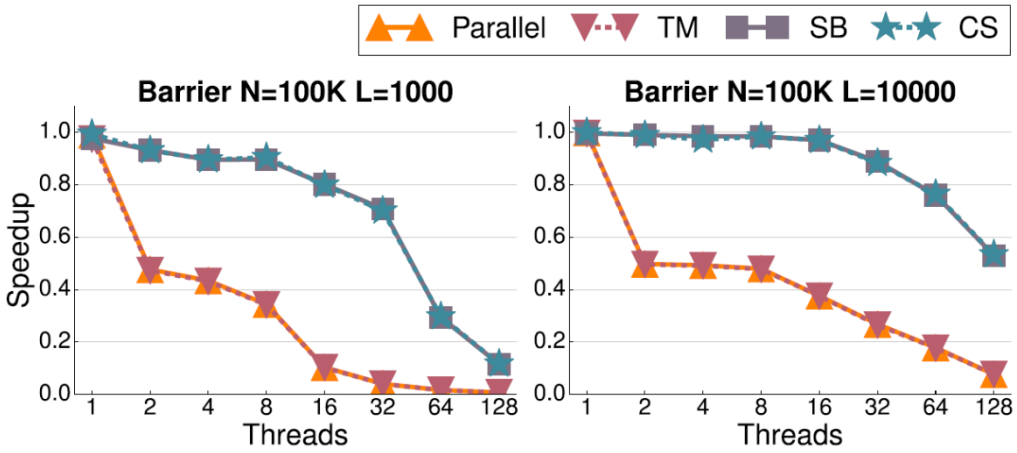
- Background
- Introduction
- Speculative Barriers (SBs)
  - API
  - SBs and non-transactional codes
- **Experimental Evaluation**
  - **Methodology**
  - **Results**
- Related Work
- Conclusions

METHODOLOGY

- Server S822LC-8335 with 2 sockets:
  - IBM Power8 processor with 10 cores each and 8 SMT → 160 threads
- Best-effort HTM:
  - 8KB cache for transactional accesses
    - The cache is shared by threads when using SMT
  - 128B words
  - Escape actions (tsuspend/tresume)
- Experiments were carried out up to 128 threads with round-robin affinity:
  - 1-8 threads → 1 socket, w/o SMT
  - 16 threads → 2 sockets, w/o SMT
  - 32+ threads → 2 sockets, with SMT

METHODOLOGY

- Experiment profiles:
  - *Unprotected*: no barriers, upper bound
  - *Parallel*: no transactions and standard barriers
  - *TM*: transactions and standard barriers
  - *SB*: transactions and speculative barriers (SBs)
  - *CS*: no transactions, SBs and CHECK_SPECs

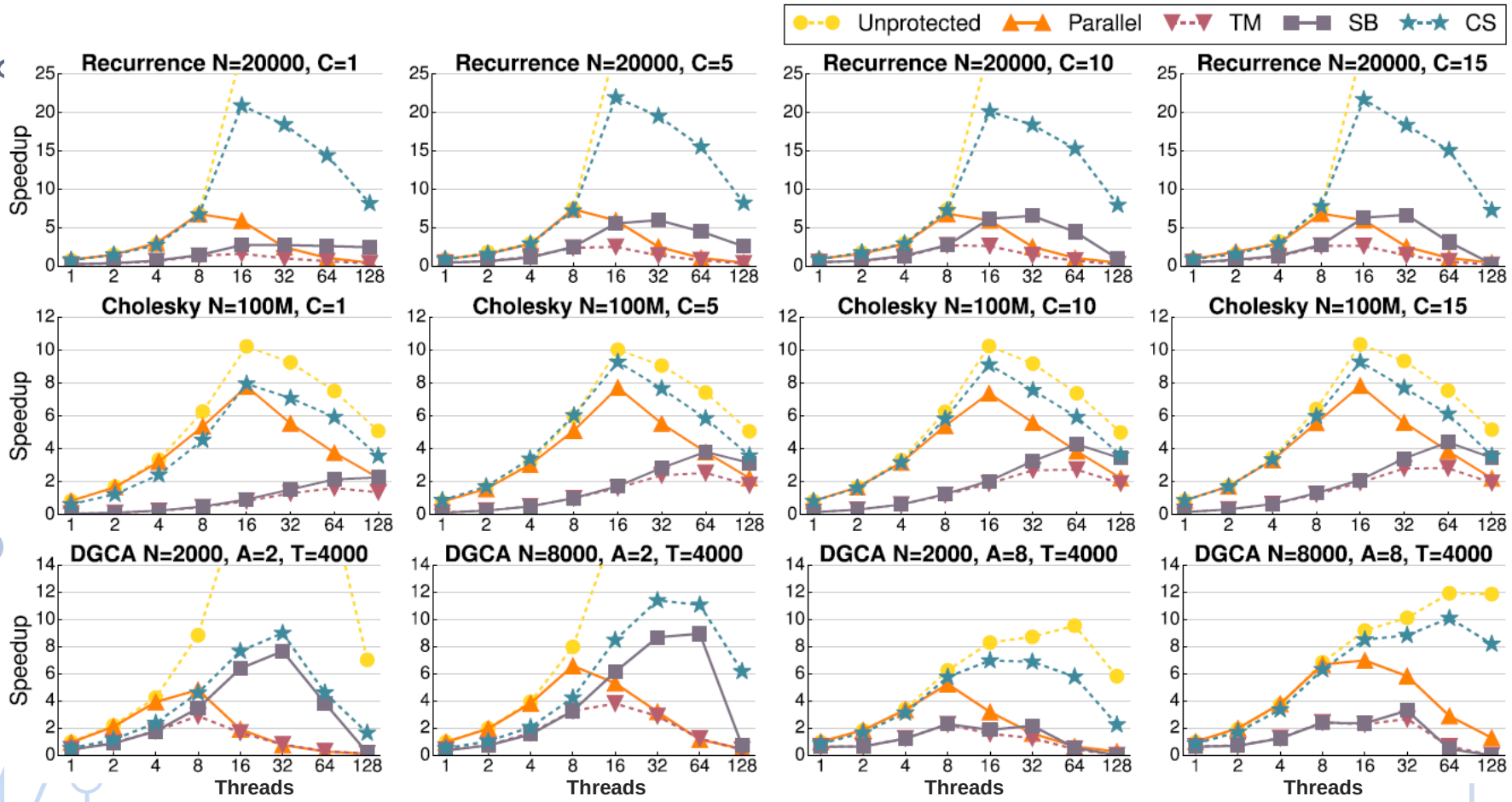


RESULTS

		Power 8		TCR (%)/SCR (%)		
	Parameters	#Barr	|tx| (M/A)	TM	SB	CS
Barr	N100K L1K	100K	5/4.5	99.9/-	99.9/99.9	-/99.9
	N100K L10K	100K	5/4.5	99.9/-	99.7/99.9	-/99.9
Recurren	N20K C1	20K	9/6.5	98.6/-	99.3/6.3	-/34.6
	N20K C5	20K	16/8.9	95.5/-	98.6/8.3	-/33.1
	N20K C10	20K	21/11.8	92.2/-	97.9/8.4	-/26.7
	N20K C15	20K	26/14.8	88.6/-	96.9/7.9	-/29.2
Cholesky	N100M C1	27	8/7.2	100/-	100/38.0	-/60.9
	N100M C5	27	10/8.5	99.9/-	100/43.8	-/58.8
	N100M C10	27	12/10	99.9/-	100/45.1	-/59.3
	N100M C15	27	12/11.6	99.9/-	100/45.7	-/57.1
DGCA	N2K A2 T4K	4K	20/18.1	99.8/-	99.9/99.6	-/99
	N8K A2 T4K	4K	20/18.1	99.9/-	99.9/99.4	-/99.7
	N2K A8 T4K	4K	27/25.1	94.2/-	93.1/42.1	-/48.6
	N8K A8 T4K	4K	26/24.6	98.1/-	97.8/29.6	-/40.3
Sten	Large T100	101	12/11.1	99.9/-	99.9/10.3	-/25.5
	Large T800	801	12/11.1	99.9/-	99.9/7.6	-/26.7
Hist	Large T1K C1	2001	6/5.9	98.9/-	98.9/0.9	-/1.2
	Large T1K C5	2001	15/13.2	30.4/-	30.9/0.2	-/0.2
SSCA2	k1 -s14-l9-p9	10	8/8	98.6/-	98.5/3	-
	k1 -s20-l3-p3	10	8/8	99.9/-	99.9/0.2	-
	k4 -s14-l9-p9	24K	6/6	71.77/-	30.07/0.51	-
	k4 -s20-l3-p3	1572K	6/6	62.61/-	30.47/0.05	-

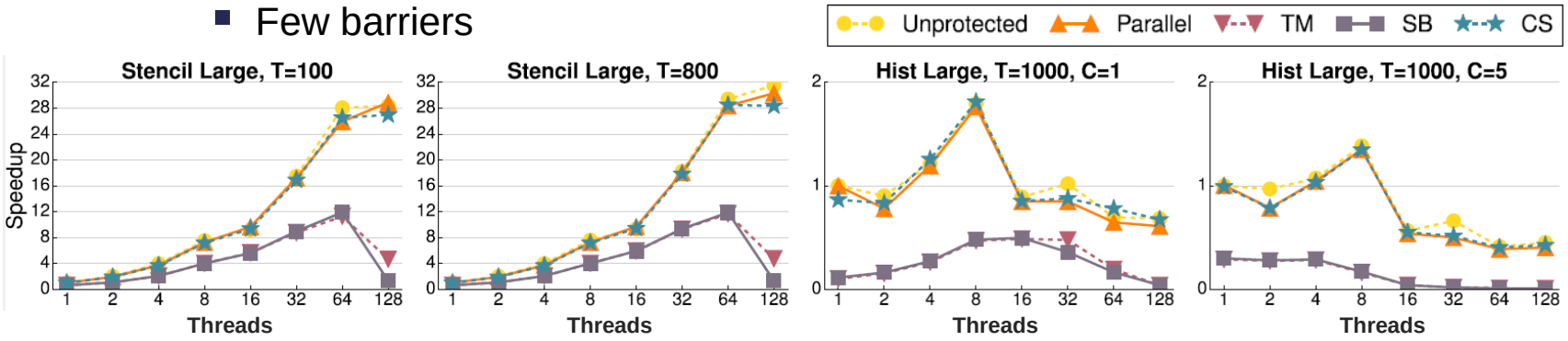
UNIVERSIDAD DE MÁLAGA

RESULTS

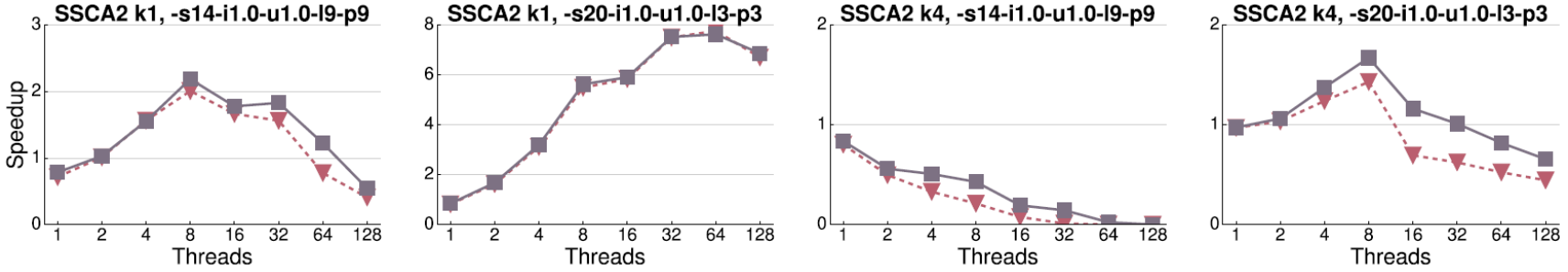


RESULTS

- Bad case scenarios:
  - No load imbalance
  - Few barriers



- Transactional workloads:
  - Only TM and SB profiles
  - More barriers and load imbalance in Kernel 4



UNIVERSIDAD DE MÁLAGA

CONTENTS

- Background
- Introduction
- Speculative Barriers (SBs)
  - API
  - SBs and non-transactional codes
- Experimental Evaluation
  - Methodology
  - Results
- **Related Work**
- Conclusions

RELATED WORK

- *[Martínez, ASPLOS'02]*
  - TLS on locks, flags and barriers
  - Dedicated hardware
  - Transaction ordering is not supported
  - Expose hardware to the user
- *[Nagarajan, LCPC'09]*
  - Two codes: with and without barriers
  - Dedicated hardware and compiler
  - 12% improvement
- *[Bonnichsen, IWOMP'15]*
  - Barrier speculation implemented with Intel's RTM
  - Transaction ordering is not supported
  - No performance gains (No escape actions)

UNIVERSIDAD DE MÁLAGA

CONTENTS

- Background
- Introduction
- Speculative Barriers (SBs)
  - API
  - SBs and non-transactional codes
- Experimental Evaluation
  - Methodology
  - Results
- Related Work
- **Conclusions**

CONCLUSIONS

- Optimistic barriers with HTM to hide latency and load imbalance
- Escape actions are used to allow communication between threads without causing aborts
- API is provided
- Improves performance in barrier-intensive applications
  - Number of threads high
  - Multi-socket configuration
- SB overload doesn't seem to penalize performance