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 36 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MEMSYS ’19, September 30–October 3, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7206-0/19/09 ... $15.00
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-pointer) data.

Besides all hardware architectural developments, data structures in modern programming are getting more complex. In the past,
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 pointed 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 and func is the function to be executed there. This function may access its lexical environment 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 pointed 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

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.

2 RELATED WORK

Near-memory accelerators have seen much interest recently [1, 3, 7, 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 Hybrid-Memory-Cube (HMC) [10].
Near-memory accelerators have also been built for even more complex operations: As examples for more numerically-oriented tasks, Négaz 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 accelerators for 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 accessing 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’](image-url)
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 trigger 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 transferred to the receiver D (into M').
(1c) S triggers the NCA graph writeback and dispatch unit, accompanied by the descriptor of task T1.
(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 T1 on D.
(3a) Based on the metadata information M', T1 (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 T2 on D upon its completion.
(4) T2 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).

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 NCAS graph writeback and dispatch unit, now accompanied by the descriptor of task T3.
(1d) The NCAS 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 T3 locally on S itself.
(1g') T3 (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 T1 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.](image-url)
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.

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.

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.

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

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 \( O(n^2) \) for object graphs of size \( n \).

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 \( h_M(x) \) parameterized by a \( k \times n \) bit matrix \( M \). The hash \( h_M(x) \) is then given by \( h_M(x) = \oplus_{i=0}^{n-1} M_ix_i \), 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 \( 2^{[\log_2 o]+1} \) 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.

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.
4.6 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 δ in the offset field. Now, we must recursively copy the object at δ (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 occurring 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.

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 \( \triangleright \) Copy data
24. O′[δ] ← O[δ]
25. go to COPYWORD(O, O′, δ + 1, 0)
26. if pointerMask(O, δ) = 10 then \( \triangleright \) Copy pointer
27. P ← O[δ]
28. if copyMap[P] = NULL then
29. go to ADVANCE(O, O′, P, δ, i)
30. else
31. O′[δ] ← copyMap[P]
32. go to COPYWORD(O, O′, δ + 1, 0)
33. if pointerMask(O, δ) = 11 then \( \triangleright \) 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 \( \triangleright \) of data
42. memcpy(O′[δ], O[δ], bytes)
43. go to COPYWORD(O, O′, δ + 3, 0)
44. if pointerMask(O, δ + 1) = 10 then \( \triangleright \) 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 \( \triangleright \) Handle transient
52. O′[δ] ← 0
53. go to COPYWORD(O, O′, δ + 1, 0)
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

### Table 2: Cache and memory parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache sets</td>
<td>2</td>
<td>LEON 3 freq</td>
<td>50 MHz</td>
</tr>
<tr>
<td>L1-I cache set size</td>
<td>16 kByte</td>
<td>L1 &amp; L2 cache freq</td>
<td>50 MHz</td>
</tr>
<tr>
<td>L1-I cache line size</td>
<td>32 Byte</td>
<td>TLM freq.</td>
<td>50 MHz</td>
</tr>
<tr>
<td>L1-D cache sets</td>
<td>2</td>
<td>MEM ctrl freq</td>
<td>100 MHz</td>
</tr>
<tr>
<td>L1-D cache set size</td>
<td>16 kByte</td>
<td>NCA freq.</td>
<td>50 MHz</td>
</tr>
<tr>
<td>L1-D cache line size</td>
<td>16 Byte</td>
<td>NMA freq.</td>
<td>100 MHz</td>
</tr>
<tr>
<td>L2 cache sets</td>
<td>4</td>
<td>L1 cache policy</td>
<td>write-through</td>
</tr>
<tr>
<td>L2 cache set size</td>
<td>128 kByte</td>
<td>L2 cache policy</td>
<td>write-back</td>
</tr>
<tr>
<td>L2 cache line size</td>
<td>32 Byte</td>
<td>L1 hit time</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Tile-local memory</td>
<td>8 MByte</td>
<td>L2 hit time</td>
<td>20 cycles</td>
</tr>
<tr>
<td>MEM controller</td>
<td>2 - 1 GByte</td>
<td>TLM acc. time</td>
<td>20 cycles</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>HW Module</th>
<th>Slices</th>
<th>LUT</th>
<th>Register</th>
<th>Mutex</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCA range-operations</td>
<td>165</td>
<td>425</td>
<td>357</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NCA graph writeback</td>
<td>662</td>
<td>2064</td>
<td>766</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NMA graph copy</td>
<td>1862</td>
<td>6078</td>
<td>1163</td>
<td>117</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>add. HMAP module</td>
<td>379</td>
<td>1273</td>
<td>554</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NMA FIFO &amp; trigger</td>
<td>547</td>
<td>1354</td>
<td>1264</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>LEON 3 core</td>
<td>2499</td>
<td>8160</td>
<td>2587</td>
<td>33</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>L2 cache</td>
<td>3440</td>
<td>6902</td>
<td>8898</td>
<td>100</td>
<td>139</td>
<td>0</td>
</tr>
<tr>
<td>MEM controller</td>
<td>4825</td>
<td>13275</td>
<td>11455</td>
<td>398</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Abbrev.</th>
<th>Description</th>
<th>Input</th>
<th>Input Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfsBellmanFord</td>
<td>BF</td>
<td>Breadth-first search in a graph using the Bellman-Ford algorithm</td>
<td>64-spmax</td>
<td>Sparse graph with 64 nodes and ( n \log n ) edges</td>
</tr>
<tr>
<td>bfsDijkstra</td>
<td>DST</td>
<td>Breadth-first search in a graph using Dijkstra’s algorithm</td>
<td>64-rn</td>
<td>Dense graph with 64 nodes and random adjacency</td>
</tr>
<tr>
<td>byzantine</td>
<td>BY</td>
<td>A solution of the Byzantine generals problem</td>
<td>16-sp-max</td>
<td>Sparse graph with 16 nodes and ( n \log n ) edges</td>
</tr>
<tr>
<td>dijkstraRouting</td>
<td>DR</td>
<td>Single-source routing through a graph with Dijkstra’s algorithm</td>
<td>32-sp-ray-max</td>
<td>Sparse graph with 32 nodes and ( n \log n ) edges of equal weight</td>
</tr>
<tr>
<td>dominatingSet</td>
<td>DS</td>
<td>Computation of a dominating set</td>
<td>32-sp-max</td>
<td>Sparse graph with 32 nodes and ( n \log n ) edges</td>
</tr>
<tr>
<td>kcommittee</td>
<td>KC</td>
<td>Partitioning a graph into K-committees</td>
<td>64-rn</td>
<td>Dense graph with 64 nodes and random adjacency</td>
</tr>
<tr>
<td>leader_elect_lcr</td>
<td>LCR</td>
<td>Leader election in a unidirectional ring network</td>
<td>64</td>
<td>Ring of 64 nodes</td>
</tr>
<tr>
<td>leader_elect_hs</td>
<td>HS</td>
<td>Leader election in a bidirectional ring network</td>
<td>64</td>
<td>Ring of 64 nodes</td>
</tr>
<tr>
<td>leader_elect_dp</td>
<td>DP</td>
<td>Leader election in a general network</td>
<td>32-sp-max</td>
<td>Sparse graph with 32 nodes and ( n \log n ) edges</td>
</tr>
<tr>
<td>mis</td>
<td>MIS</td>
<td>Finding a maximal independent set in a graph</td>
<td>64-spmax</td>
<td>Sparse graph with 64 nodes and ( n \log n ) edges</td>
</tr>
<tr>
<td>mst</td>
<td>MST</td>
<td>Computation of a minimal spanning tree</td>
<td>32-sp-max</td>
<td>Sparse graph with 32 nodes and ( n \log n ) edges</td>
</tr>
<tr>
<td>vertexColoring</td>
<td>VC</td>
<td>Coloring of a tree</td>
<td>64-rn</td>
<td>Tree with 64 nodes</td>
</tr>
</tbody>
</table>

Table 5: Object graph statistics for each benchmark. The first three columns show overall statistics: The number of graph copy size, and the number of times a graph of this size is copied during the benchmark run.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># copies</th>
<th>avg. objects</th>
<th>avg. size</th>
<th>small graphs (&lt; 1000)</th>
<th>medium graphs (&lt; 10000)</th>
<th>large graphs</th>
<th>count</th>
<th>objects</th>
<th>size count</th>
<th>objects</th>
<th>size count</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>2150</td>
<td>10.5</td>
<td>16864</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>10</td>
<td>16844</td>
<td>16848</td>
<td>1964</td>
</tr>
<tr>
<td>DST</td>
<td>8492</td>
<td>11.3</td>
<td>16548</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>11</td>
<td>16912</td>
<td>16928</td>
<td>7720</td>
</tr>
<tr>
<td>BY</td>
<td>7362</td>
<td>16.5</td>
<td>1875</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>15</td>
<td>15</td>
<td>1820</td>
<td>1832</td>
<td>6144</td>
</tr>
<tr>
<td>DR</td>
<td>13550</td>
<td>6.7</td>
<td>2382</td>
<td>2 124 – 188 4850</td>
<td>11 4580 6134</td>
<td>—</td>
<td>12</td>
<td>11</td>
<td>1694</td>
<td>1698</td>
<td>7720</td>
</tr>
<tr>
<td>DS</td>
<td>38778</td>
<td>8.7</td>
<td>2571</td>
<td>1 24 17856</td>
<td>14 4728 15764</td>
<td>—</td>
<td>15</td>
<td>15</td>
<td>4884</td>
<td>4908</td>
<td>6808</td>
</tr>
<tr>
<td>KC</td>
<td>58224</td>
<td>10.4</td>
<td>709</td>
<td>10 688 – 700 63368</td>
<td>—</td>
<td>—</td>
<td>14</td>
<td>14</td>
<td>4884</td>
<td>4908</td>
<td>6808</td>
</tr>
<tr>
<td>LCR</td>
<td>17370</td>
<td>10.0</td>
<td>670</td>
<td>10 672 – 676 17006</td>
<td>—</td>
<td>—</td>
<td>12</td>
<td>12</td>
<td>768 – 788 45992</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>HS</td>
<td>46336</td>
<td>12.0</td>
<td>764</td>
<td>12 768 – 788 65992</td>
<td>—</td>
<td>—</td>
<td>27</td>
<td>17</td>
<td>4404</td>
<td>4404</td>
<td>9856</td>
</tr>
<tr>
<td>DP</td>
<td>8830</td>
<td>15.0</td>
<td>4582</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>15</td>
<td>15</td>
<td>4884</td>
<td>4908</td>
<td>6808</td>
</tr>
<tr>
<td>MS</td>
<td>6168</td>
<td>13.4</td>
<td>16516</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>15</td>
<td>15</td>
<td>4800</td>
<td>4824</td>
<td>9856</td>
</tr>
<tr>
<td>MST</td>
<td>18486</td>
<td>13.3</td>
<td>3618</td>
<td>1 24 4404</td>
<td>15 – 16 4800 – 4824 9856</td>
<td>—</td>
<td>15</td>
<td>15</td>
<td>17624</td>
<td>17648</td>
<td>1666</td>
</tr>
</tbody>
</table>

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.
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.
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.

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).

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.

**NEMESYS-single.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPU bus load</th>
<th>NA usage</th>
<th>MEM usage</th>
<th>MEM via</th>
<th>NMA via</th>
<th>NMA util.</th>
<th>avg. cycles</th>
<th>load</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>5.3 %</td>
<td>11.5 %</td>
<td>67.2 %</td>
<td>53</td>
<td>22.7 %</td>
<td>44.5 %</td>
<td>68.3 %</td>
<td>318</td>
<td>234</td>
</tr>
<tr>
<td>DST</td>
<td>5.2 %</td>
<td>25.6 %</td>
<td>67.9 %</td>
<td>232</td>
<td>27.4 %</td>
<td>40.5 %</td>
<td>66.1 %</td>
<td>479</td>
<td>392</td>
</tr>
<tr>
<td>BY</td>
<td>6.7 %</td>
<td>13.9 %</td>
<td>45.7 %</td>
<td>52</td>
<td>26.3 %</td>
<td>19.4 %</td>
<td>55.8 %</td>
<td>289</td>
<td>212</td>
</tr>
<tr>
<td>DR</td>
<td>11.6 %</td>
<td>30.3 %</td>
<td>76.5 %</td>
<td>67</td>
<td>38.6 %</td>
<td>37.9 %</td>
<td>90.5 %</td>
<td>460</td>
<td>361</td>
</tr>
<tr>
<td>DS</td>
<td>7.6 %</td>
<td>16.4 %</td>
<td>55.3 %</td>
<td>223</td>
<td>26.0 %</td>
<td>29.3 %</td>
<td>64.7 %</td>
<td>319</td>
<td>241</td>
</tr>
<tr>
<td>KC</td>
<td>12.3 %</td>
<td>46.9 %</td>
<td>56.3 %</td>
<td>281</td>
<td>40.5 %</td>
<td>15.8 %</td>
<td>64.2 %</td>
<td>506</td>
<td>444</td>
</tr>
<tr>
<td>LCR</td>
<td>9.9 %</td>
<td>16.3 %</td>
<td>35.7 %</td>
<td>106</td>
<td>28.0 %</td>
<td>7.7 %</td>
<td>28.7 %</td>
<td>291</td>
<td>233</td>
</tr>
<tr>
<td>HS</td>
<td>7.7 %</td>
<td>10.1 %</td>
<td>32.6 %</td>
<td>266</td>
<td>23.5 %</td>
<td>9.1 %</td>
<td>33.6 %</td>
<td>220</td>
<td>150</td>
</tr>
<tr>
<td>DP</td>
<td>4.9 %</td>
<td>12.6 %</td>
<td>41.3 %</td>
<td>103</td>
<td>22.8 %</td>
<td>18.5 %</td>
<td>41.3 %</td>
<td>295</td>
<td>219</td>
</tr>
<tr>
<td>MIS</td>
<td>4.8 %</td>
<td>10.8 %</td>
<td>39.8 %</td>
<td>299</td>
<td>26.6 %</td>
<td>13.2 %</td>
<td>21.7 %</td>
<td>225</td>
<td>151</td>
</tr>
<tr>
<td>MST</td>
<td>3.7 %</td>
<td>5.8 %</td>
<td>26.4 %</td>
<td>170</td>
<td>13.7 %</td>
<td>12.7 %</td>
<td>25.2 %</td>
<td>236</td>
<td>166</td>
</tr>
<tr>
<td>VC</td>
<td>3.9 %</td>
<td>4.1 %</td>
<td>45.5 %</td>
<td>60</td>
<td>14.3 %</td>
<td>31.2 %</td>
<td>46.6 %</td>
<td>186</td>
<td>94</td>
</tr>
<tr>
<td>Mean</td>
<td>7.0 %</td>
<td>20.9 %</td>
<td>49.2 %</td>
<td>159</td>
<td>25.9 %</td>
<td>32.3 %</td>
<td>50.5 %</td>
<td>319</td>
<td>241</td>
</tr>
</tbody>
</table>

**Pegasus-single.**

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

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.
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.