Splash-4: Una Benchmark Suite moderna con Constructos Lock-Free

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

Resumen— **La piedra angular para la evaluación del rendimiento de los sistemas informáticos son los *benchmark suites*. Entre los muchos *benchmark suites* utilizados en la computación de alto rendimiento y la investigación en procesadores multinúcleo, Splash-2 ha sido fundamental para avanzar en el conocimiento tanto en el ámbito académico como en la industria. Publicada en 1995 y con más de 5276 citas y contando, esta *benchmark suite* todavía se utiliza para evaluar nuevas propuestas. Recientemente, la suite Splash-3 elimina errores importantes de rendimiento, condiciones de carrera y sincronizaciones incorrectas, que afectaban a los benchmarks de Splash-2 tras la definición formal del modelo de memoria en C.**

Sin embargo, mantenerse al día con los cambios arquitecturales mientras se mantienen las mismas cargas de trabajo y algoritmos (con fines comparativos) es un verdadero desafío. Los *benchmark suites* pueden distorsionar las características de rendimiento de un sistema si no reflejan las características disponibles del hardware, y los arquitectos de computadores pueden terminar sobreestimando el impacto de las técnicas propuestas o subestimando otras.

En este trabajo, presentamos una versión revisada de Splash-3, designada como Splash-4, que introduce técnicas de programación modernas para mejorar la escalabilidad en hardware contemporáneo. Luego caracterizamos Splash-3 y Splash-4 en una arquitectura simulada de última generación, el simulador gem5-20 imitando un Intel Ice Lake, así como en un procesador de hardware contemporáneo real (serie AMD EPYC 7002). Nuestra evaluación muestra que para una ejecución de 64 hilos, Splash-4 reduce el tiempo de ejecución normalizado en un promedio de 52 % y 34 % para AMD EPYC y gem5, respectivamente.

Palabras clave— **Benchmarks, simulación, caracterización, sincronización, operaciones atómicas, optimización**

I. Introducción

ESTÁ bien establecido que el método estándar para llevar a cabo experimentos científicos en informática es el *benchmarking*. Los arquitectos de computadoras deciden sobre una selección de *benchmarks*, que son una representación de aplicaciones de interés. Estos *benchmarks* se estudian en detalle para obtener una conclusión generalizada que luego se puede aplicar a sistemas reales. Es crucial que las cargas de trabajo seleccionadas sean lo suficientemente generales como para cubrir una amplia gama de aplicaciones de software; de lo contrario, los resultados obtenidos solo tendrán una validez muy limitada. Ejemplos de *benchmarks* comúnmente utilizadas incluyen:[1] Splash-2 [1] (5291), MiBench [2] (4546), Parsec [3] (4219), Rodinia [4] (3211), Linpack [5] (112), Parboil [6] (809), SHOC [7] (757) y PBBS[8] (227).

Splash-2 fue la primera suite de *benchmarks* paralela importante. El propósito principal de Splash-2 era demostrar la escalabilidad de la memoria compartida. De hecho, bajo las técnicas de evaluación prevalecientes en ese momento (por ejemplo, bajo un sistema de memoria perfecto), esta suite mostraba escalabilidad casi lineal en la mayoría de los *benchmarks* para hasta 64 núcleos [1]. Su existencia resultó ser fundamental en el desarrollo de multiprocesadores de memoria compartida. Actualizaciones posteriores, como Splash-2X[2], corrigieron errores de codificación menores, actualizándolo a los estándares de la época [9]. Esta revisión también introdujo "entradas" sustanciales que mejoraron la escalabilidad en sistemas cada vez más grandes y simuladores no idealizados. Sin embargo, Splash-2 muestra un comportamiento inesperado cuando se utiliza con compiladores y hardware contemporáneos. De hecho, Splash-2 contiene condiciones de carrera que introducen un comportamiento indefinido según el estándar actual de C, lo que conduce a errores lógicos y de rendimiento.

Una actualización reciente, Splash-3[10], expone estas condiciones de carrera y errores de rendimiento. Su solución es mejorar la sincronización de los *benchmarks* para resolver estos problemas. Según su propio análisis de rendimiento realizado con GEMS[11], la mayoría de los *benchmarks* alcanzan una aceleración entre 16× y 47× en un multinúcleo in-order de 64 núcleos. Por el contrario, según nuestras propias mediciones utilizando gem5-20 [12] con núcleos fuera de orden similares a Intel Ice Lake, los mismos *benchmarks* de Splash-3 agotan su escalabilidad (es decir, no muestran una mejora adicional de rendimiento) entre 16 y 32 núcleos, con una aceleración promedio de 2.3× para 64 núcleos. Finalmente, y también según nuestras propias mediciones en hardware real (AMD EPYC 7702P de 64 núcleos), la mayoría de las aplicaciones de Splash-3 dejan de escalar cuando se utilizan entre 4 y 16 núcleos, con una aceleración promedio de 4.7× para 64 núcleos.

El problema principal de escalabilidad para Splash-2 y Splash-3 es que los *benchmarks* se crean utilizando técnicas de programación obsoletas. Las operaciones atómicas son ahora prevalentes en muchos programas, debido a su amplio soporte tanto en lenguajes de programación, como C [13], C++[14], Java[15], como en ISAs, como x86 [16], IBM Power [17], ARMv8 [18] y RISC-V [19], [20]. Hoy en día, la semántica de alto nivel invoca directamente operaciones atómicas de bajo nivel, pero esto no era así en el momento en que se creó Splash-2. Splash-4 actualiza las estrategias de sincronización de Splash-3 para adaptarse a las características de programación y arquitectónicas contemporáneas, haciendo uso de *constructos lock-free* siempre que sea posible [21]. La nueva suite se caracteriza luego en un procesador similar a Intel Ice Lake utilizando gem5-20, así como en un procesador de hardware real contemporáneo, un AMD EPYC 7702P de 64 núcleos. Analizamos el rendimiento, la escalabilidad, la sobrecarga de sincronización y los cuellos de botella para la plataforma evaluada. En general, Splash-4 mejora

[1]Dpto. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mail: {eduardojose.gomez,jcebrian,aros}@um.es

[2]Department of Information Technology, Uppsala University, Sweden, e-mail: stefanos.kaxiras@it.uu.se.

Disponible en https://github.com/OdnetninI/Splash-4

[1]Recuento de citas hasta el 11/07/2022.

[2]En el resto del documento incluimos Splash-2x al referirnos a Splash-2.

la aceleración sobre las aplicaciones correspondientes de Splash-3 en hasta 9× en el procesador 7702P y hasta 4× en el procesador simulado gem5-20 Ice Lake.

```
/* 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);
```

Listado 1 Constructo While&CAS

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

Listado 2 FETCH_AND_ADD_DOUBLE

II. Optimizaciones de Splash-4

Las aplicaciones de Splash-3 utilizan una combinación de bloqueos (exclusión mutua), variables condicionales (señal y espera) y barreras (espera para todos) para sincronizarse entre los diferentes hilos. Estos constructos introducen sobrecargas en la aplicación, especialmente cuando surge contención [22]. Trabajos anteriores ya notaron que los tamaños de entrada predeterminados de Splash limitan la escalabilidad de algunas aplicaciones [23], [24]. De hecho, la computación entre puntos de sincronización no es sustancialmente más larga en comparación con el tiempo dedicado a la sincronización y sugiere el uso de conjuntos de datos más grandes para equilibrar la carga. Sin embargo, el uso de conjuntos de datos más grandes tiene el efecto de aumentar el tiempo de ejecución, y eso es un problema al usar infraestructuras de simulación. Splash-4 adopta un enfoque diferente, reemplazando operaciones de sincronización de alto costo con alternativas livianas. Esto se traduce en una mejora de rendimiento al ampliar las características arquitectónicas que los *benchmarks* pueden ejercitar. Específicamente, las secciones críticas protegidas por bloqueos se reemplazan por constructos *lock-free*, mientras que las barreras se reemplazan por una variante liviana optimizada para esperas cortas.

El objetivo es reemplazar todas las secciones críticas posibles con operaciones atómicas u otros constructos *lock-free*. El objetivo inicial son las secciones críticas que modifican una única variable compartida. Estas secciones críticas pueden reemplazarse fácilmente por una operación atómica. Luego, se analizan las secciones críticas que acceden a algunas variables compartidas para ser reemplazadas por un equivalente *lock-free*.

A. Operaciones lock-free y atómicas

Las arquitecturas modernas generalmente proporcionan un conjunto básico de operaciones atómicas que ofrecen tanto atomicidad como sincronización. Estas instrucciones se pueden utilizar para negociar la exclusión mutua entre hilos y actualmente son compatibles con muchos lenguajes de programación. Ejemplos incluyen “operaciones atómicas bloqueadas” utilizadas en x86 y “operaciones atómicas de memoria”, o instrucciones AMO, utilizadas en IBM Power, ARMv8 y RISC-V. Este conjunto básico consta de cargas y almacenamientos atómicos, operaciones de lectura-modificación-escritura (RMW) atómicas (como fetch-and-add), y algunas operaciones atómicas de comparación e intercambio (como compare-and-swap, CAS). Las instrucciones atómicas sientan las bases para crear operaciones *lock-free* más complejas.

Por ejemplo, CAS permite implementar códigos que leen un valor, lo actualizan localmente mediante operaciones no atómicas y luego lo escriben en la memoria compartida mientras verifican que no haya conflicto. Sin embargo, a diferencia de otras operaciones atómicas, CAS puede fallar (si el valor antiguo no coincide) y puede ser necesario intentarlo varias veces. Por esta razón, es común usar la operación CAS en un bucle [25], como se ve en el Listado 1. Por simplicidad, esto también se conoce como un “constructo CAS”.

Las operaciones RMW atómicas de hardware típicamente solo están disponibles para tipos enteros. CAS, por otro lado, es agnóstico al tipo, por lo que se puede utilizar para implementar operaciones RMW para tipos subyacentes más complejos. Utilizando el constructo CAS, se puede implementar un constructo FETCH_AND_ADD_DOUBLE, que es una operación RMW fetch-and-add para números de punto flotante de doble precisión (Listado 2)[25]. Específicamente, el valor antiguo se lee en un registro con una carga atómica (`LOAD`) que impone consistencia secuencial (comportamiento predeterminado cuando no se indica el orden de memoria). Luego, el nuevo valor se calcula a partir del valor leído. La instrucción CAExch realiza comprobaciones de la variable atómicamente, volviendo a cargar el valor antiguo al realizar la comprobación y devolviendo un booleano que indica si la comprobación tuvo éxito. La instrucción CAExch se puede intercambiar por la instrucción CAS, como se muestra en el Listado1 (el mismo principio se puede aplicar al resto de los *benchmarks*). Esta implementación con accesos atómicos evita las “condiciones de carrera benignas” que podrían afectar la corrección al compilar binarios para arquitecturas específicas, según se especifica en [26].

B. Barrera centralizada con inversión de sentido

Para los *benchmarks* de Splash-3, el tiempo de ejecución entre barreras es bastante corto. Para minimizar la sobrecarga de la operación de barrera, implementamos una barrera con inversión de sentido (*sense-reversing barrier*). Este constructo está optimizado para tiempos de espera cortos, excepto cuando hay más hilos que recursos disponibles (Listado 3) [27]. La instrucción de almacenamiento atómico (STORE) impone un orden secuen-

cialmente consistente (comportamiento predeterminado cuando no se indica el orden de memoria).

```
local_sense = !local_sense;
if (atomic_fetch_sub(&(count), 1) == 1) {
    count = threads;
        STORE(sense, local_sense);
} else {
        do {} while (LOAD(sense) != local_sense)
            ;
}
```

Listado 3 Sense-reversing barrier

La implementación estándar de la barrera pthread en la biblioteca estándar de glibc utiliza un bloqueo de mutex para actualizar atómicamente el recuento de hilos a medida que los hilos llegan a la barrera [28]. Esto se reemplaza con una operación atómica de fetch-and-decrement. Al hacerlo, los hilos buscan activamente la finalización de la barrera (hacen *spinning*) en lugar de dormir. Por lo tanto, el tiempo total dedicado a la barrera se reduce, ya que despertar a un hilo dormido es una operación lenta.

C. División de secciones críticas

Splash-4 utiliza *constructos lock-free* que gestionan una única dirección y corresponden naturalmente a secciones críticas que modifican una sola dirección. Dividir una sección crítica más grande que modifica más de una dirección en secciones críticas más pequeñas de una sola dirección permitiría, por lo tanto, el uso de *constructos lock-free* en más casos. Desafortunadamente, dividir secciones críticas grandes *no es posible en el caso general*, debido a garantías implícitas de atomicidad que pueden existir profundamente en el código con respecto a la actualización grupal de múltiples variables, incluso cuando estas variables parecen ser independientes.

Sin embargo, descubrimos que, para muchas secciones críticas de Splash-3, las actualizaciones (atómicas) de variables independientes se agrupan en secciones críticas más grandes sin ninguna razón aparente. En otras palabras, la atomicidad de grupo no se requiere ni se asume en ninguna parte del código. Suponemos que Splash-2 original agrupó variables "atómicamente independientes" en las mismas secciones críticas para amortizar el alto costo de las operaciones de bloqueo y desbloqueo.

III. Estudio por Aplicación

Por cuestiones de espacio, no mostramos todos los cambios de código en esta sección. Sin embargo, el código completo de los *benchmarks* está disponible públicamente[3].

Hay múltiples primitivas de sincronización utilizadas en las aplicaciones de Splash-3, pero nos centramos en mutex y barreras. Por lo tanto, en este trabajo omitimos todo lo relacionado con variables condicionales, signal/-wait y broadcast. La mayoría de las aplicaciones tienen una sola sección crítica que proporciona un identificador secuencial y único a cada hilo. Esta sección crítica se puede reemplazar trivialmente con una operación atómica de fetch-and-add, y no debería afectar el rendimiento de la aplicación en ningún caso.

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

Para aumentar el número de secciones críticas que se pueden cambiar mientras se mantiene la corrección de la aplicación, definimos las secciones críticas que se ejecutan entre dos barreras como *pertenecientes al mismo barrier group*. Dentro de los *barrier groups*, el comportamiento con un constructo *lock-free* debe ser equivalente a la estructura original de bloqueo-desbloqueo, pero los accesos en otros *barrier groups* no deben considerarse como concurrentes. Para reemplazar una sección crítica con una estructura *lock-free*, la variable compartida modificada por la sección crítica no debe entrar en conflicto con ninguna otra sección crítica que no se haya reemplazado con una estructura *lock-free* compatible *dentro del mismo barrier group*. Con pocas excepciones, no estamos cambiando una sección crítica que pertenece a un *barrier group* si *todas* las demás secciones críticas en conflicto del mismo *barrier group* no se pueden cambiar.

A. Barnes

Barnes es una simulación tridimensional de n-cuerpos. Contiene 11 secciones críticas que se pueden agrupar en tres *grupos de barrera*. Desafortunadamente, ninguna sección crítica se puede reemplazar fácilmente con operaciones atómicas.

B. Cholesky

Cholesky es una prueba que realiza una factorización de Cholesky bloqueada y dispersa. Contiene 6 secciones críticas que se pueden agrupar en 2 *barrier groups*. Casi todas estas se dedican a la gestión manual de la memoria para la asignación de objetos. Una sección crítica (Listado 4) se puede reemplazar con una operación CAExch equivalente. Una vez que un hilo obtiene un bloque libre, ya no se modifica, manteniendo la corrección del código.

```
/* Lock */
LOCK(mem_pool[home].memoryLock)
result = mem_pool[home].freeBlock[bucket];
if (result)
  mem_pool[home].freeBlock[bucket] = NEXTFREE(
      result);
UNLOCK(mem_pool[home].memoryLock)
```

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

Listado 4 malloc.c.in 138

C. FMM

FMM es una simulación n-cuerpo bidimensional y contiene 51 secciones críticas que pueden agruparse en ocho *barrier groups*. La mayoría de estas se utilizan para acceder y modificar las `boxes` que FMM utiliza para dividir el espacio de simulación. Debido a la naturaleza del algoritmo, en la mayoría de los casos la sección crítica abarca una única operación de almacenamiento o lectura, por lo que es posible eliminar simplemente la sección crítica y reemplazarla por la operación atómica equivalente. Hay una excepción a esto: la inserción de una nue-

va box en la cuadrícula, que debe reemplazarse con una operación CAExch para garantizar que la cuadrícula permanezca consistente.

D. Radiosity

Radiosity realiza un cálculo de equilibrio en la distribución de la luz. Contiene 43 secciones críticas que pueden agruparse en tres *barrier groups*. La mayoría de estas secciones críticas son demasiado grandes para ser reemplazadas con una sola operación atómica y demasiado complejas para encontrar un equivalente *lock-free*. Una sección adecuada implementa una barrera personalizada que permite el robo de trabajo (Listados 5, 6 y 7). Para mantener la corrección, todas estas secciones críticas deben cambiarse todas juntas o ninguna. Los Listados 5 y 6 pueden reemplazarse con un CAExch y una operación de decremento atómico respectivamente, mientras que el Listado 7 puede reemplazarse con una operación de carga atómica, ya que la operación de comparación no necesita formar parte de la sección crítica.

```
/* 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);
```

Listado 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);
```

Listado 6 taskman.c.in 134

Finalmente, hay otra sección crítica especial que controla cómo se distribuye el trabajo. Esta sección crítica carga y verifica dos variables compartidas diferentes. En este caso, se requieren dos operaciones CAExch encadenadas para asegurar que ambos de estos valores permanezcan consistentes (Listado 8).

```
/* 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);
```

Listado 7 taskman.c.in 140

```
/* 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;
```

Listado 8 taskman.c.in 533

E. Raytrace

Raytrace es una aplicación de renderizado de trazado de rayos tridimensional. Contiene 11 secciones críticas que pueden agruparse en dos *barrier groups*. Cuatro de estas secciones críticas se utilizan solo para asignar identificadores únicos a los rayos y pueden ser fácilmente reemplazadas con operaciones de fetch–and–add. Otras dos secciones críticas gestionan la asignación de memoria y no pueden ser reemplazadas sin modificar significativamente el algoritmo. Finalmente, la última sección crítica gestiona la asignación de trabajo y puede ser reemplazada con una operación CAExch (Listado 9).

F. Ocean

Ocean, tanto en su version contigua como no contigua, es una aplicacion de estudio de movimiento oceánico a gran escala. Contienen tres secciones críticas que pueden agruparse en dos *barrier groups*. Dos de estas secciones críticas se utilizan para acumular las variables `psibi` y `psiai`. Dado que las operaciones de fetch–and–add atómicas con variables de punto flotante no son típicamente compatibles, confiamos en nuestra construcción personalizada FETCH_AND_ADD_DOUBLE, que utiliza una operación CAExch (Listados 10 y 11).

Finalmente, la última sección crítica recopila todos los errores calculados y selecciona el más grande. Esta fun-

```
/* 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));
```

Listado 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);
```

Listado 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);
```

Listado 11 slave2.c.in 857 & 718

cionalidad se implementa en algunas ISAs mediante la instrucción *atomicMin*. Dado que la ISA x86 no implementa *atomicMin*, ofrecemos una implementación alternativa. Como el máximo y el mínimo establecen un orden global, *atomicMin* puede implementarse fácilmente como una *actualización de prioridad*[29]. Nuestra implementación utiliza una variación ligera de la *actualización de prioridad* para mantener la uniformidad del código con el resto de Splash-4 (Listado12).

```
/* 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));
```

Listado 12 multi.c.in 90

G. Volrend

Volrend es una aplicación de renderizado de volumen rotativo tridimensional utilizando ray casting. Contiene 16 secciones críticas que pueden agruparse en tres *barrier groups*. Muchas de estas secciones críticas se utilizan para establecer identificadores únicos para cada subdivisión del proceso de renderizado. Las secciones críticas restantes gestionan el tamaño de muestreo para cada región del volumen a renderizar. Todas estas secciones pueden ser simplemente reemplazadas por una operación atómica de lectura-modificación-escritura equivalente (por ejemplo, Listado 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);
```

Listado 13 adaptive.c.in 182 & 199

H. Water

Water-Nsquared y Water-Spatial son aplicaciones simuladoras de fuerzas moleculares para moléculas de agua. Water-Nsquared utiliza un algoritmo $O(n^2)$ con un predictor y un corrector, mientras que Water-Spatial establece una cuadrícula 3D para distribuir las moléculas entre hilos con un algoritmo $O(n)$. Water-Nsquared contiene siete secciones críticas que se pueden agrupar en tres *barrier groups*, y Water-Spatial contiene siete secciones críticas agrupadas en siete *barrier groups*.

H.1 Común

Ambas implementaciones comparten cuatro secciones críticas que actualizan la fuerza inter/intra-molecular global, la energía cinética y la energía potencial. Sin embargo, como sucede en Ocean, estas variables son de punto flotante y, por lo tanto, requieren el uso de construcciones personalizadas (por ejemplo, Listados 14).

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

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

Listado 14 intraf.c.in 133 & interf.c.in 146 & intraf.c.in 170 & interf.c.in 196

Por otro lado, la sección crítica que actualiza la energía potencial del sistema consiste en tres operaciones de fetch–and–add. Como se discute en la Sección II-C, hay casos en los que una sección crítica se puede dividir en

varias secciones críticas más pequeñas manteniendo la corrección (Listado 15).

```
/* 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);
```

Listado 15 poteng.c.in 159 & poteng.c.in 253

```
/* 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]);
}
```

Listado 16 interf.c.in 156 & interf.c.in 167 & interf.c.in 179

H.2 Secciones críticas de Water-Nsquared

Water-Nsquared tiene tres secciones críticas que no se comparten con Water-Spatial. Estas secciones críticas son la parte principal del algoritmo $O(n^2)$, donde todas las fuerzas se calculan utilizando una estructura compartida. Sin embargo, estas secciones críticas también se pueden dividir en varias más pequeñas, utilizando la construcción FETCH_AND_ADD_DOUBLE (Listado 16).

IV. Resumen del Benchmark

Con el fin de entender cómo los cambios de código en Splash-4 afectan las aplicaciones en tiempo de ejecución, la Tabla I muestra todas las primitivas de sincronización ejecutadas (instancias de instrucciones) para Splash 3 y 4 al ejecutar 64 hilos. Además, también mostramos un desglose estático (secciones críticas en el código), ya que estas son las que tienen cambios importantes en el código (las barreras simplemente se reemplazan por una versión de sentido inverso). Las secciones críticas se clasifican en tres categorías, según cómo estén implementadas: bloqueo de mutex, atómicos de C11 o constructos *lock-free* (CAExch). Podemos observar que, en general, el total de llamadas a primitivas de sincronización sigue siendo similar entre las dos suites, pero, como veremos más adelante, el tiempo total dedicado a esperar en cada llamada a primitiva de sincronización disminuye drásticamente para Splash-4.

V. Metodología

El propósito principal de Splash-4 es servir como herramienta de evaluación para nuevas propuestas arquitecturales. Es por eso que este artículo realiza una caracterización tanto en hardware simulado como real. Sin embargo, el objetivo no es medir la precisión del simulador en comparación con el hardware real, sino medir la eficacia de los cambios introducidos en Splash-4.

A. Entorno de evaluación

El hardware seleccionado es la CPU AMD Epyc 7702P [30] con 64 núcleos a 2 GHz, 32KB de caché L1-D y L1-I, 512KB de caché L2 y 16MB de caché L3. El *hyperthreading* está habilitado, pero solo ejecutamos un hilo en cada núcleo físico. Ejecuta Ubuntu 18.04 con el kernel de Linux 5.4.0. El simulador seleccionado es gem5-20 [12] simulando el sistema completo. Imitamos un procesador Intel Ice Lake [31] funcionando a 2 GHz. El sistema simulado ejecuta Ubuntu 16.04 con el kernel de Linux 4.9.4. Los parámetros del procesador se muestran en la Tabla II. Utilizamos Ruby y Garnet [32] para modelar la jerarquía de memoria. La latencia de ejecución y emisión se modela según lo medido en hardware real por Fog [33]. Cada aplicación se ejecuta diez veces y luego se calcula la media recortada del 30 %. Las mediciones tienen en cuenta la región de interés (ROI), es decir, la región paralela, después de la inicialización y antes de la salida de pantalla. Se han eliminado las impresiones en esta sección del código. Además, las mediciones de gem5-20 restablecen las estadísticas dentro de la ROI después de un período de calentamiento, según lo sugerido por los desarrolladores originales de Splash-2, para minimizar la variabilidad entre ejecuciones.

B. Entradas de las aplicaciones

Las entradas utilizadas en este artículo se muestran en la Tabla III junto con la huella de memoria para 1 y 64 núcleos. Estas entradas, comúnmente conocidas como simsmall, son las mismas para ambas plataformas (hardware real y simulador). Valgrind se utiliza para medir la huella de memoria.

Esta sección muestra la caracterización del rendimiento de la suite de benchmarks Splash-4. También realizamos un análisis de escalabilidad para comprender cuánto tiempo se pasa en primitivas de sincronización.

C. Efectos de rendimiento de los costes de sincronización

Nuestra evaluación comienza mostrando cómo las distintas mejoras de código de Splash-4 afectan el rendimiento de la aplicación. La Fig.1 muestra los efectos individuales de actualizar las barreras de inversión de sentido, utilizando operaciones/constructos de bloqueo libres de atómicos, y una combinación de ambos (etiquetado como *Splash-4*) en una ejecución de 64 hilos en el procesador AMD 7702P. Las mejoras en las barreras reducen el tiempo de ejecución en un 40 % en promedio. Los constructos atómicos en aislamiento reducen el tiempo de ejecución

Aplicación	Barreras		Secciones Críticas						Condicionales					
			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

Tabla I: 64 núcleos, entradas predeterminadas, cada ejecución puede variar un poco, números obtenidos con nuestra herramienta pin. St(atic) es el número de instancias presentes en el código, mientras que Dyn(amic) es el número de instancias ejecutadas en tiempo de ejecución.

Procesador	Gem5	EPYC 7702P
Ancho de Fetch	5	32 Bytes
Ancho de Decode	5	4
Ancho de Rename	5	6
Ancho de Issue	10	6
Ancho de Commit	10	8
Cola de Instr.	160	-
ROB	352	224
Cola de Loads	128	44
Cola de Stores	72	48
Registros de Enteros	180	180
Registros Flotantes	180	160
Unidades de Loads	2	2
Unidades de Stores	1	3
ALUs de Enteros	1	4
ALUs Combinadas	3	-
Subsistema de Memoria (por núcleo)		
L1I	32K 8v	32K 8v
L1D	48K 12v	32K 8v
L2	512K 8v	512K 8v
L3 Compartida	2M 16w	4M
Directorio	32708 conj. 16 vias	-
Lat. de Memoria	80ns	N/A
Red de Interconexión		
Topologia	Crossbar	Inf. Fabric 2

Tabla II: Configuración de gem5 y de la máquina Real

Aplicación	Entrada	Huella de Memoria (1/64 núcleos)
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

Tabla III: Datos de entrada de las aplicaciónes (# → Number of cores)

en un 11 % en promedio, aunque tiene un rendimiento marginalmente peor que pthreads en condiciones de baja contención. En general, el impacto de las optimizaciones en el tiempo de ejecución depende de cuánto cada aplicación dependa de los bloqueos/barreras y de su sobrecarga de sincronización (Fig.2). Por ejemplo, FFT solo tiene un bloqueo que protege la identificación del hilo (Tabla I), y solo se ejecuta una vez y sin contención, por lo que el impacto de la optimización *Atomic* en aislamiento es mínimo. La combinación de ambas técnicas proporciona una reducción significativa del tiempo de ejecución, un 52 %.

Los resultados de tiempo de ejecución muestran los beneficios de Splash-4, pero, para ser más exhaustivos, el siguiente paso es desglosar los costos de sincronización de ambas suites de benchmarks.

Para hacerlo, modificamos los códigos de las aplicaciones para incluir instrucciones ficticias que marcarán el comienzo y el final de las distintas primitivas de sincronización. Estas instrucciones se capturan en gem5-20 en la etapa de *commit*, de modo que solo se consideren las primitivas de sincronización de la ejecución correcta. La Fig.2 muestra dos barras por recuento de hilos, que representan las implementaciones de Splash-3 y Splash-4,

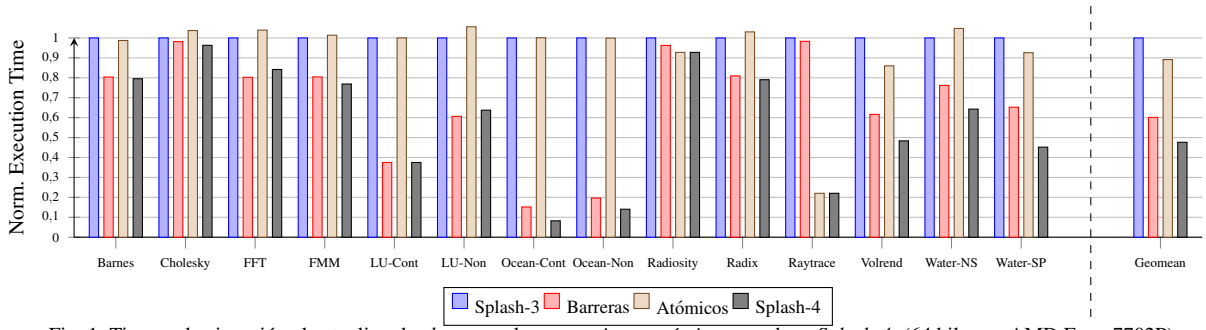

Fig. 1: Tiempo de ejecución al actualizar las *barreras*, las *operaciones atómicas* y ambas -*Splash-4*- (64 hilos en AMD Epyc 7702P)

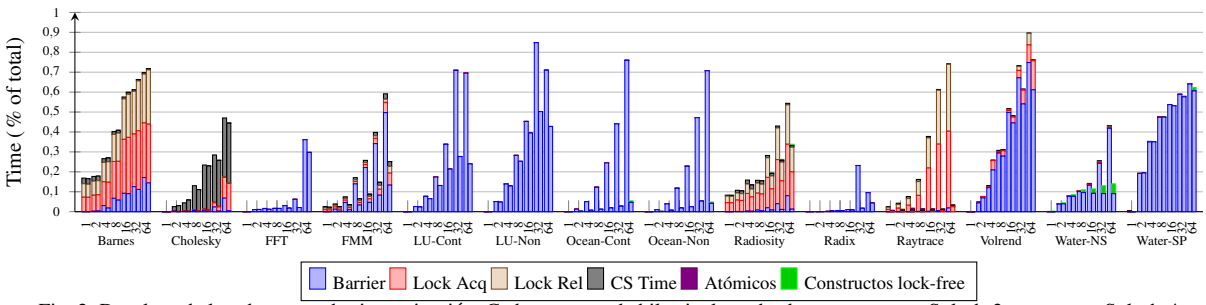

Fig. 2: Desglose de la sobrecarga de sincronización. Cada recuento de hilos incluye dos barras, una para Splash-3 y otra para Splash-4.

respectivamente. Esta Figura muestra un gran aumento en los costos de sincronización para FFT con 64 núcleos, ligeramente contenidos por las barreras de inversión de sentido de Splash-4. También podemos ver cómo disminuye drásticamente el tiempo de espera de la barrera para LU, Ocean y Water-NS. En cuanto a los costos de bloqueo, Radiosity y Raytrace son las aplicaciones que más se benefician de las mejoras de código. Es importante tener en cuenta que este desglose corresponde a un procesador simulado similar a Ice Lake (gem5-20), por lo que habrá diferencias en los valores absolutos en comparación con la Fig.1.

D. Escalabilidad

La Fig.3 y la Fig.4 muestran un análisis de escalabilidad tanto en hardware real como en simulador, respectivamente, para estudiar cómo la reducción de los costos de sincronización afecta el rendimiento. La escalabilidad se calcula solo considerando el tiempo de ejecución de la región de interés (ROI) para cada número de hilos, normalizado al tiempo de ROI para la versión de un solo hilo. La escalabilidad se muestra en una escala logarítmica. Además, es importante tener en cuenta que el objetivo de este artículo no es validar la infraestructura de simulación. Splash-4 mejora o mantiene la escalabilidad en ambos entornos, es decir, los cambios de Splash-4 no son contraproducentes en los sistemas y entradas medidos. Raytrace, una aplicación con alta contención atómica, dejó de escalar a partir de cuatro hilos en Splash-3. Con las mejoras de código, ahora puede escalar hasta 32 hilos en hardware real y 64 en la simulación. La escalabilidad de LU saltó de 16 a 64 hilos en ambas plataformas. LU es una aplicación sincronizada con barreras pero no intensiva en cómputo. Esto significa que después de 8 hilos, el costo de sincronización con las barreras predeterminadas es bastante alto en comparación con los cálculos realizados entre barreras. Ocean escala hasta 32 hilos en hardware real, pero no lo hace en una simulación, a pesar de la enorme reducción en el tiempo de espera de la barrera. Las diferencias en Radix entre el hardware real y la simulación se deben principalmente a la ubicación de reinicio de estadísticas sugerida por los autores de Splash-2. Hay una asignación de memoria justo antes del lugar de reinicio de las estadísticas. Esta asignación de memoria paralela es la razón principal por la que Radix escala mejor, y también se puede ver en la simulación cuando las estadísticas no se reinician después de la asignación. El resto de las aplicaciones experimentan pequeñas mejoras de rendimiento.

VI. Conclusiones

Splash-4 es la última revisión del conjunto de pruebas Splash, centrada en modernizar sus operaciones de sincronización y, por lo tanto, mejorar la escalabilidad de las aplicaciones. Este trabajo presenta Splash-4 y realiza un análisis detallado del rendimiento en comparación con Splash-3. Basamos nuestro análisis tanto en hardware real como en un entorno simulado utilizando gem5-20.

Nuestro estudio muestra una mejora significativa en la escalabilidad de Splash-4, pasando de 4 a 16 núcleos a 16 a 32 núcleos en la mayoría de las aplicaciones. El tiempo de ejecución se reduce tanto en el entorno simulado (34 %) como en el hardware real (52 %). También es importante destacar que las mejoras de código realizadas en Splash-4, apenas tuvieron efectos negativos en nuestro análisis, con la excepción de una ligera desaceleración para 16 núcleos, Los algoritmos siguen siendo los mismos, por lo que las aplicaciones revisadas con primitivas de sincronización actualizadas mantienen todos los patrones computacionales, pero con la ventaja de aprovechar las características modernas de sincronización hardware.

En resumen, Splash sigue siendo fundamental en la investigación de arquitectura de computadoras. Sin embargo, debería actualizarse para poder aprovechar las carac-

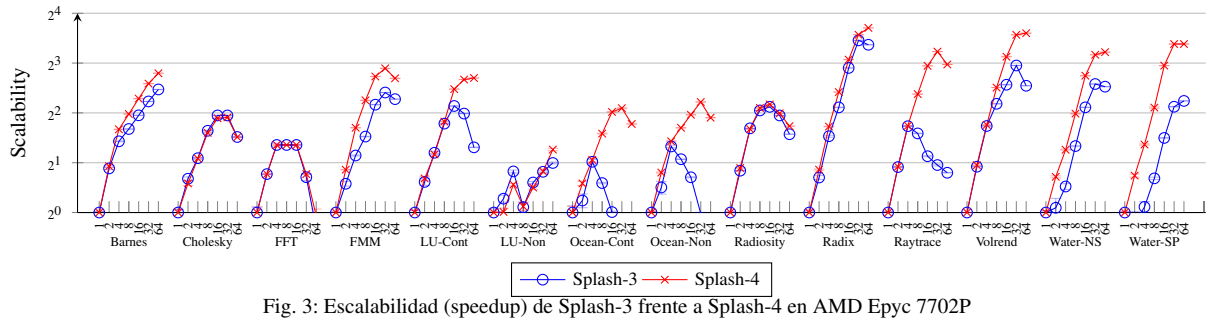

Fig. 3: Escalabilidad (speedup) de Splash-3 frente a Splash-4 en AMD Epyc 7702P

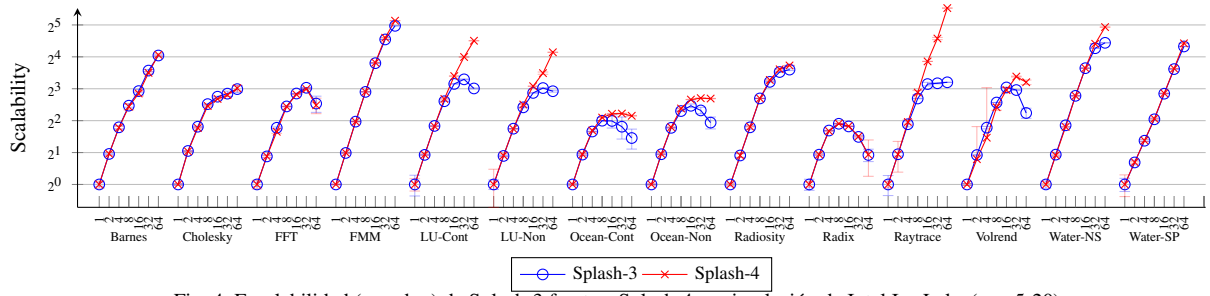

Fig. 4: Escalabilidad (speedup) de Splash-3 frente a Splash-4 en simulación de Intel Ice Lake (gem5-20)

terísticas modernas del hardware. Splash-4 logra esto al introducir mecanismos de sincronización de bajo costo. Esto elimina parte de la sobrecarga de sincronización y añade presión adicional a los núcleos, principalmente en el *backend*, dejando una puerta abierta para que los investigadores mejoren aún más sus diseños.

Agradecimientos

Este proyecto ha recibido financiamiento del Consejo Europeo de Investigación (ERC) en el marco del programa de investigación e innovación Horizon 2020 de la Unión Europea (acuerdo de subvención No 819134) y del Ministerio de Economía, Industria y Competitividad de España, a través de la Agencia Estatal de Investigación, en virtud de la subvención ERC2018-092826.

Referencias

[1] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, June 1995, pp. 24–36.

[2] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard 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.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai 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.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Int'l Symp. on Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.

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

[6] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, pp. 27, 2012.

[7] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey 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.

[8] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2012, SPAA '12, p. 68–70, Association for Computing Machinery.

[9] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai 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.

[10] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto 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.

[11] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sept. 2005.

[12] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Matthew D. Sinclair Boris Shingarov, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

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

[14] ISO, *ISO/IEC 14882:2011 Information technology — Program-*

ming languages — C++, International Organization for Standardization, Geneva, Switzerland, Feb. 2012.
[15] A. Williams, *C++ Concurrency in Action*, Manning Publications, 2019.
[16] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Number 325462-072US. Intel, May 2020.
[17] "Power ISA Version 3.1," `https://ibm.ent.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0`, May 2020.
[18] ARM, "Arm synchronization primitives development article," 2022.
[19] Andrew Waterman and Krste Asanovic, "The risc-v instruction set manual, volume i: Unprivileged isa document, version 20190608-baseratified," *RISC-V Foundation, Tech. Rep*, 2019.
[20] Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
[21] Eduardo José Gómez-Hernández, Ruixiang Shao, Christos Sakalis, Stefanos Kaxiras, and Alberto Ros, "Splash-4: Improving scalability with lock-free constructs," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, Ed., Worldwide event, Mar. 2021, pp. 235–236, IEEE Computer Society.
[22] Paul D Bryan, Jesse G Beu, Thomas M Conte, Paolo Faraboschi, and Daniel Ortega, "Our many-core benchmarks do not use that many cores," *System*, vol. 6, pp. 8, 2009.
[23] Matteo Monchiero, Jung Ho Ahn, Ayose Falcón, Daniel Ortega, and Paolo Faraboschi, "How to simulate 1000 cores," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 10–19, 2009.
[24] 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.
[25] H. Gao and W.H. Hesselink, "A general lock-free algorithm using compare-and-swap," *Information and Computation*, vol. 205, no. 2, pp. 225–241, 2007.
[26] Hans-J. Boehm, "How to miscompile programs with "benign"data races," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, USA, 2011, HotPar'11, p. 3, USENIX Association.
[27] John M. Mellor-Crummey and Michael L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.
[28] "The GNU C library (glibc)," `https://www.gnu.org/software/libc/libc.html`, 2022.
[29] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons, "Reducing contention through priority updates," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2013, SPAA '13, p. 152–163, Association for Computing Machinery.
[30] "Zen 2 - microarchitectures - amd," .
[31] "Sunny cove - microarchitectures - intel," .
[32] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj 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.
[33] Agner Fog, "Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns," 2018, Available at `http://www.agner.org/optimize/instruction_tables.pdf`.