Towards A Portable Hierarchical View of Distributed Shared Memory Systems: Challenges and Solutions

Millad Ghane
Department of Computer Science
University of Houston
TX, USA
mghane2@uh.edu,mghane@cs.uh.edu

Sunita Chandrasekaran
Department of Computer and Information Sciences
University of Delaware
DE, USA
schandra@udel.edu

Margaret S. Cheung
Physics Department
University of Houston
Center for Theoretical Biological Physics, Rice University
TX, USA
mscheung@central.uh.edu

Abstract

An ever-growing diversity in the architecture of modern supercomputers has led to challenges in developing scientific software. Utilizing heterogeneous and disruptive architectures (e.g., off-chip and, in the near future, on-chip accelerators) has increased the software complexity and worsened its maintainability. To that end, we need a productive software ecosystem that improves the usability and portability of applications for such systems while allowing every parallelism opportunity to be exploited.

In this paper, we outline several challenges that we encountered in the implementation of Gecko, a hierarchical model for distributed shared memory architectures, using a directive-based programming model, and discuss our solutions. Such challenges include: 1) inferred kernel execution with respect to the data placement, 2) workload distribution, 3) hierarchy maintenance, and 4) memory management.

We performed the experimental evaluation of our implementation by using the Stream and Rodinia benchmarks. These benchmarks represent several major scientific software applications commonly used by the domain scientists. Our results reveal how the Stream benchmark reaches a sustainable bandwidth of 80 GB/s and 1.8 TB/s for single Intel Xeon Processor and four NVIDIA V100 GPUs, respectively. Additionally, the srad_v2 in the Rodinia benchmark reaches the 88% speedup efficiency while using four GPUs.

CCS Concepts • **Computer systems organization → Heterogeneous (hybrid) systems**;

Keywords Hierarchy, Heterogeneous, Portable, Shared Memory, Programming Model, Abstraction

ACM Reference Format:
Millad Ghane, Sunita Chandrasekaran, and Margaret S. Cheung. 2020. Towards A Portable Hierarchical View of Distributed Shared Memory Systems: Challenges and Solutions. In *The 11th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'20), February 22, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3380536.3380542

PMAM'20 , February 22, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7522-1/20/02…$15.00
https://doi.org/10.1145/3380536.3380542

1 Introduction

Heterogeneity has become increasingly prevalent in the recent years given its promising role in tackling energy and power consumption crisis of high-performance computing (HPC) systems [15, 20]. Dennard scaling [14] has instigated the adaptation of heterogeneous architectures in the design of supercomputers and clusters by the HPC community. The July 2019 TOP500 [36] report shows how 126 systems in the list are heterogeneous systems configured with one or many GPUs. This is the prevailing trend in current generation of supercomputers. As an example, Summit [31], the fastest supercomputer according to the Top500 list (June 2019) [36], has two IBM POWER9 processors and six NVIDIA Volta V100 GPUs.

Computer architects have also started to integrate accelerators and conventional processors on one single chip; hence, moving from node-level parallelism, e.g., modern supercomputers, to chip-level parallelism, e.g., System-on-chips (SoC). Most likely, in the near future, computing nodes will possess chips with diverse type of computational cores and memory units on them [23]. Figure 1a displays potential and envisioned schematic architecture of a future SoC. Fat cores in the figure are characterized by their sophisticated branch prediction units, deep pipelines, instruction-level parallelism, and other architectural features that optimize the serial execution to its full extent. They are similar to the conventional processors in current systems. Thin cores, however, are the alternative class of cores, which have a less complex design, consume less energy, and have higher memory bandwidth and throughput. Such cores are designed to boost the performance of the data parallelism algorithms [4].

The increasing discrepancy between memory bandwidth and computation speed [39] has led computer architects seek disruptive methods like Processing-In-Memory (PIM) [22, 24] to solve this problem. PIM-enabled methods bring memory modules closer to the processing elements and place them on the same chip to minimize the data transfer with off-chip components. Such adaptation is manifesting itself in the form of die-stacked memories (e.g., 3D Stacked Memories [27, 38]) and scratchpad memories [6]. Figure 1a shows such on-chip memory organization. The memory modules on the chip form a network among themselves to ensure the data consistency among themselves [30], and the data transfer among computational cores on the chip is enabled with the network-on-the-chip (NoC) component [8, 17].

To that end, we are in dire need of a simple yet robust model to efficiently express the implicit memory hierarchy and utilize the parallelization opportunity in potential up and coming exascale systems [37] while improving the *programmability, usability, and portability* of the scientific applications for the exascale era.

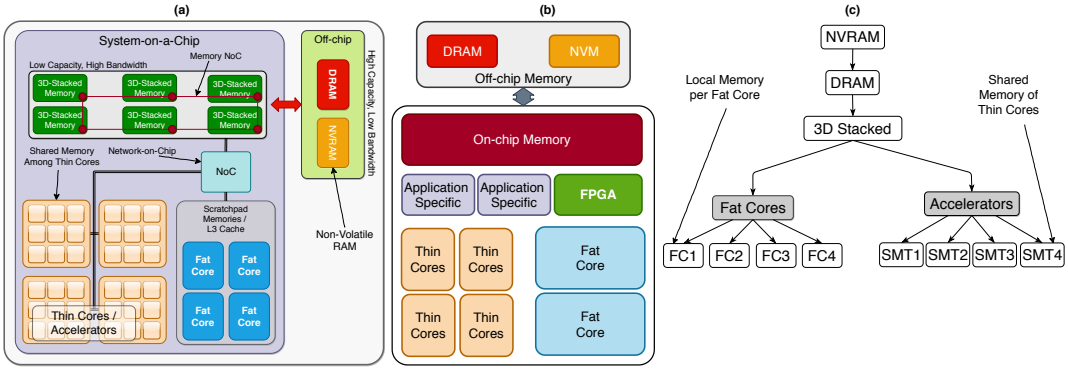


Figure 1. **(a)** A system-on-a-chip (SOC) processor, **(b)** an abstraction machine model (AMM), adopted from [4], for an exascale-class compute node, and **(c)** its corresponding Gecko model. Every component on the model in **(b)** has a corresponding location in the *Gecko* model **(c)**. Virtual locations are colored in gray.

With simplicity and portability as their main goal, Ghane et al. [18] proposed a hierarchical model, Gecko, that helps the software developers to abstract the underlying hardware of the computing systems and create portable solutions. To demonstrate the feasibility of Gecko, they also implemented Gecko as directive-based programming model with the same name. Now that Gecko is in place, this paper discusses the model's feature sets while addressing challenges along the way. Our main contributions in this paper are to declare those challenges and provide our solutions for them. Those challenges are:

1. Given a set of variables scattered in various locations on the hierarchy tree, in which of these locations Gecko would execute a kernel? We propose a novel algorithm, Most Common Descendent (MCD), to address this challenge. Discussed in Section 3.
2. Having chosen a location, how does Gecko distribute execution among the children of a location? We propose five distribution policies to address this scenario. Discussed in Section 4.
3. How does Gecko's runtime library maintain the tree hierarchy and the workload distribution internally? We use a Reverse Hash Table to propose our solutions. Discussed in Section 5.
4. Considering the dynamism that we introduce, how is memory allocated since the targeted location is only known at the execution time? We use Gecko's *multiCoreAlloc* to address this challenge. Discussed in Section 6.

The rest of the paper is organized as follows. Section 2 provides a brief overview of Gecko and its components. Section 7 discusses how we ported the Stream benchmark [13] to Gecko as an example. Section 8 discusses the performance results of the Stream and Rodinia benchmarks ported to Gecko. Finally, Sections 9 and 10 discuss the related work and conclude our work, respectively.

2 Gecko: The Hierarchical Model

2.1 Background

Although hardware offers a promising potential to provide the exascale requirements, utilizing such hardware has been a real challenge from the software standpoint. There has been a dire need for a simpler software solution at the programming level to increase developer's productivity. Bair et al. [4] proposed a set of abstract machine models (AMMs) to describe the computer architectures. AMMs are intended as communication aids between computer architects and software developers to study the performance trade-offs of a model. Figure 1b shows a promising AMM for potential heterogeneous exascale systems [4]. This model has the potential to represents both node- and chip-level systems.

Figure 1c shows the equivalent Gecko model for the abstract model of Figure 1b. The fat cores, depicted with *FCx*, have the potential to represent both the conventional processors in current systems (e.g., IBM POWER9 in Summit) and modern on-chip processors in the SOC processor (e.g., "fat cores" in Figure 1a). Other components in Figure 1c — *locations* in Gecko's terminology — have similar physical manifestations in real scenarios.

2.2 Brief Overview of Gecko

The principal components of a Gecko model are its *locations*. They are an abstraction of available memory and computational resources in a system and are connected to each other, similar to a hierarchical tree structure. Each location represents a particular memory subsystem; e.g., the host memory, the device memory on the accelerators, and so on. Potential memory locations can be grouped together and form a *virtual* location. The virtual locations in our model have no physical manifestation in the real world. Their role is to simplify the management of similar locations and to minimize code modifications. Figure 1c is an example of a Gecko model for the SOC shown in Figure 1a.

Gecko's memory model is very similar to the cache organization in modern processors. Based on the exascale report by Bair et al. [4], memory components are projected to be multi-level cached memories, in contrary to one-level, flat, non-cached memories. Gecko follows the former approach by exploiting multi-level memories, where data in a particular level is cached by another level. While following this approach, it has one exception: *memory allocated in a particular location is accessible by that location and all of its children, while it remains private with respect to its parent.*

Computations are performed by the leaf locations at the bottom of the tree. Such locations are conventional processors, GPU devices, Processing-In-Memory (PIM) [22], and other non-conventional processors like FPGAs and DSPs. Gecko's hierarchical tree structure is a dynamic structure that is described and defined at the execution time. To begin with, applications describe the type of locations

they are targeting through their lifetime. Then, the locations are defined based on the predefined types. By declaring the relationships between locations, the hierarchical structure is complete and functional. Applications are able to modify the structure on-the-fly as they encounter different architectures. Such a flexibility is a *must* for the future HPC systems due to the diversity in their design. Developing many stand-alone frameworks/applications to target various architectures is not a feasible approach anymore. Thus, software should be able to adapt itself to the different configurations in a portable manner.

Gecko, since it targets single node parallelization, is a viable model for the 'X' in the MPI+X. After workload distribution among nodes with MPI, Gecko enables an application to adapt itself to the available resources in each node with the same executable file. By changing the hierarchical tree at the execution time, we are able to execute our application with the configuration of the current MPI rank. In contrast, for the current approaches like OpenMP and OpenACC, the applications do not have such flexibility as they require all MPI ranks to have the same configuration. In those cases, the nodes should be homogeneous in their configuration. However Gecko does not have this requirement.

3 Inferred Execution

Data placement is not a trivial job. Agarwal et al. [2] and Arunkumar et al. [5] narrate how the onus is on an "expert programmer" or on an extensive proffer that can determine the efficient placement of data. Either of the two approaches has its own advantages and drawbacks. Our model, Gecko, relies on the programmer to place the data in their proper location. The programmer, using Gecko's directives, allocates a block of memory on any location. This allows Gecko to have an up-to-date knowledge of where each allocated memory is placed.

As an application progresses over time, it allocates memories in different locations. The location that is chosen depends on various criteria (e.g., bandwidth- or capacity-optimized memories). This, in turn, causes the memories allocated to be scattered around different locations within the hierarchy. Therefore, if a computational kernel utilizes multiple memories that are not in the same location, a location that is the most suitable has to be chosen, which has to be accessible by all the allocated memories throughout the hierarchy. We handle this technique by proposing a novel algorithm elaborated in the following section.

Our current work significantly differs from our previous work Ghane et al. [18, 19] in a way that our previous work does not use this novel algorithm and required the developers to choose the location manually using the `at` keyword.

3.1 Data Locality

Moving data around is quite a costly operation. Hence, reducing the number of the data transfers improves the performance significantly. Such improvement is due to the reduction of the waiting time for the completion of the data transfer operation. Our motivation is to achieve this improvement by helping developers to move *their executable code* closer to where the data is instead of moving the data. To that end, Gecko enables the migration of the computations to the targeted location instead of moving data. When a kernel is initiated on previously-allocated memories, Gecko locates the location of each memory in the hierarchy, and then it finds the proper location to execute the computational kernel based on those locations. Choosing an appropriate location depends on where the memories are scattered within the hierarchy. To find the appropriate location within the hierarchy efficiently, we propose a novel algorithm Most Common Descendent (MCD) algorithm. The next Section 3.2 discusses our algorithm and its adaptability in detail.

Algorithm 1 Most Common Descendent (MCD) Algorithm

Input: T: *Gecko*'s hierarchical tree structure.
Input: L: List of locations.
Input: $pathToRoot(t, n)$: returns the locations on the path from node n to root of the Tree t
Output: The most common descendent or NULL

```
1: function MCDALGORITHM(T, L)
2:     commonChild ← L_0
3:     commonPath ← pathToRoot(T, L_0)
4:     for each L_i ∈ L do
5:         if L_i ∉ commonPath then
6:             newPath ← pathToRoot(T, L_i)
7:             if commonChild ∉ newPath then
8:                 return NULL
9:             end if
10:            commonChild ← L_i
11:            commonPath ← commonPath
12:        end if
13:    end for
14:    return commonChild
15: end function
```

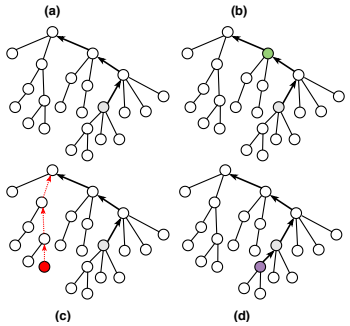


Figure 2. (a) The starting point and **(b)**, **(c)**, and **(d)** Three different scenarios of the MCD algorithm

3.2 Most Common Descendent (MCD) Algorithm

Given a hierarchy tree and a set of locations within the tree, the MCD algorithm finds a location in the set that is the child and/or grandchild of all other locations in the input set. Such a location is also the deepest location in the hierarchy among the locations in the input set. This is shown in details in Algorithm 1.

The MCD algorithm is the exact opposite of the Lowest Common Ancestor (LCA) algorithm [3]. While LCA traverses upwards in the tree to find the most common parent among a given set of locations, MCD traverses the tree downwards in order to find the most common child. Unlike LCA, MCD may not have a final answer for a given input set. In such cases, we have to use the `move` construct in Gecko to transfer a previously allocated memory from its location to another location that satisfies the criteria of the MCD algorithm.

Figure 2 demonstrates three scenarios that will happen when we run the MCD algorithm as shown in Algorithm 1. First, we set the first location as *the final answer* and record its path to the root as *the final path* (shown as **a**). Then, we loop over other locations. For each location, there are three possible scenarios, which are shown in Figure 2 **(b)**, **(c)**, and **(d)**. If a location is found on the final path (as shown with **b**), we already have the final answer, so we will skip this location. If a location has a path to the root that does not overlap with the current final path (as shown with **c**), there is no final answer

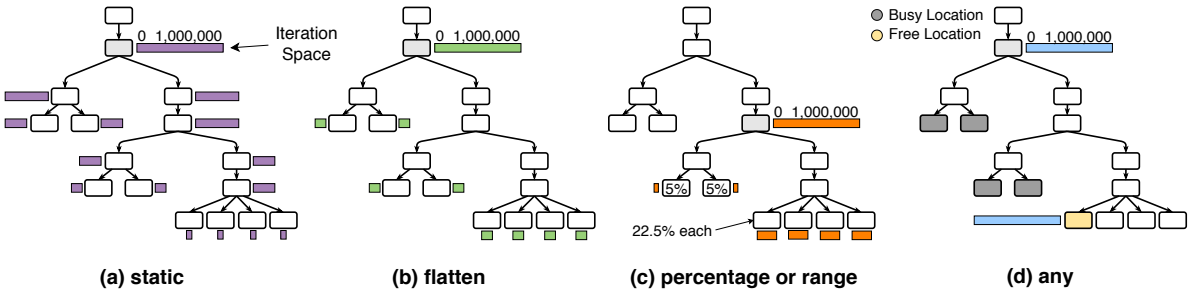


Figure 3. Four different workload distribution policies that are supported by Gecko.

that satisfies the provided input, and consequently, the program halts. Finally, if a location has a path to the root that includes the current final answer (as shown in **d**), the final answer and the final path are changed to this new location and path. To summarize, the MCD algorithm finds a common path among all input locations. If one is found, the deepest location in the hierarchy is returned. Otherwise, there is no result for the input under consideration.

Theorem: The complexity of the MCD algorithm, in the worst case scenario, is $O(n\ log(n))$. **Proof:** The for-loop on Line 4 of Algorithm 1 has to check every node in the list. Additionally, the complexity of Line 6 (`pathToRoot(T,` L_i`)`) of Algorithm 1 is related to the height of the tree. Accordingly, for a complete tree where each location has m children, extracting path from any arbitrary location to root has a complexity of $O(log_m(n))$. Thus, in the worst case scenario (where $m = 2$), the complexity becomes $O(n\ log(n))$.

4 Workload Distribution Policy

The next step after determining where to execute the kernel is to find out how to distribute the workload among children associated to that location. Gecko provides a set of distribution policies that a programmer is able to select. The distribution policy is applied to all the iterations of the `for`-loop. Similar to directive-based programming models, Gecko partitions iterations of a loop and assigns each partition to a location that will be executed. The partitioning process is governed by distribution policies. Figure 3 demonstrates how these policies partition a `for`-loop with 1,000,000 iterations. Supporting policies by Gecko are *static*, *flatten*, *percentage*, *range*, and *any*, which are discussed below.

Static: In the *static* distribution, the iteration space is divided evenly among children of that location. Figure 3 shows how the iteration space, shown as a box, is partitioned among location. Since the destination location has two children, the space is partitioned into two parts. The partition assigned to each child is further divided among its children. The leaf nodes that are closer to the root will have bigger shares in comparison to the others.

Flatten: In *flatten*, all of the leaf nodes at the bottom of the tree take an equal share of the iteration space. Figure 3b shows an example of the flatten policy. In this case, the iteration space is partitioned into eight equally-sized partitions since we have eight leaf nodes in total. Each partition is assigned to a single location.

Percentage and Range: Gecko also provides customized workload distribution among locations. A developer is able to partition the iteration space among children of a location. The *range* policy accepts an array of integers that specifies the partition size for each child. The *percentage* policy accepts an array of percentages that specifies the partition in percentages with respect to the whole iteration space. Figure 3c shows an example of how the range and percentage policies partition the iteration space. The example shows the `percentage:[5,5,22.5,22.5,22.5,22.5]` case. The locations on the left are assigned only 5% of the iteration space, while each location on the right is assigned 22.5% of the whole iteration space.

Any: In some cases, we are interested in engaging only one of the children in the execution process. In such cases, Gecko finds an idle location among children of the chosen location. Alternatively, based on the recorded history, Gecko can choose the best architecture for this kernel if we are targeting a multi-architecture virtual location. Figure 3**d** shows an example of the *any* policy. One can observe how Gecko chooses the yellow location since the first four gray ones are busy with other jobs, and the yellow location is the first available child of the light-gray location.

5 Gecko Runtime Library

5.1 Hierarchical Architecture Maintenance

Gecko Runtime Library (GRL) utilizes an internal *tree data structure* to maintain the hierarchy that Gecko proposes. Each location in the tree has another location as its parent and multiple (or no) locations as its children. The *root* location of the tree has no parent. Locations are accessed by their unique name.

Traversing Gecko's tree each time to find a location is not a performance-friendly approach. Hence, Gecko uses a Reversed Hash Table (RHT) to find and access a location. RHT is a key-value-based container that maps the name of a location to its corresponding location in the tree. Figure 4 shows RHT for a sample tree in

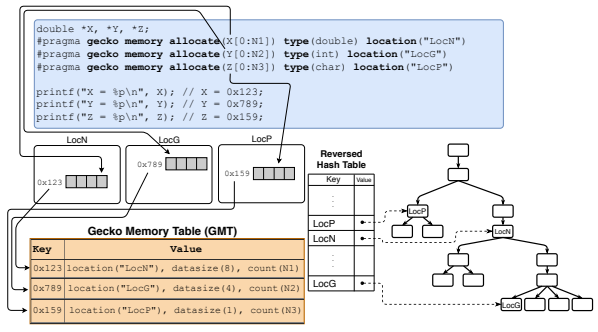


Figure 4. A visualization of Reversed Hash Table (RHT) and Gecko Memory Table (GMT) with a code snippet that shows how `memory allocate` clause in Gecko works and affects GMT.

Algorithm 2 The Region Pseudocode

```
Input: T: Gecko's hierarchical tree structure.
Input: varList: List of variables
Input: policy: The chosen distribution policy
Input: kernel: The computational kernel
 1: function REGION(T, varList, policy)
 2:     threadList ← bindThreadsToLeafLocations(T)
 3:     locList ← extractLoc(varList)
 4:     loc ← mcdAlgorithm(T, locList)
                                   ▷ splitIterSpace as shown in Figure 3
 5:     leafList, configs ← splitIterSpace(loc, policy)
 6:     for each th ∈ threadList do
 7:         if th.loc ∈ leafList then
 8:             S ← configs[th.loc]
 9:             kernel.execute(th.loc, S.begin, S.end)
10:         end if
11:     end for
12:     threadList.waitAll()
13: end function
```

Gecko. Every location has an entry in RHT. When we add or remove a location to or from the hierarchy, RHT is updated with the changes. Such a design allows Gecko to access a location in $O(1)$ complexity.

5.2 Thread Assignment

After finalizing the structure of the hierarchical tree, Gecko assigns a thread to each leaf location at the bottom of the tree. One can change the hierarchy at the execution time. If the hierarchy changes, Gecko reevaluates the tree structure and reassigns the threads to new leaves. Line 2 in Algorithm 2 shows where this binding process happens.

5.3 Workload Maintenance

GRL is also responsible for workload distribution among locations. As a program encounters for-loop that is decorated with Gecko's directive to distribute the workload, the iteration space is partitioned among children based on the execution policy that is chosen.

Algorithm 2 shows the steps that are followed by Gecko. First, as shown in Line 2, all leaf locations in the hierarchy are extracted, and a thread is assigned to them as we will discuss in Section 5.2. Each thread is responsible for initiating a job on the location and waiting for the location to finish its job. This is the synchronization mechanism that Gecko follows to coordinate devices in the system.

Second, we use MCD to find the appropriate location to execute the kernel. We extract the list of locations from the variables used in the region (Line 3). Then, Line 4 in Algorithm 2 shows how to call the MCD algorithm to choose our target location. Thirdly, on Line 5, the iteration space is partitioned among the children of a location based on the execution policy chosen as discussed in Section 4. Fourthly, Line 6 of Algorithm 2 specifies how threads dispatched on Line 2 take control of their corresponding location and execute their share of the iteration space. Finally, on Line 12, Gecko waits for all threads to finish their assigned job. After Line 12, the devices are free for the next round of execution. In case MCD does not find a common children among nodes, it returns `NULL`. Currently, in such cases, the runtime library shuts down the application gracefully with an appropriate message to the terminal output.

6 Memory Management

This section discusses challenges that Gecko faces in the memory allocations of the heterogeneous systems.

Algorithm 3 Memory Allocation Algorithm **(we only support multicore and NVIDIA GPUs in current implementation)**

```
Input: gTree: Gecko's hierarchical tree structure.
Input: loc: the target Location.
Output: allocFunc: Memory Allocation API.
 1: function MEMALLOC(gTree, loc)
 2:     allocFunc ← NULL                ▷ Selected API to perform allocation
 3:     if gTree.isLeaf(loc) then
 4:         if gTree.getType(loc) == HOST then
 5:             allocFunc ← multiCoreAlloc
 6:         else if gTree.getType(loc) == GPU then
 7:             allocFunc ← cudaMalloc
 8:         else
 9:             return ERR_UnrecognizedLocationType
10:         end if
11:     else
12:         children ← gTree.getChildren()
13:         if children.areAllMC() then
14:             allocFunc ← multiCoreAlloc
15:         else if gTree.getType(loc) ∈ {GPU, MULTICORE,VIRTUAL} then
16:             allocFunc ← cudaMallocManaged
17:         else
18:             return ERR_UnrecognizedLocationType
19:         end if
20:     end if
21:     return allocFunc
22: end function
```

6.1 Uncertainty in Location Type

Uncertainty in location type makes memory allocation a challenging problem because it is not a straightforward process. The allocation process has to be postponed to the execution time since only then Gecko has enough knowledge to perform the allocation.

Algorithm 3 shows how Gecko allocates memory in a basic configuration that utilizes *only CPUs and GPUs — since the current implementation only supports these two architectures*. It starts by recognizing if the location chosen is a leaf location in the tree or not. Allocated memories in leaf locations are private memories that are only accessible to that location. Gecko calls the corresponding memory allocation API with respect to the location type. If the leaf location is host, Gecko uses the *multiCoreAlloc*[1] API. If the leaf location is an NVIDIA GPU, Gecko uses the *cudaMalloc* API. Otherwise, due to supporting only multicore and NVIDIA GPUs in this implementation, Gecko returns an error to developers indicating the unrecognized type for the allocation. Other architectures (e.g., FPGAs or PIMs) can be supported as well by checking the type of the location at the execution time and calling the corresponding allocation API.

On the other hand, if the location chosen is not a leaf location, Gecko traverses the subtree beneath the location and determines if all of its children are multicore or not. If they are all multicore, similar to the previous case, Gecko will use *multiCoreAlloc*. If none of the children are multicore and the current location type is multicore, NVIDIA GPU, or virtual, Gecko allocates memory from Unified Virtual Memory (UVM) [26] domain[2]. Otherwise, Gecko returns an error to developers indicating the unrecognized type for the allocation.

Figure 4 displays a code snippet that utilizes Gecko's directives to allocate memories in the system. It shows the sequence of actions that takes place when a memory is allocated. First, Gecko allocates a block of memory to the designated location with the data type and the total number of elements that the user requested (known as

[1]The *multiCoreAlloc* API is a host-based memory allocation API such as `malloc`, `jemalloc` [28], `tcmalloc` [35], and `numa_alloc` [25].
[2]For upcoming architectures, a hardware or a software approach can guarantee the data consistency among architectures [12, 33].

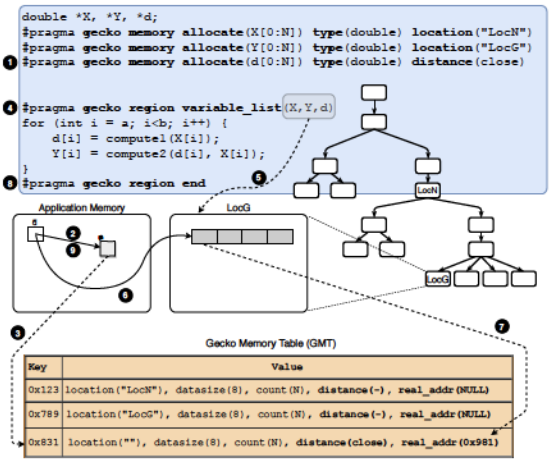


Figure 5. Steps taken by Gecko that shows how distance-based memory allocations are performed with minimum code modification. By simply annotating the memory allocation clause with `distance`, Gecko governs the correct state of the pointers internally.

memory traits). For instance, in the second line of the code snippet in Figure 4, the programmer requests `N1` double-precision elements in `LocN`. By using the API function returned by Algorithm 3, the target variable (in this case, `X`) holds the memory address (`0x123`). Then, Gecko inserts an entry into Gecko Memory Table (GMT) with 0x123 as the key and the memory traits as the value. GMT is a hash table that traces memory allocations within the system. The `extractLoc` function in Algorithm 2 uses GMT to find the location of each variable in its input parameter, `varList`. Please note that memory allocation directives can also be used within the conditional statements in the code. In such cases, the pointer provided to the `allocate` clause will be populated with a valid memory address accordingly.

6.2 Distance-based Memory Allocations

Unlike the ordinary allocations that were discussed in Section 6.1, a programmer does not have to specify the target location in distance-based allocations. In the former case, a programmer manually specifies the target location. In the latter, GRL infers the target location at the execution time. Such memory allocations are performed with respect to the location that the computational kernel is targeted to be executed.

Distance-based allocations in Gecko are declared, as either `close` or `far`. Close-memory allocations are performed within the location that runs current `region`. However, far-memory allocations are performed within the parent (or grandparents) of the targeted location. For instance, in Figure 4, if `LocG` is chosen for kernel execution, declaring a memory as `close` will allocate memory within `LocG`, while declaring it as `far:3` will allocate it in `LocN` (since it is its third grandparent).

In addition, Gecko provides `realloc` and `move` keywords for distance-based memory allocations. The `realloc` keyword allocates a memory block when entering a region and frees it when exiting a region. However, with the `move` keyword, a memory block is allocated on the first touch and is moved around within the hierarchy between the subsequent Gecko regions.

With a minimal code change, Gecko enables the distance-based memory allocation challenge by how to *declare* and *utilize* distance-based allocations, as discussed above. Figure 5 shows the sequence of actions that takes place so that Gecko performs a distance-based allocation. The code snippet in Figure 5, annotated with Gecko's directives, is utilizing `X` and `Y` variables where each variable points to `N` double precision floating-point numbers that are allocated in `LocN` and `LocG`, respectively. We declare a distance-based memory space, named `d`, that is designated as a `close` memory (❶). Gecko starts by allocating a dummy memory block on the heap (❷). The dummy block is basically a handle to distinguish the distance-based allocations from the regular ones. Then, Gecko inserts an entry into GMT to record the memory request (❸), and then allocates a memory block as soon as it determines the destination location. We updated the structure of GMT, as shown in Figure 5, to handle the distance-based allocations. The two new fields, `distance` and `real_addr`, hold the distance parameter and the address of allocated memory block in the destination target, respectively.

As we reach the `region` section in our code (❹), Gecko finds the target location to run the kernel. Based on the fact that `X` and `Y` variables are our non-distance-based variables in the Gecko region (❺), the MCD algorithm will choose `LocG` as the location to execute the region and the target location to allocate variable `d`. Then, Gecko allocates a memory block within `LocG`, reassigns the variable `d` to the new allocation (❻), and finally, updates GMT with the new address (populating the `real_addr` field as shown with ❼). Until the end of the region, the variable `d` points to the valid memory block in `LocG`. As we reach the end of the `region` (❽), the variable `d` reverts to its original value, which was the dummy variable allocated before (❾).

7 Gecko In Use

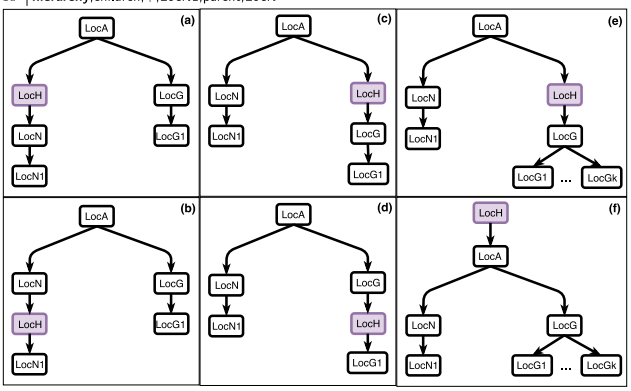
In this section, we will demonstrate how to write an application with Gecko. For a complete list of all capabilities of Gecko, please refer to Gecko's Github repository[3].

Figure 6 shows a snapshot of a sample configuration, a visualization of the different configurations and a snapshot of Stream benchmark programmed using Gecko's directives. Starting from the right side of the Figure (for easier explanation) shows the source code of the Stream benchmark in Gecko. We will go through the lines of this code and clarify what each line does. Line 1 loads the configuration file from the disk. The configuration file includes the definition of location types, the definition of locations, and the declaration of hierarchies among locations. The top left of Figure 6 shows an example of a configuration file for Configuration **a** in the bottom left of Figure 6. Lines 3-5 will allocate memory with `array_size` elements and type `T` at the location "`LocH`". We hard-coded the destination location to "`LocH`" for the three memory allocations. This provides greater flexibility to the application. We can place "`LocH`" anywhere in the hierarchy since we have defined it to be a virtual location in our configurations.

We tested our application with different configurations. A list of all configurations that we targeted are shown in the (bottom left) of the Figure 6. For example, configurations **a** and **b** target only multicore systems. However, configurations **c**-**e** target single- and multi-GPU systems. Finally, configuration **f** targets a multi-architecture system to execute our application. If we change the

[3] https://github.com/milladgit/gecko

```
1  # Content of 'gecko.conf' file that represents
2  # Configuration (a)
3
4  loctype;name,host;kind,x64,Skylake;num_cores,4;mem,4MB;
5  loctype;name,tesla;kind,NVIDIA,CC7.0;mem,4GB
6  loctype;name,NODE_MEMORY;kind,Unified_Memory;size,16GB
7
8  location;name,LocA;type,NODE_MEMORY;
9  location;name,LocN,LocG;type,virtual
10 location;name,LocN1;type,host;
11 location;name,LocG1,LocG2;type,tesla
12
13 location;name,LocH;type,virtual
14
15 hierarchy;children,+,LocH,LocG;parent,LocA
16 hierarchy;children,+,LocN;parent,LocH
17 hierarchy;children,+,LocG1;parent,LocG
18 hierarchy;children,+,LocN1;parent,LocN
```




```
1  #pragma gecko config file("gecko.conf")
2  T *a, *b, *c;
3  #pragma gecko memory allocate(a[0:array_size]) type(T) location("LocH")
4  #pragma gecko memory allocate(b[0:array_size]) type(T) location("LocH")
5  #pragma gecko memory allocate(c[0:array_size]) type(T) location("LocH")
6
7  #pragma gecko region gang vector exec_pol("runtime") variable_list(a,b,c)
8  for (int i = 0; i < array_size; i++) {
9      a[i] = initA;
10     b[i] = initB;
11     c[i] = initC;
12 }
13 #pragma gecko region end
14 #pragma gecko region pause
15
16 std::chrono::high_resolution_clock::time_point t1, t2;
17 // Run triad in loop
18 t1 = std::chrono::high_resolution_clock::now();
19 for (unsigned int k = 0; k < num_times; k++) {
20     #pragma gecko region gang vector exec_pol("runtime") \\
21                 variable_list(a,b,c)
22     for (int i = 0; i < array_size; i++) {
23         a[i] = b[i] + scalar * c[i];
24     }
25     #pragma gecko region end
26     #pragma gecko region pause
27 }
28 t2 = std::chrono::high_resolution_clock::now();
29 double runtime = duration_cast<duration<double>>(t2 - t1).count();
30
31 #pragma gecko memory free(a,b,c)
```

Figure 6. Top Left: A sample configuration file that represents Configuration (a) on the bottom left. The first two lines are comments. **Bottom Left:** Visualization of different configurations. Note how placing "LocH" in different positions in the hierarchy results in targeting different architectures. Configuration **(a)** and **(b)** target general-purpose processors. Configurations **(c)** and **(d)** target one single GPU. Configurations **(e)** and **(f)** target multi-GPU and multi-architecture systems, respectively. **Right:** A snapshot of the Stream benchmark with Gecko's directives.

configuration file so that it represents any of the configurations of **a** to **f**, without recompiling the source code, our program is able to target different architectures without significant performance loss.

Lines 7-13 show a computational kernel that initializes three arrays that were allocated previously. The `runtime` execution policy specifies that Gecko will extract the *main* policy from an environmental variable (known as `GECKO_POLICY`) at the execution time. One should set this variable to any of previously defined policies in Section 4. The `pause` statement in Line 14 asks Gecko to synchronize itself with all computational resources (in our case, CPUs and GPUs) and wait for them to finish their assigned job before continuing with the next statement in the code; in other words, `pause` is a synchronization point.

Lines 19-27 show a loop that contains the main TRIAD kernel of the Stream benchmark and calls the kernel `num_times` times. Similar to the original TRIAD kernel, it is a `for`-loop that multiplies each element in array *c* to a scalar value, adds it to an element in array *b*, and stores the final value in array *a*. Depending on the configuration and execution policy chosen at the execution time, Gecko splits the iterations of the main loop in Line 22 (from 0 to `array_size`) among the processors and GPUs. For instance, for Configuration **e** where number of GPUs is four, each GPU will process $array\_size/4$ iterations.

The benchmark calls the high resolution timers in Lines 18 and 28 before and after the for-loop to measure the total execution time of the TRIAD kernel. And finally, Line 31 asks Gecko to free all memories allocated in the system.

8 Our Evaluations

In this section, after describing the configurations of PSG and Sabine clusters, we will assess the performance of Gecko in those systems, each with Stream and Rodinia benchmarks.

8.1 Experimental Setup

We used NVIDIA Professional Services Group (PSG) cluster [29] and Sabine [32] to perform our evaluations. PSG is a dual socket 16-core Intel Haswell E5-2698v3 at 2.30GHz with 256 GB of RAM. Four NVIDIA Volta V100 GPUs are connected to this node through PCI-E bus. Each GPU has 16 GB GDDR5 memory. We used CUDA Toolkit 10.1 and PGI 18.10 (community edition) for the CUDA and OpenACC codes, respectively. Sabine is a dual socket 14-core Intel Haswell E5-2680v4 at 2.40GHz with 256 GB of RAM. Two NVIDIA Pascal P100 GPUs are connected to this node through PCI-E bus. Each GPU has 16 GB GDDR5 memory. We used CUDA Toolkit 10.1 and PGI 19.4 for the CUDA and OpenACC codes, respectively.

Note: PSG results use PGI 18.10 and Sabine results use PGI 19.4. The reason being we no longer had access to PSG when PGI 19.4 version was released. Having said we see little to no difference in results using both versions for our project.

8.2 Sustainable Bandwidth

We used BabelStream benchmark [13] to measure memory bandwidth. BabelStream provides a set of Stream benchmarks in various programming models and libraries: OpenMP, CUDA, OpenACC, Kokkos, and RAJA. We created a version of Stream benchmark

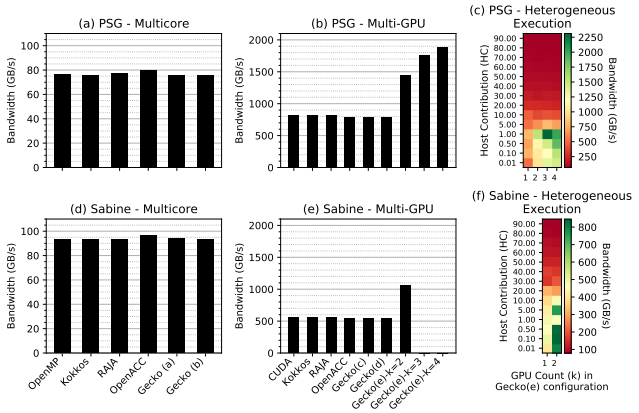


Figure 7. Sustainable bandwidth of the Stream benchmark on PSG and Sabine. **(a), (d)** multicore systems, **(b), (e)** single- and multi-GPU systems, and **(c), (f)** heterogeneous systems.

based on Gecko and then compare the sustainable bandwidth provided by each of the programming approaches as shown in Figure 7.

Results are shown for both systems (PSG and Sabine) and for three different scenarios: (1) multicore execution, (2) multi-GPU execution, and (3) heterogeneous execution. In our experiments with Stream, we set the array size to 256,000,000 double-precision elements for each array, which results in 6 GB of data in total. We run the TRIAD loop 200 times and took the average of their wall-clock time to report the execution time. The multicore results reveal a negligible difference between the Gecko version of Stream with other methods (less than 5 GB/s difference in comparison to OpenACC for both PSG and Sabine). Gecko's results are reported for configurations **(a)** and **(b)**. Sabine's main processor provides 18 GB/s more bandwidth to access the main memory in comparison to PSG. It is due to subtle difference in their memory bandwidth; the theoretical peak memory bandwidth for Sabine's dual processors is 153.6 GB/s, however, the peak bandwidth for the dual processors in PSG is 136 GB/s.

Similarly, Gecko's performance is not affected when we target GPUs. Single- and multi-GPU results are shown in Figure 7b and 7e. Despite the difference in their hierarchy, configurations **c** and **d** provide the same bandwidth since our memory allocation algorithm returns the same API function in both cases. However, since CUDA, Kokkos, RAJA, and OpenACC versions of Stream use non-UVM (Unified Virtual Memory) memories to execute the benchmark, there is a 17 GB/s difference in the bandwidth. Note that while Gecko uses our memory allocation algorithm for data placement, no source code modification is needed to support multi-GPU utilization. As a result a single code base can be maintained for both single- and multi-GPU execution. By only modifying the configuration file, Gecko's Stream source code, as shown in Figure 6, is able to utilize GPUs as well. Gecko was able to provide 1.8 TB/s and 1.9 TB/s with three and four GPUs on PSG, respectively. Figure 7b and 7e show the results of GPU execution. Gecko's results are represented with "Gecko (e)-k", where k specifies the number of GPUs in the hierarchy. Gecko supports heterogeneous execution as well with no code alteration. Heatmap plots on Figure 7 show the sustainable bandwidth for the heterogeneous execution of the Stream benchmark on the main processor and GPUs in the system, simultaneously. Host workload (HW) specifies the amount of workload assigned to the host processor and is varied from 0.01% to 90% of the total iterations of the TRIAD kernel, and the rest is divided equally between the GPUs. For instance, in case of K=4 and HC=1%, 2,560,000 out of 256,000,000 iterations are assigned to the host's processor and the rest (253,440,000 iterations) is divided among four GPUs; it means Gecko assigns 63,360,000 iterations to each GPU. We utilized the `percentage` execution policy to represent above-mentioned cases. For instance, the equivalent execution policy in Gecko for above example would be "percentage:[1.00,24.75,24.75,24.75,24.75]".

Table 1. List of benchmarks in the Rodinia Suite that were ported to *Gecko* - A: Number of kernels in the code. B: Total kernel launches. SP: Single Precision - DP: Double Precision - int: Integer - Mixed: DP+int

Application	Input	Data Type	A	B
bfs	1,000,000-edge graph	Mixed	5	39
cfd	missile.domn.0.2M	Mixed	5	9
gaussian	4096 × 4096 matrix	SP	3	12285
hotspot	1024 data points	DP	2	20
lavaMD	50 × 50 × 50 boxes	Mixed	1	1
lud	2048 data points	SP	2	4095
nn	42764 elements	SP	1	1
nw	2048 × 2048 data points	int	4	4095
particlefilter	1024 x 1024 x 40 particles	Mixed	9	391
pathfinder	width: 1,000,000	int	1	499
srad	2048 x 2048 matrix	Mixed	7	12

The other implementations of BabelStream do not support multi-GPU by default and their source codes need to be modified. However, this is not the case for their Gecko version and a multi-GPU support is available by only modifying the configuration code. Similarly, Gecko supports the heterogeneous execution with changes in the configuration file. But, this is not the case for the others and they need some code modifications to utilize both CPUs and GPUs simultaneously.

8.3 Rodinia Benchmarks

To investigate the effectiveness of Gecko in heterogeneous environment, we used the Gecko version of the Rodinia suite [18]. Table 1 shows the list of benchmarks used in this paper with their corresponding input, data type, the total number of kernels, and the total kernel launches at the execution time. Figure 8 shows speedup of all applications using multi-GPU (**(a), (d)**) and heterogeneous execution time of *cfd* and *srad_v2* (**(b), (c), (e), (f)**) on PSG and Sabine.

Speedup results were obtained by utilizing **only** the `static` execution policy to equally distribute the workload among the GPUs. For the speedup results, the X axis lists Rodinia benchmark, and the Y axis shows the speedup with respect to a single GPU. Each bar represents different number of GPUs (represented with K). For the heterogeneous execution, we followed a similar approach as we did for the Stream benchmark.

Rodinia's results in Figure 8a and 8d show how multi-GPU utilization is not a suitable option for all benchmarks in Rodinia. The *cfd* and *srad_v2* benchmark applications show promising results for scalability. The performance of *cfd* improves with each additional GPU. In case of *cfd*, the input grid elements to the solver were distributed evenly among available locations. Since computations related to each element is basically independent with respect to the other elements, we observe a good speedup as we increase number of GPUs. The heterogeneous execution of *cfd* on PSG, as shown in Figure 8b, does not lead to a performance improvement. However, the heterogeneous execution of *cfd* on Sabine (two GPUs and

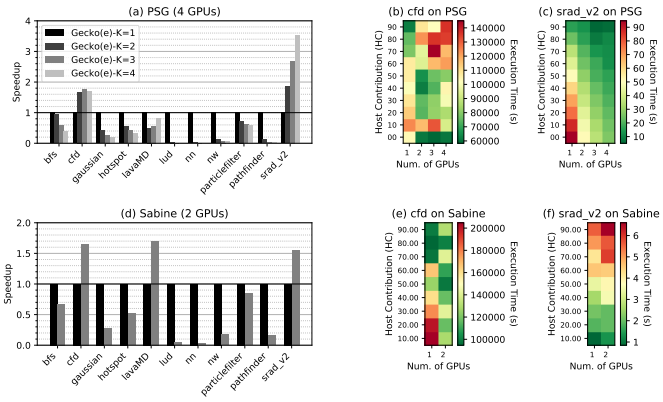


Figure 8. (a), (d) Speedup results of the multi-GPU execution of the Rodinia benchmarks on PSG and Sabine, specified with configuration **(e)** in Figure 6. Heatmap of the execution time of *cfd* (**(b), (e)**) and *srad_v2* (**(c), (f)**) for different host contributions and number of GPUs.

50% host workload) leads to 2X speedup with respect to one single GPU, as shown in Figure 8e. The reason behind the superiority of Sabine over PSG is due to two reasons: 1) Sabine has better host memory bandwidth. Results in Figure 7a and 7d had shown 17 GB/s difference in main memory bandwidth between Sabine and PSG; 2) Utilizing all available GPUs is not always a good idea. Splitting the iteration space among many GPUs leads to lesser work by each GPU, which incurs more overhead. The *cfd* results on PSG confirms our finding as well. For all values of HW (except HW=0), utilizing two GPUs leads to a better performance in comparison to three or four GPUs. The *srad_v2* benchmark benefits more from the heterogeneous execution as the heatmap results show. In comparison to one single GPU (one GPU and HW=0), if we utilize three or four GPUs while HW is 90%, the speedup is 15X for PSG. Similarly, for Sabine, the speedup becomes 11.5X, when two GPUs are utilized and HW is 90%.

Other benchmark applications (*bfs*, *lavaMD*, and *particlefilter*) do not scale as we increase the number of utilized GPUs. The performance degradation is due to uncoalesced and random memory accesses in such applications. The *bfs* benchmark traverses all the connected components in a graph. Thus, the memory accesses follow a random pattern. The *lavaMD* benchmark goes through all atoms in the system and computes the force, velocity, and new position of each atom. It uses a cutoff range to limit the unnecessary computations. However, such cutoff ranges may include atoms that are currently residing in another GPU device. The *particlefilter* benchmark visits elements of a matrix using two nested *for*-loops. Heterogeneous execution of *bfs*, *lavaMD*, and *particlefilter* benchmarks does not improve the speedup either. Utilizing the host processor has caused a gradual performance degradation for these benchmarks. In the case of other benchmarks (*gaussian*, *hotspot*, *lud*, *nn*, *nw*, and *pathfinder*), the performance loss is severe. Algorithms that follow a very random memory access pattern like *bfs* and *gaussian* are not a suitable option for either multi-GPU or heterogeneous execution.

The false sharing [21] effect on the inter-device level is the primary source of performance degradation. It is highly possible that when device d_1 is executing iteration i needs accessing another data that currently resides on device d_2. In such cases, many memory pages have to be invalidated to perform the iteration i. The invalidated page has to travel through PCI-E and NVlink, which are not performance-friendly. Gecko tries to alleviate this by partitioning the iteration space fairly among locations (with respect to the chosen execution policy). However, for cases like *bfs* or *gaussian* where memory access pattern is random, Gecko is unable to improve performance due to unpredictable nature of memory accesses.

9 Related work

Heterogeneous systems have become increasingly prevalent. There are not many solutions out there that can handle systems with multiple devices. *VirtCL* [41] uses an OpenCL approach to utilize multiple homogeneous GPUs replicating an array on the host and other devices ensuring the consistency by locking arrays on any devices that are using it. CoreTSAR [34] introduces directives that are similar to Gecko's. GPU-SM [9] enable multi-GPU utilization on a single node and ensure data consistency with peer-to-peer (P2P) communications. In this paper, however, our implementation of Gecko utilized UVM to make sure the consistency among multiple devices.

Built upon the Parallel Memory Hierarchy (PMH) model [1], Sequoia [16] represents available memory spaces in a hierarchical manner, and workloads are distributed statically through their definition. Hierarchical Place Trees (HPT) [40] bears similarity to Sequoia and Gecko in the exploitation of the "location" concept. However, HPT lacks the dynamic features of Gecko, such as dynamic memory allocation and dynamic hierarchy. Many modern languages that provide facilities to describe data distribution and parallelism through an abstract model for both data and computation have also been introduced (e.g., Legion [7], Chapel [10], and X10 [11]). However, unlike above-mentioned models, application developers can utilize Gecko with their current legacy codes with minimum code modification and they do not need to rewrite the whole application from scratch.

The current implementation of Gecko as a directive-based programming language has similarities to OpenMP and OpenACC. The `region` and `pause` constructs in Gecko have equivalent counterparts in both OpenMP and OpenACC (in terms of their work-sharing strategies). However, unlike Gecko, applications developed with OpenMP and OpenACC only target one architecture (either multicore or GPUs) when compiled. Gecko supports for the simultaneous execution of the kernels on both architectures and enables switching between them at the execution time. Moreover, Gecko also supports the inferred execution of computational kernels as discussed in Section 3, which is not supported in OpenMP and OpenACC. With regards to memory management, Gecko bears some similarities to the "Memory Allocators" of OpenMP 5.0; they both accept traits from the developer to allocate the memory. However, Gecko uses its hierarchy to provide those capabilities, while OpenMP 5.0 asks developers to define an allocator per memory space. Furthermore, the hierarchy in Gecko enables developers to easily incorporate new memory hierarchies with minimum efforts by only changing the location, which it is not the case for OpenMP.

10 Conclusion and Future Work

This paper presents Gecko that investigates challenges while developing a hierarchical portable abstraction for upcoming heterogeneous systems. Gecko raises the solution to the high-level, follows

a directive-based mechanism and targets heterogeneous shared memory architectures commonly found in modern high performance systems. Gecko is highly user-friendly, dynamic, flexible and architecture agnostic. Our experiments show how Gecko delivers a scalable solution primarily for benchmarks where the false sharing effect among devices is minimal. In the near future, we will explore the feasibility of automatic data transfer between different locations with Gecko.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant MCB-1412532 and OAC-1531814.

References

[1] 1993. Modeling parallel computers as memory hierarchies. In *Proceedings of Workshop on Programming Models for Massively Parallel Computers*. 116–123.

[2] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 607–618.

[3] A V Aho, J E Hopcroft, and J D Ullman. 1973. On Finding Lowest Common Ancestors in Trees. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC '73)*. ACM, New York, NY, USA, 253–265.

[4] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. 2014. Abstract machine models and proxy architectures for exascale computing. In *Proceedings of Co-HPC 2014: 1st International Workshop on Hardware-Software Co-Design for High Performance Computing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis*. 25–32.

[5] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 320–332.

[6] R Banakar, S Steinke, Bo-Sik Lee, M Balakrishnan, and P Marwedel. 2002. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*. 73–78.

[7] M Bauer, S Treichler, E Slaughter, and A Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

[8] Luca Benini and Giovanni De Micheli. 2002. Networks on chips: a new SoC paradigm. *Computer* 35, 1 (2002), 70–78.

[9] Javier Cabezas, Marc Jordà, Isaac Gelado, Nacho Navarro, and Wen-mei Hwu. 2015. GPU-SM: shared memory multi-GPU programming. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs - GPGPU 2015 (GPGPU-8)*. ACM, New York, NY, USA, 13–24.

[10] B. L. Chamberlain, D. Callahan, and H. P. Zima. 2007. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications* 21, 3 (aug 2007), 291–312.

[11] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05 (OOPSLA '05)*. ACM, New York, NY, USA, 519.

[12] Loïc Cudennec. 2018. Software-Distributed Shared Memory over Heterogeneous Micro-server Architecture. In *Euro-Par 2017: Parallel Processing Workshops*. Springer International Publishing, Cham, 366–377.

[13] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In *International Conference on High Performance Computing*. Springer International Publishing, 489–507.

[14] Robert H. Dennard, Fritz H. Gaensslen, Y. U. Hwa-Nien, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. 1999. Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions. *Proc. IEEE* 87, 4 (1999), 668–678.

[15] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. 2007. Power Provisioning for a Warehouse-sized Computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 13–23.

[16] Kayvon Fatahalian, William J. Dally, Pat Hanrahan, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, and Alex Aiken. 2006. Sequoia: programming the memory hierarchy. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC '06)*. ACM, New York, NY, USA.

[17] Millad Ghane, Mohammad Arjomand, and Hamid Sarbazi-Azad. 2014. An opto-electrical NoC with traffic flow prediction in chip multiprocessors. In *Proceedings of 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '14)*. 440–443.

[18] Millad Ghane, Sunita Chandrasekaran, and Margaret S Cheung. 2019. Gecko: Hierarchical Distributed View of Heterogeneous Shared Memory Architectures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '19)*. ACM, New York, NY, USA, 21–30.

[19] M. Ghane, S. Chandrasekaran, R. Searles, M.S. Cheung, and O. Hernandez. 2018. Path forward for softwarization to tackle evolving hardware. In *Proceedings of SPIE - The International Society for Optical Engineering (SPIE '18)*.

[20] Millad Ghane, Jeff Larkin, Larry Shi, Sunita Chandrasekaran, and Margaret S. Cheung. 2018. Power and Energy-efficiency Roofline Model for GPUs. In *arXiv*. arXiv:1809.09206

[21] Millad Ghane, Abid M. Malik, Barbara Chapman, and Ahmad Qawasmeh. 2015. False sharing detection in OpenMP applications using OMPT API. In *International Workshop on OpenMP (IWOMP '15)*. Springer International Publishing, 102–114.

[22] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. 2018. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. *arXiv* (2018).

[23] HPC Wire. 2019. Cerebras to Supply DOE with Wafer-Scale AI Supercomputing Technology.

[24] N Jayasena, D P Zhang, Amin Farmahini Farahani, and Mike Ignatowski. 2015. Realizing the Full Potential of Heterogeneity through Processing in Memory. In *3rd Workshop on Near-Data Processing*.

[25] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview. *Queue* 11, 7 (2013), 40.

[26] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin Herbordt. 2014. An investigation of Unified Memory Access performance in CUDA. In *IEEE High Performance Extreme Computing Conference (HPEC '14)*. 1–6.

[27] Gabriel H. Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *Proceedings of International Symposium on Computer Architecture (ISCA '08)*. 453–464.

[28] Jemalloc: memory allocator. 2019. http://jemalloc.net/.

[29] NVIDIA PSG Cluster. 2017. http://psgcluster.nvidia.com/trac.

[30] Matheus Almeida Ogleari, Ye Yu, Chen Qian, Ethan L Miller, and Jishen Zhao. 2019. String Figure: A Scalable and Elastic Memory Network Architecture. In *the Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*.

[31] ORNL's Summit Supercomputer. 2018. https://www.olcf.ornl.gov/for-users/system-user-guides/summit/.

[32] Sabine Cluster. 2019. https://www.uh.edu/cacds/resources/hpc/sabine.

[33] Daniel J Scales and Kourosh Gharachorloo. 1997. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 157–169.

[34] T R W Scogland, W Feng, B Rountree, and B R de Supinski. 2015. CoreTSAR: Core Task-Size Adapting Runtime. *IEEE Transactions on Parallel and Distributed Systems* 26, 11 (2015), 2970–2983.

[35] TCMalloc: Thread-Caching Malloc. 2018. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[36] Top500. 2019. https://www.top500.org.

[37] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericas. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 1–1.

[38] Dong Hyuk Woo, Nak Hee Seong, Dean L. Lewis, and Hsien-Hsin S. Lee. 2010. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA '10)*. 1–12.

[39] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[40] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2010. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 172–187.

[41] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. 2015. VirtCL: a framework for OpenCL device abstraction and management. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP 2015 (PPoPP 2015)*. ACM, New York, NY, USA, 161–172.