SAND2019-9952C

Enabling Resilience in Asynchronous Many-Task Programming Models

Sri Raj Paul[1], Akihiro Hayashi[2], Nicole Slattengren[3], Hemanth Kolla[3], Matthew Whitlock[3], Seonmyeong Bak[1], Keita Teranishi[3], Jackson Mayo[3], and Vivek Sarkar[1]

August 29, 2019

[1] : Georgia Institute of Technology
[2] : Rice University
[3] : Sandia National Laboratories

Extreme-Scale Computing Challenges

- **Large number of processing unit**
  - **Bandwidth and latency constraints**
- **Heterogeneity**
  - **CPU, GPU, FPGA**
  - **NVRAM, Scratchpad, NMP**

Extreme-Scale Computing Challenges

- **Large number of processing unit**
  - **Bandwidth and latency constraints**
- **Heterogeneity**
  - **CPU, GPU, FPGA**
  - **NVRAM, Scratchpad, NMP**
- **Shorter mean times to failure than before**
  - **Includes transient failures**

Extreme-Scale Computing Challenges

- Large number of processing unit
  - Bandwidth and latency constraints
- Heterogeneity
  - CPU, GPU, FPGA
  - NVRAM, Scratchpad, NMP

https://www.crystalgraphicsimages.com/search/cartoon-man-computer-images

- Shorter mean times to failure than before
  - Includes transient failures

Extreme-Scale Programming Challenge

- **Application programmers do not want to manage all system resources or hardwire their code to a specific platform**

- **Decouple specification of computation and data from system mapping for programmability and performance portability**
  - **Computation can be decomposed into a large number of asynchronous tasks**
  - **Support task migration using relocatable objects virtualized from actual memory**

Extreme-Scale Programming Challenge

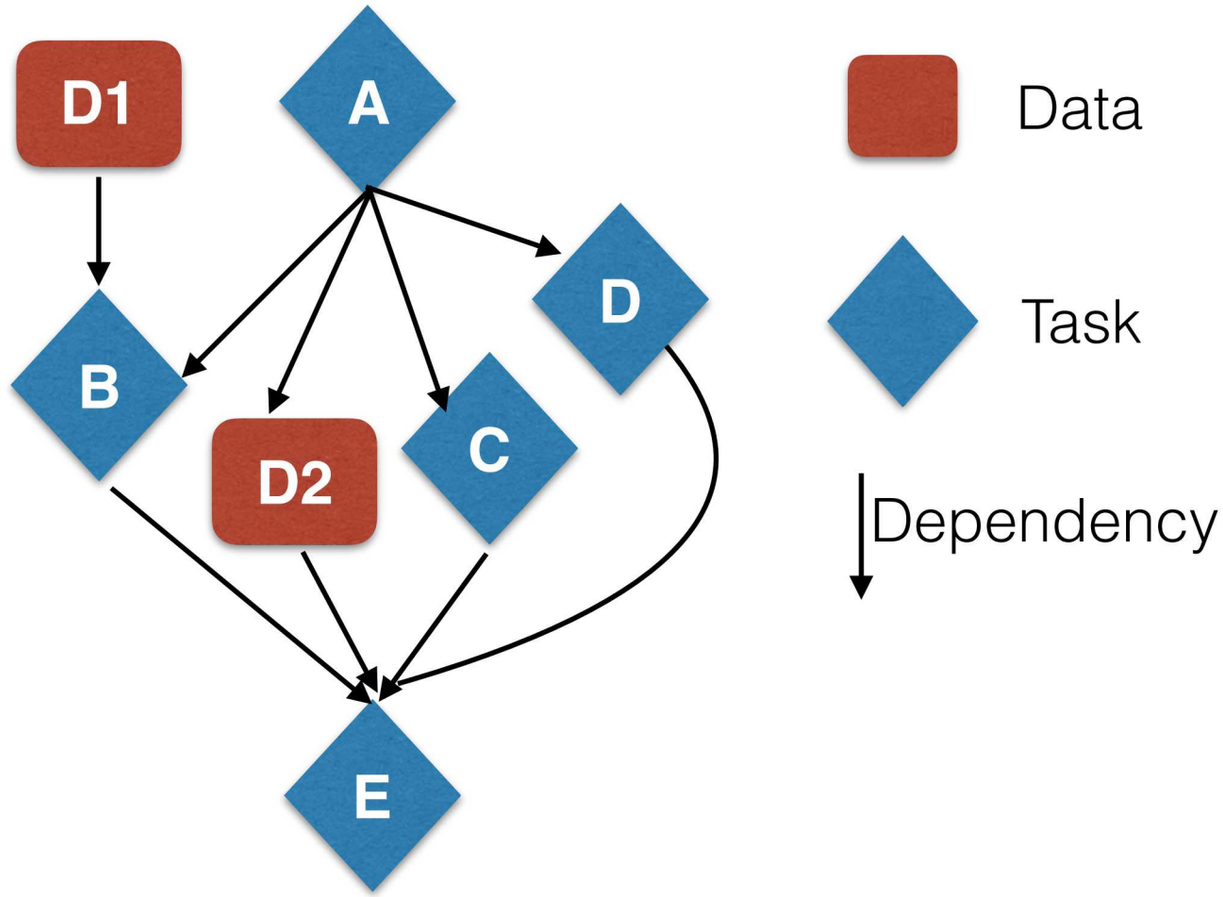
- **Application programmers do not want to manage all system resources or hardwire their code to a specific platform**

- **Decouple specification of computation and data from system mapping for programmability and performance portability**
  - **Computation can be decomposed into a large number of asynchronous tasks**
  - **Support task migration using relocatable objects virtualized from actual memory**

- **A practical realization of these ideas can be found in Asynchronous Many Task (AMT) models**

Asynchronous Many Task (AMT) Runtime

- **Application programmer expresses available parallelism in terms of**
  - **Tasks,**
  - **Dependencies between tasks**
  - **Data dependencies of tasks**
- **The runtime performs**
  - **Scheduling of tasks**
  - **Migration of tasks/data between nodes/cores to ensure load balance and locality**
- **Advantages: more flexible resource management, resilience support**
- **HClib, OCR, PaRSEC, Uintah …**



Type of Errors

- **Fail-stop error**
  - **Process halts**
  - **Need to restart the application**
  - **Can restart from intermediate state if checkpointing was performed**

- **Fail-continue error or soft/transient errors**
  - **Likely to be most important type of fault in exascale systems [1]**
  - **DUE - Detected Uncorrected Errors**
  - **SDC - Silent data corruptions**
    - **Errors such as multiple faults that cancel each other, preventing the HW from detecting it**
    - **Require software mechanisms such as replication or checksum to detect such errors**

[1]Franck Cappello et al. 2014. Toward Exascale Resilience: 2014 Update. Supercomput. Front. Innov.: Int. J. 1, 1 (April 2014), 5–28.

Type of Errors

- **Fail-stop error**
  - **Process halts**
  - **Need to restart the application**
  - **Can restart from intermediate state if checkpointing was performed**

- **Fail-continue error or soft/transient errors**
  - **Likely to be most important type of fault in exascale systems [1]**
  - **DUE - Detected Uncorrected Errors**
  - **SDC - Silent data corruptions**
    - **Errors such as multiple faults that cancel each other, preventing the HW from detecting it**
    - **Require software mechanisms such as replication or checksum to detect such errors**

[1]Franck Cappello et al. 2014. Toward Exascale Resilience: 2014 Update. Supercomput. Front. Innov.: Int. J. 1, 1 (April 2014), 5–28.

Contributions

- **Programming model extensions to enable resilience techniques for AMT applications**

- **Support for arbitrary compositions of these extensions**

- **Unified execution of resilient and non-resilient tasks in a single framework**

AMT Runtime and Resiliency

- Tasks can be migrated to achieve load balance
  - Tasks can be migrated around faults
    - Replay task in a different worker

- Tasks provide a natural boundary on where to perform error-checking/checkpointing
  - SPMD model needs to take a global checkpoint
  - No need for global checkpoint with AMT tasks

- Input/Output to tasks provides information on what needs to be checked/checkpointed
  - Input/Output of tasks can be done through promise/futures

Habanero-C/C++ Library (HClib)

- **Library-based tasking runtime and API**
  - **Semantically derived from X10 programming model**
- **Focused on: lightweight, minimal overheads; flexible synchronization; locality control;**
- **Simplified deployment: no custom compiler, entirely library-based**
- **Uses runtime-managed call stacks to avoid blocking**

HClib - AMT Runtime

- Tasks are the basic unit of computation
- Tasks can wait on completion of other tasks :

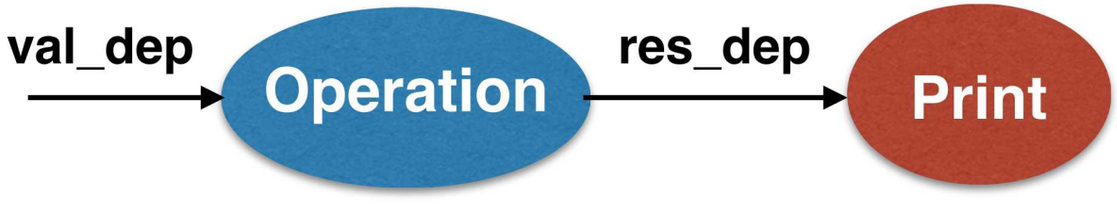
  async_await(task_body, dependency_1, dependency_2, ..);

- Promise: Store a value (put) using single assignment semantics
- Future: Fetch a value (get) stored in a promise
  - Can be used as a dependency to a task
- Write to a Promise is synchronized with the read from its corresponding Future

HClib - Task Example

```
void async_await(lambda, hclib_future_t *f1, …, hclib_future_t *f4)
```

```
void main() {
    operation_val();
    print_result();
    val = new value(get_val());
    val_dep->put(val);
}
void operation_val() {
  async_await(
   [=] { val = val_dep->get_future()->get();
    res = operation(val);
    res_dep->put(res);
   }, val_dep->get_future());
}
```

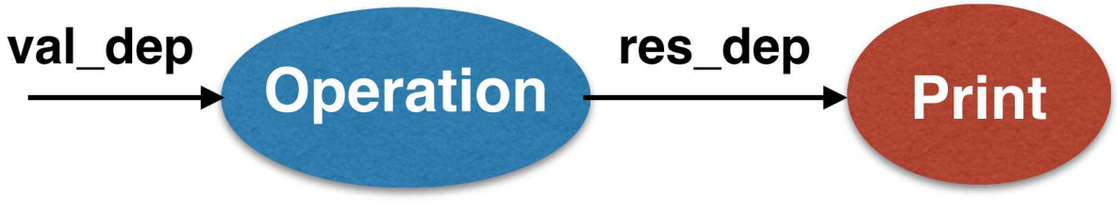



```
void print_result() {
  async_await(
   [=] { res = res_dep->get_future()->get();
    print(res);
  }, res_dep->get_future());
}
auto val_dep = new promise();
auto res_dep = new promise();
```

HClib - Task Example

```
void async_await(lambda, hclib_future_t *f1, …, hclib_future_t *f4)
```

```
void main() {
    operation_val();
    print_result();
    val = new value(get_val());
    val_dep->put(val);
}
void operation_val() {
  async_await(
   [=] { val = val_dep->get_future()->get();
    res = operation(val);
    res_dep->put(res);
   }, val_dep->get_future());
}
```

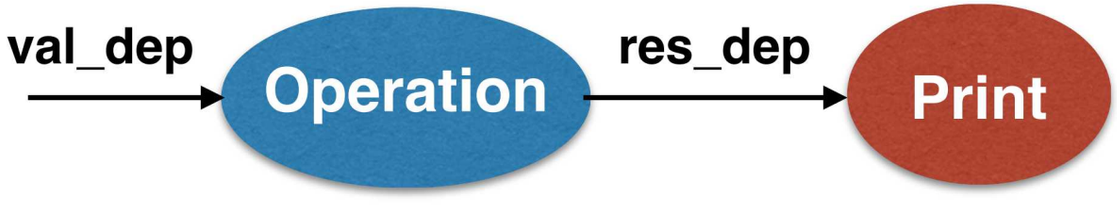



```
void print_result() {
  async_await(
   [=] { res = res_dep->get_future()->get();
    print(res);
  }, res_dep->get_future());
}
auto val_dep = new promise();
auto res_dep = new promise();
```

HClib - Task Example

```
void async_await(lambda, hclib_future_t *f1, …, hclib_future_t *f4)
```

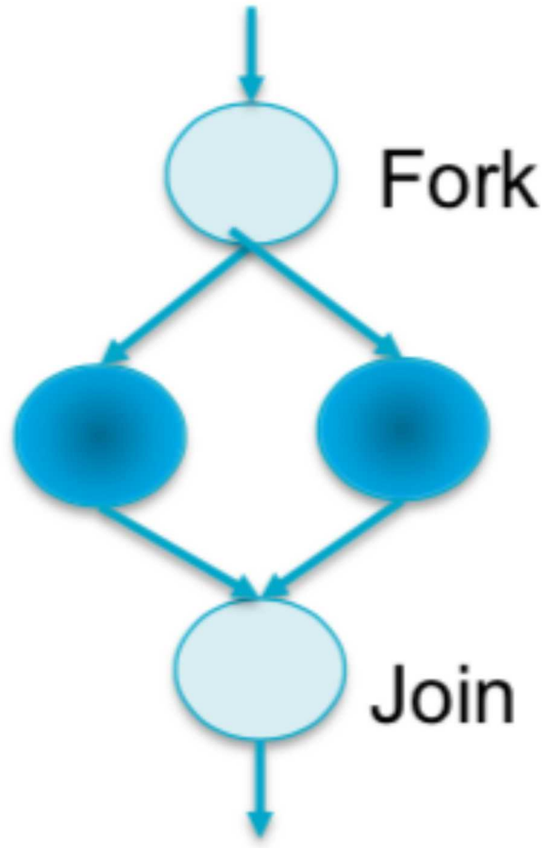
```
void main() {
    operation_val();
    print_result();
    val = new value(get_val());
    val_dep->put(val);
}
void operation_val() {
  async_await(
   [=] { val = val_dep->get_future()->get();
    res = operation(val);
    res_dep->put(res);
   }, val_dep->get_future());
}
```




```
void print_result() {
  async_await(
   [=] { res = res_dep->get_future()->get();
    print(res);
   }, res_dep->get_future());
}
auto val_dep = new promise();
auto res_dep = new promise();
```

1. Task Replication

- N-way replicate the task and checks for equality of put operations at the end of the task
- If error checking succeeds, actual puts are done
- If error checking fails, puts are ignored and the error is reported using an output promise

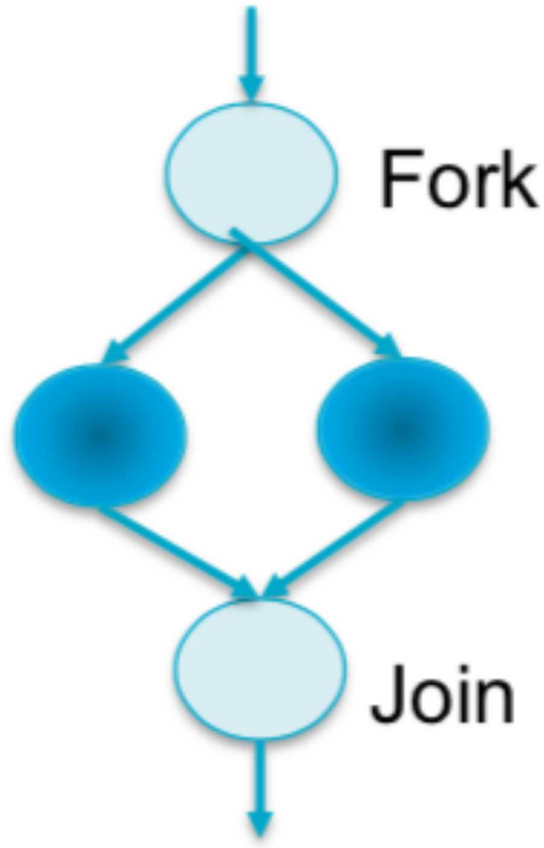


1. Task Replication

- **N-way replicate the task and checks for equality of put operations at the end of the task**

- **If error checking succeeds, actual puts are done**

- **If error checking fails, puts are ignored and the error is reported using an output promise**




```
void async_await_check<N>(lambda, hclib::promise<int> out,
                          hclib_future_t *f1, …, hclib_future_t *f4)
```

Resilient Promise

- **Extend promise to include additional storage for replicated put operation**

- `put` **is not published until the N-way task is finished and correctness check succeeds**
  - **Added task-local storage to collect the promise for publishing at the end of the task**

- **Correctness is checked using equals() method provided by the user object**

Data Management

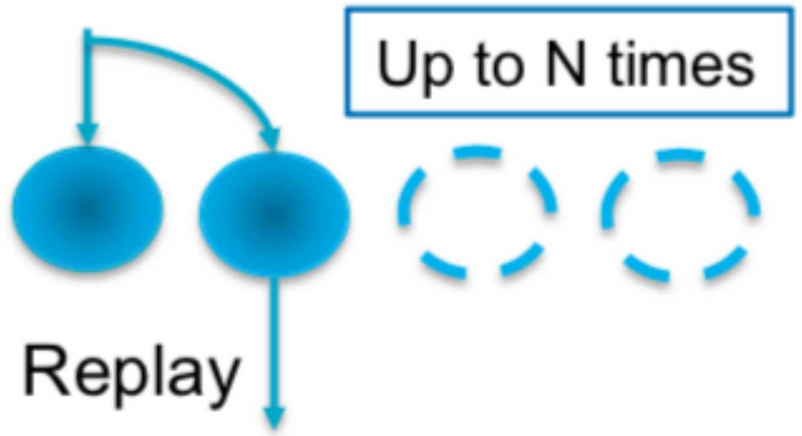
- **Problem:**
  - **User needs to keep track of the last task that uses a promise and then deallocate data**
  - **Resiliency might involve multiple executions of the task**
  - **User needs to keep track of correct vs bad execution**
- **Solution:**
  - **Added a reference count to a promise**
    - **Count indicates the number of tasks that depends on the promise**
  - **The promise is deallocated after the specified number of tasks finish execution**

2. Replay Tasks

- Executes the task and checks for error using the error checking function
  - User-provided error checking function returns true if there is no error

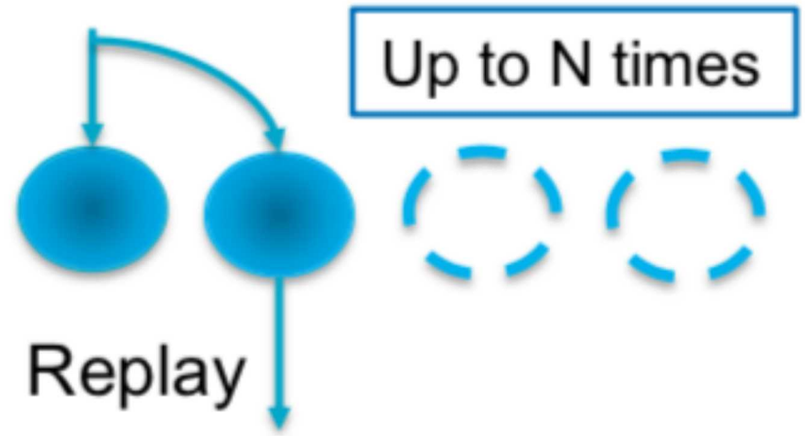
- The task is executed a maximum of N times if there is any error

- If error checking fails, puts are ignored and the error is reported using an output promise
  - Puts should be performed on a resilient promise



2. Replay Tasks

- Executes the task and checks for error using the error checking function
  - User-provided error checking function returns true if there is no error
- The task is executed a maximum of N times if there is any error
- If error checking fails, puts are ignored and the error is reported using an output promise
  - Puts should be performed on a resilient promise



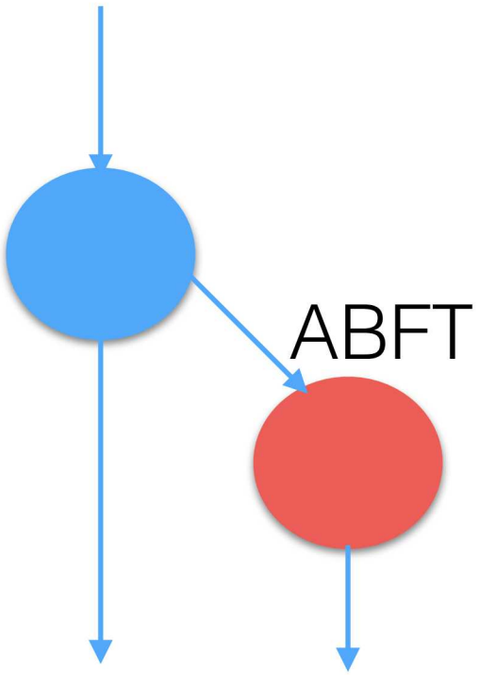

```
void async_await_check<N>(lambda, hclib::promise<int> out,
                          std::function<int(void*)> error_check_fn, void * params,
                          hclib_future_t *f1, …, hclib_future_t *f4)
```

Algorithm Based Fault Tolerance (ABFT)

- Use numerical properties of the algorithm to perform error correction

- Executes the task and checks for error using the error checking function
  - Error checking function returns true if there is no error

- If error checking fails, error correction routine is executed and checked for error again at its end
  - If error checking fails, puts are ignored and the error is reported using an output promise

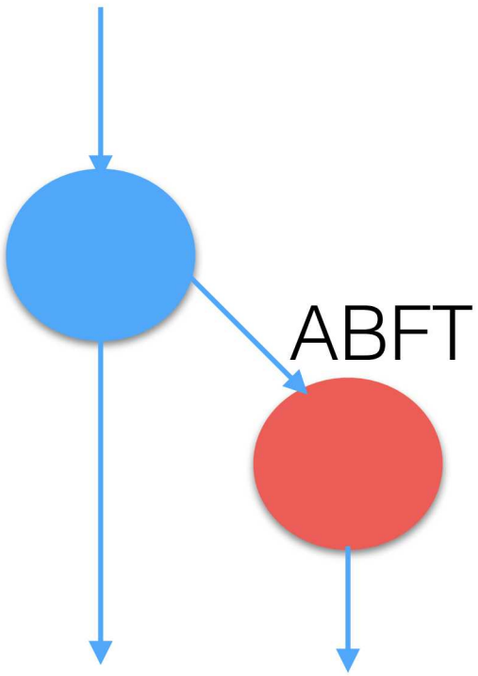


Algorithm Based Fault Tolerance (ABFT)

- **Use numerical properties of the algorithm to perform error correction**

- **Executes the task and checks for error using the error checking function**
  - **Error checking function returns true if there is no error**

- **If error checking fails, error correction routine is executed and checked for error again at its end**
  - **If error checking fails, puts are ignored and the error is reported using an output promise**




```
void async_await_check(lambda, hclib::promise<int> out,
                       std::function<int(void*)> error_check_fn, void * params,
                       ABFT_lambda, hclib_future_t *f1, …, hclib_future_t *f4)
```

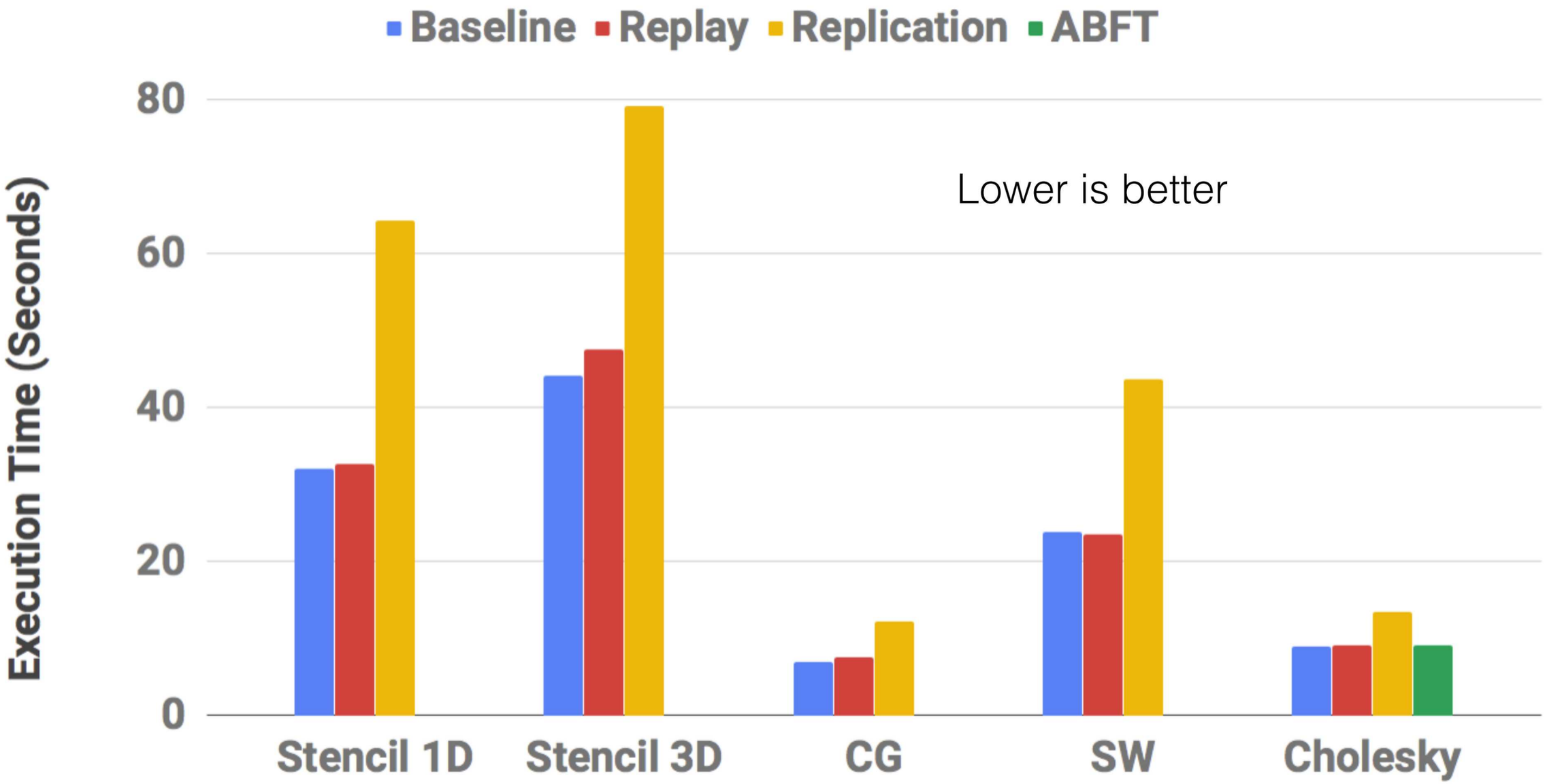
Preliminary Evaluation

- **Cray XC40™ Supercomputer @ NERSC (Cori)**
  - **Node**
    - **Intel Xeon E5-2698 v3 @ 2.30GHz x 32 cores**
    - **128GB of RAM**
    - **Replication: defaults to 2-way and if error is found 3-rd replica is created.**
    - **Replay: 3-way i.e if error is reported, maximum of 2 reruns are allowed.**

Benchmarks

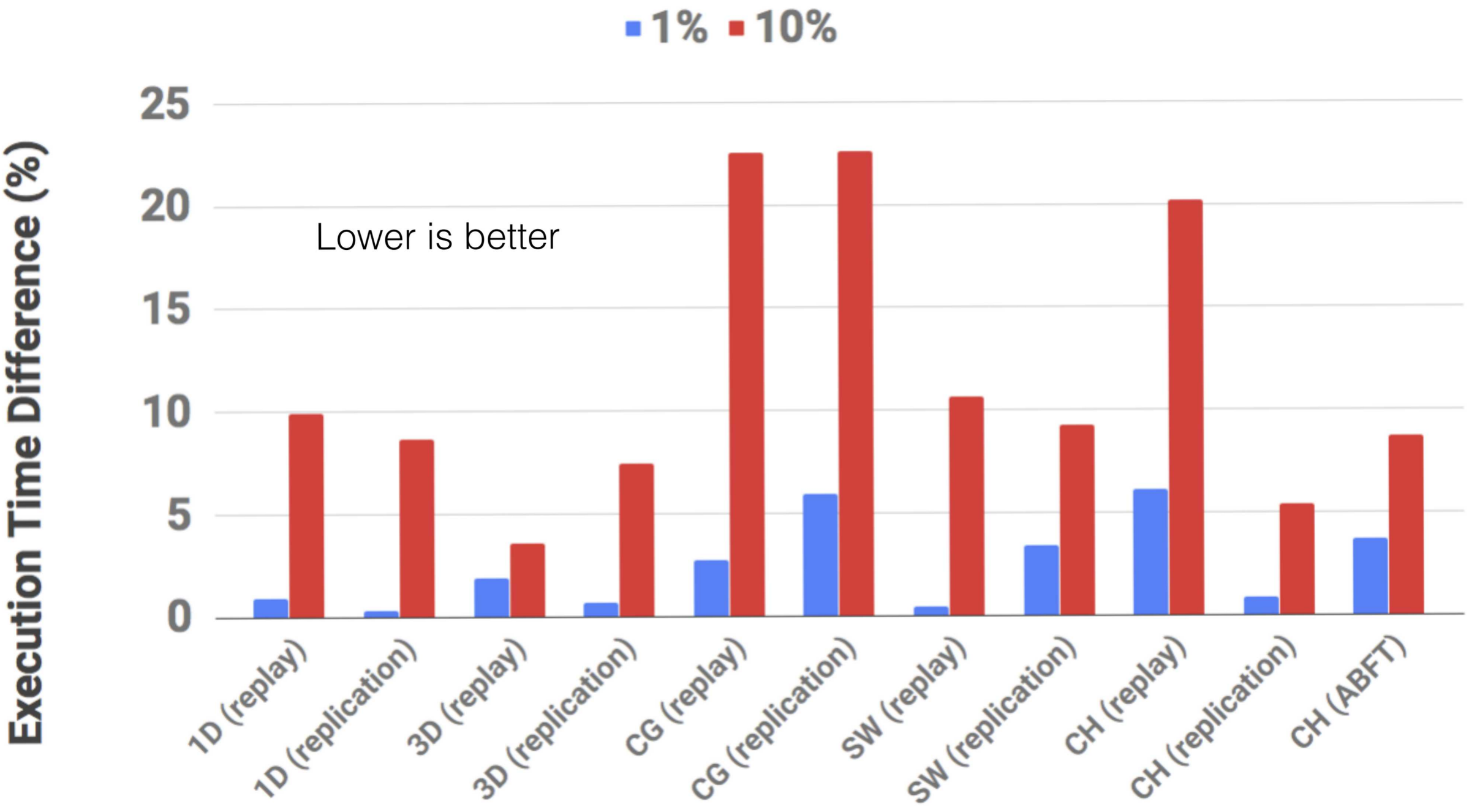
- **Stencil 1D : 3-point stencil with 128 tiles of size 16000 doubles, 128 time steps per iteration, and 8192 iterations.**
- **Stencil 3D : 7-point stencil with 16x16x16 cubes, each cube representing a subdomain of size 32x32x32 and 1024 iterations.**
- **Conjugate Gradient : Square matrix of 52804 rows/columns, having 5,333,507 non-zeros with 128 tiles and 500 iterations**
- **Smith Waterman : Strings of sizes 185600 and 192000, divided among 4096 tiles arranged as 64x64**
- **Cholesky : Matrix of size 24000x24000 divided into tiles of size 400x400**

Results - No Failures

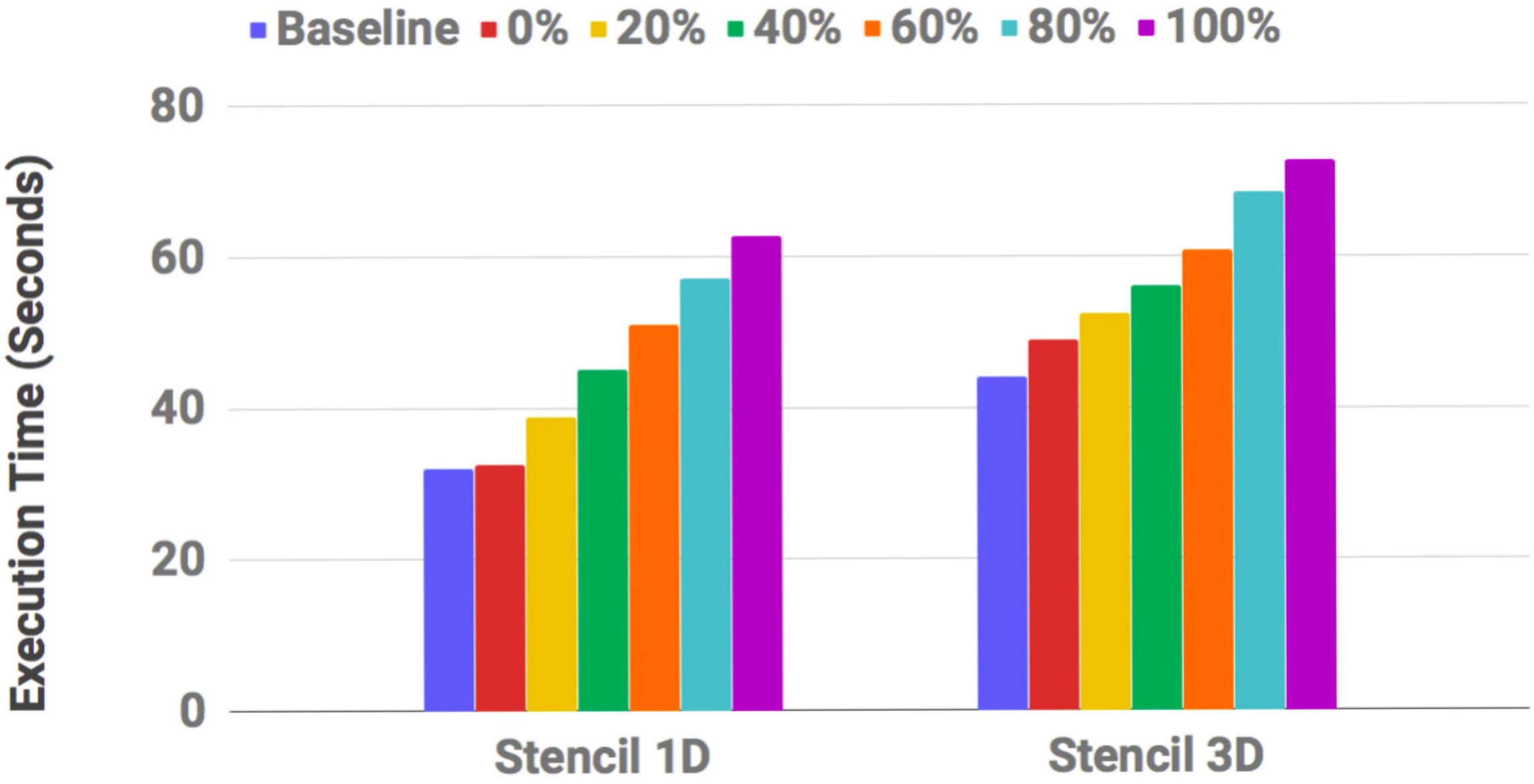


- Replay overhead causes a maximum 10% slowdown.
- Replication overhead causes the execution time to double.

Results - With Failures



Results - Mixing



- Mixing Replication (x%) and Replay (100-x)%
- Baseline is non-resilient execution time
- Percentage of replication shown in graph

Related Work

- **BSC (OmpSS): Task Replication**
- **OSU+PNNL: Task Replay**
- **UTK+ORNL: ABFT**
- **ENS Lyon: Rigorous model for task replication**
- **BLCR (LBL): Checkpoint/Restart**

- **We demonstrate how to incorporate all these various resilience mechanisms on a promise based tasking infrastructure**

Conclusion

- **AMT runtimes can support wide variety of resilience mechanisms**
- **Adapted various benchmarks to be resilient to transient errors**
- **Seamless composition of various resilience mechanisms**

Future Work

- **Composing with communication**
- **Nesting resilient tasks**
- **Exploration of multilevel checkpointing**
- **Study characteristics of faults to perform experiments with more realistic fault injection**

- **Backup**

RICE

Results - Cache Reuse

- **Replication is using less than twice the time**

3D Stencil	Time	L1_DCM	L2_TCM	L3_TCM
Replay	40.58 sec	7.97E+10	3.32E+10	6.21E+08
Replication	73.18 sec	1.61E+11	6.36E+10	9.19E+08