PLOS 2021: 11th Workshop on Programming Languages and Operating Systems

Asterope: A Cross-Platform Optimization Method for Fast Memory Copy

Giorgos Kappes, Stergios V. Anastasiadis

University of Ioannina, Ioannina 45110, Greece

Memory copy

Memory copy (memcpy) is used in every software stack

- Applications, libraries, language runtime, operating-system kernel

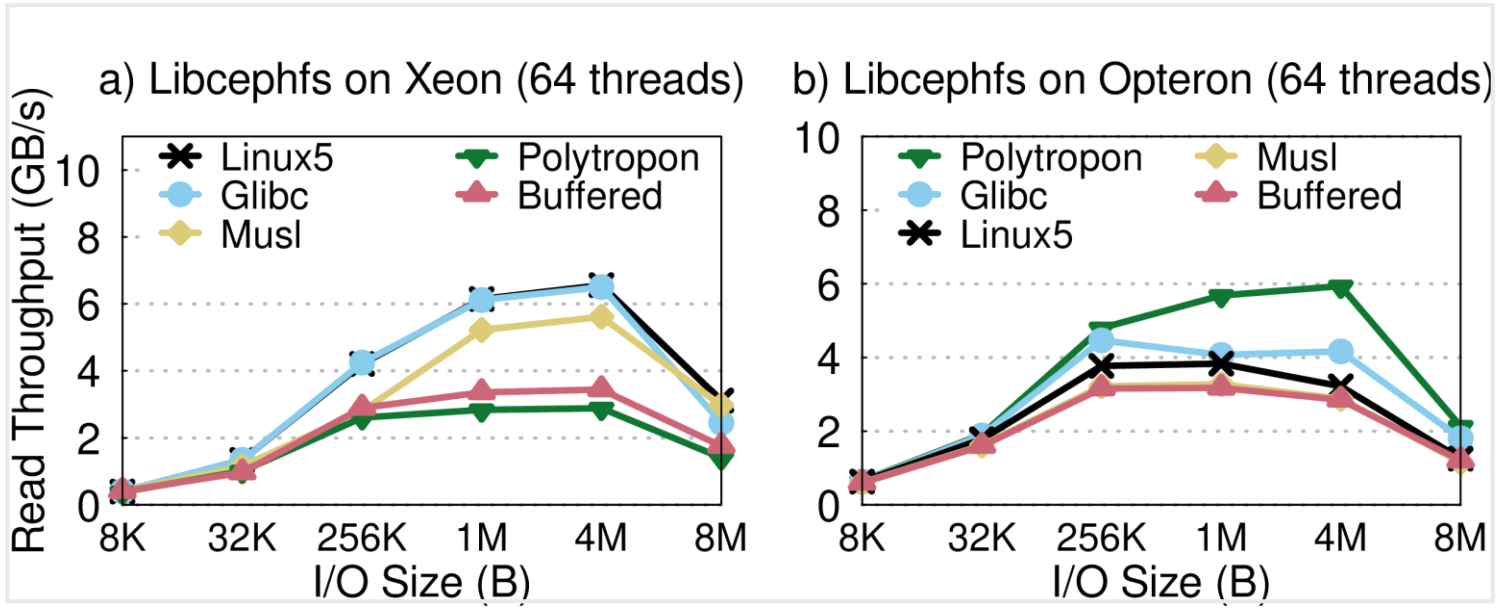
Copy performance is critical

- Used in critical path of application-kernel interaction, inter-process communication, device access

Memory copy is resource intensive & sensitive to system parameters

- Numerous optimizations developed over the years
- Software routines, compiler intrinsic functions, architecture-specific hardware instructions
- Sensitive to: Hardware configuration, system settings, colocated workloads

Performance of various memcpy routines



Cached read-only workload: read I/O of dataset cached at client side

- Alternative memcpy routines at client of distributed filesystem
- E.g., Polytropon (SoCC 2020), Buffered (transfer through memory Buffer, Intel 2009)

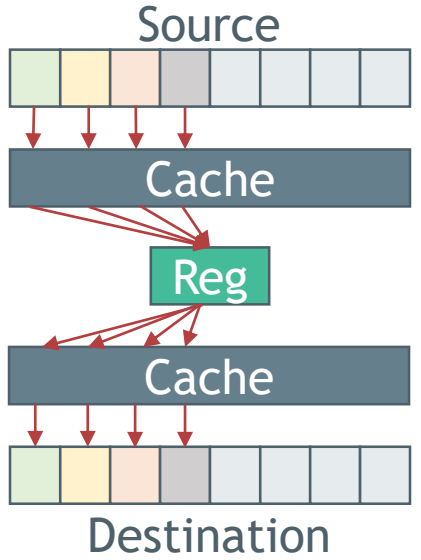
Copy performance is platform-dependent

- Polytropon memcpy achieves best performance on Opteron but not on Xeon
- Performance of Glibc memcpy drops on Opteron

Simplest memory copy

Load & Store, Byte-by-Byte

```
void *memcpy(void *dest, void const *src, size_t len)
{
        char *_dest = (char *) dest;
        char const *_src = (char const *) src;
        while (len--)
                *_dest++ = *_src++;
        return dest;
}
```



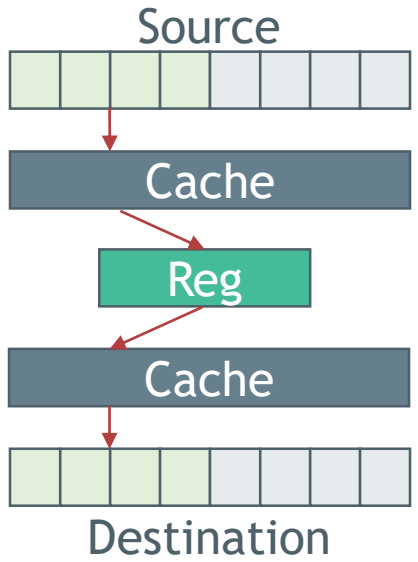
What is wrong with the above function?

1. Does not employ single instruction, multiple data (SIMD)
2. Has no explicit instruction-level parallelism
3. Does not prevent possible cache pollution
4. Does not optimize memory loads with software prefetching

Optimization #1: Maximize memory bandwidth

SIMD Instructions

- Larger register sizes
- Move batches of data between registers & memory
- Multiple loads/stores per cycle
- Fewer instructions to complete a transfer
- Larger instruction opcodes



Examples

- MMX, SSE, SSE2, SSE4.1, AVX, AVX512

Restrictions

- Platform-specific
- Architectural power restrictions may lower core clock frequency

Optimization #2: Do not pollute the cache

Streaming Instructions

- Non-temporal loads/stores that bypass (some) caches
- Reduce cache pollution
- Reduce cache-coherence traffic

Non-temporal loads

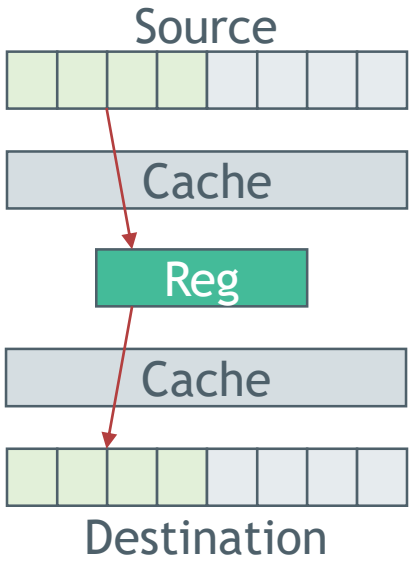
- Use temporal internal buffers (e.g., Line Fill Buffers/LFBs) to transfer data

Non-temporal stores

- Use write-combining, weakly-ordered, uncachable, non-write-allocate operations to transfer data

Restrictions

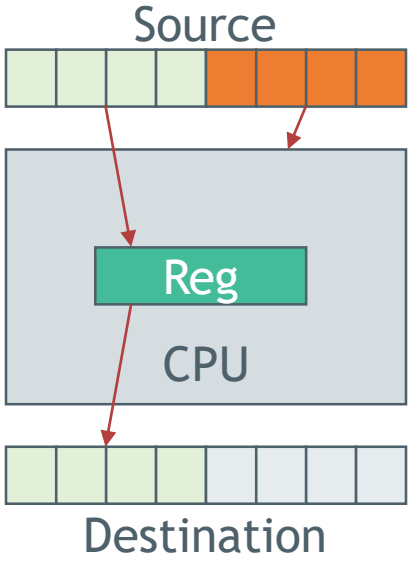
- Performance benefit depends on architecture



Optimization #3: Optimize load performance

Hardware prefetching

- Automatically initiated when cpu detects predictable access pattern at the cache level
- Restrictions of hardware prefetching
  - Cache type (instruction/data) & cache level
  - Cache misses needed to trigger prefetching
  - Prefetched data located in single page



Software prefetching

- Special instructions hint data transfer from memory to processor in advance without register involvement
- Restrictions of software prefetching
  - Performance benefits of prefetch parameters vary across systems

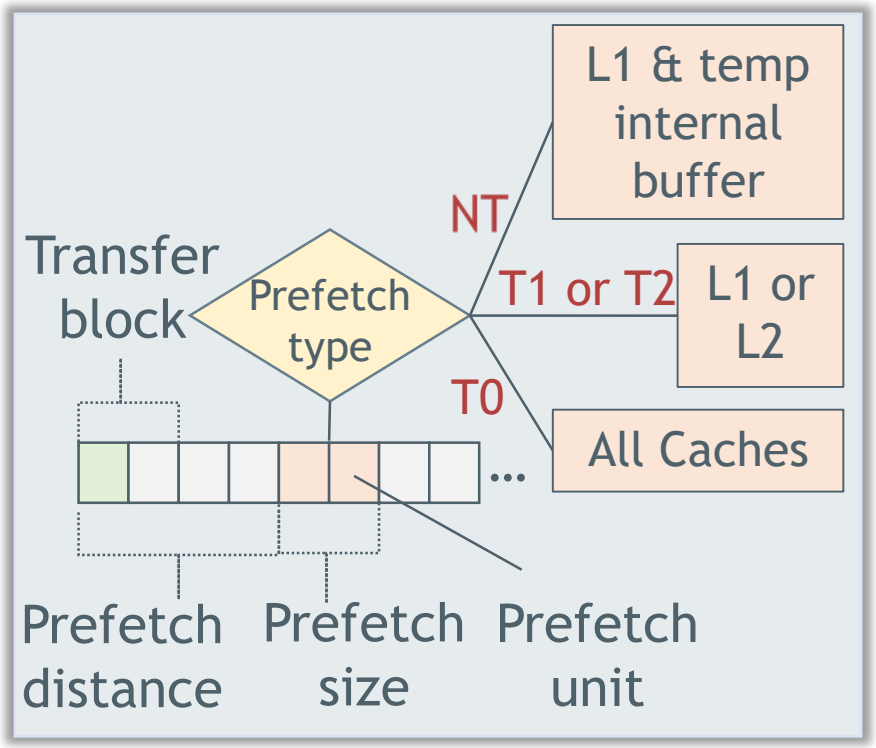
Parameters of software prefetching

Transfer block

- Data transferred per memcpy iteration (2 cache lines of 64B each)

Prefetch type

- Temporal (T0/1/2): Hint data transfer to cpu caches
- Non-temporal (NT): Hint data transfer to L1 & cpu temporary internal buffers, limit cache pollution



Prefetch unit

- Data fetched by single instruction (1 cache line)

Prefetch size

- Data fetched by a sequence of instructions

Prefetch distance

- Length in bytes by which data is requested ahead of its actual load

Problems

Large design space for memory copy optimization

- Register type & size
- Temporal or non-temporal instructions
- Prefetch type, size & distance

Optimization effectiveness depends on the platform

- E.g., number of Line Fill Buffers (LFBs), frequency scaling

Plethora of custom-crafted memory copy routines

- Their performance varies significantly on different platforms
- Systems can use runtime dispatching techniques to select a particular memory copy function during initialization but they lack automatic adaptation at finer-grain parameters

Fast memory copy with Asterope

Goal

- Automatically identify memcpy settings for max performance per transfer size across different platforms

Solution: Apply exhaustive search over low-level system parameters

- Automate optimizations #1, #2, #3
- Run once offline at initial system setup
- Run again on major system modifications (e.g., memory upgrade)

Explored system parameters

- Registers, instructions, prefetching

Data transfer pipeline

- Prefetch stage: Software-prefetch data up to prefetch size
- Load stage: Load a block of data from source memory into registers
- Store stage: Store a block of data from registers to target memory

Explored parameters

Registers

- RSI/RDI (1B), MM (8B), XMM (16B) YMM (32B), ZMM (64B)

Load Instructions (x86 & SIMD)

- Non-temporal: requires custom kernel module to activate write-combining memory (movntdqa, vmovntdqa)
- Temporal (rep movsq, movq, movaps, vmoaps)

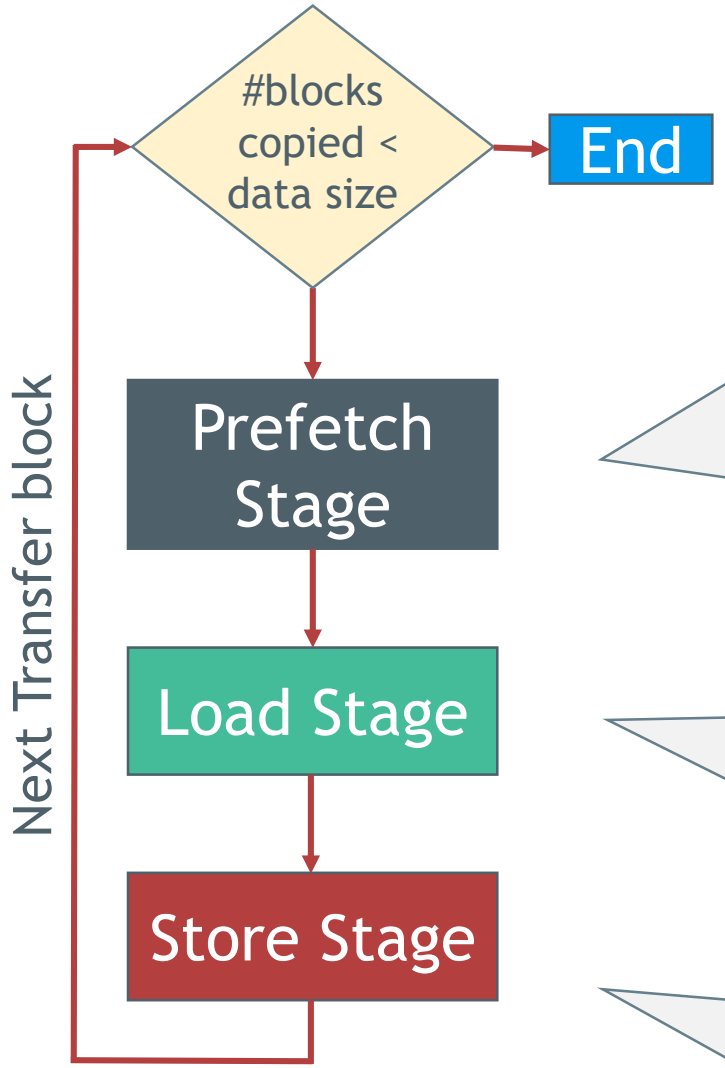
Store Instructions (x86 & SIMD)

- Non-Temporal (movntdq, movntps, vmovntps)
- Temporal (rep movsq, movq, movaps, vmovaps)

Prefetching

- Temporal (T0, T1, T2), Non-temporal (NT)
- Size & distance

Data transfer pipeline



Goal: Prefetch data ahead of load but not too early to risk replacement before store
Parameters: Prefetch Type (T0, T1, NT), Prefetch Size, Prefetch Distance

Goal: Load next block to CPU registers
Parameters: Register type (RSI/RDI, MM, XMM, YMM, ZMM), Load type (Temporal, Non-Temporal)

Goal: Store next block to destination memory
Parameters: Register type (RSI/RDI, MM, XMM, YMM, ZMM), Store type (Temporal, Non-Temporal)

Asterope optimization algorithm

Search for (ps, pd, pt, lt, st, rt) with max memcpy performance per transfer size

```
// to: source, from: destination, xfsz[]: transfer sizes
For each transfer size xi (index to xfsz[])
  For each prefetch size ps to Max Prefetch Size (blocks)
    For each prefetch distance pd to Max Prefetch Distance (blocks)
      For each prefetch type pt
        For each load type lt
          For each store type st
            For each register type rt
              Clear cpu caches
              Record memcpy duration
              For each block bi of transfer size xfsz[xi]
                For each block pi of ps // if !(bi mod ps)
                   blkmemprf(from + (pd+pi) * BSZ, pt)
                blkmemcpy(to, from, lt, st, rt);
                // Next transfer block
                from += BSZ
                to += BSZ
              Fence for non-temporal store
```

Examined parameters

Prefetch stage

Load & Store stages

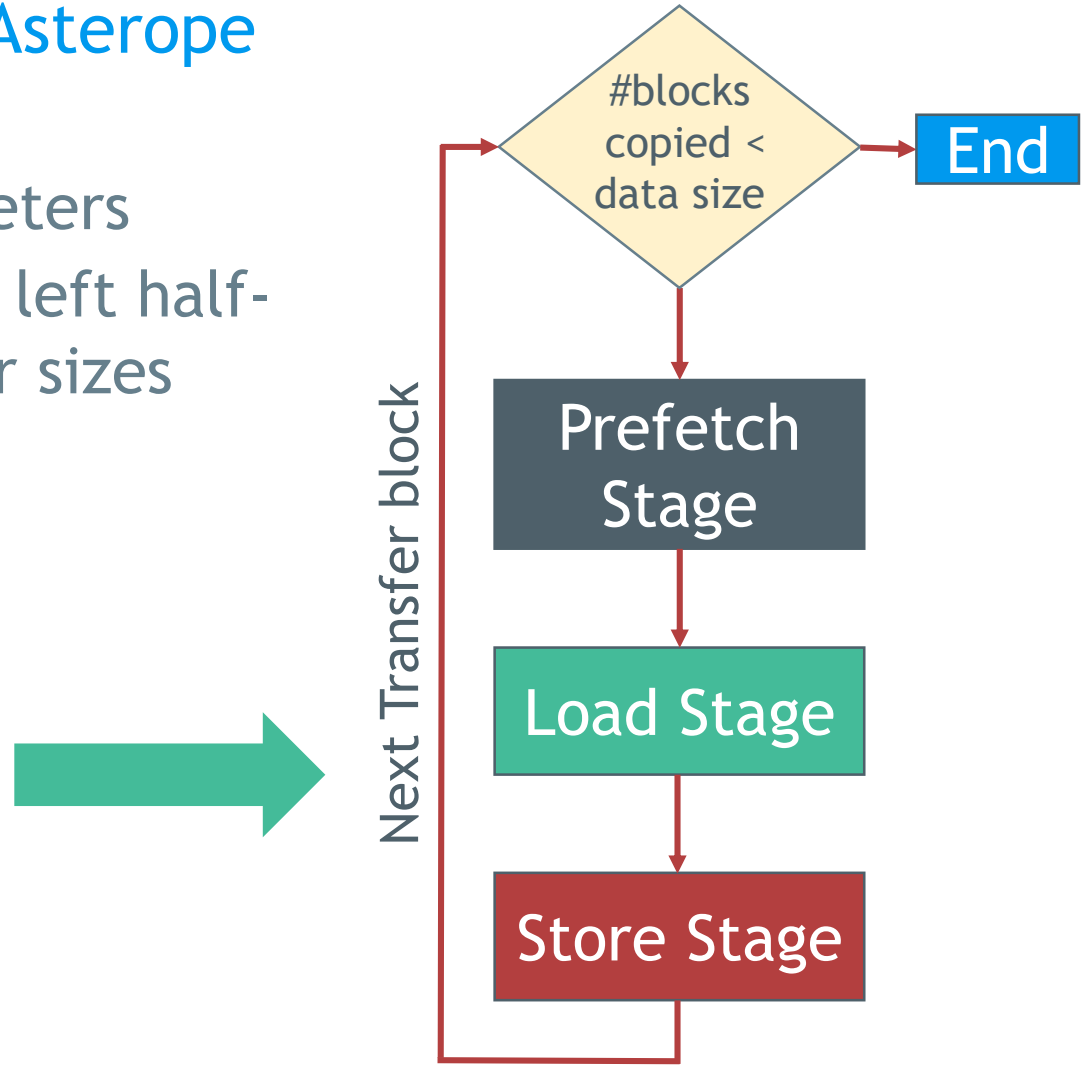
Copy xfsz[xi] Bytes from source to destination

Asterope output

Optimized memcpy routine produced by Asterope

- Common 3-stage copy pipeline
- Per platform optimized memcpy parameters
- Domain of transfer sizes: partitioned to left half-open intervals by the evaluated transfer sizes
- Handles unaligned memory addresses

Size	RT	LT	ST	PT	PS	PD
128B	MMX	T	NT	T0	2	0
8KB	YMM	T	T	T1	10	0
256KB	XMM	T	T	T0	8	30
1MB	ZMM	T	T	NT	4	28
4MB	XMM	T	T	NT	2	20
32MB	MMX	T	T	NT	2	20



Experimentation environment

Machine A: 2 x Intel Xeon Gold 5218

- 128GB RAM, 32 physical cores (64 HTs)
- 88.9ns Measured Latency
- Per socket
  - 1MB L1, 16MB L2, 22MB L3
  - 2 Memory Controllers, 6 Channels
  - 19.87GB/s BW/Channel

Machine B: 4 x AMD Opteron 6378

- 256GB RAM, 64 physical cores
- 84.7ns Measured Latency
- Per socket
  - 768KB L1, 16MB L2, 16MB L3
  - 1 Memory Controller, 4 Channels
  - 12.8GB/s BW/Channel

Automatic Settings by Asterope (Xeon)

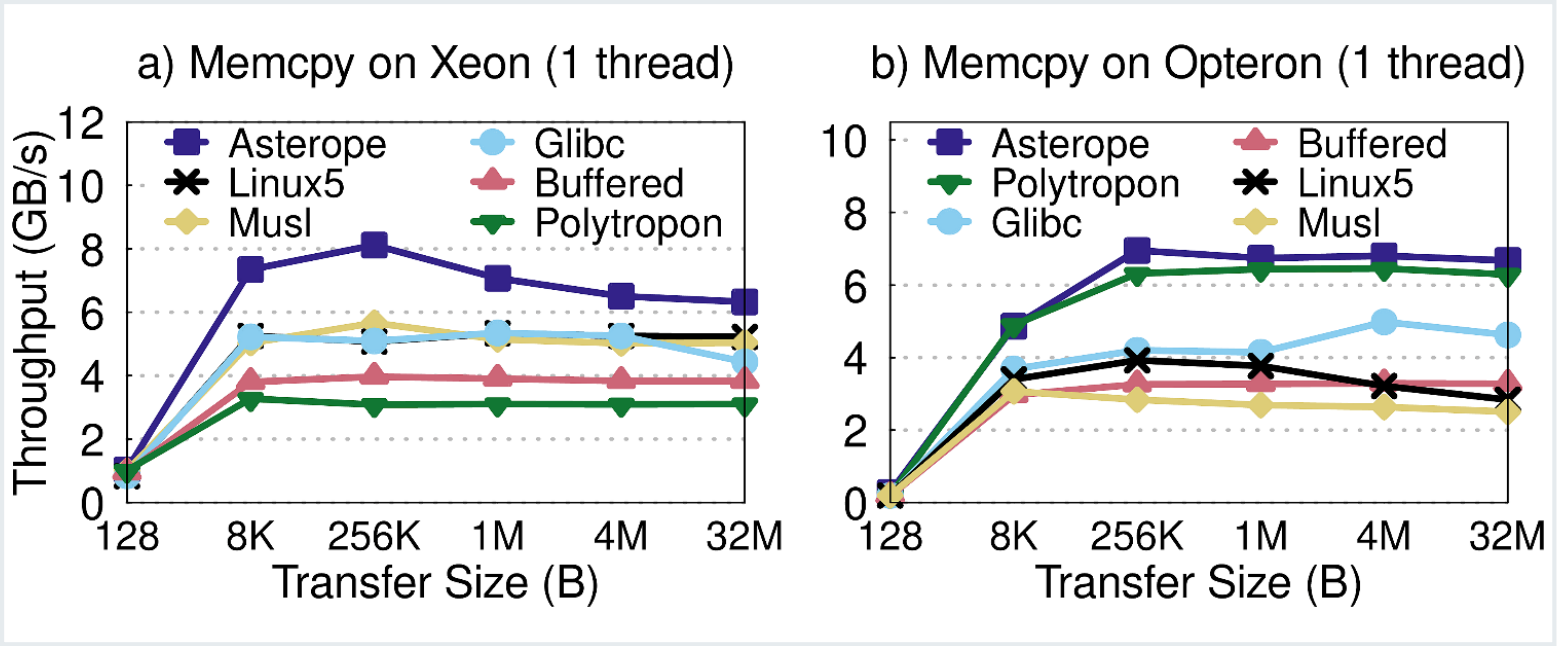
Size	RT	LT	ST	PT	PS	PD	GB/s
128B	MMX	T	NT	T0	2	0	1.01
8KB	YMM	T	T	T1	14	0	7.17
256KB	XMM	T	T	T0	4	30	7.93
1MB	MMX	T	T	T0	2	28	6.9
4MB	MMX	T	T	T0	2	28	6.36
32MB	MMX	T	T	T0	2	30	6.18

Automatic Settings by Asterope (Opteron)

128B	MMX	T	T	T0	2	0	0.27
8KB	XMM	T	NT	NT	2	12	4.75
256KB	XMM	T	NT	NT	2	30	6.79
1MB	XMM	T	NT	NT	2	30	6.58
4MB	XMM	T	NT	NT	4	28	6.65
32MB	XMM	T	NT	NT	2	20	6.6

Register, instruction & Prefetch parameters of Asterope for max memcpy performance

Microbenchmark memory copy performance



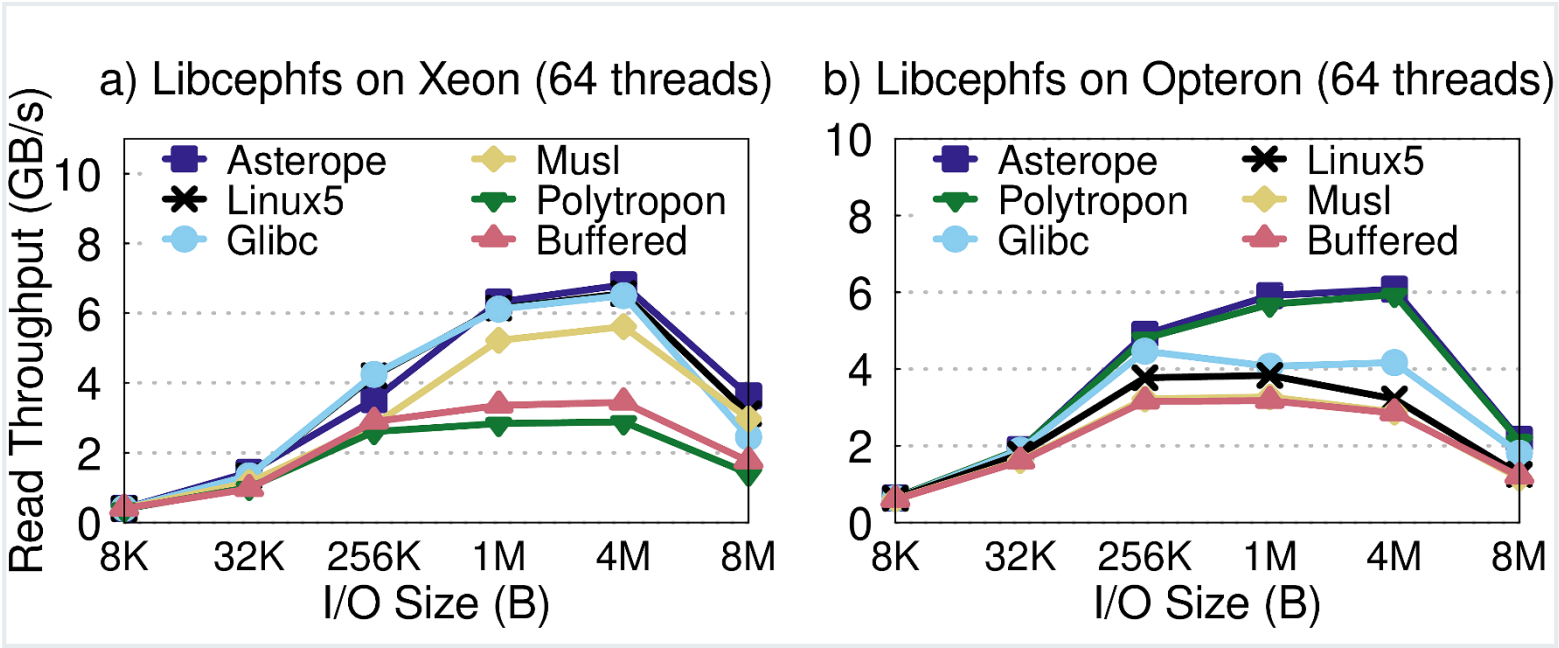
Asterope is the fastest routine in both systems

- Up to 1.6x faster than Glibc & Linux5 (Xeon)
- Up to 1.7x faster than Glibc, up to 1.5x faster than Polytropon (Opteron)

Ceph client performance

Ceph distributed filesystem (v10.2.7)

- Modified memcpy routine at client between application buffers and cache
- Read-only workload that fits in client cache



Asterope tracks or improves the performance of the fastest memory copy routine in Xeon and Opteron

- Up to 1.5x faster than the default Glibc memcpy in both systems
- Up to 1.9x faster than Linux5 memcpy in both systems

Conclusions

Multiple memcpy versions that optimize copy performance

- Design space for memcpy optimization too large
- Distinct routine per CPU, lack automatic adaptation at finer grain
- Performance sensitivity to system instructions & parameters

Asterope: Automatic cross-platform optimization for fast memcpy

- 3-stage copy routine: prefetch, load, store
- Exhaustive search to find parameters that optimize copy performance per transfer size on specific platform
- Optimization conducted once for a specific platform configuration

Result

- Asterope memcpy tracks or increases the fastest performance achieved by existing routines across different platforms

Future work

Apply Asterope across a range of performance-critical functions

- Dependent on features of CPU, memory, interconnect, devices
- E.g., data load, caching, communication, list iteration, string/stream processing

Explore more system parameters

- E.g., kernel/userspace, mode/context switching, caching, isolation

Optimize parameter exploration

- The one-time offline search takes several hours
- Approach 1: Dynamically explore performance for different parameters as the system serves data transfers during normal operation
- Approach: Use model-based learning to approximate memcpy performance per parameter tuple