NEMESYS: Near-Memory Graph Copy Enhanced System-Software

Sven Rheindt
Technical University of Munich
sven.rheindt@tum.de

Andreas Fried
Karlsruhe Institute of Technology

Oliver Lenke
Technical University of Munich

Lars Nolte
Technical University of Munich

Thomas Wild
Technical University of Munich

Andreas Herkersdorf
Technical University of Munich

ABSTRACT

Despite tackling the memory and power walls over the last decades, new challenges for manycore architectures arose due to the emergence of ever increasing memory intensiveness of applications with big, irregular and cache unfriendly data sets. As data-to-task locality is of key importance for system performance, the MEMSYS 2017 keynote speaker Peter Kogge showed evidence for the so-called "locality wall", that paved the path to near- and in-memory computing. The reduction of data movement is especially challenging on tile-based architectures with physically distributed memory as they often omit inter-tile cache coherence and thus require a different programming model (e.g. PGAS).

Inter-tile communication in the PGAS paradigm is allowed via a remote procedure call (RPC)-like programming language construct. The more modern PGAS languages are object-oriented and thus the RPC mechanism requires object graphs to be copied between tiles. It is the system-software's job to provide an efficient implementation of it since the transfer of such object graphs is crucial for the performance of object-oriented applications on PGAS architectures.

We therefore propose NEMESYS: NEar-Memory Graph Copy Enhanced SYstem-Software, which outsources the memory-intensive and cache unfriendly graph copy operation to near-memory hardware accelerators. As NEMESYS is an efficient implementation of the PGAS RPC, it integrates these near-memory accelerators into the system-software, opaque to the application programmer.

We integrated NEMESYS into an FPGA prototype and a distributed operating system running on a 4x4-tile design with a total of 56 application cores and two memory tiles. The evaluation with the X10 IMSuite benchmarks, featuring distributed graph algorithm kernels, showed a speedup in execution time between 1.35x and 3.85x compared to a state of the art approach. The overall reduction in communication time was between 40% and 82%.

KEYWORDS

Near-Memory Computing, Graph Copy Accelerator, PGAS, System-Software, Data-to-Task Locality

MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357545

ACM Reference Format:
Sven Rheindt, Andreas Fried, Oliver Lenke, Lars Nolte, Thomas Wild, and Andreas Herkersdorf. 2019. NEMESYS: Near-Memory Graph Copy Enhanced System-Software. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3357526.3357545

1 INTRODUCTION

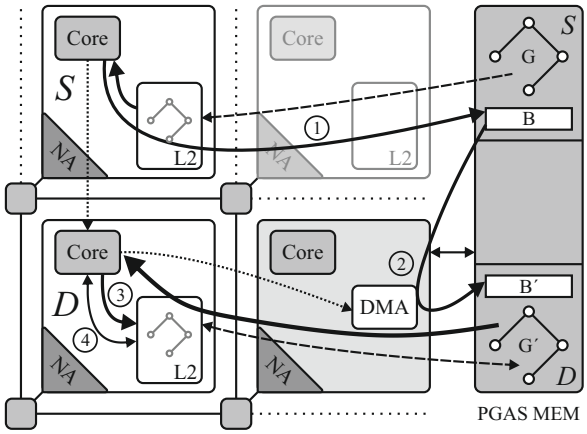
Over the last decades, computer architecture overcame several performance hindering obstacles. The memory and power walls have been tackled by integrating manycore systems with sophisticated memory architectures and cache hierarchies, as well as shifting to tile-based architectures with physically distributed memories and processing nodes [10, 12].

Data-to-task locality plays a vital role for the performance of applications on such architectures. However, the emergence of memory intensive applications – dominated by data access and movement, with big and irregular data sets that become more and more cache unfriendly – poses new challenges. The MEMSYS 2017 keynote speaker Peter Kogge therefore showed evidence that a new wall, the so-called "locality wall", exists and has to be overcome [30]. Not only is this wall hindering performance, but data-to-task locality is also highly responsible for the energy footprint of an application. Many recent approaches therefore reduce data movement by leveraging in- or near-memory computing [1, 21, 26, 29, 35, 40].

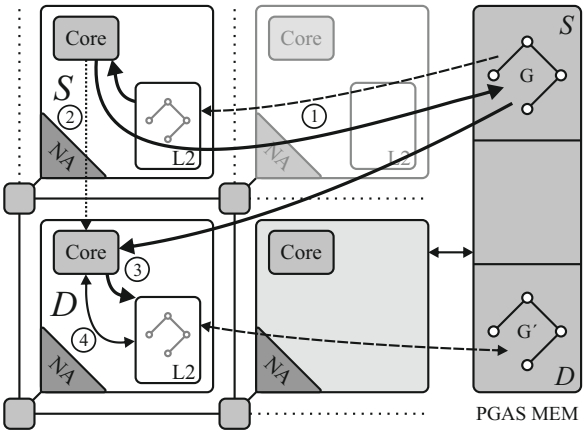
Tile-based architectures further face the challenge of providing efficient and scalable inter-tile cache coherence and consistency. The community is therefore unclear whether global coherence is here to stay [4, 24]. While some architectures provide global [12] or partial coherence [37], others omit hardware support for inter-tile cache coherence and consistency [6, 7, 14, 20, 22, 31]. As those non-coherent architectures do not easily support the shared memory programming model, a different paradigm with explicit inter-tile communication via messages has to be used.

One example is the partitioned global address space (PGAS) programming model [11, 17, 19, 33], which divides the global address space into partitions and assigns each of them to one tile. Threads running on a tile may only freely access data in its partition of memory. To access another tile's data, the thread needs to use a special, remote procedure call (RPC)-like programming language construct [19, 33]. In order to assist the inter-tile message passing and relieve the processing cores of this duty, such architectures usually provide dedicated hardware support, like a direct memory access (DMA) engine, that speeds up copying of flat (i. e. non-pointered) data.

Besides all hardware architectural developments, data structures in modern programming are getting more complex. In the past,



(a) Message passing. (1) Core on S serializes G to source buffer B (2) DMA unit transfers B to destination buffer B' (3) Core on D deserializes B' to G' in destination partition D (4) Core on D uses G'.



(b) Pegasus. (1) Core on S writes back G to source partition S (2) Core on S signals D to begin copying (3) Core on D copies graph to G' in destination partition D (4) Core on D uses G'.

Figure 1: Different approaches to graph copy. In both cases, an object graph is to be copied from tile S to tile D. Bold arrows denote bulk data transfer, dotted arrows denote control/metadata messages, dashed arrows denote possible traffic due to cache misses/evictions. To write back a graph to the memory a complete graph traversal is needed.

data was simply structured in arrays and records, whereas today's high-level languages (e.g. Java) organize data in *object graphs* with several objects pointing to each other.

We consider the PGAS programming language X10 [33], where pointered data structures are prevalent as well. As mentioned above, the PGAS programming model requires a special language construct to transfer data between tiles. In X10 this RPC construct is called "`at` statement", which has the form **`at (p) func`**, where `p` denotes the destination *place* [1] and `func` is the function to be executed there. This function may access its *lexical environment* [2] on the remote place `p`. To support this, the run-time system implicitly creates an object graph (the *closure*) including all variables and objects visible at the point of the `at` statement. It then transfers the closure to the destination place's partition, executes the function on the destination place, and transfers the result (if present) back to the source partition. To ensure that all object pointers stay intact, a *graph copy* with proper pointer adjustment is necessary.

Hence, a good amount of effort has been invested into accelerating the transfer of such object graphs [25, 27]. As object graphs are in general pointered data structures, a DMA engine is not able to directly copy them without using costly (de-) serialization. Otherwise, all the pointers in the copied graph would still point to the objects in the original graph.

In the classical message passing variant, as depicted in Figure 1a, the following steps are taken: (1) a core on the source tile S has to serialize the object graph G into a buffer B, (2) the flattened structure B is transferred via DMA into a buffer B' on the destination tile D, (3) a core on D deserializes B' and reconstructs the graph G'.

Some message passing approaches either reduce the need for the buffer B' [8, 38], or even avoid the (de-) serialization overhead [25, 27]. Mohr et al. presented Pegasus [25], an efficient way to perform serialization-free graph copy operations in software. Since data is crossing cache coherence boundaries, the run-time system needs to manage the involved caches. As depicted in Figure 1b, the Pegasus approach performs the following steps: (1) the sender S has to traverse the graph G and write back[3] all objects of G from its caches to its memory partition S, (2) S then signals and passes metadata to the receiver D, (3) D in turn copies the graph directly from the partition S into its partition D.

Despite the benefits of this approach, the aforementioned data-to-task locality issue has not been resolved. Since the graph copy operation (3) is performed via remote load-store operations through the network-on-chip (NoC), its performance is limited. Furthermore, the operation is not only performed "far from memory", but also an accelerator that relieves the CPUs from the graph copy duty is missing (similar to a DMA unit for non-pointered data).

Due to the high importance of data-to-task locality and the relevance of efficiently transferring object graphs, we propose NEMESYS: NEar-Memory Graph Copy Enhanced SYstem-Software for tile-based manycore architectures. We integrate near-memory and near-cache graph (copy) accelerators into an architecture-aware system-software. This synergy provides a performant and scalable solution to mitigate the locality wall.

2 RELATED WORK

Near-memory accelerators have seen much interest recently [1, 18, 26, 29, 35, 39, 40], as they promise to bridge the widening gap between processor and memory performance, i. e. the memory wall.

Yitbarek et al. [40] proposed near-memory accelerators for four different memory intensive tasks, namely *string-compare*, *memcpy*, *sorting* and *hashtable lookup*. These operations are widely used by many different applications. They integrated them into the bottom-layer of each vault of a Hybird-Memory-Cube (HMC) [10].

[1] In the X10 language, a tile and its memory partition are represented as a *place*.
[2] The *lexical environment* denotes variables that were defined outside the `at` statement

[3] We adhere to the following differentiation in cache terminology: 1) invalidate a cache line, 2) writeback of a cache line to memory without invalidating it and 3) flush, which is a combination of invalidate and writeback.

Table 1: Related work feature comparison.

Related Work		Near-Memory	HW-Acc	System-Software	object oriented	graph copy	PGAS
Yitbarek	[40]	✓	✓	✗	✗	✗	✗
Maas	[23]	✗	✓	✓	✓	✗	✗
Nguyen	[27]	✗	✗	✓	✓	✓	✗
Mohr	[25]	✗	✗	✓	✓	✓	✓
NEMESYS		✓	✓	✓	✓	✓	✓

Near-memory accelerators have also been built for even more complex operations: As examples for more numerically-oriented tasks, Neggaz et al. [26] have developed a near-memory accelerator for matrix multiplication, and Schuiki et al. [35] use an accelerator to improve the performance of training neural networks. These numerical tasks are highly regular, so the accelerators can easily iterate over the data in hardware.

Ahn et al. [1], Ozdal et al. [29], and Li et al. [21] presented different near-memory accelerators to efficiently process graphs. Their approaches partition the graphs in memory and use several graph processors in parallel to compute metrics such as PageRank on the graphs. They attain flexibility by combining fixed-function accleratorsfor common patterns such as scatter-gather with programmable processing units.

Many of these near-memory accelerators have in common that they are implemented on the HMC. However, they only use HMC's multiprocessing and distributed memory features for data parallelism. They do not address the challenges arising from multiple tasks acessing data concurrently.

Concerning accelerators dealing with object graphs, much work has been done on hardware-assisted or fully hardware-implemented garbage collection. Maas et al. [23] presented an accelerator that implements a concurrent mark-and-sweep garbage collector. However, the garbage collector is tightly integrated with the CPU rather than a separate near-memory unit. Bacon et al. [3] also developed a hardware garbage collector, but they target systems implemented on an FPGA, without necessarily having a CPU. Their garbage collector is also not near-memory integrated.

On the other hand, there is also previous work that improves object graph handling in software. Nguyen et al. [27] presented an approach to transfer objects between Java Virtual Machine heaps without full serialization. Their approach uses an object model where objects can be transferred almost as they are to an intermediate buffer, sent over the network, and put directly into the remote heap. The sender and receiver only slightly adjust the objects, e. g. correcting for different heap origin addresses.

For systems which have a common address space, such as PGAS systems, Mohr and Tradowsky [25] presented the more efficient Pegasus approach. They do not need an intermediate buffer to transfer an object graph, because the receiver can remotely read the sender's memory. Their approach allows for the sender's and receiver's caches to be incoherent, and synchronizes them in software. However, they did not use hardware support except for range-based cache operations.

Our work represents a synergy of the aspects discussed above: We leverage near-memory accelerators to work on object graphs in hardware, and integrate them into the runtime software of a PGAS system. Our system is thereby able to copy pointered data between the heaps of a PGAS system more efficiently.

Although we chose X10 [19, 33] as the programming language to implement our work in, it applies to other languages as well. First and foremost, the Chapel language [19] is quite similar to X10: Activities run on *locales* (X10's places) and can migrate between them with the RPC-like *on statement* (X10's `at`). While on the other locale, they can implicitly access their lexical environment. Therefore, given a suitable hardware architecture, Chapel can also benefit from our work.

In contrast, earlier PGAS languages like UPC [11] use the single-program-multiple-data (SPMD) paradigm. This means that they have no mechanism to migrate activities, and instead provide explicit remote load-store primitives. To transfer a whole object graph, the application programmer needs to write specialized code for the concrete type(s) to be copied. We expect this approach to become obsolete in future.

3 NEMESYS CONCEPT

Our NEMESYS approach mitigates the locality wall by leveraging near-memory accelerated graph copy operations that are inherently integrated into the system-software of a tile-based manycore architecture. NEMESYS is an efficient implementation of the PGAS remote procedure call and follows the hardware-software co-design approach that is illustrated in Figure 2. Similar to the Pegasus RPC (Figure 1b), the writeback of the source graph G to memory (1), the signaling of the receiver D (2), the copying of the graph G to G' (3) and the execution of the function `func` (4) have to be performed. However, the actual memory-intensive and cache unfriendly graph writeback (1) and graph copy operations (3) are outsourced to near-cache (NCA) and near-memory accelerators (NMA), respectively. The following sections provide a brief overview of the NEMESYS concept, while a more detailed description of the architecture is given in Section 4.

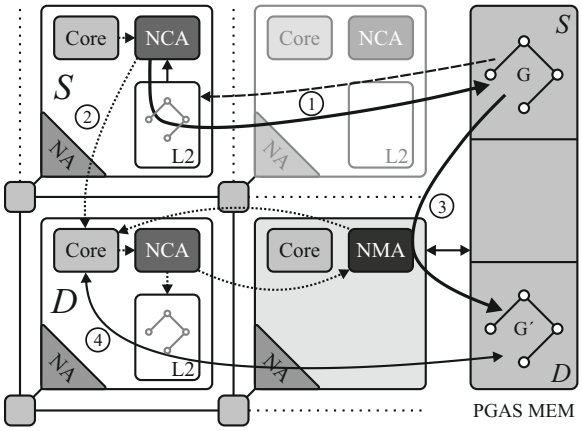


Figure 2: NEMESYS. Bold arrows denote bulk data transfer, dotted arrows denote control/metadata messages, dashed arrows denote possible traffic due to cache misses/evictions. (1) NCA writes back G to source partition S (2) NCA signals core on D (3) NMA copies G to G' (4) Core on D uses G'.

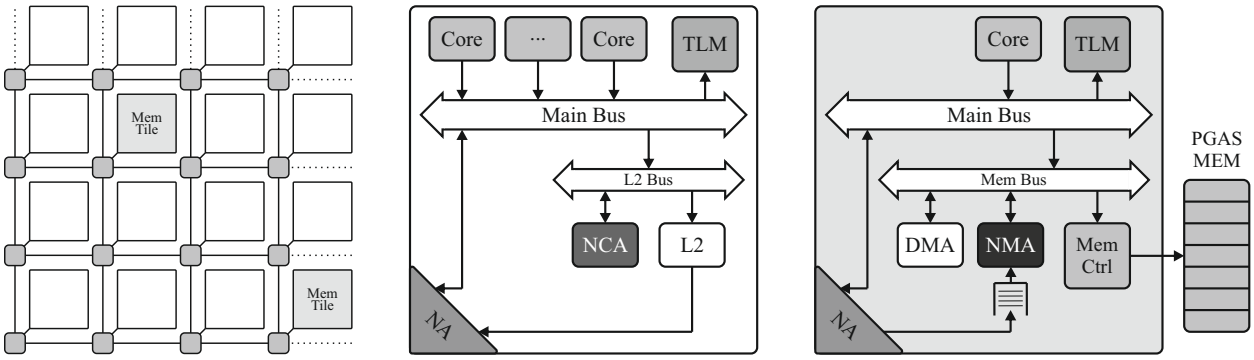


Figure 3: An overview of our hardware platform. Left: The 4x4 tile design with two memory tiles and the NoC. Unmarked tiles are compute tiles, the small gray squares are NoC routers. Center/Right: Block diagrams of a compute and memory tile respectively. TLM: Tile-local memory, NA: Network adapter. The CPU core on the memory tile only runs the operating system. NEMESYS adds the near-cache accelerator (NCA) to the compute tiles, and the near-memory accelerator (NMA) to the memory tiles. The requests to the NMA are buffered in a FIFO.

3.1 Near-Memory Integration

The key feature of NEMESYS is its near-memory and near-cache integration. We thereby achieve increased data-to-task locality, whose absence is a major reason for the locality wall. Data movement is reduced by bringing the graph copy operation close to the memory instead of copying it remotely via the NoC. This not only lowers data access latencies, but also decreases the NoC traffic, resulting in an overall performance increase and energy savings.

Contrary to the approach presented by Mohr et al. [25], the receiving tile D no longer has to perform the graph copy operation remotely through its L2 cache. Therefore no unnecessary cache pollution arises by the graph copy operation. This is especially beneficial and important when the used data sets outgrow the available cache capacity, which is a realistic scenario for many applications [30]. We analyze this phenomenon in Section 5.4.6.

3.2 Graph Accelerators

The efficient support of object-oriented programming languages for PGAS architectures is a key driver for NEMESYS. Integrating a graph copy accelerator is advantageous for multiple reasons: (1) the cores are relieved from the graph copy duty, (2) a dedicated hardware module works more efficiently in terms of performance, power, as well as resources, (3) it is the natural replacement or enhancement of a DMA unit to efficiently support graph- and pointer-based data structures.

As depicted in Figures 2 and 3, each memory tile is equipped with a near-memory accelerator (NMA) for graph copy that can be triggered by any CPU in the system. The NMA has a FIFO to buffer incoming requests. It automatically creates back-pressure, so that no global locking is required in software. Upon completion, the graph copy unit directly spawns a task to a core on the receiving tile D, that executes the function `func` of the `at` statement **`at (D) func`** on the copied graph G'.

The proposed accelerator is capable of copying arbitrarily structured and sized object graphs by leveraging (1) the idea of pointer reversal introduced by Schorr and Waite [34], as well as (2) an extension of the object model as will be described in Section 4.3.1.

As tile-based architectures generally contain several memory tiles in order to avoid access hot-spots, NEMESYS is also able to efficiently copy graphs between memory partitions located on different tiles. The corresponding advanced mechanism is described in Section 4.2.

As also depicted in Figures 2 and 3, each compute tile is equipped with a near-cache accelerator unit (NCA) that is capable of two different operations: (1) traversing an arbitrary object graph and issuing cache writeback commands for each of the objects. Upon completion, it can directly dispatch a user-defined task to the receiving tile without additional system calls on sender side. (2) Performing range-based cache operations (similar to [25]) with a subsequent triggering of the graph copy accelerator with user-defined parameters.

3.3 System-Software Integration

The NEMESYS approach tightly integrates near-memory and near-cache accelerators at system-software instead of application level. In contrast to several approaches [1, 21, 26, 29, 35] that utilize near-memory accelerators directly in the application, NEMESYS does not require any changes to the API or the application code. While maintaining ease of programmability, the application programmer can profit from the benefits of NEMESYS for all applications that use PGAS remote procedure calls.

During the NEMESYS RPC, explained in Section 4.1, the graph copy related operations are asynchronously offloaded to the NCA and NMA, respectively. The involved cores are therefore not only relieved of the graph writeback and copy duties, but can in parallel cope with other tasks instead of synchronously waiting on the completion of the outsourced operations. The accelerators are therefore equipped with task spawning capability to avoid unnecessary system calls. The whole transfer of the graph G from partition S to partition D, including the necessary cache management, can thus be performed with minimal software involvement. Only the allocation of the destination buffer on tile D and the triggering of the near-cache and near-memory accelerators remains a software task.

4 NEMESYS ARCHITECTURE

Having described NEMESYS conceptually, we now present the architectural contribution. We describe the employed hardware-software co-design approach with special focus on the system-software integration (Section 4.1) and the handling of inter-memory copying (Section 4.2). We then give an overview of the necessary extensions to the object model (Section 4.3.1) and the proposed hardware units (Section 4.4). We further provide an abstract overview of the hardware graph copy algorithm (Section 4.6) and details on an efficient copy map (Section 4.5).

4.1 Hardware-Software Co-Design

We have already described the basic workings of a PGAS remote procedure call in Section 1. Now, we present the necessary steps in more detail, and also point out where hardware acceleration units come into play. Figure 4 also illustrates the steps in a message sequence chart.

We assume that an RPC call is to be made from the source tile S to the destination tile D. In order to manage the RPC, the source tile has to transmit some metadata, which is stored in the structure M. This metadata includes the pointer G to the closure, its size, and synchronization objects. The NEMESYS RPC then takes the following steps:

(1a) The sender S allocates a metadata buffer M' on D.
(1b) It then allocates and sets up a metadata structure M that needs to be transfered to the receiver D (into M').
(1c) S triggers the NCA *graph writeback and dispatch unit*, accompanied by the descriptor of task T_1.
(1d) The NCA on tile S writes back the source graph G into its memory partition and additionally measures G, before it
(1e) appends the measured graph size to the metadata M.
(2) The NCA on tile S initiates a DMA of the metadata M to M' with subsequent invocation of the task T_1 on D.
(3a) Based on the metadata information M', T_1 (on D) allocates a destination buffer G' in the memory partition of D, which is to hold the copy of G.
(3b) D triggers the NCA *invalidate-and-trigger unit*, accompanied with the command to trigger the NMA *graph copy unit*.
(3c) The NCA on tile D invalidates G' before it triggers the NMA to avoid cache evictions during the graph copy.
(3d) The NMA copies the graph G to G' and spawns a task T_2 on D upon its completion.
(4) T_2 then executes the function `func` on the copied graph G'.
(5) Either, the termination of the remote procedure call is signaled back to S, or the result graph of (4) is copied back to S applying the same mechanism (while reusing the metadata structures M and M').

Owing to the asynchronous offloading to the NCA and NMA, the cores on S are relieved from RPC duties between steps (1c) to (5). The cores on D are only needed for the RPC during (3a),(3b) and (4). We thus provide an efficient remote procedure call with minimal software involvement.

As the core's L1 caches follow a write-through policy to allow for tile-local snooping-based coherence, they only need to be invalidated at the beginning of step (3b) and (4).

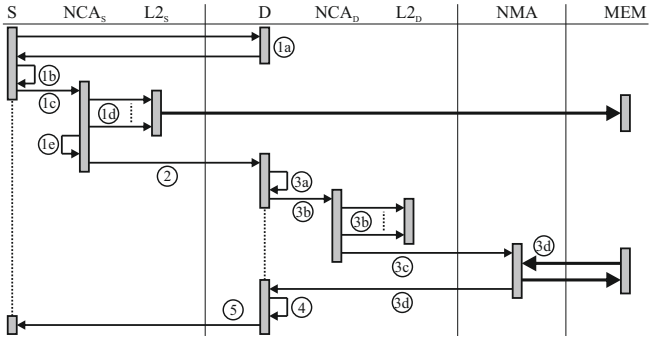


Figure 4: Message sequence chart illustrating the hardware-software co-design approach followed by the NEMESYS remote procedure call.

4.2 Inter-Memory Graph Copy

To cope with several physical distributed memories, NEMESYS is capable of an efficient *inter-memory* graph copy mechanism. If the involved partitions are located on different physical memories, a *near-memory* graph copy is not directly possible, as the NMA could only access G' remotely over the NoC via individual load-store operations.

We therefore copy G via an intermediate buffer G* located in the same physical memory as G. This requires a slight modification (marked in **bold**) of the steps (1c)-(2).

(1c') S triggers the NCA$_S$ *graph writeback and dispatch unit*, **now** accompanied by the descriptor of task $\mathbf{T_3}$.
(1d) The NCA$_S$ writes back the source graph G into its memory partition and additionally measures G, before it
(1e) appends the measured graph size to the metadata M,
(1f') **and invokes the task T_3 locally on S itself.**
(1g') **T_3 (on S) allocates and invalidates the intermediate buffer G* and appends a reference of it to the metadata M.**
(2') **S then** initiates a DMA of the metadata M to M' with subsequent invocation of the task T_1 on D.

In step (3c), the graph copy unit located in the tile of the source graph G is triggered. This graph copy unit then follows a two-step

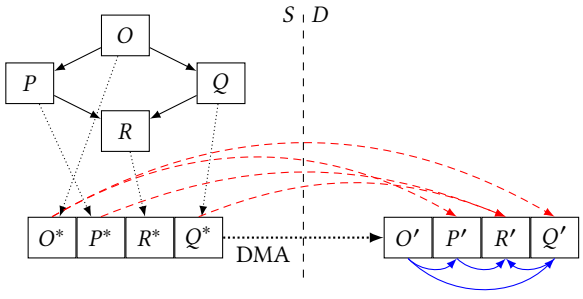


Figure 5: Inter-memory graph copy of the source graph G to the destination graph G' via the intermediate buffer G*. The graph copy unit writes intermediate pointers (dashed, red) to G*, which become valid (blue) in G' after the DMA.

procedure for step (3d), also depicted in Figure 5: (1) it copies G into the intermediate buffer G* with all pointers adapted (dashed, red) to point to the final addresses in the destination buffer G', (2) G* is transferred to G' via normal DMA over the NoC.

There is no deserialization or other postprocessing needed at the final destination. This is possible because the NMA knows about the address of the final destination buffer. It can therefore set up the pointers in the intermediate buffer (red dashed arrows in Figure 5) in such a way that they match up after the DMA (blue arrows in Figure 5).

The DMA is triggered by the graph copy unit and is performed asynchronously, so that the graph copy unit can already serve the next job from its FIFO queue. The task T_2 is passed alongside the DMA and is spawned onto tile D by the DMA-unit upon the completion of the inter-memory data transfer. The performance overhead of the additional DMA will be evaluated in Section 5.4.3.

4.3 Object Model

Before we describe the functionality of the graph accelerator hardware units (Section 4.4), we present the underlying object model and its necessary extension to be compatible with the accelerator. See Figure 6 for an example object.

We use a simple object model for our prototype, which supports Java-like object-oriented languages. Each object begins with a *header* followed by the *payload*. Each word (32 bits) of payload is either a piece of primitive data, a pointer to another object, or part of an *array descriptor*. An array descriptor consists of two words. The first is a pointer to the array's *backing store*, which holds the data, and the second holds the number of elements in the array. The object model allows arrays of primitive data or of pointers to other objects.

The object header contains only the *vptr* (virtual pointer), which points to the *vtable* (virtual table) of the object's class. The first word of the *vtable* in turn points to the class's *RTTI* (run-time type information) structure; the following words hold the function pointers to the class's virtually bound methods.

The RTTI structure contains class metadata used by run-time operations such as checked type casts or reflection. Most importantly for our purpose, it contains the size of the objects of this class, and the *pointer mask*.

The pointer mask provides metadata about the object's memory layout. We follow the example of other runtime systems such as the HotSpot VM [13] or the Go run-time [2], where similar metadata is present to support the garbage collector. There is one pointer mask for each type, which is located at the end of the type's RTTI structure, so that it can have variable size. The pointer mask is a bit field, where two bits correspond to each word in an object. This gives us four kinds of words to distinguish:

- `00` marks a word of data
- `01` marks a pointer to another object
- `10` marks a *transient* word (pointer or data)
- `11` marks the first word of an array descriptor

A transient word is one that must not be copied to the receiver's place but set to 0. Classes designate words as transient if they only use them for caching data which the receiver can easily recompute.

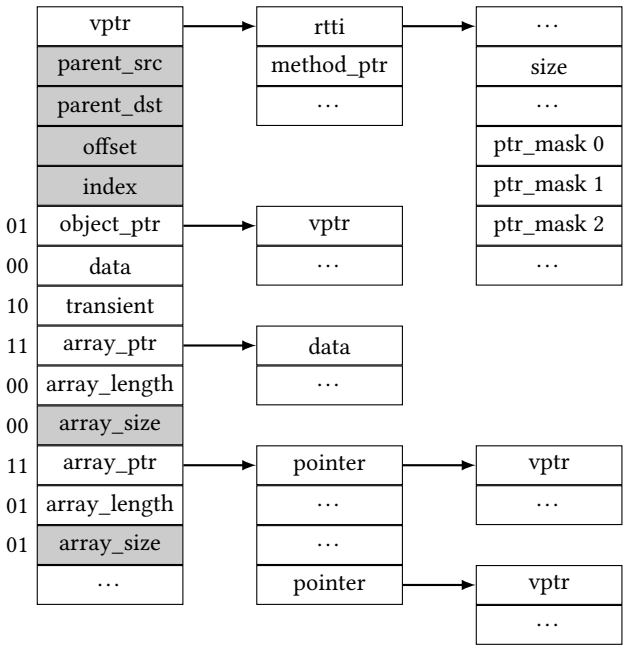


Figure 6: Layout of an example object. Members added to support NEMESYS are highlighted in gray.

To further distinguish between arrays of primitive data and arrays of pointers, we use the pointer mask bits corresponding to the second word of the array descriptor. Thus, arrays of data are marked `1100` and arrays of pointers are marked `1101`.

4.3.1 Object Model Extensions. We extend the object model in two places in order to support the accelerator.

First, we add the size of the array in bytes to the array descriptor. The compiler can compute this size from static type information, but the accelerator unit needs it explicitly.

Second, the traversal algorithm needs some scratch space in every object. We therefore extend the object header by four additional words after the vptr. These are collectively known as the object's *transition structure*. We discuss its purpose in detail in Section 4.6.

4.4 Graph Accelerator Hardware Units

The near-cache and near-memory accelerator units, proposed in Section 3.2 and described below in more detail, are deliberately placed on dedicated buses next to the memory or cache, respectively. This allows for their faster access, less load on the main bus, as well as independence of the particular memory or cache controller.

4.4.1 The NCA Range-Operations Unit. Each NCA is equipped with a range-operations unit that can perform cache operations on a continuous address range, thereby relieving the CPUs of potentially long-lasting cache operations. A range-operations unit was already part of the Pegasus system [25], but we extend it with the capability to directly trigger the NMA upon completion of the cache operations. We refer to this extended unit as the *invalidate-and-trigger unit*.

4.4.2 The NCA Graph-Writeback-and-Dispatch Unit. Additionally, each NCA features a graph writeback unit that traverses an object graph and issues writeback commands for every cache line of each object. The cache carries out these writeback commands if the corresponding cache line is in modified state, else returns immediately. In addition, the NCA counts the number of objects in the graph, and sums up their total size including array backing stores, obtained from the objects RTTI. The graph writeback follows a simple breadth-first-search algorithm: (1) It issues writeback commands for the whole object. (2) Each pointer in an object is pushed to a stack of outstanding objects, that is statically allocated in the tiles memory partition. (3) It pops the next object from the stack and proceeds with (1) until the stack is empty.

In order not to revisit objects in a cyclic graph, the NCA writes a marker into the transition structure of each processed object, which is checked before the writeback is issued.

To support asynchronous offloading, the graph writeback unit can directly spawn tasks, even on other tiles. The core triggering the NCA passes it the metadata gathered so far and a descriptor of the task T_1. After that, the core is no longer involved in the process.

When the writeback is complete, the NCA stores the object count and total graph size in the metadata M, issues a DMA transfer of M to M', and invokes the task T_1 on the receiving tile D.

4.4.3 The NMA Graph-Copy Unit. The core component of the NEMESYS approach is the near-memory graph copy accelerator, which is capable of copying an object graph without intermediate software interaction. To avoid these up-calls to the operating system, the receiver D allocates an appropriately sized *destination buffer* beforehand, based on the result of the graph writeback unit present in the metadata M'. The graph copy unit then uses an internal bump allocator to allocate the objects of the copied graph consecutively in the destination buffer.

To avoid using a separate recursion stack, the graph copy algorithm builds on the idea of pointer reversal introduced by Schorr and Waite [34]. However, as we do not only deal with cons cells, our algorithm is more complicated, and uses the transition structure in the objects in G' to keep its state. We describe the algorithm in more detail in Section 4.6.

We cannot use the transition structures of objects in G for two reasons: (1) Another thread could be preparing to copy parts of G concurrently. During its step (1d), it would overwrite G's transition structure with stale data from the tile's L2 cache. (2) As G is still in the L2 cache, it may be evicted if another cache line (even from an unrelated thread) aliases with it. Again, the eviction would overwrite G's transition structure. Therefore, we use the transition structures of objects in G', as G' resides in its own buffer, which is not in any L2 caches and not yet accessible to user threads.

In addition to handling recursion, we also need to detect cycles in the object graph, or identify objects which we have already started copying. Besides identifying them, we also need to obtain the address of their copies in order to set the pointers to them. For example, consider the object graphs in Figure 7. Assuming we have already copied O, P, R to O', P', R' and are now copying Q to Q', we must not make a new copy of R, but Q' must point to R' instead.

Therefore, we use a separate *copy map* to associate objects with their copies. To find out whether we have already created a copy of the object R, we search for R in the copy map. If R is found in the copy map, it returns R'. Otherwise, if the search returns NULL, we create a new object R' and add the mapping $R \mapsto R'$ to the copy map. We will now discuss the copy map in more detail.

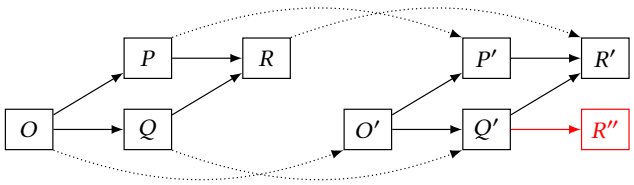


Figure 7: A simple example of a cyclic object graph: Object R is reachable from O in two ways. A correct graph copy still must only make one copy of R, and not the separate R'' marked in red.

4.5 Copy Map

We set aside a statically allocated buffer for the copy map in every memory tile. This buffer is divided into two halves. To store the mapping $R \mapsto R'$ in the copy map, we store the address of R at a certain offset in the first half, and the address of R' at the same offset in the second half. The offset where R is stored is decided by the implementation chosen for the copy map.

The graph copy unit contains two such implementations, one based on linear search, and one based on hashing. They provide a trade-off between the initial setup time and the scaling behavior of the copy map.

The linear search implementation stores the original addresses consecutively in the first half of the copy map, and the addresses of the copies in the same order in the second half. Searching for an object requires iterating over the list. This method therefore scales in $O(o^2)$ for object graphs of size o.

The second copy map implementation improves the performance for large graphs by using a hashtable. For each new mapping $R \mapsto R'$, R is inserted at the offset hash(R) in the first half, with collisions being resolved by linear probing. R' is inserted at the same offset in the second half.

We use the family of universal hash functions H_3 by Carter and Wegman [5]: To hash n bits of input down to k bits, H_3 defines a set of functions h_M parameterized by a $k \times n$ bit matrix M. The hash $h_M(x)$ is then given by $h_M(x) = \bigoplus_{i=0}^{n-1} M_i x_i$, where M_i is the i-th column of M, x_i is the i-th bit of x, and $\oplus$ is the exclusive or operation. This function is easily and cheaply implemented in hardware, requiring nk gates for a fixed matrix.

In order to recognize empty slots, the first half of the hashmap has to be initialized by zeroing it. We keep this overhead as low as possible, by using hashmaps of variable size depending on the number of objects in the graph. For an object graph with o objects, the hashmap has $2^{\lceil \log_2 o \rceil + 1}$ slots. This yields a loading factor of between 50 % (if o is a power of two) and $\approx$ 25 % (if o is a power of two + 1). Since the number of hashmap slots is always a power of two, we can simply use fewer bits of the hash function's output to select a slot.

The decision between the two variants, as well as the size of the hashmap can be changed dynamically at run-time for every triggering of the unit.

4.6 Hardware Graph Copy Algorithm

The two main operations of the graph copy algorithm are *advancing* to a new object, and *retreating* to an object already visited. See Algorithm 1 for a summary in pseudo code. Note that we present the algorithm in a state-machine style, similar to the hardware implementation.

As an example for advancing, consider the following situation: We are copying an object from address O to O', and find a pointer P at offset δ. Now, we must recursively copy the object at P (say, to P') and then store P' at offset δ in O', i.e., we must *advance* to P, copy it recursively, and then *retreat* to O.

Our concern is that when we retreat from P to O, we can resume copying O. Therefore, we store the following in the transition structure of P': O in the *parent_src* field and O' in the *parent_dst* field. If we are copying a single pointer field, we store its offset δ in the *offset* field of O''s transition structure. On the other hand, if we iterate over an array of pointers, δ is the offset of the array descriptor, and we additionally store the current array index in the *index* field.

When we have finished copying P to P', it is time to *retreat* from it. We have to go back to P's parent and set up to continue where we left off when advancing to P.

First, we obtain O and O' from the transition structure of P', and then δ (and i if necessary) from the transition structure of O'. Then, we store P' at offset δ in O' (or in the array) and increment δ (or the array index). If we are still within O, we continue copying, otherwise we retreat to O's parent.

When we retreat from an object whose parent is NULL, we know that we have completed copying the object graph's root and therefore the whole graph.

5 EVALUATION

In order to evaluate the effectiveness of our approach, we have implemented the hardware units described in Section 4.4 into the prototype platform described below (Section 5.1).

We will first discuss the hardware requirements of our implementation (Section 5.2), and then turn to performance measurements.

To evaluate the performance of the graph copy unit itself, we measure its complexity over a range of microbenchmarks (Section 5.3).

Finally, we also evaluate the performance of NEMESYS in the context of complete applications (Section 5.4). We also measure how using NEMESYS affects the load of the rest of the system (Section 5.4.4), and we investigate the properties of the objects graphs occuring in these "real-world" programs (Section 5.4.1).

5.1 Prototype Platform

We integrated NEMESYS into an existing tile-based MPSoC prototype platform and a distributed run-time system. The prototype implementation features a 4×4 tile design with 14 compute and two memory tiles, arranged as already depicted in Figure 3. The global memory is physically distributed to the memory tiles, which are each connected to an off-chip DDR-3 memory. Each compute tile contains 5 cores with private L1 caches and a *tile-local memory* (TLM), which holds the program text, OS data, and temporary user data. All cores are Gaisler SPARC V8 LEON 3 [9, 36] processors. The

Algorithm 1 The graph traversal algorithm

```
1:  state ADVANCE(O, O', P, δ, i)
2:      O'.offset ← δ
3:      O'.index ← i
4:      P' ← create(size(P))
5:      copyMap[P] ← P'
6:      P'.parent_src ← O
7:      P'.parent_dst ← O'
8:      go to COPYWORD(P, P', 0, 0)
9:  state RETREAT(P')
10:     O ← P'.parent_src
11:     O' ← P'.parent_dst
12:     δ ← O'.offset
13:     if pointerMask(O, δ) = 11 then
14:         i ← O'.index
15:         O'[δ][i] ← P'
16:         go to COPYWORD(O, O', δ, i + 1)
17:     else
18:         O'[δ] ← P'
19:         go to COPYWORD(O, O', δ + 1, 0)
20: state COPYWORD(O, O', δ, i)
21:     if δ ≥ size(O) then
22:         go to RETREAT(O')
23:     if pointerMask(O, δ) = 00 then                ▷ Copy data
24:         O'[δ] ← O[δ]
25:         go to COPYWORD(O, O', δ + 1, 0)
26:     if pointerMask(O, δ) = 01 then                ▷ Copy pointer
27:         P ← O[δ]
28:         if copyMap[P] = NULL then
29:             go to ADVANCE(O, O', P, δ, −1)
30:         else
31:             O'[δ] ← copyMap[P]
32:             go to COPYWORD(O, O', δ + 1, 0)
33:     if pointerMask(O, δ) = 11 then                ▷ Copy array ...
34:         if i = 0 then
35:             bytes ← O[δ + 2]
36:             O'[δ] ← create(bytes)
37:             O'[δ + 1] ← O[δ + 1]
38:             O'[δ + 2] ← bytes
39:         if i ≥ O[δ + 1] then
40:             go to COPYWORD(O, O', δ + 3, 0)
41:         if pointerMask(O, δ + 1) = 00 then        ▷ ... of data
42:             memcpy(O'[δ], O[δ], bytes)
43:             go to COPYWORD(O, O', δ + 3, 0)
44:         if pointerMask(O, δ + 1) = 01 then        ▷ ... of pointers
45:             P ← O[δ]
46:             if copyMap[P] = NULL then
47:                 go to ADVANCE(O, O', P, δ, i)
48:             else
49:                 O'[δ][i] ← copyMap[P]
50:                 go to COPYWORD(O, O', δ, i + 1)
51:     if pointerMask(O, δ) = 10 then                ▷ Handle transient
52:         O'[δ] ← 0
53:         go to COPYWORD(O, O', δ + 1, 0)
```

Table 2: Cache and memory parameters.

Parameter	Value	Parameter	Value
L1-I cache sets	2	LEON 3 freq.	50 MHz
L1-I cache set size	16 kByte	L1 & L2 cache freq.	50 MHz
L1-I cache line size	32 Byte	TLM freq.	50 MHz
L1-D cache sets	2	MEM ctrl freq.	100 MHz
L1-D cache set size	16 kByte	NCA freq.	50 MHz
L1-D cache line size	16 Byte	NMA freq.	100 MHz
L2 cache sets	4	L1 cache policy	write-through
L2 cache set size	128 kByte	L2 cache policy	write-back
L2 cache line size	32 Byte	L1 hit time	1 cycle
		L2 hit time	20 cycles
Tile-local memory	8 MByte	L2 miss time	90 cycles
PGAS MEM	$2 \cdot 1$ GByte	TLM acc. time	20 cycles

Table 3: Resource utilization on a Virtex-7 2000T FPGA. One slice contains 4 LUTs, 8 Registers and 2 Muxes.

HW Module	Slices	LUT	Register	Mux	BRAM	DSP
NCA range-operations	165	425	357	0	0	0
NCA graph writeback	662	2064	766	12	0	0
NMA graph copy	1862	6078	1163	117	0	0
add. HMAP module	379	1273	554	10	0	0
NMA FIFO & trigger	547	1354	1264	0	5	0
LEON 3 core	2499	8160	2587	33	18	4
L2 cache	3440	6092	8898	100	139	0
MEM controller	4825	13275	11455	398	0	0

L1 caches inside a tile are kept coherent with classical bus snooping coherence. Since one core per tile is dedicated to system tasks like interrupt handling, a total of 56 cores are available for application use. Each compute tile is further equipped with a L2 cache, that caches accesses to the global memory. Table 2 gives an overview of the core, cache, accelerator and memory configuration parameters. The LEON 3 cores further run with enabled branch prediction and floating-point unit.

The tiles are connected to the NoC routers by a network adapter (NA), that is amongst others connected to the L2 cache back-end to carry out its remote load-store operations. Besides that, the NA can also bypass the L2 cache to perform DMA transfers, forward remote task invocations, perform remote atomic operations [32], as well as trigger commands to the NMA.

The implemented prototype system is synthesized on a proFPGA system consisting of four Xilinx Virtex-7 2000T FPGAs [15]. The whole prototype is operated at a clock frequency of 50 MHz due to bottlenecks in components other than NCA and NMA. The DDR-3 memory controller runs at a minimum frequency of 100 MHz.

We run a distributed operating system [28] that is able to exploit the described hardware features.
For benchmarking, we distinguish three variants of the platform:

(1) 4×4 design (twin) using 14 compute tiles and both memory tiles.
(2) 4×4 design (single) using only the memory tile at grid position (1,1). The memory tile at (3,3) is unused.
(3) 2×2 design using only the compute tiles at grid positions (0,0), (0,1), (1,0), and the memory tile at (1,1).

5.2 Hardware Evaluation

When synthesized onto the Virtex-7 2000T FPGAs, the individual modules have the resource utilization shown in Table 3. The NMA graph copy unit including the hardware hashmap module is smaller than a single LEON 3 core and much smaller than the memory controller. The FIFO and the trigger and completion logic, connected to the network adapter, adds 30% resource overhead. The FIFO is dimensioned to hold 16 incoming graph copy requests and further performs the clock domain crossing between the network adapter and the NMA. The NMA, with and without hashmap module, could run at 164 MHz. Each NCA on the other hand is able to run at 261 MHz. As it is less complex than the graph copy unit, it also requires significantly less resources than the NMA. When compared to the L2 cache, the resources utilization is roughly one quarter, excluding the cache memory itself. Both NMA and NCA thus have a reasonable resource utilization and frequency.

5.3 Microbenchmarks

The goal of these measurements is to evaluate the performance of the graph copy algorithm in terms of its setup time and scaling behavior. To this end, we use four regular families of graphs, and measure the time required for copying them as the graphs become larger. Unless otherwise specified, we use the linear search copy map in all benchmarks. Furthermore, we compare NEMESYS to the software algorithm from Pegasus [25].

These benchmarks run as a bare-metal application on the memory tile. Thus, we measure only the effect of the NMA, without influences from NoC timing or caching.

5.3.1 Single Large Object. First, we consider graphs consisting of only one single object of increasing size without any pointers or arrays. In this case, the graph copy decays to a normal DMA operation. However, the graph copy unit still has to check the pointer mask for every word, which will incur some overhead.

We can see in Figure 8a that the time taken is very regular, with a fixed setup time of 25 μs, 0.04 μs (4 cycles) to 0.1 μs (10 cycles) for every word copied, and 0.2 μs to 0.3 μs (20 to 30 cycles) extra when the graph copy unit needs to fetch a new pointer mask.

Out of the 25 μs setup time, 2.8 μs are spent by the hardware unit to copy the object metadata, and 22 μs are spent by the operating system. Indeed, we find a constant difference of about 22 μs between the time taken by the hardware unit and the total time, irrespective of the size of the object.

The software implementation has less setup time (12 μs), but scales worse at 1.4 μs per word copied. Thus, it becomes slower than NEMESYS for objects larger than 8 words.

5.3.2 Primitive Array. In the second microbenchmark, the object graphs consist of one small object containing only an array descriptor, which points to an array of primitive data with increasing size. Here, we expect the graph copy unit to perform as well as a DMA unit, because it only needs to read once from the pointer mask before copying the array.

The results in Figure 8b again show a constant setup time of 25 μs to start up and copy the first object. Starting at about 2048 array elements, the graph copy unit reaches its full performance of 0.02 μs (2 cycles) for every word copied. This is as fast as a DMA unit could run on our platform.

The operating system time again stays at a constant 22 μs, which becomes negligible with growing array size. For clarity, we do not

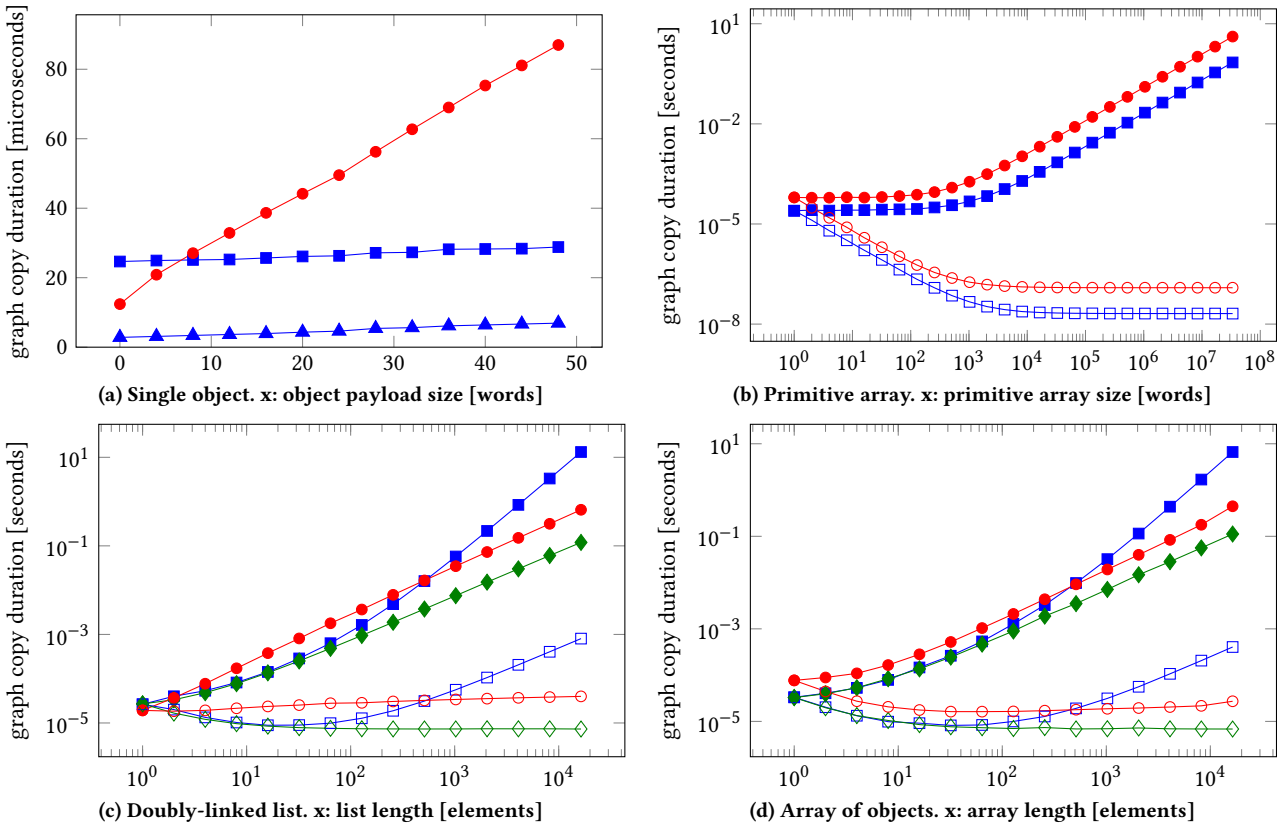


Figure 8: Microbenchmark results. For each benchmark we compare the time taken by Pegasus (software) (—●—), and by NEMESYS (hardware) with linear search (—■—) and with hashing (—◆—). In Figure 8a, we also plot the time that the graph copy unit is active (—▲—) to show the operating system overhead. In Figures 8b to 8d, we plot the time taken per array element or object with hollow markers (—○—, —□—, and —◇— respectively).

plot the total time and the time taken by the hardware separately for the following benchmarks.

On the other hand, the software implementation already starts with a higher setup time of 46 μs, and scales at 0.12 μs per word.

5.3.3 Doubly-linked list. Third, we measure the performance of the graph copy unit on pointered structures. The first structure that is evaluated is a doubly-linked list. For this benchmark, we also compare the linear search and hashing approaches for the copy map.

Figure 8c shows the results. Comparing the runtime of the linear search and hashing copy maps for small objects, we see that the overhead of initializing the hashtable is in fact negligible. Only for lists of length 1 is there a 0.4 μs difference in the time that the graph copy unit is active, but that difference is masked by operating system jitter.

The hashmap gains a clear advantage over linear search beginning at lists of length 64. From this point onwards, it takes a constant 7.4 μs on average to copy each list element. On the other hand, the time taken by the linear search is decidedly super-linear, showing roughly the quadratic growth one would expect.

The software implementation, which also uses a hashmap, again has a higher setup time than either hardware implementation. It scales worse than the hashmap implementation of NEMESYS, but beats the linear search starting at lists of length 1024.

From these measurements, we can conclude that hashing is the preferred implementation of the copy map. However, if hardware resources are at a premium, linear search gives competitive results up to graphs of 64 objects.

5.3.4 Object Array. As the second benchmark with pointered structures, we investigate an array of objects. The object graphs consist of one small object containing only an array descriptor that refers to an array of pointers with increasing size. All these pointers refer to different objects with a single word of payload.

This benchmark differs from the doubly-linked list benchmark, because there is only one pointer to each object. This means that all copy map searches will be unsuccessful, i. e. return NULL. The linear search algorithm therefore always has to traverse the whole list to verify that the pointer queried is not in it.

We find similar results in this benchmark as in the doubly-linked list benchmark: Hashing outperforms linear search starting at 64 elements, and the software implementation outperforms the NEMESYS approach using linear search for graphs larger than 1024 elements.

We can therefore conclude that our findings apply to a range of differently structured object graphs.

Table 4: An overview of the IMSuite benchmarks with the input sets we use for each of them. For more information about the benchmarks, see the IMSuite documentation [16].

Benchmark	Abbrev.	Description	Input	Input Description
bfsBellmanFord	BF	Breadth-first search in a graph using the Bellman-Ford algorithm	64-spmax	Sparse graph with 64 nodes and $n \log n$ edges
bfsDijkstra	DST	Breadth-first search in a graph using Dijkstra's algorithm	64-rn	Dense graph with 64 nodes and random adjacency
byzantine	BY	A solution of the Byzantine generals problem	16-spar-max	Sparse graph with 16 nodes and $n \log n$ edges
dijkstraRouting	DR	Single-source routing through a graph with Dijkstra's algorithm	32-spar-weq-max	Sparse graph with 32 nodes and $n \log n$ edges of equal weight
dominatingSet	DS	Computation of a dominating set	32-spar-max	Sparse graph with 32 nodes and $n \log n$ edges
kcommitte	KC	Partitioning a graph into k-committees	64-rn	Dense graph with 64 nodes and random adjacency
leader_elect_lcr	LCR	Leader election in a unidirectional ring network	64	Ring of 64 nodes
leader_elect_hs	HS	Leader election in a bidirectional ring network	64	Ring of 64 nodes
leader_elect_dp	DP	Leader election in a general network	32-spar-max	Sparse graph with 32 nodes and $n \log n$ edges
mis	MIS	Finding a maximal independent set in a graph	64-spmax	Sparse graph with 64 nodes and $n \log n$ edges
mst	MST	Computation of a minimal spanning tree	32-spmax	Sparse graph with 32 nodes and $n \log n$ edges
vertexColoring	VC	3-coloring of a tree	64-rn	Tree with 64 nodes

Table 5: Object graph statistics for each benchmark. The first three columns show overall statistics: The number of graph copy operations performed by each benchmark, and the average number of objects and bytes copied in one operation. The rest shows typical combinations of object count and graph size for each benchmark. The benchmarks DR, DS, and MST have two common types of graph, the rest have only one. For each combination, we give the number of objects in the graph, its total size, and the number of times a graph of this size is copied during the benchmark run.

				small graphs (< 1000)			medium graphs (< 10000)			large graphs		
Benchmark	# copies	avg. objects	avg. size	objects	size	count	objects	size	count	objects	size	count
BF	2150	10.5	16864	—	—	—	—	—	—	10	16844 – 16848	1964
DST	8492	11.3	16548	—	—	—	—	—	—	11	16912 – 16928	7720
BY	7362	16.5	1875	—	—	—	15	1820 – 1832	6144	—	—	—
DR	13550	6.7	2382	2	124 – 188	4850	11	4580	6134	—	—	—
DS	38778	8.7	2571	1	24	17856	14	4728	15764	—	—	—
KC	58224	10.4	709	10	688 – 700	63368	—	—	—	—	—	—
LCR	17370	10.0	670	10	672 – 676	17006	—	—	—	—	—	—
HS	46336	12.0	764	12	768 – 788	45992	—	—	—	—	—	—
DP	8830	15.0	4582	—	—	—	14 – 15	4884 – 4908	6808	—	—	—
MIS	6168	13.4	16516	—	—	—	—	—	—	13	16984 – 16988	5538
MST	18486	13.3	3618	1	24	4404	15 – 16	4800 – 4824	9856	—	—	—
VC	2388	16.5	16725	—	—	—	—	—	—	15 – 16	17624 – 17648	1666

5.4 Macrobenchmarks

To investigate the influence of NEMESYS on whole applications, we use the IMSuite benchmarks [16]. We describe their setup and characterize their communication behavior by analyzing the object graphs that are copied in Section 5.4.1. Using these benchmarks and the evaluation setup described in Section 5.4.2, we investigate a number of performance metrics in Sections 5.4.3 and 5.4.4. We compare the overall run-time of NEMESYS against both message-passing, the most-common related work, and Pegasus, the closest related work. An in-depth analysis is then performed between NEMESYS and Pegasus, including communication time and performance counter metrics. We then continue with an analysis of the design scalability (Section 5.4.5) and cache friendliness (Section 5.4.6), as well as the effect of only using the NMA without the NCA (Section 5.4.7).

5.4.1 Benchmark Description and Analysis. The IMSuite is a collection of classical distributed algorithm kernels written to exploit the features of PGAS and X10. We use these benchmarks in their iterative, concurrent, distributed variant without clocks (IMSuite_Iterative/X10-FA). See Table 4 for an overview of the benchmarks and the input sets we use. All IMSuite benchmarks are built in a way that they distribute data, compute on it, and finally gather and verify the results. Like the IMSuite authors, we only measure the computation phase, the "region of interest" (RoI).

For each benchmark, we make a separate run and log all object graphs that are copied (both closures and result values) and measure their number of objects and their total size. Table 5 shows the number of copy operations for each benchmark and the average number of objects and graph size. Analyzing the data further, we find that the majority of graphs to be copied have a similar structure. With small variations, each benchmark has one or two typical combinations of object count and graph size. These typical combinations are also presented in Table 5.

5.4.2 Evaluation Setup. Based on the analysis of Table 5, our graphs are so small that the hashing copy map will not perform better than linear search. We therefore use the linear search copy map for all benchmark runs.

We further carry out all measurements with the parameters specified in Table 2. Only when analyzing the cache friendliness in Section 5.4.6, we vary the L2 cache set size between 8 and 128 kBytes.

We adhere to the following naming convention: "{MP, NEMESYS, Pegasus}-{single, twin}" refers to message passing (MP), NEMESYS, and Pegasus, using only one (single) or both (twin) memory tiles present in the 4×4 design.

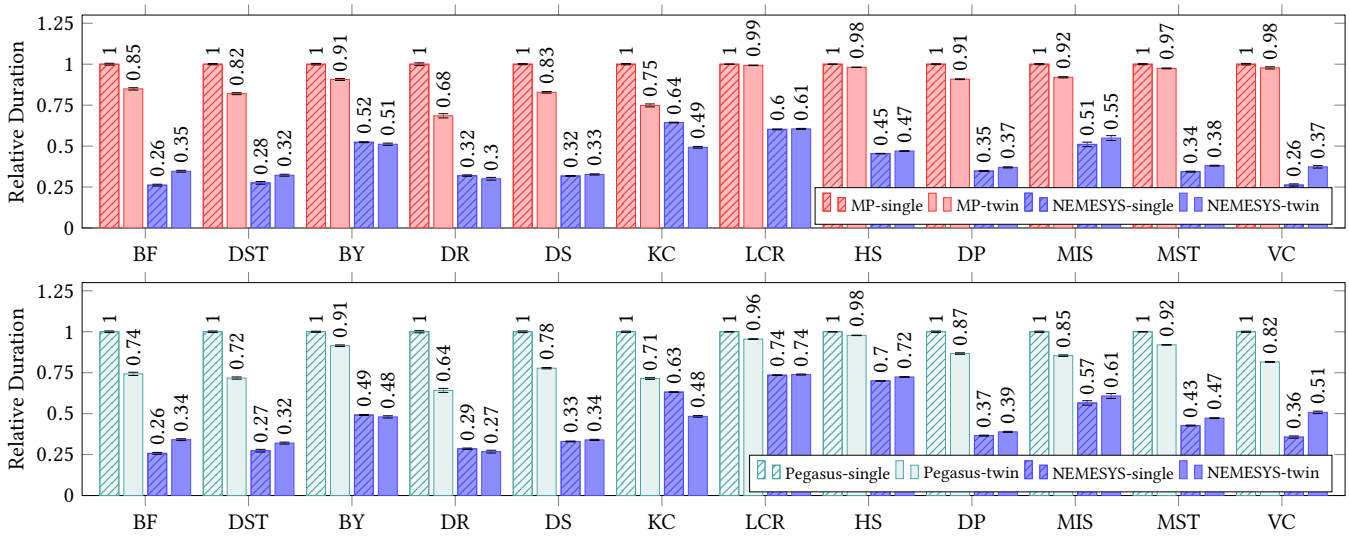


Figure 9: Runtime measurements of the IMSuite benchmarks in the 4x4 configuration. Top: NEMESYS vs. Message-passing (MP) normalized to MP-single. Bottom: NEMESYS vs. Pegasus normalized to Pegasus-single.

In order to gain more performance metrics than the overall execution time, we add several performance counters both to the hardware and the software. We reset them at the beginning of the region of interest and read their values at its end.

In hardware, we first of all added performance counters to the tiles that show the bus load and memory utilization. Second, each network adapter provides metrics for its usage, i. e. how much time it spends performing remote load-store operations. The NA further gives average round-trip times for remote load-store operations. Third, the NMA graph copy unit supplies metrics on its overall runtime, as well as its memory accesses.

In software, we integrated two additional timers T_{at} and T_{com} that measure the overall time spent inside of `at` statements and the communication time inside the `at` statement, respectively. An `at` statement is composed of the communication time and the time for the actual execution of the remote function T_{func} so that the equation $T_{at} = T_{com} + T_{func}$ holds true.

5.4.3 Overall Run- and Communication Time. First of all, we compare the overall runtime between the message passing (MP), Pegasus, and NEMESYS approaches. Figure 9 Top and Figure 9 Bottom show the results for the overall execution times normalized to MP-single and Pegasus-single, respectively. Figure 10 Top compares the communication times T_{com} for the NEMESYS and Pegasus variants normalized to Pegasus-single. Figure 10 Bottom shows the fraction T_{com} / T_{at} of the communication time inside of the `at` statement for each individual benchmark and variant.

Analyzing the runtimes, we observe:

(1) NEMESYS outperforms message-passing and Pegasus in every case and mostly by far. The HS and LCR benchmark show the least speedup due to their small graph sizes and the thereby higher relative base overhead introduced by the operating system (task creation, scheduling).
(2) Pegasus-twin performs better than Pegasus-single since the two physically distributed memory tiles mitigate the memory access hot-spot.
(3) NEMESYS-twin has a small performance degradation compared to the single variant. This is due to the overhead of the extended RPC mechanism using the additional inter-memory DMA.
(4) Although the total communication time reduces substantially with NEMESYS (Figure 10 Top), the fraction of time spent in communication stays roughly equal (Figure 10 Bottom). This means that the computation itself also runs more quickly with NEMESYS because the CPUs can focus on executing application rather than runtime system code.

5.4.4 Effect on System Load. Since NEMESYS is a near-memory approach, it relieves the NoC and the tile buses of memory traffic during graph copies. The integrated hardware performance counter numbers, provided in Table 6, yielded seven further important observations that underline the results presented in Section 5.4.3. The following observations become especially apparent by looking at the mean values over all benchmarks.

(5) NEMESYS has fewer remote memory accesses and thus its NA usage is reduced compared to Pegasus.
(6) In the Pegasus-twin variant, the resolved hot-spots lead to reduced average round-trip times for remote load-store operations. This also lowers the percentage in NA usage, since the average latency is lower.
(7) NEMESYS produces fewer total memory accesses (MEM usage in MBytes) due to reduced cache pollution: The object graph does not evict other data from the caches since it is not copied by a CPU.
(8) NEMESYS-twin produces additional memory accesses due to the intermediate copy followed by the inter-memory DMA.

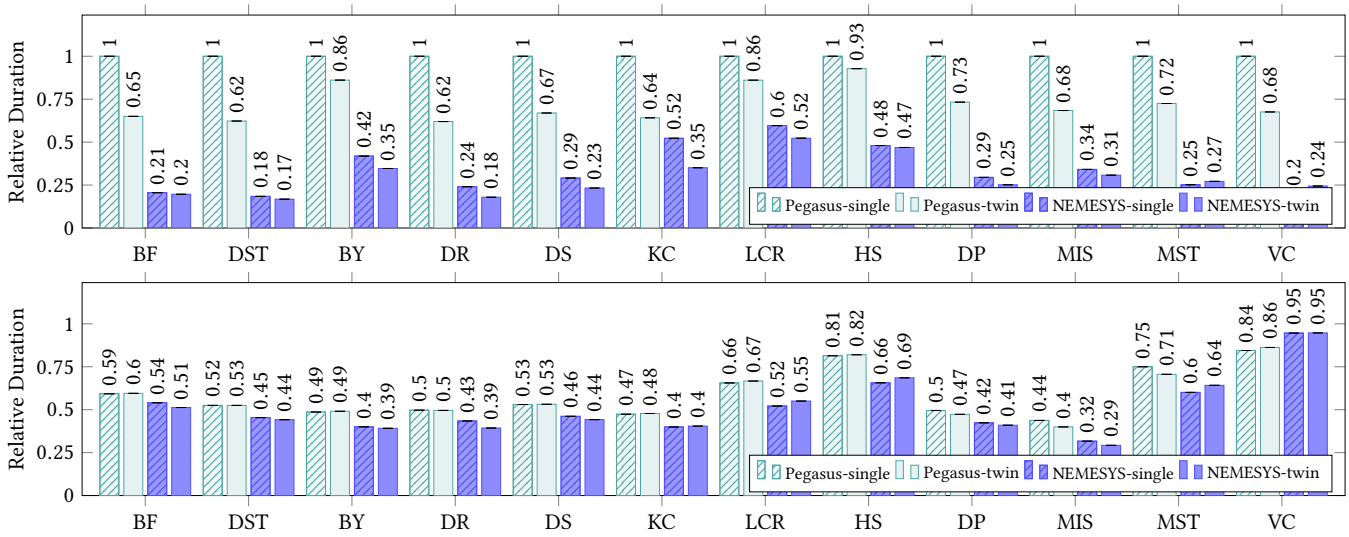


Figure 10: Communication time measurements of the IMSuite benchmarks in the 4x4 configuration. Top: Total communication time T_{com} normalized to Pegasus-single. Bottom: Fraction T_{com} / T_{at} of the communication time inside of the `at` statement for each individual benchmark and variant.

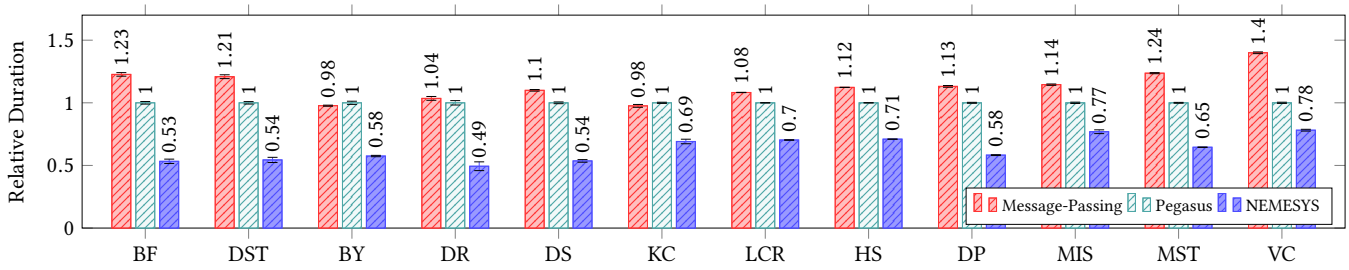


Figure 11: Overall runtime of the IMSuite benchmarks with message passing, Pegasus, and NEMESYS on the 2x2 configuration. Times are normalized to Pegasus.

(9) NEMESYS further has more memory accesses per time. In both twin variants, this number is lower than their single pendant since it distributes onto two different memory tiles.
(10) The analysis further yields that no variant is compute bound since the memory access times dominate. Further trying various core counts per tile did not significantly influence the results (not shown).
(11) Especially for the high performing NEMESYS benchmarks, the NMA utilization and memory accesses are much higher than for LCR or HS.

Table 6 further shows that one single NMA can take on the communication needs of at least 14 compute tiles. We can see that even with one memory tile (i. e. one NMA serving 14 compute tiles), the mean NMA utilization is roughly 50 % with a peak usage of 90.5 % for the DR benchmark. Using two memory tiles each NMA utilization halves to 25.1 % with a peak usage of only 52 %. As real world applications consist of a mixture of the benchmark kernels the mean utilization is a good indicator for the to be expected NMA utilization. We can therefore conclude that it is reasonable to provide one NMA for at least 14 compute tiles since it does not yet run at its capacity limit. In particular, the NEMESYS-twin variant shows that 7 compute tiles cannot fully load one NMA.

5.4.5 Scalability Analysis. Figure 11 shows results for a 2×2 design running on 3 compute tiles. Comparing these results with those in Figure 9 Top and Figure 9 Bottom, we can investigate how NEMESYS compares to Pegasus in its scaling behavior.

For most benchmarks, we see that NEMESYS yields a larger speedup on a 4×4 design than on a 2×2 design, both with one and two memory tiles. This is because the amount of communication grows super-linearly with the number of tiles (see Gupta and Nandivada [16]), and the NMA does not run at full capacity on the 2×2 design. Then, on the 4×4 design, the NMA gets utilized better, whereas the software implementation has to deal with more communication work per CPU available.

On the other hand, some benchmarks do not show such a marked increase in speedup, most notably LCR and HS. This is because these benchmarks communicate with many small graphs between tiles (see Table 5). Therefore, the communication overhead in this case does not lie in the copy operation itself, but in the operating system (task creation, scheduling).

Table 6: Performance counter metrics for NEMESYS and Pegasus, each for both the single and twin variant. The shown numbers are averaged over all compute or memory tiles, respectively. CPU bus load: % of time the CPU accesses the bus. NA usage: % of time performing remote load-store operations. MEM usage: % of total runtime and total accessed Megabytes. Avg. cycles: average latency for a remote load-store operation.

(a) NEMESYS-single.

Benchmark	CPU bus load	NA usage	MEM usage in %	MEM usage in MBytes	MEM via NA	MEM via NMA	NMA util.	avg. cycles load	avg. cycles store
BF	5.3 %	11.5 %	67.2 %	53	22.7 %	44.5 %	68.3 %	318	234
DST	5.2 %	25.6 %	67.9 %	232	27.4 %	40.5 %	66.1 %	479	392
BY	6.7 %	13.9 %	45.7 %	52	26.3 %	19.4 %	55.8 %	289	212
DR	11.6 %	30.3 %	76.5 %	67	38.6 %	37.9 %	90.5 %	460	361
DS	7.6 %	16.4 %	55.3 %	223	26.0 %	29.3 %	64.7 %	319	241
KC	12.3 %	46.9 %	56.3 %	281	40.5 %	15.8 %	64.2 %	506	444
LCR	9.9 %	16.3 %	35.7 %	106	28.0 %	7.7 %	28.7 %	291	233
HS	7.7 %	10.1 %	32.6 %	266	23.5 %	9.1 %	33.0 %	220	150
DP	4.9 %	12.6 %	41.3 %	103	22.8 %	18.5 %	41.3 %	295	219
MIS	4.8 %	10.8 %	39.8 %	299	26.6 %	13.2 %	21.7 %	225	151
MST	3.7 %	5.8 %	26.4 %	170	13.7 %	12.7 %	25.2 %	236	166
VC	3.9 %	4.1 %	45.5 %	60	14.3 %	31.2 %	46.0 %	186	94
Mean	7.0 %	20.9 %	49.2 %	159	25.9 %	32.3 %	50.5 %	319	241

(b) Pegasus-single.

Benchmark	CPU bus load	NA usage	MEM usage in %	MEM usage in MBytes	avg. cycles load	avg. cycles store
BF	3.5 %	33.5 %	27.5 %	133	346	296
DST	3.4 %	38.4 %	44.6 %	554	381	331
BY	6.7 %	14.2 %	31.4 %	77	244	204
DR	7.0 %	48.5 %	50.5 %	167	445	389
DS	5.0 %	30.8 %	40.7 %	543	338	288
KC	12.5 %	45.7 %	45.5 %	402	445	403
LCR	10.4 %	13.7 %	30.2 %	128	226	183
HS	8.3 %	10.5 %	27.5 %	346	191	135
DP	3.7 %	19.6 %	32.0 %	230	287	234
MIS	4.1 %	23.4 %	39.3 %	523	279	220
MST	3.1 %	13.6 %	23.0 %	359	273	218
VC	3.5 %	22.5 %	35.7 %	127	283	232
Mean	5.9 %	26.2 %	37.2 %	299	311	261

(c) NEMESYS-twin.

Benchmark	CPU bus load	NA usage	MEM usage in %	MEM usage in MBytes	MEM via NA	MEM via NMA	NMA util.	avg. cycles load	avg. cycles store
BF	4.3 %	15.6 %	41.1 %	86	24.5 %	16.6 %	28.4 %	571	436
DST	5.0 %	25.3 %	44.5 %	362	27.4 %	17.1 %	30.6 %	542	421
BY	7.3 %	11.3 %	28.1 %	65	18.2 %	9.9 %	27.5 %	228	146
DR	14.2 %	35.7 %	56.6 %	97	36.6 %	20.0 %	52.0 %	507	367
DS	8.2 %	19.0 %	36.6 %	317	22.5 %	14.1 %	33.8 %	382	256
KC	17.4 %	31.6 %	40.7 %	322	30.6 %	10.1 %	40.3 %	276	205
LCR	10.5 %	10.6 %	19.1 %	116	15.3 %	3.8 %	13.5 %	191	128
HS	7.7 %	7.8 %	17.1 %	298	12.8 %	4.3 %	14.2 %	172	104
DP	4.9 %	11.6 %	25.6 %	141	17.0 %	8.6 %	20.2 %	287	183
MIS	4.6 %	10.4 %	24.0 %	393	17.9 %	6.1 %	11.0 %	228	144
MST	3.6 %	5.1 %	11.0 %	217	7.0 %	4.0 %	12.0 %	221	130
VC	2.9 %	4.6 %	25.1 %	94	14.3 %	10.8 %	17.9 %	256	123
Mean	7.6 %	15.7 %	30.8 %	209	20.4 %	10.5 %	25.1 %	322	220

(d) Pegasus-twin.

Benchmark	CPU bus load	NA usage	MEM usage in %	MEM usage in MBytes	avg. cycles load	avg. cycles store
BF	4.7 %	25.7 %	30.9 %	133	208	150
DST	4.7 %	28.6 %	31.1 %	555	215	158
BY	7.3 %	9.8 %	17.1 %	77	160	105
DR	10.8 %	36.3 %	39.3 %	167	225	166
DS	6.9 %	21.4 %	26.2 %	543	194	135
KC	17.8 %	28.6 %	31.8 %	402	212	159
LCR	11.1 %	9.7 %	15.8 %	128	157	102
HS	8.8 %	8.4 %	14.1 %	346	151	93
DP	4.3 %	13.7 %	18.5 %	230	182	124
MIS	4.8 %	17.7 %	23.0 %	523	188	124
MST	3.4 %	8.9 %	8.8 %	341	172	112
VC	4.2 %	16.7 %	21.9 %	127	185	124
Mean	7.4 %	18.8 %	23.2 %	298	188	129

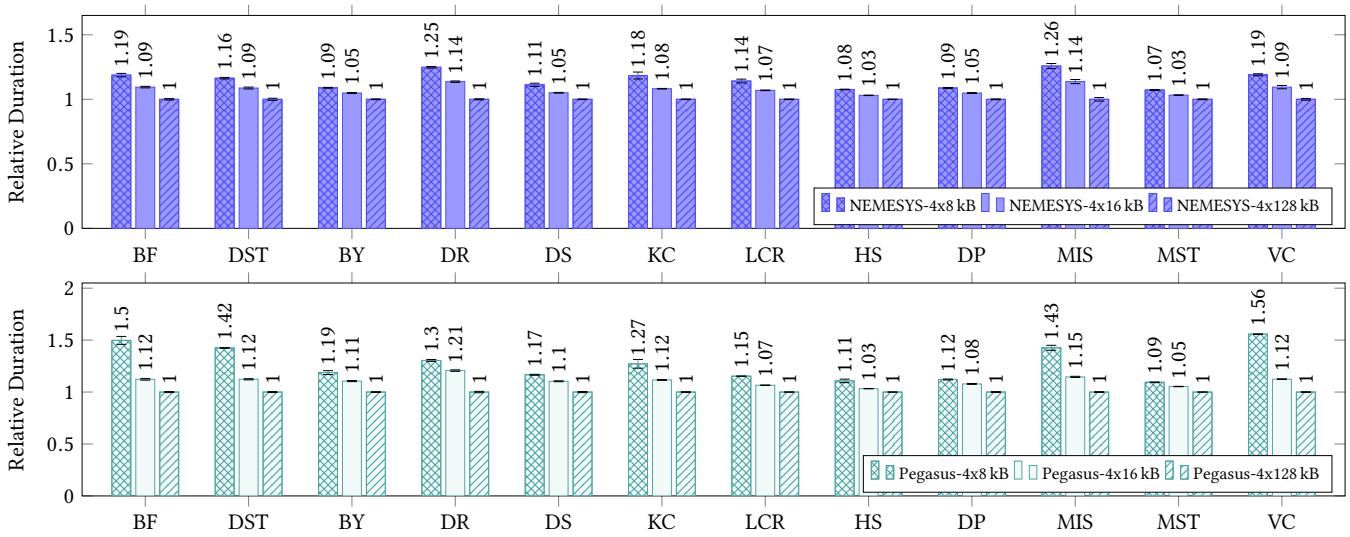


Figure 12: Influence of cache size on the overall runtime of the IMSuite benchmarks. Top: NEMESYS-single. Bottom: Pegasus-single. Both variants use the 4x4 configuration with single DDR and 4-way associative cache. Times are normalized to a L2 cache size of 4x128 kByte, i. e. 4 sets à 128 kByte.

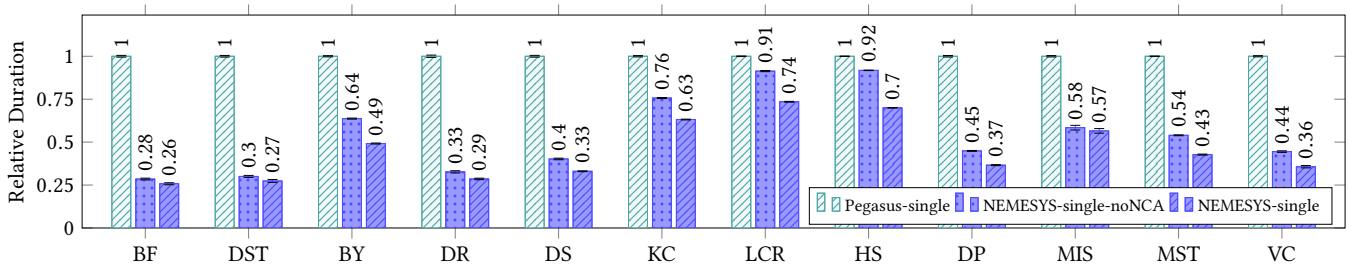


Figure 13: Effect of the near-cache accelerators (NCA) on the overall runtime. This plot compares Pegasus (no accelerators), noNCA (only the NMA) and NEMESYS (both NMA and NCA). Each configuration uses the 4x4 configuration with single DDR.

5.4.6 Analysis of Cache Friendliness. In order to analyze whether the size of the data sets has an unfair influence on the evaluation, we ran the benchmarks with different L2 cache sizes. As shown in Table 5, the biggest graph size is roughly 16 kBytes. We thus use the 4-way associative L2 cache with small (8 kByte), medium (16 kByte) or large (128 kByte) set sizes, respectively.

Section 5.4.4 Top and Section 5.4.4 Bottom show cache analysis results for NEMESYS-single and Pegasus-single. We omit the figures for the twin variants as they lead to almost identical ratios between Pegasus and NEMESYS. The analysis shows that Pegasus' runtime generally slows down more with smaller caches. This is because the graph copy operations run on the CPUs, so larger caches help avoiding remote load-store operations.

5.4.7 Effect of Near-Cache Accelerators. In some designs, including the NCA on every tile may be too costly in terms of hardware resources. Therefore, we also investigate how much benefit we get from just having the NMA on every memory tile, and triggering the (much simpler) range-operations unit from software.

Figure 13 shows the results. We can see that we already get most of the speedup just from having the NMA, while the NCA still adds a measurable speedup on top of that. Again, LCR and HS stand out. This is because their small object graphs take a relatively longer time to write back to DDR. On the other hand, the large arrays used by the other benchmarks just require one range-operation to write back the whole array.

6 FUTURE WORK

6.1 Garbage Collector Integration

Being part of an object-oriented system, NEMESYS should play well with garbage collection. However, the garbage collector does not see the destination buffer as separate objects. There are three possible solutions: (1) Build an *object-aware* garbage collector, i. e. one which re-uses the object metadata provided in the RTTI structures. This garbage collector would identify the contents of the destination buffer as separate objects by their metadata and handle them accordingly. (2) If the allocator/garbage collector needs separate metadata, the graph copy unit could leave gaps between the objects. Then, we could extend the garbage collector with an operation that takes an existing buffer in memory and adds it to the garbage collector heap as a separate object. Thus, we divide the destination buffer into individual objects which the garbage collector can reclaim independently. Or (3) allocate separate buffers for the objects beforehand and pass a list of pointers to these buffers to the NMA.

For now, our concern is that the benchmarks against the software implementation are fair. Therefore, we emulate higher allocator load by allocating as many dummy objects after each hardware graph copy as there were objects in the graph.

6.2 Hardware Garbage Collection

Furthermore, NEMESYS itself can be used as a garbage collector with some extensions. It already implements the core functionality of a semi-space garbage collector: it can traverse an object graph starting at its root and copy all reachable objects to a newly allocated buffer. However, a garbage collector usually keeps a set of several root objects (global and local variables, stack slots, etc.). The object graphs rooted at these objects must be viewed as one graph with multiple roots for the purposes of garbage collection. This is easily achieved with NEMESYS by not resetting the copy map and the destination buffer after each copy. The garbage collection driver can then pass each garbage collection root to NEMESYS in turn.

7 CONCLUSION

We presented NEMESYS, a technique to speed up copying object graphs using a near-memory accelerator. We applied our technique to the runtime system of a PGAS prototype platform, and evaluated its performance on distributed algorithm kernels. NEMESYS achieved speedups of up to 3.8× in benchmarks where large object graphs had to be copied, and 1.35× when the object graphs were small. Furthermore, we found that with NEMESYS the CPUs spend more time executing user rather than runtime system code.

We envision that hardware units like ours will be tightly integrated into the memories they operate on in the future. They will become the generalized equivalent of DMA units as software engineering shifts towards high-level languages. NEMESYS thus provides a more efficient way to move data where it is needed in a distributed system. This allows applications to benefit from data locality more often and overcome the looming locality wall.

ACKNOWLEDGEMENTS

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 146371743 – TRR 89: Invasive Computing. We thank Richard Petri for the implementation and evaluation of performance counter metrics, as well as Nora Pohle, Anh Vu Doan, Florian Schmaus, and the anonymous reviewers for their valuable comments.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. 105–117. https://doi.org/10.1145/2749469.2750386

[2] The Go authors. 2019. *mbitmap.go*. https://go.googlesource.com/go/+/d41a0a0690ccb699401c7c8904999895b2c92511/src/runtime/mbitmap.go

[3] David F. Bacon, Perry Cheng, and Sunil Shukla. 2013. And then There Were None: A Stall-free Real-time Garbage Collector for Reconfigurable Hardware. *Commun. ACM* 56, 12 (Dec. 2013), 101–109. https://doi.org/10.1145/2534706.2534726

[4] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. 1–16. http://www.usenix.org/events/osdi10/tech/full_papers/Boyd-Wickizer.pdf

[5] J.Lawrence Carter and Mark N. Wegman. 1979. Universal classes of hash functions. *J. Comput. System Sci.* 18, 2 (1979), 143 – 154. https://doi.org/10.1016/0022-0000(79)90044-8

[6] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. 2013. Runnemede: An architecture for Ubiquitous High-Performance Computing. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 198–209. https://doi.org/10.1109/HPCA.2013.6522319

[7] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*. 155–166. https://doi.org/10.1109/PACT.2011.21

[8] Steffen Christgau and Bettina Schnor. 2016. Software-managed Cache Coherence for fast One-Sided Communication. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. 69–77. https://doi.org/10.1145/2883404.2883409

[9] Cobham Gaisler. 2010. *LEON 3*.

[10] Hybrid Memory Cube Consortium. 2019. *Hybird Memory Cube Specification 2.1*. http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf

[11] UPC Consortium, Dan Bonachea, and Gary Funck. 2013. *UPC Language and Library Specifications, Version 1.3*. Technical Report. https://doi.org/10.2172/1134233

[12] Tilera Corp. 2009. *TILE-Gx Processor Family Product Brief (archived)*. https://web.archive.org/web/20100411035435/http://www.tilera.com/pdf/PB025_TILE-Gx_Processor_A_v3.pdf

[13] Oracle Corporation. 2019. *instanceKlass.hpp*. https://hg.openjdk.java.net/jdk10/jdk10/hotspot/file/5ab7a67bc155/src/share/vm/oops/instanceKlass.hpp

[14] Y. Durand, P. M. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, and J. Thomson. 2014. EUROSERVER: Energy Efficient Node for European Micro-Servers. In *2014 17th Euromicro Conference on Digital System Design*. 206–213. https://doi.org/10.1109/DSD.2014.15

[15] PRO DESIGN Electronic GmbH. 2019. *FPGA Module XC7V2000T*. https://www.profpga.com/products/fpga-modules-overview/virtex-7-based/profpga-xc7v2000t

[16] Suyash Gupta and V. Krishna Nandivada. 2015. IMSuite: A benchmark suite for simulating distributed algorithms. *J. Parallel Distrib. Comput.* 75 (2015), 1–19. https://doi.org/10.1016/j.jpdc.2014.10.010

[17] Paul N. Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. 2001. *Titanium Language Reference Manual*. Technical Report. Berkeley, CA, USA.

[18] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. 2016. Accelerating Linked-list Traversal Through Near-Data Processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*. 113–124. https://doi.org/10.1145/2967938.2967958

[19] Cray Inc. 2019. *Chapel Language Specification*. https://chapel-lang.org/docs/_downloads/chapelLanguageSpec.pdf

[20] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. 2011. Cohesion: An Adaptive Hybrid Memory Model for Accelerators. *IEEE Micro* 31, 1 (2011), 42–55. https://doi.org/10.1109/MM.2011.8

[21] Gushu Li, Guohao Dai, Shuangchen Li, Yu Wang, and Yuan Xie. 2018. GraphIA: an in-situ accelerator for large-scale graph processing. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2018, Old Town Alexandria, VA, USA, October 01-04, 2018*. 79–84. https://doi.org/10.1145/3240302.3240312

[22] S. Lyberis, G. Kalokerinos, M. Lygerakis, V. Papaefstathiou, D. Tsaliagkos, M. Katevenis, D. Pnevmatikatos, and D. Nikolopoulos. 2012. Formic: Cost-efficient and Scalable Prototyping of Manycore Architectures. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 61–64.

[23] Martin Maas, Krste Asanović, and John Kubiatowicz. 2018. A Hardware Accelerator for Tracing Garbage Collection. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 138–151. https://doi.org/10.1109/ISCA.2018.00022

[24] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. 2012. Why on-chip cache coherence is here to stay. *Commun. ACM* 55, 7 (2012), 78–89. https://doi.org/10.1145/2209249.2209269

[25] Manuel Mohr and Carsten Tradowsky. 2017. Pegasus: efficient data transfers for PGAS languages on non-cache-coherent many-cores. In *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 1785–1790.

[26] Mohamed Ayoub Neggaz, Hasan Erdem Yantir, Smaïl Niar, Ahmed M. Eltawil, and Fadi J. Kurdahi. 2018. Rapid in-memory matrix multiplication using associative processor. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. 985–990. https://doi.org/10.23919/DATE.2018.8342152

[27] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting managed heaps in distributed big data systems. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 56–69.

[28] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2011. OctoPOS: A Parallel Operating System for Invasive Computing. In *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA)* (2011-04-10/2011-04-13) *(Sixth International ACM/EuroSys European Conference on Computer Systems (EuroSys))*, Ross McIlroy, Joe Sventek, Tim Harris, and Timothy Roscoe (Eds.), Vol. USB Proceedings. EuroSys, 9–14.

[29] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven M. Burns, and Özcan Özturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. 166–177. https://doi.org/10.1109/ISCA.2016.24

[30] Peter Kogge. 2017. *Memory Intensive Computing, the 3rdWall, and the Need for Innovation in Architecture*. Univ. of Notre Dame. https://memsys.io/wp-content/uploads/2017/12/The_Wall.pdf

[31] Sven Rheindt, Sebastian Maier, Florian Schmaus, Thomas Wild, Wolfgang Schröder-Preikschat, and Andreas Herkersdorf. 2019. SHARQ: Software-Defined Hardware-Managed Queues for Tile-Based Manycore Architectures. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIX)*. Samos, Greece.

[32] Sven Rheindt, Andreas Schenk, Akshay Srivatsa, Thomas Wild, and Andreas Herkersdorf. 2018. CaCAO: Complex and Compositional Atomic Operations for NoC-Based Manycore Platforms. In *Architecture of Computing Systems - ARCS 2018 - 31st International Conference, Braunschweig, Germany, April 9-12, 2018, Proceedings*. 139–152. https://doi.org/10.1007/978-3-319-77610-1_11

[33] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. 2019. *X10 Language Specification*. http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf

[34] H. Schorr and W. M. Waite. 1967. An Efficient Machine-independent Procedure for Garbage Collection in Various List Structures. *Commun. ACM* 10, 8 (Aug. 1967), 501–506. https://doi.org/10.1145/363534.363554

[35] Fabian Schuiki, Michael Schaffner, Frank K. Gürkaynak, and Luca Benini. 2019. A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets. *IEEE Trans. Computers* 68, 4 (2019), 484–497. https://doi.org/10.1109/TC.2018.2876312

[36] SPARC Inc. 1992. *The SPARC Architecture Manual, Version 8* (sav080si9308 ed.).

[37] Akshay Srivatsa, Sven Rheindt, Dirk Gabriel, Thomas Wild, and Andreas Herkersdorf. 2019. CoD: Coherence-on-Demand – Runtime Adaptable Working Set Coherence for DSM-based Manycore Architectures. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIX)*. Samos, Greece.

[38] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. 2011. RCKMPI - Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). In *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*. 208–217. https://doi.org/10.1007/978-3-642-24449-0_24

[39] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2013. Navigating big data with high-throughput, energy-efficient data partitioning. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*. 249–260. https://doi.org/10.1145/2485922.2485944

[40] Salessawi Ferede Yitbarek, Tao Yang, Reetuparna Das, and Todd M. Austin. 2016. Exploring specialized near-memory processing for data intensive operations. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*. 1449–1452.