Splash-4: A Modern Benchmark Suite with Lock-Free Constructs

Eduardo José Gómez-Hernández[1] Juan M. Cebrian[1] Stefanos Kaxiras[2] Alberto Ros[1]

[1]*University of Murcia, Murcia, Spain*
[2]*Uppsala University, Uppsala, Sweden*

Abstract

The cornerstone for the performance evaluation of computer systems is the benchmark suite. Among the many benchmark suites used in high-performance computing and multicore research, Splash-2 has been instrumental in advancing knowledge for both academia and industry. Published in 1995 and with over 5276 citations and counting, this benchmark suite is still in use to evaluate novel architectural proposals. Recently, the Splash-3 suite eliminates important performance bugs, data races, and improper synchronization that plagued Splash-2 benchmarks after the formal definition of the C memory model.

However, keeping up with architectural changes while maintaining the same workloads and algorithms (for comparative purposes) is a real challenge. Benchmark suites can misrepresent the performance characteristics of a computer system if they do not reflect the available features of the hardware and architects may end up overestimating the impact of proposed techniques or underestimating others.

In this work we introduce a revised version of Splash-3, designated Splash-4, that introduces modern programming techniques to improve scalability on contemporary hardware. We then characterize Splash-3 and Splash-4 in a state-of-the-art simulated architecture, Intel's Ice Lake with gem5-20 simulator, as well as a real contemporary hardware processor (AMD's EPYC 7002 series). Our evaluation shows that for a 64-thread execution Splash-4 reduces the normalized execution time by an average of 52% and 34% for AMD's EPYC and Intel's Ice Lake, respectively.

1. Introduction

It is well established that the standard method to conduct scientific experiments in computer science is benchmarking. Computer architects decide on a selection of benchmarks, which are a representation of applications of interest. These benchmarks are studied in detail to obtain a generalized conclusion that can be then applied to real computer systems. It is crucial that the selected workloads are general enough to cover a wide range of software applications, otherwise obtained results will only be of very limited validity. Examples of commonly used benchmark suites include:[1] Splash-2 [53] (5291), MiBench [28] (4546), Parsec [8] (4219), Rodinia [16] (3211), Linpack [19] (112), Parboil [50] (809), SHOC [18] (757) and PBBS [48] (227).

Splash-2 was the first major parallel benchmark suite. The main purpose of Splash-2 was to demonstrate shared-memory scalability. Indeed, under the evaluation techniques prevailing at the time (e.g., under a perfect memory system), this suite showed near-linear scalability in most of the benchmarks for up to 64 cores [53]. Its existence proved to be instrumental in the development of shared-memory multiprocessors. Subsequent updates, such as Splash-2X[2], fixed minor coding errors, updating it to the standards of the time [57]. This revision also introduced sizeable inputs that improved scalability on increasingly large systems and non-idealized simulators. However, Splash-2 shows unexpected behavior when used within contemporary compilers and hardware. Indeed, Splash-2 contains data races that introduce undefined behavior under the current C standard, leading to both logic and performance bugs.

A recent update, Splash-3 [46], exposes these data races and performance bugs. Their solution is to improve benchmark synchronization to resolve these issues. According to their own performance analysis done using GEMS [39], most benchmarks reach a speedup between 16× to 47× in a 64-core in-order multicore. Conversely, according to our own measurements using gem5-20 [37] with Intel Ice Lake-like out-of-order cores, the same Splash-3 benchmarks exhaust their scalability (i.e., show no further performance improvement) between 16 and 32 cores, with an average speedup of 2.3× for 64 cores. Finally, and also according to our own measurements on real hardware (64-Core AMD EPYC 7702P), most Splash-3 applications stop scaling when using between 4 and 16 cores, with an average speedup of 4.7× for 64 cores.

The main scalability problem for Splash-2 and Splash-3 is that benchmarks are crafted using outdated programming techniques. Atomic operations are now prevalent in many programs, due to their wide support in both programming languages, e.g., C [35], C++ [36], Java [52], and ISAs, e.g., x86 [34], IBM Power [33], ARMv8 [4] and RISC-V [32, 51]. Today, high-level language semantics directly invoke low-level hardware atomic operations, but this was not the case at the time when Splash-2 was created. Splash-4 updates

. Available at `https://github.com/OdnetninI/Splash-4`

. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134) and the Spanish Ministerio de Economía, Industria y Competitividad – Agencia Estatal de Investigación, under grant ERC2018-092826.

1. Citation count as of 11/07/2022.

2. For the rest of the paper we include Splash-2x when referring to Splash-2.

Splash-3 synchronization strategies to match contemporary programming and architectural features, making use of *lock-free constructs* whenever possible [27]. The new suite is then characterized on an Intel Ice Lake-like processor using gem5-20, as well as a contemporary real hardware processor, a 64-Core AMD EPYC 7702P. We analyze performance, scalability, synchronization overhead, and bottlenecks for the evaluated platform. Overall, Splash-4 improves the speedup over the corresponding Splash-3 applications by up to 9× on the 7702P processor, and up to 4× in the gem5-20 simulated Ice Lake processor.

2. Splash-4 optimizations

Splash-3 applications use a combination of locks (mutual exclusion), conditional variables (signal and wait), and barriers (wait for all) to synchronize between the different threads. These constructs introduce overheads in the application, especially when contention arises [12]. Previous works already noticed that the default input sizes of the Splash benchmark suite limit the scalability of some applications [7, 42]. Indeed, the computation between synchronization points is not substantially longer compared to the time spent in the synchronization and suggests using larger datasets to balance the load. However, using larger datasets has the effect of increasing the execution time, and that is a problem when using simulation infrastructures. Splash-4 takes a different approach, replacing high-overhead synchronization operations with lightweight alternatives. This translates into a performance improvement by expanding the architectural features that the benchmarks can exercise. More specifically, critical sections guarded by locks are replaced by lock-free constructs, while barriers are reinstated by a lightweight variant optimized for short waits.

The goal is to replace all possible critical sections with atomic operations or other lock-free constructs. The initial target are critical sections that modify a single shared variable. These critical sections can be easily replaced by an atomic operation. Then, critical sections that access a few shared variables are analyzed to be replaced by a lock-free equivalent.

2.1. Lock-Free and Atomic Operations

Modern ISAs typically provide a basic set of atomic operations that offer both atomicity and synchronization. These instructions can be used to negotiate mutual exclusion among threads and are currently supported by many programming languages. Examples include, "locked atomic operations" used in x86 and "atomic memory operations", or AMO instructions, used in IBM Power, ARMv8, and RISC-V. This basic set consists of atomic loads and stores, atomic read–modify–write (RMW) operations (such as fetch–and–add), and some atomic comparisons and exchange operations (such as compare–and–swap, CAS). Atomics instructions set the foundation to create more complex lock-free operations.

For example, CAS allows implementing codes that read a value, update it locally using non-atomic operations, and then write it to shared memory while checking that there is no conflict. However, and unlike other atomic operations, CAS can fail (if the old value does not match) and might need to retry multiple times. For this reason, it is common to use the CAS operation in a loop [24], as seen in Listing 1. For simplicity, this is also referred to as a "CAS construct".

Typical hardware RMW atomics are only available for integer types. CAS on the other hand is type-agnostic, so it can be used to implement RMW operations for more complex underlying types. Using the CAS construct, a FETCH_AND_ADD_DOUBLE construct can be implemented, which is a RMW fetch–and–add operation for double-precision floating-point numbers (Listing 2) [24]. Specifically, the old value is read into a register with an atomic load (LOAD) that enforces sequential consistency (default behaviour when not indicating the memory order). Then the new value is calculated from the read one. The CAExch instruction performs checks of the variable atomically, thus reloading the old value when performing the check, and returning a boolean indicating if the check succeeded. The CAExch instruction can be exchanged by the CAS instruction as shown in Listing 1 (the same principle can be applied to the rest of the listings on the benchmarks). This implementation with atomic accesses prevents "benign" data races that could affect correctness when building binaries for specific architectures as specified in [10].

```
/* CAExch */
var oldValue = LOAD(ptr);
var newValue;
do {
  newValue = new;
} while (!CAExch(ptr, oldValue, newValue))
    ;
```

```
/* CAS */
var readValue = LOAD(ptr);
var oldValue;
var newValue;
do {
  oldValue = readValue;
  newValue = new;
} while ((readValue = CAS(ptr, oldValue,
    newValue)) != oldValue);
```

Listing 1: While&CAS structure

```
double oldValue = LOAD(ptr);
double newValue;
do {
 newValue = oldValue + addition;
} while (!CAExch(ptr, oldValue, newValue))
    ;
```

Listing 2: FETCH_AND_ADD_DOUBLE operation

2.2. Sense-Reversing Centralized Barrier

For the Splash-3 benchmarks, the execution time between barriers is fairly short. To minimize the overhead of the barrier operation, we implement a sense-reversing barrier. This construct is optimized for short waiting times, except when oversubscribing threads (Listing 3) [40]. The atomic

store (STORE) enforces a sequentially-consistent ordering (default behavior when not indicating the memory order).

```
1 local_sense = !local_sense;
2 if (atomic_fetch_sub(&(count), 1) == 1) {
3     count = threads;
4   STORE(sense, local_sense);
5 } else {
6   do {} while (LOAD(sense) != local_sense)
      ;
7 }
```

Listing 3: Sense-reversing barrier

The standard glibc pthread barrier implementation uses a mutex lock to atomically update the thread count as threads arrive to the barrier [26]. This is replaced with a fetch–and–decrement atomic operation. By doing so, threads are actively looking for the barrier completion (spinning) instead of sleeping. Therefore, overall time spent in the barrier is reduced, since waking a sleeping thread is a slow operation.

2.3. Splitting Critical Sections

Splash-4 employs *lock-free constructs* that manage a single address and naturally correspond to critical sections that modify a single address. Splitting a larger critical section that modifies more than one address in smaller single-address critical sections would, therefore, enable the use of *lock-free constructs* in more cases. Unfortunately, splitting large critical sections, *is not possible in the general case*, because of implicit atomicity guarantees that may exist deep in the code regarding the group update of multiple variables—even when such variables are seemingly independent.

However, we found out that for many Splash-3 critical sections, (atomic) updates of independent variables are clumped together in larger critical sections for no apparent reason. In other words, *group* atomicity is not required and neither is assumed anywhere in the code. They surmise that the original Splash-2 clustered "atomically-independent" variables in the same critical sections to amortize the high cost of lock and unlock operations.

3. Per Application Study

For the sake of space we do not show all code changes in this section. The full benchmark code is however publicly available [3].

There are multiple synchronization primitives used in Splash-3 applications, but we are focusing on mutexes and barriers. Therefore, in this work we omit everything related to conditional variables, signal/wait, and broadcasts. Most applications have a single critical section that provides a sequential and unique ID to each thread. Such a critical section can be trivially replaced with an atomic fetch–and–add operation, and should not affect the performance of the application in any case.

To increase the number of critical sections that can be changed while maintaining the correctness of the application, we define the critical sections that execute between two barriers as *belonging to the same barrier group*. Within barrier groups, the behavior with a lock-free construct must be equivalent to the original lock-unlock structure, but accesses in other barrier groups should not be considered as concurrent. To replace a critical section with a lock-free structure, the shared variable modified by the critical section should not conflict with any other critical section that is not replaced with a compatible lock-free structure *inside the same barrier group*. With few exceptions, we are not changing a critical section belonging to a barrier group if *all* other conflicting critical sections of the same barrier group cannot be changed.

3. https://github.com/OdnetninI/Splash-4

3.1. Barnes

Barnes is a three-dimensional n-body simulation. It contains 11 critical sections that can be grouped into three *barrier groups*. Unfortunately, no critical section can be easily replaced with atomic operations.

3.2. Cholesky

Cholesky is a benchmark that performs a blocked sparse Cholesky factorization. It contains 6 critical sections that can be grouped into 2 *barrier groups*. Almost all of these are dedicated to manual memory management for object allocation. A critical section (Listing 4) can be replaced with an equivalent CAExch operation. Once a thread obtains a free block, it is no longer changed, maintaining the correctness of the code.

```
1 /* Lock */
2 LOCK(mem_pool[home].memoryLock)
3 result = mem_pool[home].freeBlock[bucket];
4 if (result)
5   mem_pool[home].freeBlock[bucket] =
      NEXTFREE(result);
6 UNLOCK(mem_pool[home].memoryLock)
```

```
1 /* Lock-free */
2 result = LOAD(mem_pool[home].freeBlock[
      bucket]);
3 do {
4   if (!result) break;
5 } while (!CAExch(mem_pool[home].freeBlock[
      bucket], result, NEXTFREE(result)));
```

Listing 4: malloc.c.in 138

3.3. FMM

FMM is a two-dimensional n-body simulation and contains 51 critical sections that can be grouped into eight *barrier groups*. The majority of these are used to access and modify the boxes that FMM uses to split the simulation space. Due to the nature of the algorithm, in most cases the critical section covers a single store or load, so it is possible to simply remove the critical section and replace it with the equivalent atomic operation. There is one exception to this: inserting a new box into the grid, which needs to be replaced with a CAExch operation to ensure that the grid remains consistent.

3.4. Radiosity

Radiosity performs a light distribution equilibrium computation. It contains 43 critical sections that can be grouped into three *barrier groups*. The majority of these critical sections are too large to be replaced with a single atomic operation and too complex to find a lock-free equivalent. A suitable section implements a custom barrier that allows for work stealing (Listings 5, 6, and 7). To maintain correctness, all these critical sections have to be changed all together or not at all. Listings 5 and 6 can be replaced with a CAExch and an atomic decrement operation respectively, while Listing 7 can be replaced with an atomic load operation, as the comparison operation does not need to be part of the critical section.

```
/* Lock */
LOCK(global->pbar_lock);
// Reset the barrier counter if not
    initialized
if( global->pbar_count >= n_processors )
  global->pbar_count = 0 ;

// Increment the counter
global->pbar_count++ ;

// barrier spin-wait loop
long bar_done = !(global->pbar_count <
    n_processors);
UNLOCK(global->pbar_lock);
```

```
/* Lock-free */
long expected = LOAD(global->pbar_count);
long result;
do {
  if( expected >= n_processors ) result =
    1;
  else result = expected + 1;
} while(!CAExch(global->pbar_count,
    expected, result));
long bar_done = !(result < n_processors);
```

Listing 5: taskman.c.in 108

```
/* Lock */
LOCK(global->pbar_lock);
global->pbar_count-- ;
UNLOCK(global->pbar_lock);
```

```
/* Lock-free */
FETCH_AND_SUB(global->pbar_count, 1);
```

Listing 6: taskman.c.in 134

```
/* Lock */
LOCK(global->pbar_lock);
bar_done = !(global->pbar_count <
    n_processors);
UNLOCK(global->pbar_lock);
```

```
/* Lock-free */
bar_done = !(LOAD(global->pbar_count) <
    n_processors);
```

Listing 7: taskman.c.in 140

Finally, there is another special critical section that controls how the work is distributed. This critical section loads and checks two different shared variables. In this case, two chained CAExch operations are required to ensure that both of these values remain consistent (Listing 8).

```
/* Lock */
LOCK(tq->q_lock);
if( tq->n_tasks > 0) {
  if ( tq->top ) {
    task_found = 1;
  }
  UNLOCK(tq->q_lock);
  break ;
}
UNLOCK(tq->q_lock);
```

```
/* Lock-free */
int exit;
Task *expectedTop = LOAD(tq->top);
long expectedNTasks = LOAD(tq->n_tasks);
do {
  exit = 0;
  if (expectedNTasks <= 0) break;
  exit = 1;
  do {
    task_found = 0;
    if (!expectedTop) break;
      task_found = 1;
  } while (!CAExch(tq->top, expectedTop,
    expectedTop));
} while (!CAExch(tq->n_tasks,
    expectedNTasks, expectedNTasks));
if (exit) break;
```

Listing 8: taskman.c.in 533

3.5. Raytrace

Raytrace is a three-dimensional raytracing rendering application. It contains 11 critical sections that can be grouped into two *barrier groups*. Four of these critical sections are only used for assigning unique IDs to the rays and can be easily replaced with fetch–and–add operations. Two other critical sections manage the memory allocation and cannot be replaced without significantly modifying the algorithm. Finally, the last critical section manages work assignment and can be replaced with a CAExch operation (Listing 9).

3.6. Ocean

Ocean, both the contiguous and non-contiguous versions, are large-scale ocean movement study applications. They contain three critical sections that can be grouped into two *barrier groups*. Two of these critical sections are used to accumulate the `psibi` and `psiai` variables. Since atomic fetch–and–add operation with floating point variables are not typically supported, we rely on our custom FETCH_AND_ADD_DOUBLE construct, that uses a CAExch operation (Listings 10 and 11).

Finally, the last critical section gathers all computed errors and selects the largest one. This functionality is

```
/* Lock */
ALOCK(gm->wplock, pid)
wpentry = gm->workpool[pid][0];

if (!wpentry) {
  AULOCK(gm->wplock, pid)
  return (WPS_EMPTY);
}
gm->workpool[pid][0] = wpentry->next;
AULOCK(gm->wplock, pid)
```

```
/* Lock-free */
wpentry = LOAD(gm->workpool[pid][0]);
do {
  if (!wpentry) return WPS_EMPTY;
} while (!CAExch(gm->workpool[pid][0],
    wpentry, wpentry->next));
```

Listing 9: workpool.c.in 152

```
/* Lock */
LOCK(locks->psibilock)
global->psibi = global->psibi + psibipriv;
UNLOCK(locks->psibilock)
```

```
/* Lock-free */
FETCH_AND_ADD_DOUBLE(global->psibi,
    psibipriv);
```

Listing 10: slave1.c.in 508 & 344

```
/* Lock */
LOCK(locks->psiailock)
global->psiai = global->psiai + psiaipriv;
UNLOCK(locks->psiailock)
```

```
/* Lock-free */
FETCH_AND_ADD_DOUBLE(global->psiai,
    psiaipriv);
```

Listing 11: slave2.c.in 857 & 718

implemented on some ISAs by the *atomicMin* instruction. Since the x86 ISA does not implement *atomicMin* we offer an alternative implementation. As maximum and minimum establish a global order, *atomicMin* can be easily implemented as a *priority update* [47]. Our implementation uses a slight variation *priority update* to maintain code uniformity with the rest of Splash-4 (Listing 12).

```
/* Lock */
LOCK(locks->error_lock)
if (local_err > multi->err_multi) {
  multi->err_multi = local_err;
}
UNLOCK(locks->error_lock)
```

```
/* Lock-free */
double expected = LOAD(multi->err_multi);
do {
  if (local_err <= expected) break;
} while (!CAExch(multi->err_multi,
    expected, local_err));
```

Listing 12: multi.c.in 90

3.7. Volrend

Volrend is a three-dimensional rotating volume rendering application using ray casting. It contains 16 critical sections that can be grouped into three *barrier groups*. Many of these critical sections are used to establish unique IDs for each subdivision of the rendering process. The remaining critical sections manage the sampling size for each region of the volume to render. All these sections can be simply replaced by an equivalent atomic RMW (e.g. Listing 13).

```
/* Lock */
ALOCK(Global->QLock,local_node);
work = Global->Queue[local_node][0];
Global->Queue[local_node][0] += 1;
AULOCK(Global->QLock,local_node);
```

```
/* Lock-free */
work = FETCH_ADD(Global->Queue[local_node
    ][0], 1);
```

Listing 13: adaptive.c.in 182 & 199

3.8. Water

Water-Nsquared and Water-Spatial are force molecular simulator applications for water molecules. Water-Nsquared uses an $O(n^2)$ algorithm with a predictor and a corrector, while Water-Spatial establishes a 3D grid to distribute the molecules between threads with an $O(n)$ algorithm. Water-Nsquared contains seven critical sections that can be grouped in three *barrier groups* and Water-Spatial contains seven critical sections grouped in seven *barrier groups*.

3.8.1. Common. Both implementations share four critical sections that update the global inter/intra-molecular force, the kinetic energy, and the potential energy. However, as it happens in Ocean, these variables are floating point, and thus require the use of custom constructs (e.g., Listings 14).

```
/* Lock */
LOCK(gl->IntrafVirLock);
*VIR =  *VIR + LVIR; // LVIR/2.0 (Spatial
    Interf)
UNLOCK(gl->IntrafVirLock);
```

```
/* Lock-free */
FETCH_AND_ADD_DOUBLE(VIR, LVIR);
```

Listing 14: intraf.c.in 133 & interf.c.in 146 & intraf.c.in 170 & interf.c.in 196

On the other hand, the critical section that updates the potential energy of the system consists of three fetch–and–add operations. As discussed in Section 2.3, there are cases where a critical section can be split into multiple smaller critical sections while maintaining correctness (Listing 15).

3.8.2. Water-Nsquared Critical Sections. Water-Nsquared has three critical sections that are not shared with Water-Spatial. These critical sections are the main part of the $O(n^2)$ algorithm, where all the forces are computed using a shared structure. However, these critical sections can also be split into multiple smaller ones, using the FETCH_AND_ADD_DOUBLE construct (Listing 16).

Application	Barriers		Critical Sections						Conditionals					
			Mutex		C11		CAExch		Wait		Signal		Broad	
	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn
Splash-3														
Barnes	6	19	10	2140090	0	0	0	0	1	360	0	0	2	23539
Cholesky	4	6	8	95182	0	0	0	0	1	4588	1	20508	0	0
Fft	7	9	1	64	0	0	0	0	0	0	0	0	0	0
Fmm	13	36	38	488126	0	0	0	0	8	1467	1	6207	5	23282
Lu	5	69	1	64	0	0	0	0	0	0	0	0	0	0
Lu-NonContiguous	5	69	1	64	0	0	0	0	0	0	0	0	0	0
Ocean	20	902	4	13312	0	0	0	0	0	0	0	0	0	0
Ocean-NonContiguous	19	872	4	13312	0	0	0	0	0	0	0	0	0	0
Radiosity	5	12	48	3861123	0	0	0	0	0	0	0	0	0	0
Radix	7	17	1	64	0	0	0	0	0	0	0	0	0	0
Raytrace	3	3	8	355184	0	0	0	0	0	0	0	0	0	0
Volrend	15	146	12	311164	0	0	0	0	0	0	0	0	0	0
Water-Nsquared	9	22	8	68672	0	0	0	0	0	0	0	0	0	0
Water-Spatial	9	22	6	1217	0	0	0	0	0	0	0	0	0	0
Splash-4														
Barnes	6	19	9	2140056	1	64	0	0	1	352	0	0	2	23539
Cholesky	4	6	6	68979	1	64	1	26238	1	3911	1	20508	0	0
Fft	7	9	0	0	1	64	0	0	0	0	0	0	0	0
Fmm	13	36	26	442838	1	64	1	5	8	1485	1	6207	5	23282
Lu	5	69	0	0	1	64	0	0	0	0	0	0	0	0
Lu-NonContiguous	5	69	0	0	1	64	0	0	0	0	0	0	0	0
Ocean	20	902	0	0	1	64	3	13248	0	0	0	0	0	0
Ocean-NonContiguous	19	872	0	0	1	64	3	13248	0	0	0	0	0	0
Radiosity	5	12	36	3478298	3	50497	3	6394618	0	0	0	0	0	0
Radix	7	17	0	0	1	64	0	0	0	0	0	0	0	0
Raytrace	3	3	2	252498	5	92455	1	8816	0	0	0	0	0	0
Volrend	15	146	1	1536	8	245519	0	0	0	0	0	0	0	0
Water-Nsquared	9	22	0	0	1	64	15	608384	0	0	0	0	0	0
Water-Spatial	9	22	0	0	1	64	6	1280	0	0	0	0	0	0

TABLE 1: 64 cores, default entries, each execution may vary the numbers a bit, numbers obtained with our pin tool. St(atic) is the number of instances present in the code, while Dyn(amic) is the number of instances executed in runtime.

```
/* Lock */
LOCK(gl->PotengSumLock);
*POTA = *POTA + LPOTA;
*POTR = *POTR + LPOTR;
*PTRF = *PTRF + LPTRF;
UNLOCK(gl->PotengSumLock);
```

```
/* Lock-free */
FETCH_AND_ADD_DOUBLE(POTA, LPOTA);
FETCH_AND_ADD_DOUBLE(POTR, LPOTR);
FETCH_AND_ADD_DOUBLE(PTRF, LPTRF);
```

Listing 15: poteng.c.in 159 & poteng.c.in 253

4. Benchmark Overview

In order to understand how Splash-4 code changes affect the applications at runtime, Table 1 shows all the synchronization primitives executed (instances of instructions) for Splash 3 and 4 when running 64 threads. In addition, we also show static (critical sections in the code) breakdown, since these are the ones with major code changes (barriers are simply replaced by a sense-reversing version). Critical sections are classified into three categories, depending on how they are implemented: mutex lock, C11 atomics, or Lock-Free constructs (CAExch). We can see that, overall, total synchronization primitive calls remain similar between the two suites, but, as we will see later, the total time

```
/* Lock */
ALOCK(gl->MolLock, mol % MAXLCKS);
for ( dir = XDIR; dir  <= ZDIR; dir++) {
  temp_p = VAR[mol].F[DEST][dir];
  temp_p[H1] += PFORCES[ProcID][mol][dir][
    H1];
  temp_p[O]  += PFORCES[ProcID][mol][dir][
    O];
  temp_p[H2] += PFORCES[ProcID][mol][dir][
    H2];
}
AULOCK(gl->MolLock, mol % MAXLCKS);
```

```
/* Lock-free */
for ( dir = XDIR; dir  <= ZDIR; dir++) {
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][
    dir][H1]), PFORCES[ProcID][mol][dir][H1
    ]);
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][
    dir][O]), PFORCES[ProcID][mol][dir][O])
    ;
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][
    dir][H2]), PFORCES[ProcID][mol][dir][H2
    ]);
}
```

Listing 16: interf.c.in 156 & interf.c.in 167 & interf.c.in 179

spent waiting in each synchronization primitive call drops drastically for Splash-4.

5. Methodology

The main purpose of Splash-4 is to serve as an evaluation tool for novel architectural proposals. This is why the present paper performs a characterization in both simulated and real hardware. However, the objective is not to measure the accuracy of the simulator against the real hardware, but to measure how effective the changes introduced in Splash-4 are.

5.1. Evaluation Environment

The selected hardware is AMD Epyc 7702P CPU [2] with 64 cores @ 2GHz, 32KB L1-D and L1-I, 512KB L2, and 16MB L3 caches. Hyper-threading is enabled, but we only run one thread on each physical core. It runs Ubuntu 18.04 with Linux kernel 5.4.0. The selected simulator is gem5-20 [37] running on full-system mode. We mimic an Intel's Ice Lake processor [1] running at 2GHz. The simulated system runs Ubuntu 16.04 with Linux kernel 4.9.4. The processor parameters are shown in Table 2. We use Ruby and Garnet [3] to model memory hierarchy. The execution and issue latency is modeled as measured on real hardware by Fog [23]. Each application is run ten times, and then a trimmed mean of 30% is computed. Measurements account for the region of interest (ROI), that is, the parallel region, after initialization and before screen output. Prints in this code section have been removed. In addition, gem5-20 measurements reset stats within the ROI after a warm-up period, as suggested by the original Splash-2 developers, to minimize variability between runs.

Processor	Ice Lake	EPYC 7702P
Fetch width	5	32 Bytes
Decode width	5	4
Rename width	5	6
Issue width	10	6
Commit width	10	8
Instruction queue	160	-
Reorder buffer	352	224
Load queue	128	44
Store queue	72	48
Integer Registers	180	180
Float Registers	180	160
Load Units	2	2
Store Units	1	3
Integer ALUs	1	4
Combined FP/Int ALUs	3	-
Memory Subsystem (per core)		
L1-I	32K 8w	32K 8w
L1-D	48K 12w	32K 8w
L2	512K 8w	512K 8w
Shared L3	2M 16w	4M
Directory	32708 sets 16 ways	-
Memory latency	80ns	N/A
Network		
Topology	Crossbar	Inf. Fabric 2

TABLE 2: gem5 System Configuration

5.2. Application Inputs

The inputs used in this paper are shown in Table 3 along with the memory footprint for 1 and 64 cores. These inputs, commonly known as simsmall, are the same for both platforms (real hardware and simulator). Valgrind is used to measure the memory footprint.

Application	**Input**	**Memory Footprint (1/64 cores)**
Barnes	< inputs/n16384-p#	10MB/10MB
Cholesky	-p# < inputs/tk15.O	16MB/40MB
FFT	-p# -m16	6MB/8MB
FMM	< inputs/input.#.16384	12MB/42MB
LU	-p# -n512	4MB/5MB
LU-NonContiguous	-p# -n512	4MB/5MB
Ocean	-p# -n258	17MB/20MB
Ocean-NonContiguous	-p# -n258	46MB/47MB
Radiosity	-p # -ae 5000 -bf 0.1 -en 0.05 -room -batch	219MB/219MB
Radix	-p# -n1048576	19MB/25MB
Raytrace	-p# -m64 inputs/car.env	58MB/59MB
Volrend	# inputs/head 8	32MB/33MB
Water-Nsquared	< inputs/n512-p#	3MB/8MB
Water-Spatial	< inputs/n512-p#	3MB/4MB

TABLE 3: Application Inputs (# $\rightarrow$ Number of cores)

6. Evaluation

This section shows the performance characterization of the Splash-4 benchmark suite. We also perform a scalability analysis, analyzing how much time is spent in synchronization primitives. Finally, we show a breakdown of core active running time, based on performance counter information, to pinpoint possible bottlenecks.

6.1. Performance Effects of Synchronization Overheads

Our evaluation starts by showing how the different code upgrades from Splash-4 affect application performance. Fig. 1 shows the individual effects of upgrading to sense-reversing barriers (labeled *Barrier*), using atomic-based lock-free operations/constructs (labeled *Atomics*) and a combination of both (labeled *Splash-4*) on a 64-thread execution in the AMD 7702P processor. Barrier upgrades reduce execution time by 40% on average. Atomic constructs in isolation reduce execution time by 11% on average, although the *Atomic* implementation performs marginally worse than pthreads under low contention. In general, the impact of the optimizations on execution time depends on how heavily each application relies on locks/barriers and their synchronization overhead (Fig. 2). For example, FFT only has one lock that protects the thread id (Table 1), and is only executed once and without contention, so the impact of *Atomic* optimization in isolation is minimal. The combination of both techniques provides a significant 52% reduction in execution time. Execution time results show the benefits of Splash-4, but, in order to be more thorough, the next step is to break down the synchronization overheads of both benchmark suites.

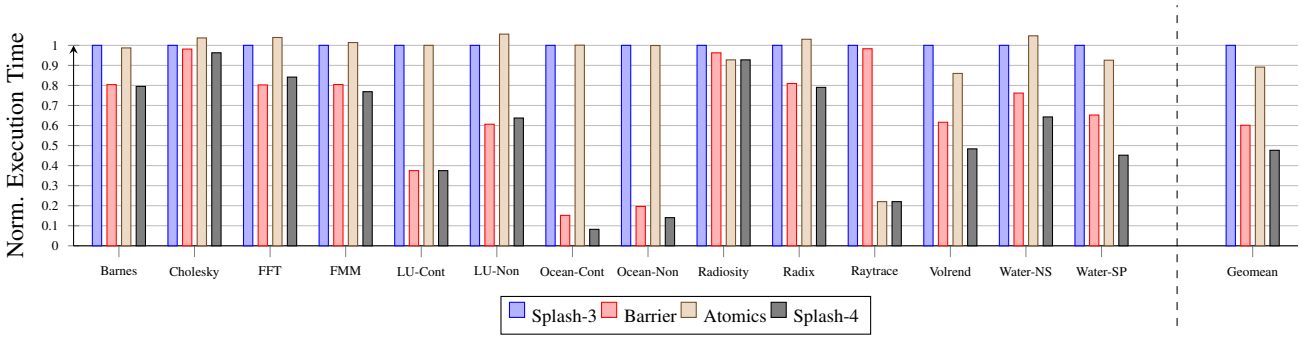

Figure 1: Execution time when upgrading Splash-3 *barriers*, *atomics* and both -*Splash-4*- (64-threads on AMD Epyc 7702P)

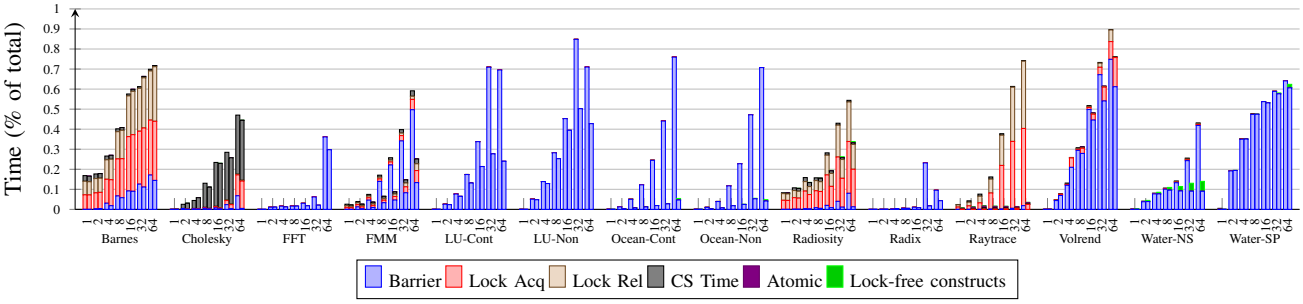

Figure 2: Synchronization overhead breakdown. Each thread count includes two bars, one for Splash-3 and one for Splash-4

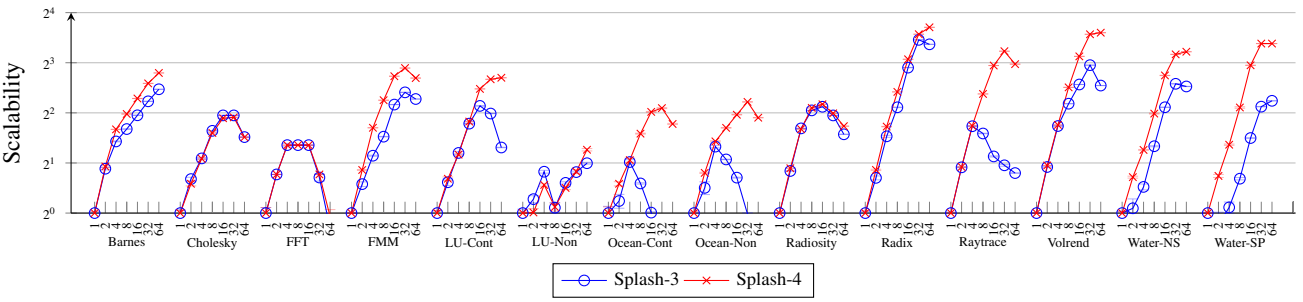

Figure 3: Splash-3 vs Splash-4 Scalability on AMD Epyc 7702P

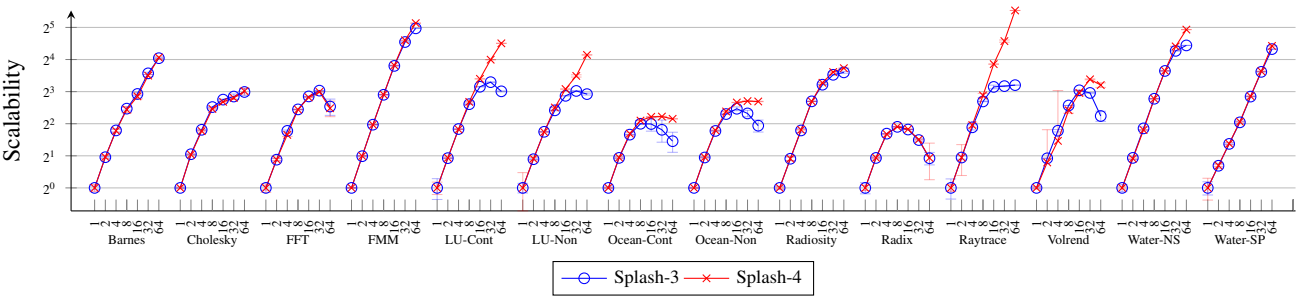

Figure 4: Splash-3 vs Splash-4 Scalability in simulated Intel Ice Lake (gem5-20)

To do so, we modified the application codes to include dummy instructions that will mark the beginning and end of the different synchronization primitives. These instructions are captured in gem5-20 at the commit stage so that only synchronization primitives from the correct execution path are considered. Fig. 2 shows two bars per thread count, that represent Splash-3 and Splash-4 implementations, respectively. This Figure shows a huge jump in synchronization overhead for FFT 64 cores, mildly contained by the sense-reversing barriers of Splash-4. Although we see that the cycles stalled waiting for L1 misses almost doubles when going from 32-threads to 64, meaning the level of memory-boundness increases as core count increases, most likely due to conflicts in L3 (since L3 size is fixed regardless of core-count). None of the analyzed performance counter information suggests any specific issue with the execution cores. We can also see how barrier waiting time drops drastically for LU, Ocean and Water-NS. Regarding lock overheads, Radiosity and Raytrace are the applications that benefit most from the code upgrades. Please note that this breakdown corresponds to an Ice Lake-like simulated processor (gem5-20), so there will be absolute value differences compared with Fig. 1.

6.2. Scalability

Fig. 3 and Fig. 4 show a scalability analysis on both real hardware and simulator, respectively, to study how the reduction of synchronization overheads affects performance. Scalability is computed only considering execution time of the region of interest (ROI) for each number of threads normalized to the ROI time for the single-thread version. Scalability is shown on a logarithmic scale. Also, remember the goal of this paper is not to validate the simulation infrastructure. Splash-4 either improves or preserves scalability for both platforms, that is, Splash-4 changes are not counterproductive on the measured systems and inputs. Raytrace, an application with high atomic contention, stopped scaling at four threads on Splash-3. With the code upgrades, it can now scale up to 32 threads on real hardware and 64 on simulation. LU scalability jumps from 16 to 64 threads on both platforms. LU is an application synchronized with barriers but not compute-intensive. This means that after 8 threads, the cost of synchronization with the default barriers is quite expensive compared with the computations performed between barriers. Ocean scales up to 32 threads on real hardware, but it does not in a simulation, despite the huge reduction in barrier time. We will explain Ocean behavior with performance counter information in the next Section. Differences in Radix between real hardware and simulation are mainly due to the reset stats location suggested by Splash-2 authors. There is a memory allocation just before the reset stat emplacement. This parallel memory allocation is the main reason why Radix scales better, and also can be seen in the simulator when stats are not reset after the allocation. The rest of the applications experience small performance improvements.

6.3. Execution Breakdown and Bottlenecks

We have seen how Splash-4 outperforms Splash-3 in terms of scalability, however, for some of the applications, the reduction on synchronization overhead does not proportionally translate into performance benefits. When synchronization overheads are removed, cores now spend execution time either waiting for resources or memory, preventing further scalability of the applications. This means that synchronization overheads were hiding other performance limiting factors. To elucidate why this is happening, we perform a detailed analysis of performance counter information during the active core time (performance stats in gem5-20). We base our analysis on Intel's Top-Down methodology [55]. This model is described by the authors as "a practical method to quickly identify true bottlenecks in out-of-order processors". For each benchmark, we show a breakdown Fig. 5-left showing the retiring, frontend-bound, bad-speculation, and backend-bound (core and memory). Retiring accounts for μops finishing normally. Bad-speculation represents those μops squashed due to a branch misprediction. Frontend-bound represents the ratio of μops stalled in fetch/decode, while backend-bound covers μops that cannot rename/issue/execute due to resource unavailability. Backend-bound μops are categorized into CPU or memory bound, depending on the resources causing these stalls. Top-Down model code changes for gem5-20 are based on those provided by authors in [13]. Please note that the Top-Down normalization is based on the clock cycles and instruction count of each specific implementation. Therefore, we can see cases like FFT, LU, Volrend and Water-SP where *retiring* part of the model rises compared to Splash-3. This is mostly related to the increase on the number of committed instructions by around 500%, 100%, 130% and 150%, respectively, due to active waiting when running on 64 threads. However, the *retiring* fraction does not increase as much as the committed instructions, and given that synchronization time is being reduced, it can only mean that there are more overall *Backend-bound* cycles for Splash-4 than Splash-3. The model shows how applications like FFT, Cholesky, LU, Ocean and Radix increase the pressure on the backend of the processor as the core count increases.

Fig. 5-right shows a breakdown of core-bound active cycles. Resource-related stalls are broken down into reorder-buffer (ROB), instruction window -aka reservation stations- (IQ), load queue/buffer (LQ), store queue/buffer (SQ), registers (REG), and data dependencies (Data). There are several limitations to this Figure regarding how stats are computed in gem5-20. Stall stats are accounted in a case statement in the same order as described above. This means that if an application stalls due to ROB resources, we do now know if it would stall due to any of the other resources, because only the first resource is considered. However, if an application stalls due to a lack of REGs, we know it has no problem with ROB, IQ, LQ, or SQ since checks for those resources are performed before REGs. In addition, data hazard stalls can appear in parallel with resource stalls, so they overlap, but we cannot detect that to draw the Figures. Therefore, this Figure

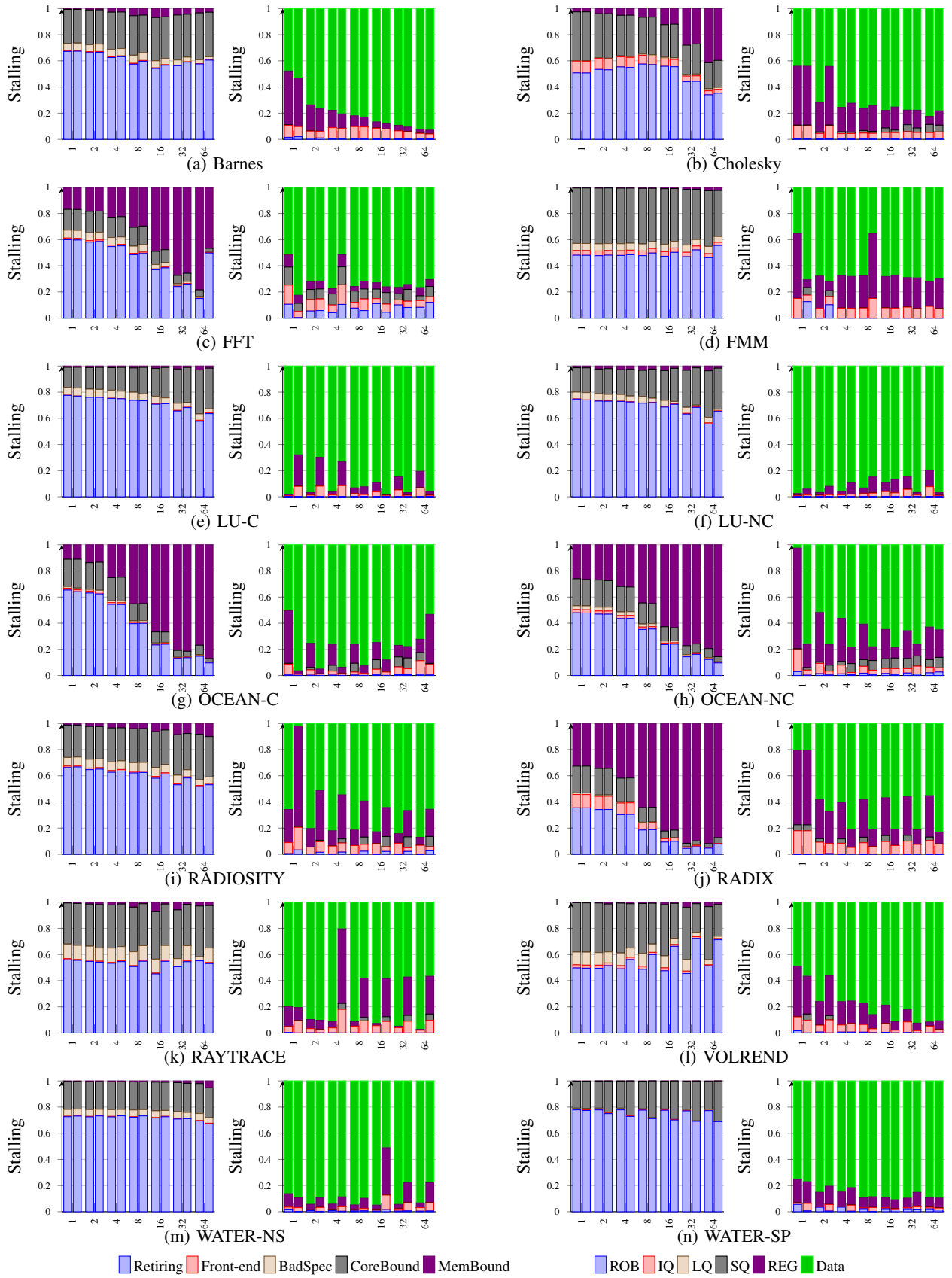

Figure 5: Per-application Top-Down model breakdown (left) and Core-bound resource stall breakdown (right)

must be used to get an overall idea of the core-bound stalls, but it lacks the precision to be part of the official Top-Down model. There are several conclusions we can get from this Figure. First, register-related stalls are abundant for many of the applications. Second, there are applications like Ocean and Cholesky that increase the pressure in the store buffer, preventing further scalability as core count increases, even with the simulated 72-entry store buffer. Our guess is that in Splash 3 cores had time to drain the store buffer between barrier waits, but since Splash 4 drastically reduces this time, cores stall more often. This behavior is not perceptible in the evaluated real AMD processor, probably because of a better memory subsystem.

7. Related Work

The main characterization papers of the Splash benchmark suite were performed by Woo and Bienia [7, 53]. However, they were performed in over-idealized simulation environments or outdated hardware, mostly focusing on metrics such as runtime and cache behavior. Other works focus on analyzing the crucial communication and sharing behavior of SPLASH and PARSEC applications [6, 38]. A better way to characterize the applications, at least from a hardware engineering perspective, is to provide performance counter information about resource utilization and possible bottlenecks.

Intel's top-down model has been used to analyze multiple benchmark suites, including SPEC2017 [30, 43], Cloudsuite [56], NAS [5, 41], BigDataBench [25]. SPEC2017 and BigDataBench show high variability in resource requirements that does not appear when analyzing other metrics like IPC or runtime. Cloudsuite analysis showed that BDA workloads suffer from overheads related to managing the data rather than accessing the data. NAS analysis revealed that core counters can be used as indicators for high utilization, whereas un-core counters are.

Other benchmark suites have also been characterized in the past. For example, authors in [29] characterize the SPEC2017 benchmark suite. However, they use an oversimplified memory system and simulation infrastructure, yet offer wide coverage since it does not focus on a single CPU or memory implementation. Authors include a phase analysis based on misses per kilo-instruction. Other techniques for finding critical code sections to focus optimization efforts have also been proposed [17], including techniques that create models to identify scaling bottlenecks of multi-threaded applications which are based on linear regression [20]. Authors in [21, 22, 31] propose the use of *cycle/speedup stacks* to estimate how scalability bottleneck limit the scalability of a multi-threaded program. It thereby quantifies how much its scalability can be improved by eliminating a given bottleneck.

Regarding scalability analysis, most articles focus on the PARSEC benchmark suite, but their conclusions can be extrapolated to other benchmark suites. In particular, those results reveal that half of the benchmarks show the same scalability for either ROI or full application runs [49]. They also found that the runtime scalability of the simulation inputs differs significantly from that of the native input sets. This is mainly due to the synchronization overheads. Similar conclusions are shown in [14]. The easiest way to deal with this problem is by increasing the input size [9]. However, this is a big issue for simulation environments, since the simulation of big inputs is impractical in terms of simulation time, especially in parallel simulations, where multi-checkpointing is harder to achieve (e.g., [44, 45, 54]). Other papers analyze techniques to improve scalability, like the use of task-level parallelism (i.e., task flow) [11]. This strategy allows the algorithm to be decoupled from the data distribution and the underlying hardware since the algorithm is entirely expressed as the flow of data. There are also tools for extrapolating the scalability of in-memory applications [15].

8. Conclusions

Splash-4 is the latest revision of the Splash benchmark suite, focused on modernizing its synchronization operations, therefore improving the scalability of the applications. This work presents Splash-4 and performs a detailed performance analysis comparing with Splash-3. We base our analysis both on real hardware and a simulated environment using gem5-20. We also use Intel's Top-Down model to detect bottlenecks on the simulated cores.

Our study shows a significant improvement in the scalability of Splash-4, going from 4-16 cores to 16-32 cores on most applications. Execution time is reduced on both the simulated environment (34%) and real hardware (52%). It is also important to note that the code upgrades performed in Splash-4, barely had any negative performance effects in our evaluation, with the exception of a minor slowdown for 16-cores, The algorithms remain the same, therefore the revised applications with upgraded synchronization primitives maintain all the computational patterns, but with the advantage of exploiting modern synchronization hardware features.

Performance counters revealed important information about scalability limitations. For certain applications, such as Ocean and Radix or FFT, there is increasing core backend pressure when synchronization overhead is reduced. Having less synchronization wait time means that cores have less time to empty their store buffers from committed stores, increasing store buffer-related stall time. On the other hand, resource stalls act as a synchronization mechanism, reducing barrier synchronization overhead.

In summary, the Splash benchmark suite is still a cornerstone in computer architecture research. However, it should be updated to be able to exploit modern hardware features. Splash-4 achieves this, by introducing low-cost synchronization mechanisms. This removes some of the synchronization overheads and adds additional pressure to the cores, mostly at the backend, leaving a door open for researchers to further improve their designs.

Appendix

1. Abstract

This artifact reproduces the results shown in Fig. 1 and Fig. 3, showing the step-by-step performance improvements from Splash-3 to Splash-4 in an AMD EPYC 7702P. The artifact includes the intermediate and final Splash-4 code and scripts to reproduce automatically the results.

2. Artifact check-list (meta-information)

- **Program**: Splash-3, Splash-3-Atomics, Splash-3-Barriers, and Splash-4
- **Compilation**: GNU GCC 7.5.0, GNU M4 1.4.18
- **Data set**: simsmall (included in the artifact)
- **Run-time environment**: GNU/Linux x86_64
- **Hardware**: AMD EPYC 7702P
- **Metrics**: Execution time, performance scalability
- **Output**: PDF files for Figures Fig. 1 and Fig. 3 (alternatively text files with the same results can be obtained)
- **How much disk space is required (approximately)?**: Less than 1 GB, less than 10GB if using docker
- **How much time is needed to prepare workflow (approximately)?**: One (1) hour
- **How much time is needed to complete experiments (approximately)?**: Four (4) hours
- **Publicly available?**: yes
- **Code licenses (if publicly available)?**: Original Splash, Splash-2, and Splash-3 Licenses. The rest of the code and scripts GPL-2.0
- **Archived (provide DOI)?**: 10.5281/zenodo.7086143

3. Description

3.1. How to access. Please download the code archive (tar-gz) from the artifact page `github.com/OdnetninI/Splash-4-Artifact` A Dockerfile is also provided in case is needed.

3.2. Hardware dependencies. Most of the benchmarks consume a few megabytes when running, so anything modern should be fine. However, our simulations were done on an AMD EPYC 7702P, therefore expect some differences when running on different hardware. To run all the experiments, 64 cores are expected (no SMT cores if possible).

3.3. Software dependencies. We provide a Dockerfile with all dependencies solved. However, it can be run in a native environment. These are the required dependencies (Ubuntu/Debian fashion) per step:

- Compiling and Running: gcc g++ m4 ivtools-dev make
- Analyze results: python3 python3-pandas python3-scipy
- Generate the graphs: texlive-latex-base texlive-fonts-recommended texlive-fonts-extra texlive-latex-extra
- Dockerfile (only for running inside docker): docker.io

4. Installation

When running native on the machine, just unpack the artifact package, and install the dependencies listed in the previous section. Then, go into the `splash` directory

When running on docker, install docker using your operating system package manager. Like on Ubuntu:

```
apt install docker.io
```

Then, unpack the artifact package and build the image (it will take a while):

```
docker build . -t splash
```

To start the container run:

```
docker run --name splash -it splash
```

5. Experiment workflow

First, make sure the current directory is the `splash` folder. The scripts to run are numbered from 0 to 9.

```
./0_compile.sh
```

Compile all the benchmarks. Some warnings will appear when compiling with gcc version 7.5.0, but applications should compile correctly. Then, execute the benchmarks using the following scripts (do not run them at the same time):

```
./1_run_splash_3.sh
./2_run_splash_3_atomics.sh
./3_run_splash_3_barriers.sh
./4_run_splash_4.sh
```

After the execution of the benchmarks, to discard the 30% of outliers and prepare the data for the next step, run:

```
./5.trimmed_mean.sh
```

The data used in Figures 1 and 3, can be view reading files `Splash3-Splash4.scala` and `Splash3-Atomics-Barriers-Splash4.measure`, after running:

```
./6_scala.sh
./8_cmp.sh
```

However, if you want to generate the graph, first you need to produce the `tex.data` files, running:

```
./7_scala_to_latex.sh
./9_cmp_to_latex.sh
```

Then go inside the `Figures` directory and run `make`. Two pdf files should be generated with the results (`Scalability.pdf` and `Compare.pdf`).

If everything was run inside the docker container, you can get the files outside the container using:

```
docker cp splash:/root/splash/Figures/
    Scalability.pdf .
docker cp splash:/root/splash/Figures/
    Compare.pdf .
```

The same applies to the results files if you want to examine them:

```
docker cp splash:/root/splash/Splash3-
    Splash4.scala .
docker cp splash:/root/splash/Splash3-
    Atomics-Barriers-Splash4.measure .
```

6. Evaluation and expected results

The obtained results should reflect Fig. 1 and Fig. 3. Some discrepancy is normal as times vary from run to run. However, in the general case, Splash-4 should scale better than Splash-3 providing the same or better performance.

7. Experiment customization

All the experiment workflow expects to get 1, 2, 4, 8, 16, 32, and 64 core results. However, by default, we bind the threads to the cores. If the machine used has less than 64 logical cores, the benchmarks will not run correctly.

To disable thread to core binding, edit the `Makefile.config` file inside the `Splash-3`, `Splash-3-Atomics`, `Splash-3-Barriers` and `Splash-4` directories to remove the `-D BIND_CORES` on lines 10 and 11.

For certain, very specific, configurations, the option `-D BIND_THREADS` can be used instead. However, we discourage its usage. It changes the binding order, so instead of binding thread 0 to core 0 and thread 1 to core 1, will take into account some SMT configurations and bind thread 0 to core 0 but thread 1 to core 2. However, this is uncommon on current Linux systems.

8. Notes

Due to a known bug since Splash-2, FMM can livelock. When running if no progress is made, kill the FMM process to continue with the rest of the benchmarks. As every benchmark is run multiple times, one or two missing executions should not be relevant. Docker commands usually require super-user permissions.

References

[1] "Sunny cove - microarchitectures - intel." [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

[2] "Zen 2 - microarchitectures - amd." [Online]. Available: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 33–42.

[4] ARM, "Arm synchronization primitives development article," 2022. [Online]. Available: https://developer.arm.com/documentation/dht0008/a/arm-synchronization-primitives/exclusive-accesses

[5] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[6] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of splash-2 and parsec," in *2009 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2009, pp. 86–97.

[7] C. Bienia, S. Kumar, and Kai Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *2008 IEEE International Symposium on Workload Characterization*, 2008, pp. 47–56.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[9] C. Bienia and K. Li, "Scaling of the parsec benchmark inputs," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, p. 561–562.

[10] H.-J. Boehm, "How to miscompile programs with "benign" data races," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, ser. HotPar'11. USA: USENIX Association, 2011, p. 3.

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[12] P. D. Bryan, J. G. Beu, T. M. Conte, P. Faraboschi, and D. Ortega, "Our many-core benchmarks do not use that many cores," *System*, vol. 6, p. 8, 2009.

[13] J. M. Cebrian, A. Barredo, H. Caminal, M. Moretó, M. Casas, and M. Valero, "Semi-automatic validation of cycle-accurate simulation infrastructures: The case for gem5-x86," *Future Generation Computer Systems*, Jun. 2020.

[14] J. M. Cebrian, L. Natvig, and M. Jahre, "Scalability analysis of avx-512 extensions," *The Journal of Supercomputing*, vol. 76, no. 3, pp. 2082–2097, 2020.

[15] G. Chatzopoulos, A. Dragojević, and R. Guerraoui, "Estima: Extrapolating scalability of in-memory applications," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 2, pp. 1–28, 2017.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Int'l Symp. on Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.

[17] C. Curtsinger and E. D. Berger, "Coz: Finding code that counts with causal profiling," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 184–197.

[18] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.

[19] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[20] S. Dutta, S. Manakkadu, and D. Kagaris, "Classifying performance bottlenecks in multi-threaded applications," in *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, 2014, pp. 341–345.

[21] D. Eklov, N. Nikoleris, and E. Hagersten, "A software based profiling method for obtaining speedup stacks on commodity multi-cores," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 148–157.

[22] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2012, pp. 145–155.

[23] A. Fog, "Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns," 2018, Available at http://www.agner.org/optimize/instruction_tables.pdf.

[24] H. Gao and W. Hesselink, "A general lock-free algorithm using compare-and-swap," *Information and Computation*, vol. 205, no. 2, pp. 225–241, 2007.

[25] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, X. Wen, R. Ren, C. Zheng, X. He, H. Ye *et al.*, "Bigdatabench: A scalable and unified big data and ai benchmark suite," *arXiv preprint arXiv:1802.08254*, 2018.

[26] "The GNU C library (glibc)," https://www.gnu.org/software/libc/libc.html, 2022. [Online]. Available: https://www.gnu.org/software/libc/libc.html

[27] E. J. Gómez-Hernández, R. Shao, C. Sakalis, S. Kaxiras, and A. Ros, "Splash-4: Improving scalability with lock-free constructs," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, I. C. Society, Ed. Worldwide event: IEEE Computer Society, Mar. 2021, pp. 235–236.

[28] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[29] M. Hassan, C. H. Park, and D. Black-Schaffer, "A reusable characterization of the memory system behavior of spec2017 and spec2006," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 2, pp. 1–20, 2021.

[30] R. Hebbar SR and A. Milenković, "Spec cpu2017: Performance, event, and energy characterization on the core i7-8700k," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 111–118.

[31] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2011, pp. 38–49.

[32] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[33] "Power ISA Version 3.1," https://ibm.ent.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0, May 2020. [Online]. Available: https://ibm.ent.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0

[34] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, May 2020, no. 325462-072US.

[35] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, December 2011.

[36] ——, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012.

[37] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, M. D. S. Boris Shingarov, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[38] D. Marinov, D. Magdic, A. Milenkovic, J. Protic, I. Tartalja, and V. Milutinovic, "Scowl: a tool for characterization of parallel workload and its use on splash-2 application suite," in *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No. PR00728)*. IEEE, 2000, pp. 207–213.

[39] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[40] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[41] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel, "Detecting memory-boundedness with hardware performance counters," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 27–38.

[42] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 10–19, 2009.

[43] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a decade: Did spec cpu 2017 broaden the performance horizon?" in *24th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, Feb. 2018, pp. 271–282.

[44] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[45] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting phases in parallel applications on shared memory architectures," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 10–pp.

[46] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[47] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons, "Reducing contention through priority updates," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 152–163. [Online]. Available: https://doi.org/10.1145/2486159.2486189

[48] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 68–70. [Online]. Available: https://doi.org/10.1145/2312005.2312018

[49] G. Southern and J. Renau, "Analysis of parsec workload scalability," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 133–142.

[50] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, p. 27, 2012.

[51] A. Waterman and K. Asanovic, "The risc-v instruction set manual, volume i: Unprivileged isa document, version 20190608-baseratified," *RISC-V Foundation, Tech. Rep*, 2019.

[52] A. Williams, *C++ Concurrency in Action*. Manning Publications, 2019.

[53] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[54] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003, pp. 84–97.

[55] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014, pp. 35–44.

[56] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive analysis of the data analytics workload in cloudsuite," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 202–211.

[57] X. Zhan, Y. Bao, C. Bienia, and K. Li, "Parsec3.0: A multicore benchmark suite with network stacks and splash-2x," *SIGARCH Comput. Archit. News*, vol. 44, no. 5, pp. 1–16, Feb. 2017.