Compiler-assisted type-safe checkpointing

TECHNISCHE UNIVERSITÄT DARMSTADT

Compiler-assisted Correctness Checking and Performance Optimization for HPC

June 25th, 2020, Virtual Conference

Jan-Patrick Lehr @jplehr
jan-patrick.lehr@tu-darmstadt.de

Alexander Hück alexander.hueck@tu-darmstadt.de
Moritz F. Fischer moritz_friedrich.fischer@stud.tu-darmstadt.de
Christian Bischof christian.bischof@tu-darmstadt.de

Motivation

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double *pA = (double *) malloc(N * sizeof(double));
// memId: starts at pA, with size N*sizeof(double) bytes
VELOC_Mem_protect(memId, pA, N, sizeof(double));
VELOC_Checkpoint("CPLabel", CPVersion);
VELOC_Mem_unprotect(memId);
```

- Many checkpoint/restart (CPR) libraries offer (type) unsafe, low-level APIs:
  - memory region ID
  - *pointer to start of memory region*
  - *number of elements*
  - *size (in bytes) per element*

SC Scientific Computing

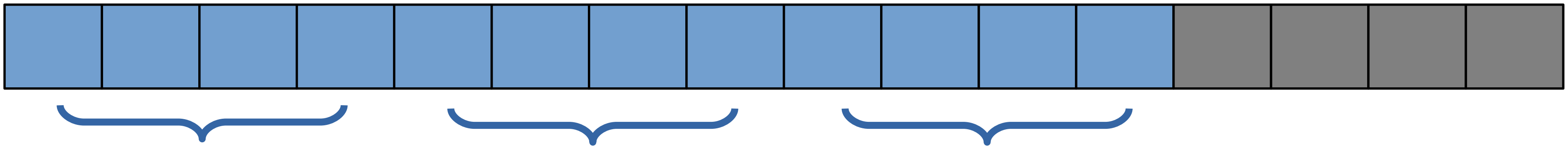
Motivation

```
int *pi = (int *) malloc(N * sizeof(int));
// element count error
VELOC_Mem_protect(1, pi, X, sizeof(int));
// data type error
VELOC_Mem_protect(1, pi, N, sizeof(double));
```

- Errors that confuse number of elements or element size are easily made
  - Result in wrong number of bytes to be captured by CPR library

Motivation

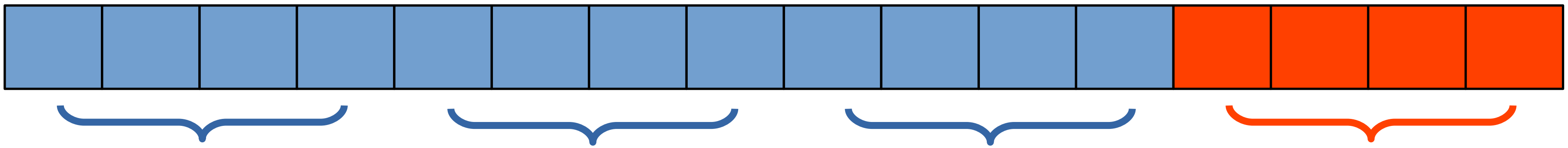
pi->



Allocation holding 3 integer values (assuming 4 byte per integer)

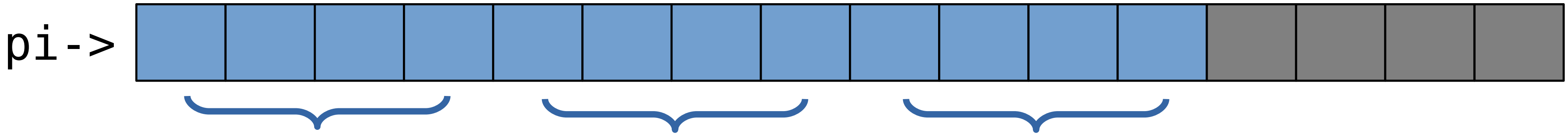
Checkpointing too many elements

pi->



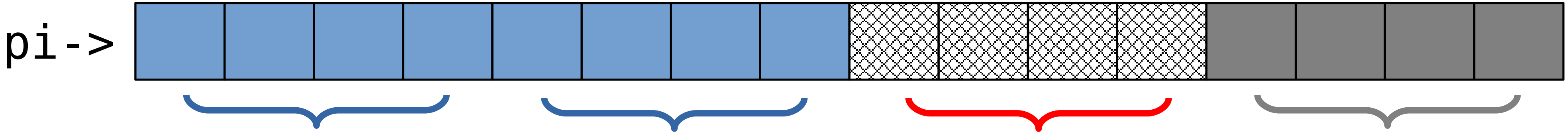
Checkpoint: Illegally reads behind allocation end (crash or silent error)
Restart: Illegally writes behind allocation end (crash or silent error)

Motivation



Allocation holding 3 integer values (assuming 4 byte per integer)

Checkpointing too few elements



Checkpoint: Data is not captured (silent error)
Restart: Memory is not fully initialized (crash or bogus results)

Error Types

- **Element Count Error**
  Developer specifies the number of elements in the allocation erroneously
- **Data Type Error**
  Developer specifies the type of the elements erroneously
- **Change Allocation Before Checkpoint**
  The already registered allocation is changed, such that either an Element Count Error or a Data Type Error are present

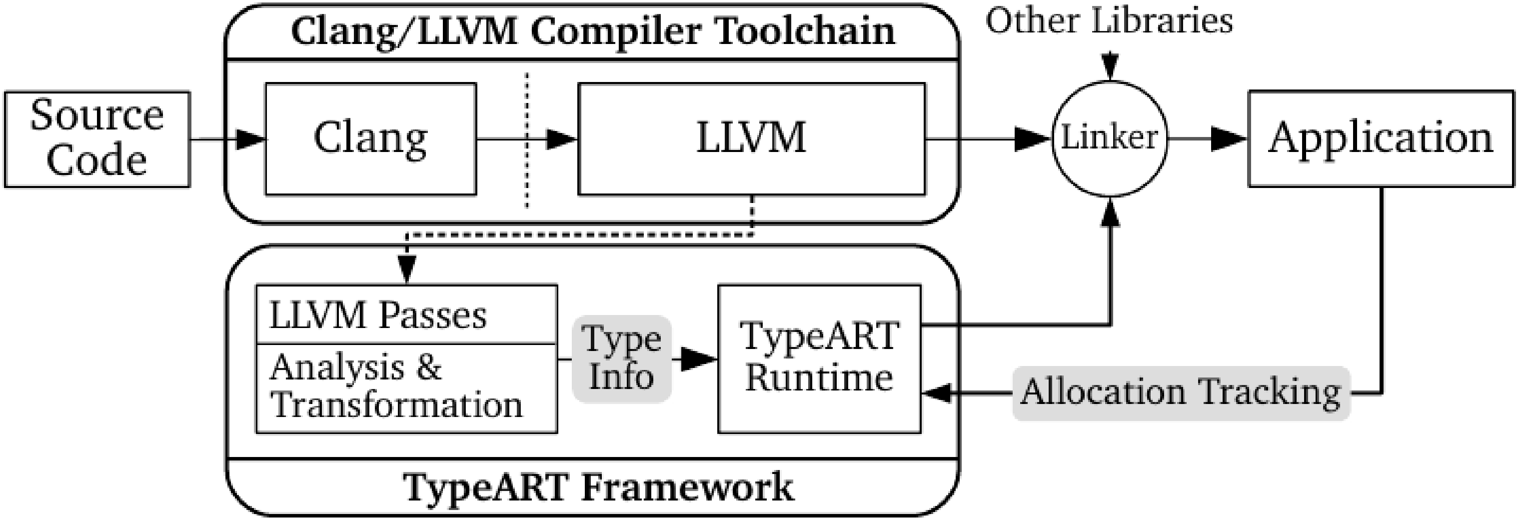
→ **Can be detected automatically using allocation tracking**

Approach

- Implement a mechanism to state requirements on memory regions
- For each CPR-registered memory region developer states the requirements
- At runtime: check the given requirements for validity
  - On a successful check: continue execution with checkpoint
  - On a failed check: abort execution with error message

- Similar to information used in MPI Type checking [1]
  - → Use **TypeART** to implement approach

TypeART



- TypeART is a memory allocation tracking and sanitizer built on LLVM
  - Uses combination of compile-time analysis and instrumentation

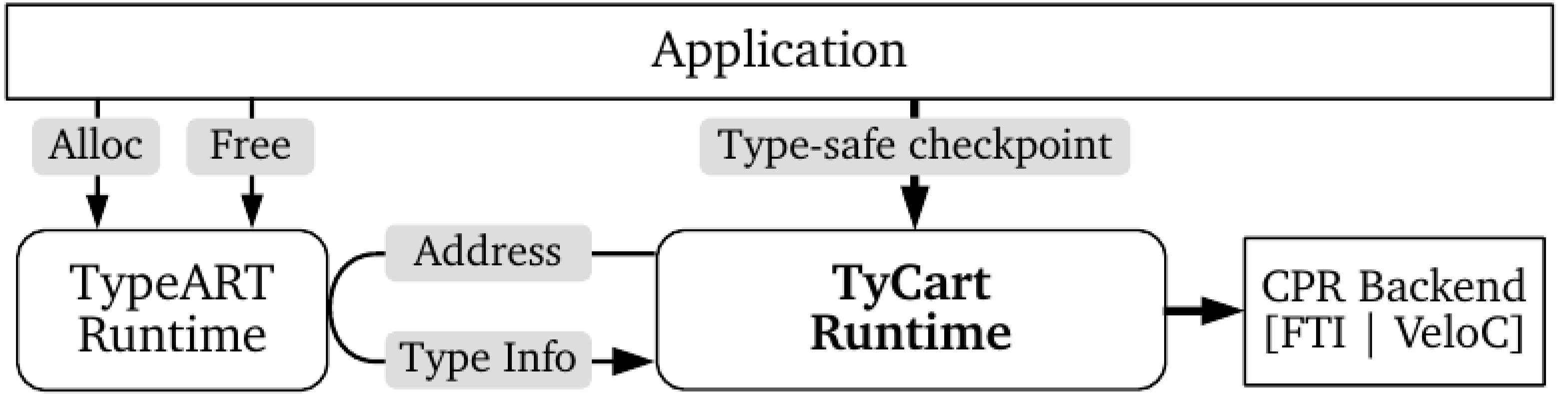
TypeART

```
%1 = call i8* @malloc(i64 %0) // %0 = n * sizeof(float)
%2 = udiv i64 %0, 4           // %2 = %0 / sizeof(float)
call void @__typeart_alloc( i8* %1,  i32 5,  i64 %2 )
%3 = bitcast i8* %1 to float*
```

Pointer Type id Extent

- Adds instrumentation to all relevant memory allocations, e.g., heap allocation
  - Generates type ids for each type in the program
- Allows to query the TypeART runtime with a memory address for:
  - Number of elements
  - Type of elements

TyCart



- Provides a CPR-library interface, similar to VeloC [2] and FTI [3]
- Implements runtime checks for specified type and number of elements
  - Uses TypeART as a service to provide required runtime type information

TyCart Interface

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double *pA = (double *) malloc(N * sizeof(double));
// memId: starts at pA with N elements of type double
TY_protect(memId, pA, N, double);
TY_checkpoint("CPLabel", CPId, CPVersion, CPLevel);
TY_unprotect(memId);
```

- **TY_protect**
  - memory id
  - pointer to start address of memory region
  - number of elements in memory region
  - type of an element in memory region

SC Scientific Computing

Type Assert

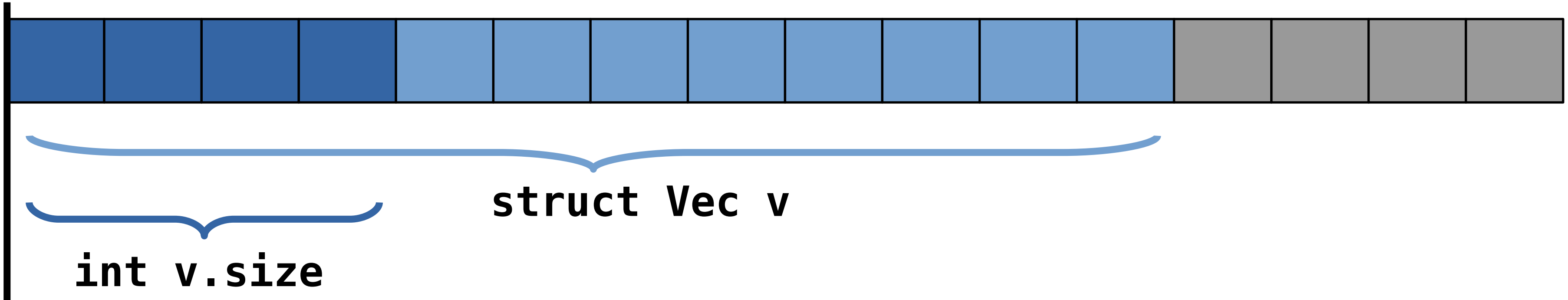
```
#define TY_protect(id, pointer, count, type)
{
  type* __ptr_ = NULL;
  __tycart_assert_stub((void*)pointer, __ptr_, count, id);
}
```

- Pointer of requested type is introduced and passed to stub function
- Stub function is replaced in LLVM compiler pass with call to TyCart runtime library function

```
void __tycart_assert(int id, void* addr, size_t count,
                     size_t typeSize, int typeId)
```

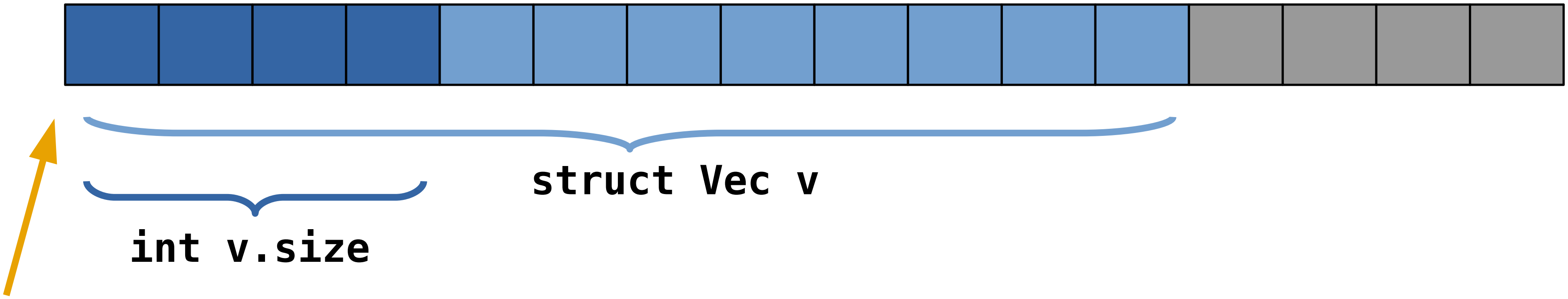
User-defined Types

```
struct Vec {int size; double *data;} v;
// All values are initialized correctly, register v.size
TY_protect(memId, &v.size, 1, int);
```



→ Start address of **v** equals member **size**, i.e., **&v == &v.size**

User-defined Types



assert 1 integer at address

1) assert fails (int != Vec)
2) get layout from Vec
3) assert 1 int with v.size
4) assert succeeds

```
assert weak (addr, type):
  if addr is not type:
    if type is user defined:
      subtype = get member type (type)
      return assert weak (addr, subtype)
  return assert(addr, type)
```

Evaluation

Experiments conducted on Lichtenberg high-performance computer (TU Darmstadt)

- Intel Haswell E5-2680v3 (fixed @ 2.5GHz), 64 GB main memory
- Results denote median over 10 consecutive runs
  - standard deviation is 3% or less except FTI+TyCart in driven cavity

- driven cavity: C++ adoption from MINPACK-2 collection
- eos-mbpt: C++ (astro)physics simulation [4]
- game of life: C++ implementation of Conway's game of life
- heatdis: C example from FTI repository (MPI-parallel)
- LULESH: C++ mini app for shock hydro dynamics (MPI-parallel)

Evaluation

Table 1: Compile-time: number of instrumented heap allocations and frees; global and stack variables (percentage filtered). Checkpoint: allocations registered for checkpointing; file size per checkpoint per MPI process.

	Compile Time			Checkpoint	
Benchmark	Heap	Globals (%)	Stack (%)	Regions	Size (MB)
driven cavity	15 / 32	1 (93)	17 (6)	10,003	764.0
eos-mbpt	482 / 160	203 (68)	549 (21)	7,845	6.7
game of life	12 / 28	1 (94)	7 (46)	4	48.0
heatdis	14 / 30	2 (89)	18 (0)	5	129.0
LULESH	14 / 30	6 (91)	39 (38)	57	7.0

Evaluation

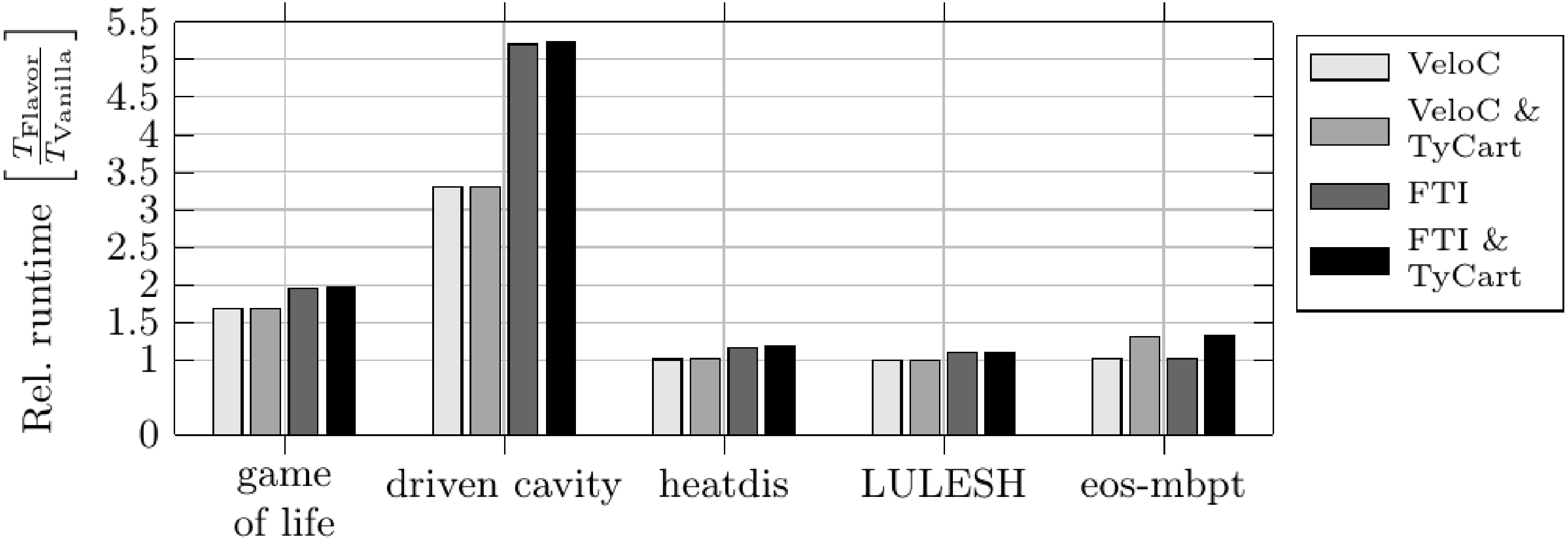


Fig. 3: Runtime overhead w.r.t. vanilla. Vanilla runtime: (1) game of life: 34.08s, (2) driven cavity: 88.61s, (3) heatdis: 206.30s, (4) LULESH 2.0: 70.69s, (5) eos-mbpt: 1,462.3s.

Evaluation

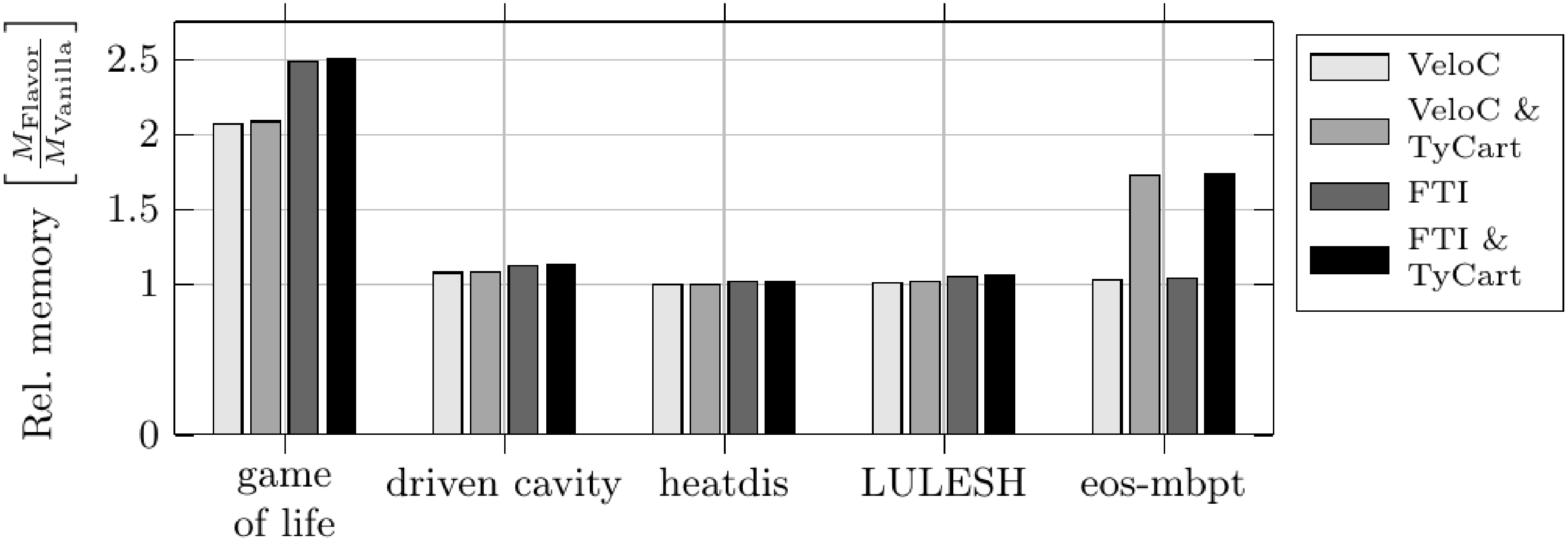


Fig. 4: Memory overhead w.r.t. vanilla. Vanilla RSS: (1) game of life: 52MB, (2) driven cavity: 764MB, (3) heatdis: 315MB, (4) LULESH 2.0: 80MB, (5) eos-mbpt: 1,827MB.

Evaluation

Table 2: Total executed instrumentation calls for heap and stack allocations; information-tracking memory consumption as computed by the TypeART runtime; maximum number of allocations tracked simultaneously.

Benchmark	TyCart Runtime				
	Tot. Heap	Tot. Stack	Mem. (KiB)	Max. Heap	Max. Stack
driven cavity	10,003	23	782.4	10,003	23
eos-mbpt	32,508,427	48,751,262	1,270,170.4	16,257,906	250
game of life	2	6	0.6	2	6
heatdis	3	30,012	0.8	3	15
LULESH	406,261	44,714	6.9	77	23

Discussion

- Allows to effectively check validity of stated type requirements for a CPR call
  - Runtime and memory overhead within reasonable margins
- Works with C and C++

- Currently, does not support incremental checkpointing
  - Prevents partial initialization of memory
- Currently, handling of user-defined types offers limited configurability for resolution
  - Introduce resolution-level to specify particular occurrence of sub type
- Restart not handled explicitly

  → potentially exploit meta data in CPR files to also check at application restart

Conclusion

- TyCart is a tool for type-safe checkpoint/restart built on top of TypeART
  - Implementation exists for the FTI and VeloC
- Implements type asserts for C and C++
  - Specify requirements on memory regions, i.e., type and number of elements
- Introduces reasonable runtime and memory overhead
  - Improving compile-time filtering should reduce overheads further

Available under **BSD 3-clause license** (branch feat/tycart)

github.com/tudasc/TypeART

References

- [1] A. Hück et al., “Compiler-aided Type Tracking for Correctness Checking of MPI Applications”, 2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness), Dallas, TX, USA, 2018, pp. 51-58. doi: 10.1109/Correctness.2018.00011.
- [2] B. Nicolae et al., “VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale”, 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, 2019, pp. 911-920. doi: 10.1109/IPDPS.2019.00099.
- [3] Leonardo Bautista-Gomez et al., “FTI: high performance fault tolerance interface for hybrid systems”, 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’11). Association for Computing Machinery, New York, NY, USA, Article 32, 1–32. doi: 10.1145/2063384.2063427.
- [4] C. Drischler et al., “Chiral interactions up to next-to-next-to-next-to-leading order and nuclear saturation”, Physical Review Letters 122, 042501 (Jan 2019). doi: 10.1103/PhysRevLett.122.042501.