PEACH: Data Allocation and Movement using a PErformance Aware Compiler framework for Heterogeneous memory system

[1]Onkar Patil, [2]Latchesar Ionkov, [2]Jason Lee, [1]Frank Mueller, [2]Michael Lang

[1]Dept. of Computer Science, North Carolina State University

[2]Ultrascale Research Center, Los Alamos National Laboratory

Problem Statement

- Heterogeneous Memory Systems
  - HBM-DRAM, DRAM-NVM, HBM-DRAM-NVM, etc.
  - Local and Remote NUMA regions
  - Performance, Power and Capacity varies
- Data Allocation and Movement
  - Not a trivial problem
  - HPC applications have 100-1000s data structures
  - Data Structures can scale up to TBs or PBs
  - Too many factors to be considered by application programmers to make optimal decisions
    - Balance compute, memory and communication resources while maintaining a power budget

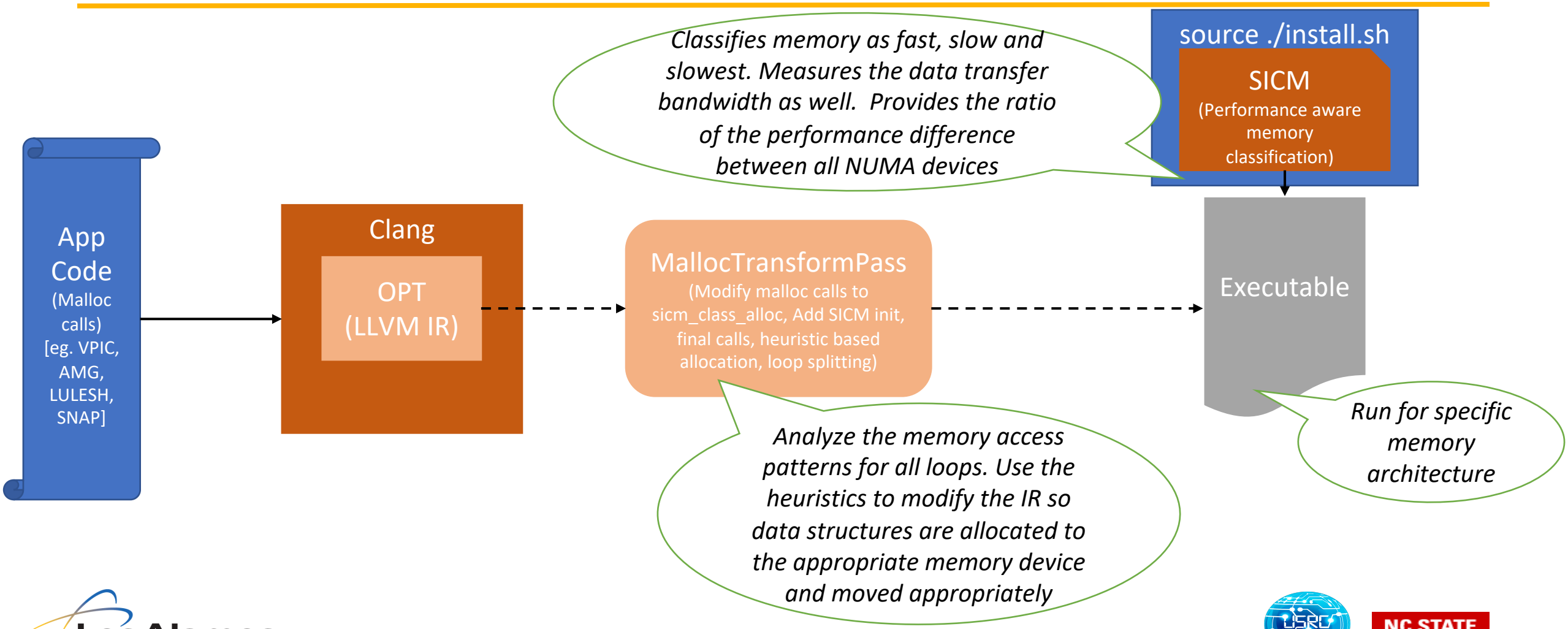
Problem Statement

- Compiler and Runtime frameworks
  - Work in tandem to provide user-level Memory Management
  - Mostly designed for Homogeneous NUMA memory systems
  - Focused primarily on reducing fragmentation, memory leaks and memory corruption

Hypothesis

- The onus of allocating memory for an application on a heterogeneous memory system should move from the programmer to the compiler
- As heterogeneous memory systems become more prominent, compiler and runtime frameworks need to be more performance and capacity aware of the underlying memory devices to ensure optimal performance of applications

PEACH: Compiler Framework



Simple Interface for Complex Memory (SICM)

- Data allocation and movement enabling API library
- Uses jemalloc underneath
- Extends the arena based allocation concepts of jemalloc to heterogenous memory system

Performance Awareness

- SICM identified NUMA devices but did not have any information about performance (Memory Bandwidth)
- Extended SICM to classify NUMA devices based on performance
  - At installation, install.sh runs a set of performance benchmarks that classify each NUMA device as fast, slow, slowest for each CPU/GPU group
    - Uses a Wr-only and single write and multiple read benchmark to measure bandwidth
    - Uses a simple k-means clustering to classify the NUMA devices
  - Also measures data movement bandwidth
  - Installation time increases by 45-75 seconds depending on the no. of NUMA devices
- Classification is read at runtime from a file
- Extended the SICM low-level API to incorporate the performance classification
  - Modified sicm_init() and sicm_fini() to sicm_class_init() and sicm_class_fini()
  - Added sicm_class_alloc(), sicm_class_realloc(), sicm_class_move() and sicm_class_free()

Performance Awareness

- NUMA Classification file
  - sicm_config
  - Sourced into runtime using an environment variable

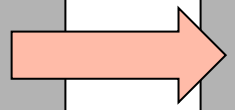
```
NUMA Device classification
CPU ID  NUMA ID Type    Init(Mb/s)      Triad(Mb/s)
72-95   1       fast    83665.204771    67649.395964
72-95   0       slow    41630.203821    33573.023831
72-95   3       slowest 4073.821638     6189.551163
72-95   2       slowest 1220.172135     810.849809
CPU ID  NUMA ID Type    Init(Mb/s)      Triad(Mb/s)
48-71   0       fast    80094.647213    69189.514867
48-71   1       slow    44094.883771    34690.394709
48-71   2       slowest 4138.825899     6079.981850
48-71   3       slowest 1202.866903     799.498232
…
Page Migration measurement
CPU ID  SRC     DEST    PgMigration(Mb/s)
72-95   0       1       17733886.936360
72-95   0       2       35102711.638282
72-95   0       3       35168571.209754
72-95   1       0       19843487.223887
72-95   1       2       35407006.497107
72-95   1       3       35253389.253937
72-95   2       0       9014960.473411
72-95   2       1       36200631.351450
72-95   2       3       40998997.652807
72-95   3       0       13322124.177216
72-95   3       1       34191903.297198
72-95   3       2       33858166.176772
…
```

AllocTransform Pass

- LLVM IR-level pass
  - Completely integrated into clang/LLVM pipeline (load using –Xclang), SICM linked dynamically
  - Inserts sicm_class_init(), sicm_class_fini() at the entry and return of main()
  - malloc(size) → sicm_class_alloc(speed, size) where speed(0/1/2) indicates the WR BW
  - realloc(ptr, size) → sicm_class_realloc(speed, ptr, size) where speed(0/1/2) indicates the WR BW
  - free(ptr) → sicm_class_free(ptr)
  - sicm_class_move(speed, ptr) where speed(0/1/2) indicates the WR BW
  - Other APIs, Data structure added but are used internally
- Each allocation has its own arena. Mapping is maintained in a data structure

AllocTransform Pass

```
define dso_local i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
…
%40 = load i64, i64* %size10, align 8
  %call58 = call noalias i8* @malloc(i64 %40) #5
  %41 = bitcast i8* %call58 to double*
  store double* %41, double** %a, align 8
  %42 = load i64, i64* %size10, align 8
  %call59 = call noalias i8* @malloc(i64 %42) #5
  %43 = bitcast i8* %call59 to double*
  store double* %43, double** %b, align 8
…
 ret i32 0
}
```



```
define dso_local i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  call void @sicm_class_init()
…
%43 = load i64, i64* %size10, align 8
  %44 = zext i32 0 to i64
  %sicmalloccall3 = call i8* @sicm_class_alloc(i64 %44, i64 %43)
  %45 = bitcast i8* %sicmalloccall3 to double*
  store double* %45, double** %a, align 8
  %46 = load i64, i64* %size10, align 8
  %47 = zext i32 0 to i64
  %sicmalloccall4 = call i8* @sicm_class_alloc(i64 %47, i64 %46)
  %48 = bitcast i8* %sicmalloccall4 to double*
  store double* %48, double** %b, align 8
…
 call void @sicm_fini()
 ret i32 0
}
```

Future Work

- Add a heuristic based policy for the compiler
  - Identify access patterns in loops and allocate the data structures in the appropriate memory class
  - Introduce loop splitting and scheduling memory movement and loop execution
- Work on characterizing LULESH, VPIC, SNAP
  - AMG will be hard to run with the compiler as it uses memory wrappers
  - Add CFG analysis/transformations to help work around them

Conclusion

- PEACH enables applications allocate and move data between heterogeneous memory devices on a system
- Application programmers can continue to design their applications the same as before