# M4Secure

## Faster and Secure Memory Management

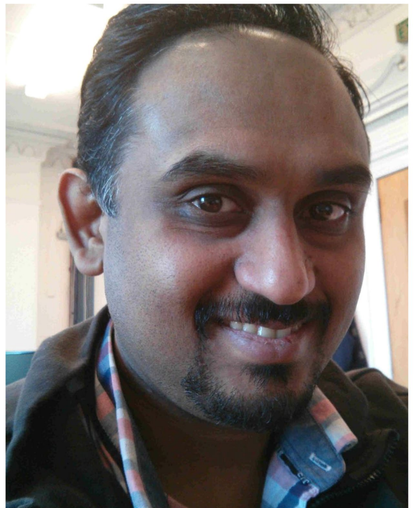



Engineering and Physical Sciences Research Council

Jeremy Singer

Alice Miller

Zheng Wang



Dejice Jacob

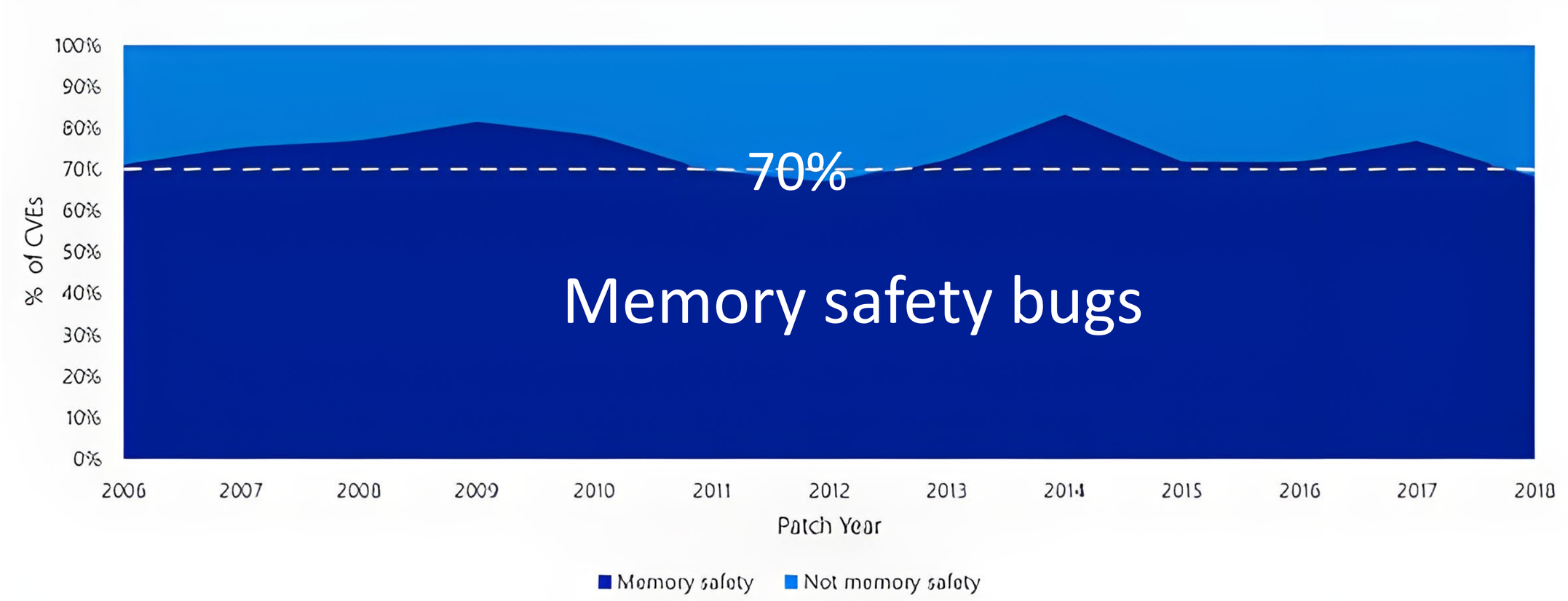
Xiaoyang Sun

## Project Partners

**Microsoft**

**Meta**

# Memory related bugs are serious issues




~70% of the vulnerabilities  found by Microsoft and in Google Chrome web browser are memory safety issues

https://www.chromium.org/Home/chromium-security/memory-safety

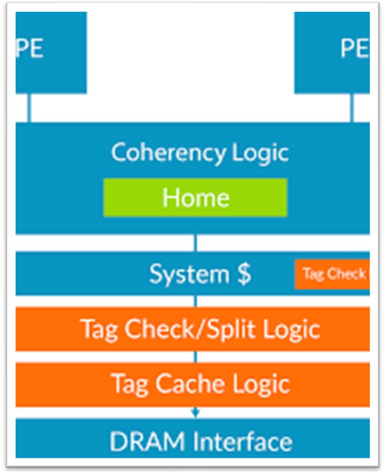
# Memory allocators must also be fast

Hundreds of millions of `malloc` calls per second in typical data-intensive workloads

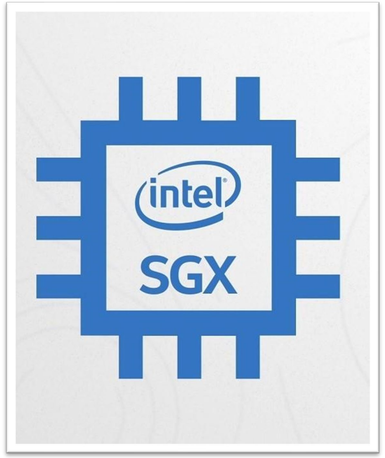
4.9 billions of `malloc` calls in 7.9 seconds for a binary tree benchmark on M1 macbook

# Customised memory allocators for hardware and applications

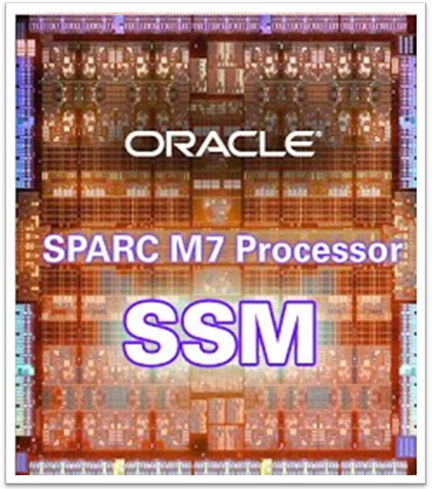

CHERI Capabilities




Arm memory tag extension




Intel Software Guard Extensions




Oracle Silicon Secured Memory

APACHE HTTP SERVER PROJECT

MEMCACHED

Firefox OS

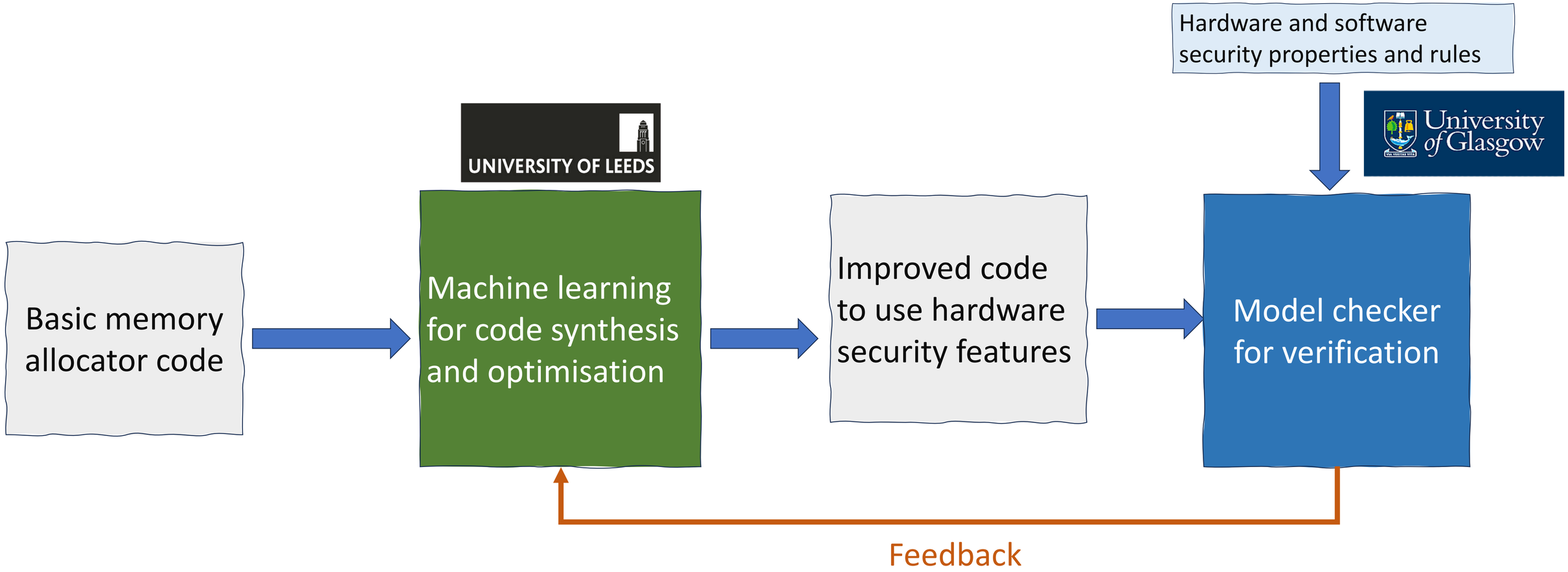
# But memory allocators are expensive to build

## e.g. Jemalloc would cost £2.1m to develop

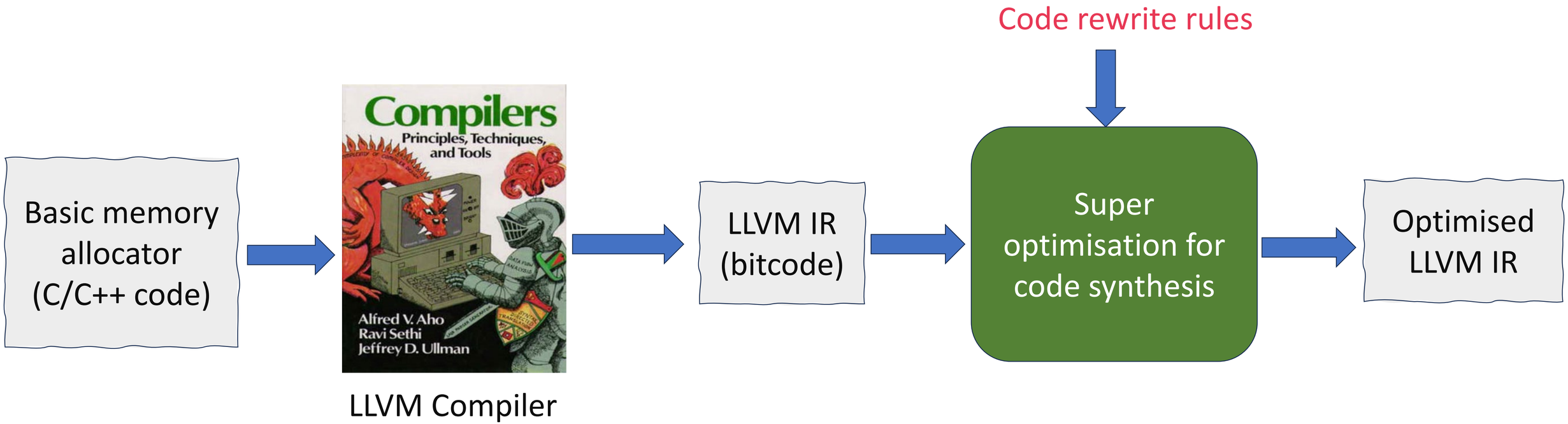
Estimate based David A. Wheeler's SLOCCount estimation tool: https://dwheeler.com/sloccount/ and a typical UK software engineer salary by Glassdoor




# M4Secure – making it easier to develop fast, correct and secure memory management libraries



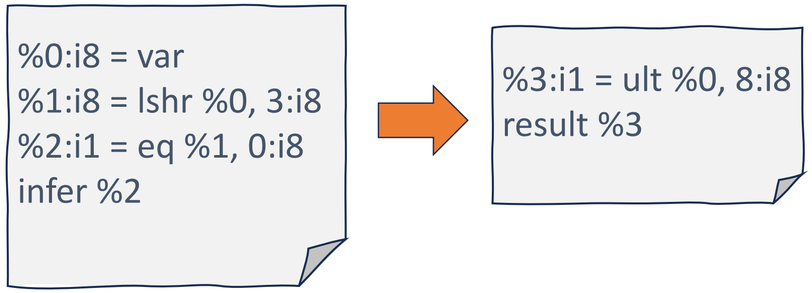
# Machine learning for fast memory allocators

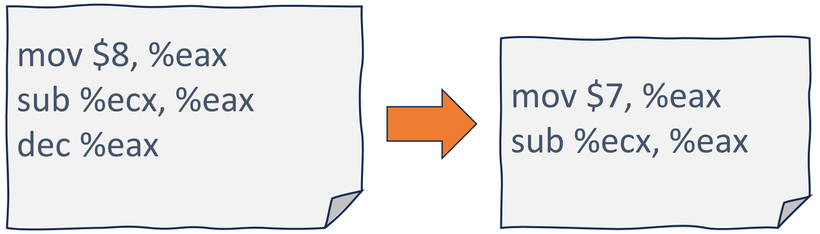



LLVM Compiler

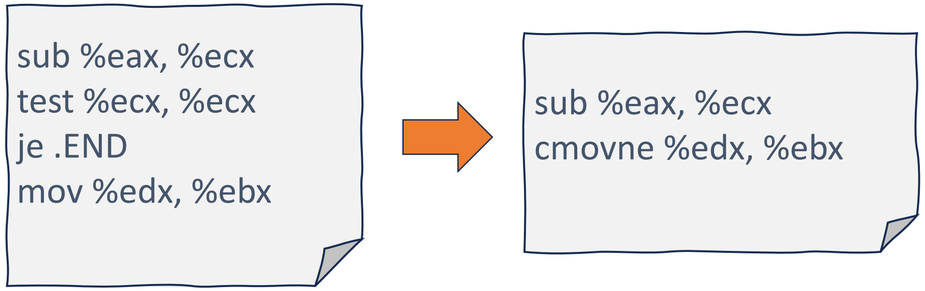
Leeds is developing a super optimiser with mutation rules designed for Cheri (ARM Morello) and LLVM IR

# Super optimisation examples



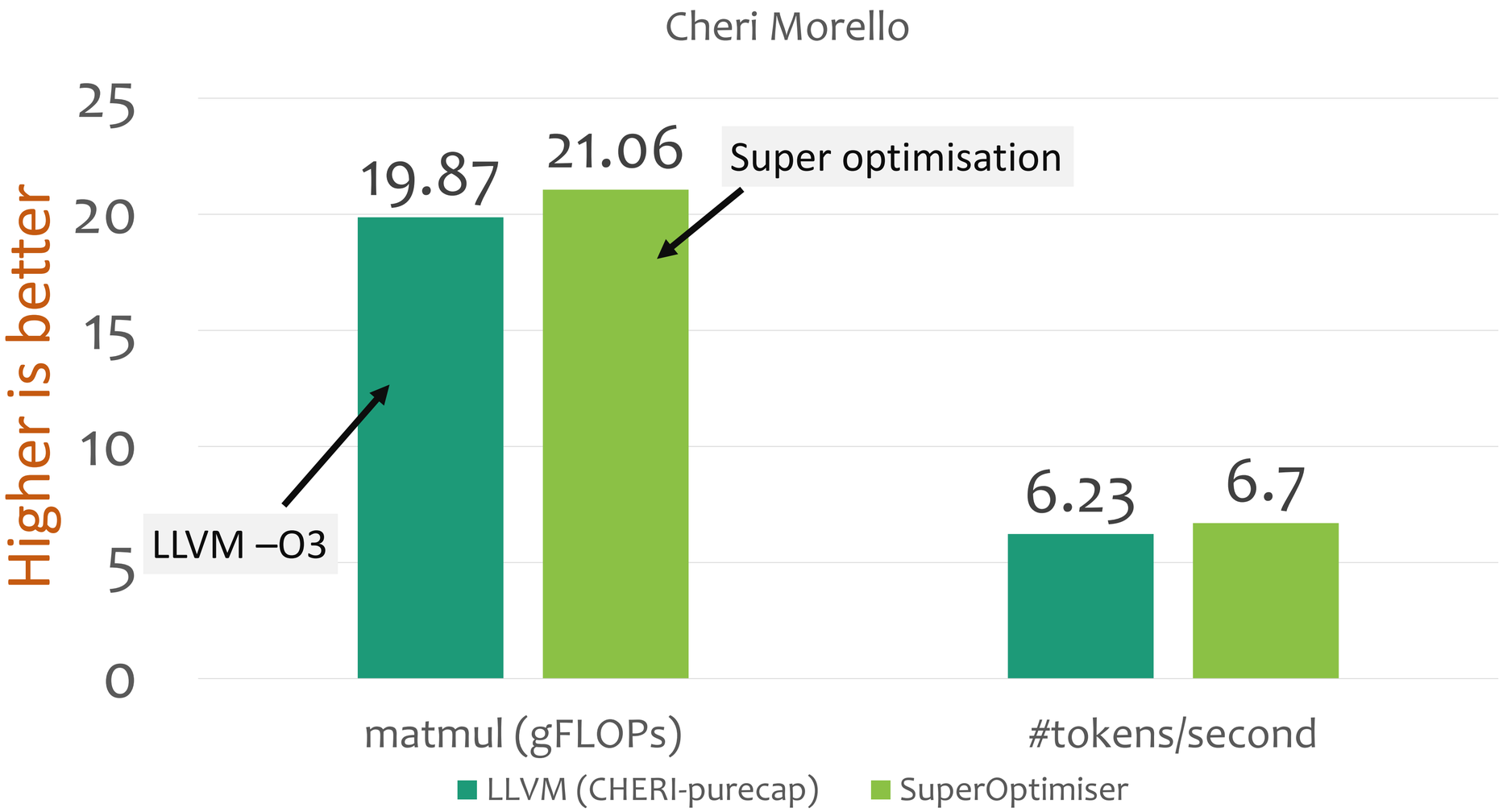




constant folding



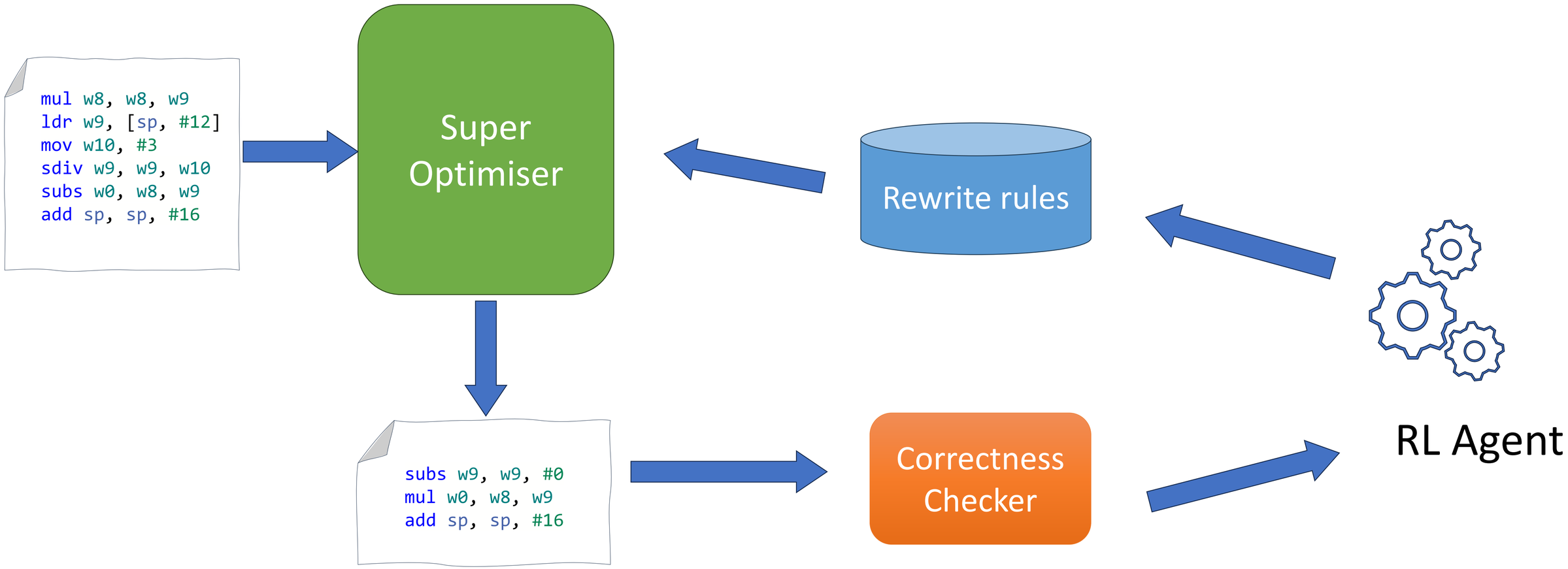

Branch elimination

# Super optimisation for LLaMA large language model




5% performance improvement without changing the user code!

# Reinforcement learning (RL) to drive super optimisation




Extend super optimisation to higher levels, e.g. C code, algorithms and allocation strategies

# Security properties for memory allocators

- e.g. Linear Temporal Logic (LTL) property:

$$\square((\diamond q) \rightarrow ((!qUp)\&\&(!rUq)\&\&(\diamond r)))$$

- Where q is **`client_access_mem`**, p is **`malloc`** and r is **`free`**.
- *"if in the future q will happen, then p must occur before q, and r must occur eventually, but not before q "*

# Model checking to verify security properties

- Develop domain specific language to generate models on the fly
- Labels in code to mark the functionalities
  - E.g. mem_allocate_start/end to trigger relevant flags

- Currently use SPIN or stripped-down version of nested depth-first search for LTL model checking
  - and support realtime verification of safety properties

# Cheri benchmarks for memory allocators

- 10 allocation-intensive C benchmarks
- Regular and irregular allocation patterns
- https://github.com/glasgowPLI/alloc-bench

```
130  struct batch* dequeue_batch() {
131    pthread_mutex_lock(&lock);
132    while (batches == NULL && !atomic_load(&done_flag)) {
133      pthread_cond_wait(&empty_cv, &lock);
134    }
135    struct batch* result = batches;
136    if (result) {
137      batches = result->next_batch;
138      batch_count--;
139      pthread_cond_signal(&full_cv);
140    }
141    pthread_mutex_unlock(&lock);
142    return result;
143  }
144
145  void *mem_allocator (void *arg) {
146    int thread_id = *(int *)arg;
147    struct lran2_st lr;
148    lran2_init(&lr, thread_id);
149
150    while (!atomic_load(&done_flag)) {
151      struct batch *b = xmalloc(sizeof(*b));
152      for (int i = 0; i < OBJECTS_PER_BATCH; i++) {
153        size_t siz = object_size > 0 ? object_size : possible_sizes[lran2(&lr)%n_sizes];
154        b->objects[i] = xmalloc(siz);
155        memset(b->objects[i],i%256,(siz > 128 ? 128 : siz));
156      }
157      enqueue_batch(b);
158    }
159    return NULL;
160  }
```

# Conclusions

- It is high time to make memory management secure while high-performant and correct
- **ML** + **Formal Verification** for fast, secure, and correct memory allocators
  - Should generalise beyond CHERI
- Lots of opportunities for collaboration