Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores

Manuel Mohr
Programming Paradigms Group
Karlsruhe Institute of Technology
manuel.mohr@kit.edu

Carsten Tradowsky
Institute for Information Processing Technologies
Karlsruhe Institute of Technology
carsten.tradowsky@kit.edu

***Abstract*—To improve scalability, some many-core architectures abandon global cache coherence, but still provide a shared address space. Partitioning the shared memory and communicating via messages is a safe way of programming such machines. However, accessing pointered data structures from a foreign memory partition is expensive due to the required serialization.**

In this paper, we propose a novel data transfer technique that avoids serialization overhead for pointered data structures by managing cache coherence in software at object granularity. We show that for PGAS programming languages, the compiler and runtime system can completely handle the necessary cache management, thus requiring no changes to application code. Moreover, we explain how cache operations working on address ranges complement our data transfer technique. We propose a novel non-blocking implementation of range-based cache operations by offloading them to an enhanced cache controller. We evaluate our approach on a non-cache-coherent many-core architecture using a distributed-kernel benchmark suite and demonstrate a reduction of communication time of up to 39.8%.

I. Introduction

With increasing core counts, guaranteeing cache coherence while maintaining performance and power efficiency is becoming increasingly difficult. Classical snooping protocols do not scale beyond a relatively low number of cores while directory-based protocols considerably increase latency and power consumption [1, 2]. Hence, some architectures [3–6] drop global cache coherence to improve scalability.

Fig. 1 shows a schematic view of such an architecture. Cores have private caches and are connected by a scalable interconnect, such as a network-on-chip, that handles on-chip communication. Such architectures have a shared physical address space, hence cores can read from and write to DRAM. They also cache results in their private caches, however, the hardware gives no coherence guarantees. Hence, these machines cannot be directly programmed using the common shared memory programming model.

One approach to deal with this situation is to provide the missing cache coherence (at least partially) in software [7–10]. An alternative is to partition the shared memory between the cores and let the cores communicate via explicit messages, for which such architectures usually provide special hardware support. Partitioning the memory means that every core only accesses (and caches) addresses in its own memory partition, hence the missing cache coherence does not cause problems.

However, message passing of pointered data structures entails costly (de-)serialization. Consider the situation that core S has a linked list in its memory partition and wants to send it to core R. Core S must first convert the list to a format suitable for message passing, i.e. serialize it to a byte stream, which R then receives to reconstruct (deserialize) a copy of the original list. The (de-)serialization causes a large overhead, both memory-wise and computation-wise. As such pointered data structures occur frequently in general-purpose applications, especially if written in high-level object-oriented languages, it is important to accelerate their transfer.

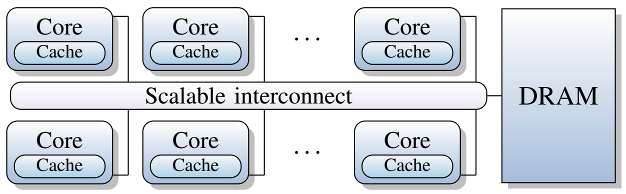


Fig. 1: A schematic view of a non-cache-coherent system. Private caches, shared memory (DRAM), but no hardware cache coherence.

In this paper, we propose a novel approach for transferring pointered data structures between shared memory partitions without requiring coherent caches. We propose that the receiver directly accesses the data structure in the sender's memory partition and makes a deep copy of it, i.e. clones it, in the receiver's partition, thereby avoiding the need for serialization and temporary buffers. To guarantee correctness, the software forces the necessary cache writebacks and invalidations. We show that in a programming language following the partitioned global address space (PGAS) model, the compiler and runtime system can issue the cache operations fully automatically, thus existing software does not have to be modified. As we manage cache coherence in software on a coarse granularity (whole objects), we show the need for range-based cache operations.

The contributions of this paper are:

1) a novel technique for transferring pointered data structures via shared memory on non-cache-coherent architectures based on software-managed cache coherence,
2) a compiler-assisted implementation that is fully automatic, safe, and has zero overhead based on the PGAS programming language X10,
3) an extensive evaluation measuring running times of distributed algorithm kernels on a non-cache-coherent many-core architecture, demonstrating communication time reductions of up to 39.8%, and
4) a novel implementation and an evaluation of non-blocking range-based cache operations that offload work to an enhanced cache controller with an area overhead of 15% compared to the original cache controller.

The rest of the paper is organized as follows. In Section II we first formally state our problem and then study two existing message-passing-based data transfer techniques. Section III describes our novel cloning approach and presents our compiler-assisted implementation for X10. In Section IV, we demonstrate that range-based cache operations complement our technique and present a novel non-blocking implementation. Next, Section V presents the performance improvements as measured on a non-cache-coherent many-core architecture and an evaluation of our hardware extension. Finally, Section VI gives an overview of related work and Section VII concludes the paper.

II. Motivation and Background

In this section, we first describe the problem of transferring pointered data structures in more detail. We then study two message passing-based approaches before we turn to our novel cloning technique in Section III.

Problem Statement: With partitioned shared memory, the programming model prevents accesses to foreign memory partitions. Hence, if a core wants to work on some piece of data, there must be a local copy of it in the core's private memory partition. For distributing data, non-cache-coherent many-core architectures provide efficient message passing mechanisms. If, for example, one core wants to distribute input data stored in its own memory partition, it sends messages carrying the data to the other cores. Each core then works on its local copy.

This works well if the input data has a "flat" memory layout, e.g., a simple array of some primitive numeric type, which is already in a format well-suited for a message. However, programs, especially those written in high-level languages, frequently use pointered data structures, such as linked lists or trees. Additionally, we expect this to be common in programs ported from a shared-memory programming model to run on a non-cache-coherent architecture. It is important to understand what it means to make a copy of a pointered data structure.

In general, we can represent a data structure by an *object graph*, which is a directed graph where the vertices are objects and an edge (x, y) means that x points to y. All object graphs have a designated root object. Such an object graph can contain cycles, e.g., the graph of a cyclic linked list.

We call data structures *flat* if their respective object graph has a single vertex and no edges, and *pointered* otherwise. Making a copy of an object graph in a different memory partition requires creating a *deep copy*. Hence, we must copy all objects reachable from the root and at the same time modify the contained pointers so that they point to the newly created objects. A shallow copy, obtained by bytewise copying of the root object, is not sufficient as the contained references would point to a different memory partition, which is unsafe.

Cache Terminology: Throughout this paper, we assume a cache that offers three operations: invalidate, writeback, and flush. Furthermore, we assume that all operations can be executed on the cache line associated with a specific address or on the whole cache. Invalidate marks a cache line as invalid, meaning that the next time an address from the cached range is accessed, it will be fetched from memory. Writeback writes a dirty cache line back to memory. The cache line stays valid after this operation. Flush combines writeback and invalidate.

Our starting point for all three approaches described in the following is the transfer of an object graph G of a pointered data structure from a sending core S to a receiving core R. Fig. 2 shows schematics for all three approaches.

A. Message Passing

Classical message passing (MP) proceeds according to the following three steps (see Fig. 2a):

1) S serializes G into a contiguous buffer B located in its private memory partition.
2) S sends the contents of B via one (or multiple, if the maximum message size is limited) message(s) to R. This may entail additional overhead for copying buffer contents from shared memory to specialized local memory used for message passing[1] and for splitting large messages into smaller parts.
3) R writes the message contents to a contiguous buffer B' in R's private memory partition. R then deserializes a copy G' of the object graph from B'.

In total, MP requires four times as much memory as the initial object graph G. Additionally, assembling B and reading B' evicts large parts of the caches of S and R. However, MP also works on machines without shared memory.

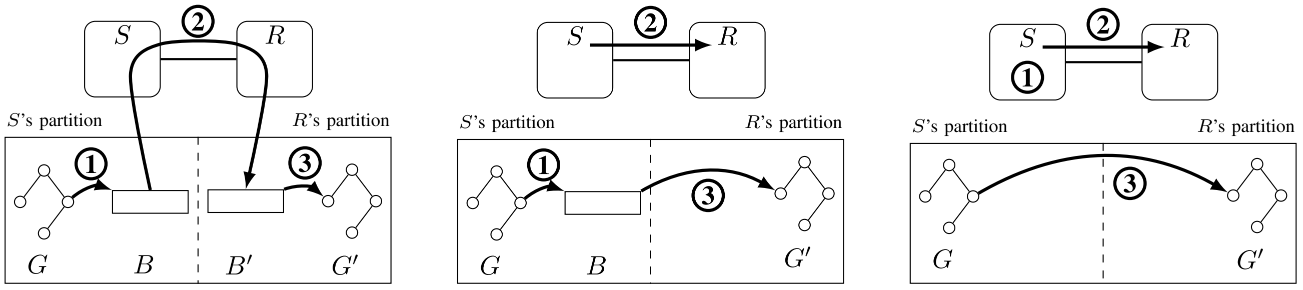
B. Message Passing via Shared Memory

Passing messages via non-cache-coherent shared memory (MP-SHM) follows these three steps (see Fig. 2b):

1) *write & writeback*: S serializes G into a buffer B located in its private memory partition. Then, S forces a writeback for the cache lines of B from its local cache. The writeback guarantees that R can read up-to-date values for B from memory. It is trivial to determine the relevant cache lines, as B is contiguous in memory and we know its starting address and size. S waits until all relevant cache lines have been committed to memory.
2) *notify*: S sends a message carrying the starting address of B to R. This informs R that it is now safe to read B.
3) *read & invalidate*: R deserializes from B a copy G' of the object graph. Then, R invalidates the cache lines relevant for B. The cache invalidation is necessary to ensure that B is actually read from memory, even if S reuses B.

Note that waiting until all updates to B are visible to other cores may require special hardware support, e.g., some kind of memory barrier. The order in which the hardware commits updated cache lines to memory is not important, allowing weaker memory models. MP-SHM avoids sending the contents of B via a message by passing them via shared memory and avoids allocating a buffer on the receiving side. Therefore, MP-SHM is also beneficial for transferring flat data structures [11, 12]. However, MP-SHM still requires (de-)serializing the object graph, potentially evicting parts of S's and R's caches.

[1]Because the shape of G and therefore the size of B is only known at runtime (as G can depend on input data), it is in general not possible to directly serialize to local memory as it may be too small to hold B.



(a) Message passing. (b) Message passing via shared memory. (c) Cloning.

Fig. 2: Schematic comparison of approaches to transfer an object graph G from sending core S to receiving core R. Temporary buffers are denoted by B, B'; and G' is the resulting copy of G.

III. Cloning

Our novel cloning approach (CLONE) works according to the following three step scheme (see Fig. 2c):

1) *writeback*: S forces a writeback of all objects in G. For each object we know its starting address and size. Hence, by traversing G, we can write back the relevant cache lines of each object. Then, S waits until all relevant cache lines have been committed to memory.
2) *notify*: S sends a message carrying the address of the root object of G to R. This notifies R that it is now safe to clone G.
3) *clone & invalidate*: R clones G, resulting in G'. The clone operation is a depth-first traversal of G with cycle detection (like serialization). Thus, we visit each object o exactly once and directly create a copy o' in R's memory partition. After creating o', R invalidates the relevant cache lines for o. Hence, after cloning, R's cache does not contain data from S's memory partition (analogous to MP-SHM).

The main difference between CLONE and message passing-based approaches is that it avoids serialization and requires no temporary buffers. Thus, it is much more cache-friendly. For flat data structures, CLONE is equivalent to MP-SHM. In this case, there is no need for serialization on the sending side ("$G = B$") and "deserialization" is equivalent to copying the single object, i.e. cloning it. Viewed this way, CLONE is a generalization of MP-SHM from flat to pointered data structures. Viewed another way, CLONE augments the widely-used object cloning technique with explicit writebacks and invalidations to allow its use on non-cache-coherent systems.

Implementation for PGAS Programming Languages

The PGAS model combines the message-passing and shared-memory programming models: it explicitly exposes data locality like in a distributed setting, but provides the illusion of a shared global address space with the ability to reference remote data items. PGAS programming languages tightly integrate this model. Here, the compiler inserts communication operations if remote data is accessed. Therefore, the compiler has a full view of all types in the program and at the same time controls the communication.

PGAS languages enable a compiler-assisted implementation of CLONE that is 1) **fully automatic**, i.e. requires no program changes, 2) **safe**, i.e. ensures that exactly the necessary cache lines are written back or invalidated, and 3) has **zero overhead**, i.e. requires no additional data structures or communication for coherence management. In our implementation of CLONE for the PGAS language X10, the compiler generates specialized writeback and cloning functions (corresponding to steps 1 and 3 of CLONE) per type. When a remote data item is accessed, the compiler knows its type and generates code to invoke the matching writeback and cloning functions on sender and receiver, respectively.

The PGAS model prevents accesses to shared data from different coherence domains on a logical level. Hence, we do not need per-object data structures to manage access to shared objects. Additionally, CLONE does not cause additional communication for coherence management: the sending core knows it has the most up-to-date version of its data as it is located in the core's private memory partition.

It is not intuitively clear that it is safe for R to access G from S's partition. For example, what if S modifies G during cloning? However, in this case the program contains a data race, as it modifies a data item concurrent to a transfer of that item. In programs with such data races, data transfers can be corrupted independent of whether data is serialized or cloned. Therefore, cloning is safe for correctly synchronized programs.

IV. Hardware Extension

Both MP-SHM and CLONE manage cache coherence at coarse granularity, i.e. operate on address ranges. When forcing a writeback or an invalidation of an address range A, we have to trigger the cache operation individually for each cache line relevant for A. This can take a long time, so we would like to accelerate this type of operation in hardware.

Moreover, we can hide the latency of the cache operations by performing other actions. We force invalidations or writebacks during object cloning when visiting each object. Hence, we would like to trigger the necessary cache operations for one object and have them executed in the background while we continue with the next object in the graph.

In the following, we present our novel concept and implementation of non-blocking range-based cache operations (or range operations for short) that achieves both of our goals. Our range operations offload the work to an enhanced cache controller. The underlying processor for our implementation is a Gaisler LEON 3, which implements the SPARC V8 ISA.

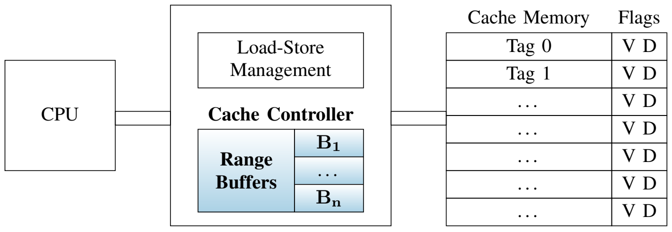


Fig. 3: Schematic view of our modified cache architecture.

However, neither our concept nor our implementation are tied to this particular ISA or microarchitecture.

Fig. 3 shows a schematic view of the modified cache architecture with changed parts of the cache controller highlighted bold. First, we extend the cache controller with the ability to invalidate, write back or flush multiple cache lines. This adds the software interface (CPU view) according to the cache operations and the hardware interface (cache view) to interact with the cache memory and reset the flags accordingly. The cache controller can modify one cache line per cycle.

Second, to make the range operations non-blocking, we add *range buffers* B_i to the cache controller. Each range buffer holds a triple (s, e, t) of start address s, end address e, and operation type t (invalidation, writeback, or flush). Each time the processor executes a range operation on a range A, the cache controller stores A along with its operation type in a range buffer according to the following rules:

(i) If there is no free range buffer, we halt the processor until a buffer becomes free.
(ii) If A overlaps with a range A' already stored in another buffer, we halt the processor until A' has been processed.
(iii) Otherwise, we store A and its type in a free range buffer.

Then, the processor continues executing the program. Every time it executes a load or store to an address D, the cache controller checks D against all stored ranges. If $D \in A$ for a stored range A, we halt the processor until the operation on A has finished. Otherwise, we perform a cache lookup as usual.

In every cycle, during which the processor does not execute a load or store, the cache controller uses this spare cycle to work on range operations. As long as there is at least one range A stored in a range buffer, the cache controller applies the respective operation to the next cache line relevant for A, e.g., clearing a line's valid bit for an invalidation. The cache controller keeps track of its progress using an internal register. It therefore takes at most n spare cycles to apply an operation to a range spanning n cache lines.

V. Evaluation

We analyze the performance of Clone and compare it to MP and MP-SHM. We first consider individual data transfers and then look at distributed kernel benchmarks. We perform all experiments on a non-cache-coherent architecture without our hardware extensions. Finally, we investigate overhead and benefit of our cache controller extension.

A. Setup

We conducted all running time measurements on an FPGA-based implementation of a non-cache-coherent tiled many-core architecture without our hardware extensions. The architecture consists of 4 tiles with 4 cores each. Each tile forms a coherence island, where cache coherence is guaranteed by a classical bus snooping protocol. However, there is no cache coherence between tiles. The tiles are connected by a network on chip [13] (NoC).

All cores are Gaisler SPARC V8 LEON 3 [14, 15] processors. Each processor has a private $16\,\mathrm{KiB}$ 2-way instruction cache and a private $8\,\mathrm{KiB}$ 2-way write-through L1 data cache. Additionally, the 4 cores of each tile share a $64\,\mathrm{KiB}$ 4-way write-back L2 cache. Each tile has $8\,\mathrm{MiB}$ of SRAM-based on-chip memory. Message passing between tiles is implemented using DMA transfers between on-chip memories. One of the tiles has $256\,\mathrm{MiB}$ of DDR3 memory, used as shared memory, attached to its internal bus. We do not use this tile during our experiments; hence, the used cores all access the shared memory via the NoC. The hardware design was synthesized to a CHIPit Platinum system, a multi-FPGA platform based on Xilinx Virtex 5 LX 330 FPGAs.

On the software side, we use X10 [16] as our PGAS language. We use a modified X10 compiler [17] based on version 2.3, with an adapted compiler backend to generate SPARC code for our platform. As our operating system, we use OctoPOS [18]. We compiled all C components of our software stack using the official SPARC toolchain [19] provided by Gaisler. We use platform-specific operations [20, sections 68.3.3 and 71.10.7] to force writebacks or invalidations of L2 cache lines associated with specific addresses. The MP approach did not have to split messages in our experiments.

We repeated each experiment 50 times. The standard deviation for all runs was below 0.1%, so we omit giving standard deviations and report minimum running times.

B. Individual Data Transfers

In the following, we look at individual transfers of pointered data structures. We transfer a circular doubly linked list, varying the number of elements n and the size per element E. Table I shows the speedup of Clone over MP-SHM for lists from 1 to 256 elements with element sizes up to $4\,\mathrm{KiB}$.

We see that, in general, speedups increase with increasing element size and increasing total data size. Clone is always as fast as MP-SHM and provides speedups of up to $7.45\times$. Interestingly, if the object graph consists of many small elements, Clone provides little or no benefit over MP-SHM. Here, the overhead for traversing the object graph, which is needed for both approaches, dominates and whether we serialize or clone the data has little influence on the running time. For object graphs that are significantly larger than the cache size we observe high speedups. In these cases, serializing the object graph into a buffer puts heavy load on the memory subsystem, which is avoided by cloning.

C. Distributed Benchmark Kernels

We now compare the running times of X10 applications using MP, MP-SHM, and Clone. We use the X10 programs from the IMSuite benchmark suite [21] as our test inputs. IMSuite consists of 12 programs that implement popular,

Table I: Speedup of CLONE over MP-SHM for copying a circular doubly linked list with n elements of size E.

	Element size E (in bytes)						
n	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
2^0	$1.32\times$	$1.33\times$	$1.34\times$	$1.35\times$	$1.39\times$	$1.39\times$	$1.40\times$
2^1	$1.28\times$	$1.30\times$	$1.36\times$	$1.38\times$	$1.45\times$	$1.42\times$	$1.45\times$
2^2	$1.26\times$	$1.33\times$	$1.36\times$	$1.39\times$	$1.40\times$	$1.47\times$	$1.52\times$
2^3	$1.25\times$	$1.31\times$	$1.37\times$	$1.38\times$	$1.45\times$	$1.51\times$	$1.58\times$
2^4	$1.13\times$	$1.21\times$	$1.31\times$	$1.30\times$	$1.44\times$	$1.57\times$	$1.77\times$
2^5	$1.05\times$	$1.22\times$	$1.27\times$	$1.36\times$	$1.54\times$	$1.73\times$	$1.86\times$
2^6	$1.01\times$	$1.17\times$	$1.30\times$	$1.47\times$	$1.68\times$	$1.78\times$	$1.84\times$
2^7	$1.03\times$	$1.16\times$	$1.33\times$	$1.54\times$	$1.69\times$	$1.77\times$	$5.62\times$
2^8	$1.04\times$	$1.19\times$	$1.36\times$	$1.54\times$	$1.70\times$	$5.20\times$	$7.45\times$

mostly graph-based distributed algorithm kernels, such as computation of dominating sets, spanning trees, and vertex colorings. Being distributed in nature means that, when run on a non-cache-coherent architecture, the programs must communicate between coherence domains. Hence, they are a good fit for assessing data transfer performance. The sizes of the test programs range from $300\,\mathrm{loc}$ to $1000\,\mathrm{loc}$.

We use the iterative X10-FA configuration of the benchmark programs with the input data set of size 64. We use the running time measurement infrastructure already present in the programs. We modified the programs so that they contain their input data as our test platform does not provide a file system. Input data is read during the initialization phase, which is not included in the running time measurements.

The upper half of Table II shows the running times of all benchmarks for the three tested variants. First, we see clear differences in the running times between the three variants, which means that due to their distributed nature, the benchmark kernels spend a significant portion of their running time for communication. We see that exploiting shared memory for data transfers on non-cache-coherent architectures is crucial: for most benchmarks, there is a large gap between MP and the other two variants as MP does not exploit shared memory.

The lower table half shows the reduction of the time spent for communication as well as the overall speedup of CLONE over MP and MP-SHM. We instrumented the programs to determine the time spent for communication. On average, CLONE provides a 34.5% reduction in communication time relative to MP, translating into an average speedup of $1.17\times$. Compared to MP-SHM, CLONE achieves an average communication time reduction of 8.1%, resulting in an average speedup of $1.05\times$.

For every test case, CLONE is at least as fast as MP-SHM. Here, the speedup depends on how expensive it is to serialize the data structures transferred by the test cases. Instrumentation revealed that the programs where the speedup is significantly above average frequently transfer larger pointered data structures, e.g., of size $8\,\mathrm{KiB}$ for MST, where CLONE achieves a communication time reduction of 39.8%, resulting in a speedup of $1.24\times$.

D. Hardware Support

We implemented our proposed range operations as an extension to the cache controller of the Gaisler LEON3 processor [14]. Table III shows that compared to the unmodified cache controller, about 15% of additional logic is necessary to implement non-blocking range operations with one range buffer on the Xilinx XUPV5 Virtex-5 FPGA.

As explained in Section IV, our implementation needs at most n spare cycles to execute a range operation on a range spanning n cache lines. We instrumented the programs from IMSuite and found that the average object graph size is $257.3\,\mathrm{B}$. On our system, the minimum cache line size is $16\,\mathrm{B}$. Hence, there must be at least 17 spare cycles between two range operations to avoid blocking. Analysis of the generated code for performing writebacks and invalidations showed that this is fulfilled. In both cases, we use a resizable hash set to detect cycles in the object graph. Operating on the hash set involves enough arithmetic and control flow instructions to hide the range operation's latency. Therefore, executing a range operation during CLONE takes one cycle from the view of the processor for the average object graph.

VI. Related Work

Data Transfers: Ureña et al. [11] present an MPI implementation that transfers large messages via shared memory on the Intel SCC. This is basically the MP-SHM approach, however, shared memory is marked uncacheable, thus Ureña et al. do not need to force writebacks or invalidations. There is prior work on using X10 on the Intel SCC [22]. However, the authors used the default MP approach. Christgau et al. [12] present an approach for software-managed cache coherence to accelerate MPI one-sided communication on the Intel SCC. In contrast to our work, they only consider flat objects.

Most closely related is the work of Prescher et al. [23, 24]. They present MESH, a C++ framework for distributed shared memory that supports non-cache-coherent architectures. While we focus on data replication, MESH allows choosing between different sharing models (replication, central instance, and mixtures of both). However, MESH is library-based as opposed to our compiler- and language-based approach. As such, existing software must be modified to be used with MESH. Moreover, their implementation requires a consistency controller object per shared object and triggers additional communication for coherence management. We avoid this overhead, as we manage coherence in a more restricted environment under control of the compiler.

The evaluation of all previously mentioned papers was hindered by the fact that the SCC does not provide fine-grained cache control. This supports our case for range-based cache operations. To the best of our knowledge, it makes our evaluation the first to investigate software-based cache coherence on a non-cache-coherent architecture with fine-grained cache control.

Range-Based Cache Operations: Range-based cache operations have been implemented before. The ARM1136J(F)-S

Table II: Upper half: Running times (in seconds) of all test programs from IMSuite for each of the three variants MP, MP-SHM, and CLONE. Lower half: Reduction of communication time and overall speedups of CLONE over MP and MP-SHM.

	Benchmark												
	BF	DST	BY	DR	DS	MIS	KC	DP	HS	LCR	MST	VC	Geomean
MP	1.30	9.35	736.79	83.22	50.92	1.75	27.10	36.59	43.86	14.24	69.82	1.60	
MP-SHM	1.17	7.94	677.27	82.13	47.24	1.60	25.86	34.14	34.81	11.92	62.87	1.30	
CLONE	1.13	7.35	658.39	80.42	45.49	1.57	25.84	32.61	34.00	11.88	50.70	1.26	
Reduction$_{\text{MP}}$	33.7%	57.6%	28.2%	22.5%	22.0%	33.7%	12.5%	35.2%	56.2%	49.0%	50.9%	50.1%	34.5%
Reduction$_{\text{MP-SHM}}$	9.7%	28.4%	8.6%	15.0%	8.3%	7.7%	0.3%	17.3%	9.9%	1.8%	39.8%	9.5%	8.1%
Speedup$_{\text{MP}}$	1.15×	1.27×	1.12×	1.03×	1.12×	1.12×	1.05×	1.12×	1.29×	1.20×	1.38×	1.27×	1.17×
Speedup$_{\text{MP-SHM}}$	1.03×	1.08×	1.03×	1.02×	1.04×	1.02×	1.00×	1.05×	1.02×	1.00×	1.24×	1.03×	1.05×

Table III: Additional resources for the implementation of non-blocking range operations compared to original cache controller.

	Additional resources	
	absolute	relative
Slices	1489	15.2%
Register	623	14.6%
LUT	1491	15.0%
BRAM	1	4.9%

processors [25, sec. 3.3.17] can perform writebacks and invalidations of address ranges via a system control coprocessor. In contrast, our concept does only require an enhanced cache controller instead of a full-blown coprocessor. Other articles [12, 24] conclude that cache operations working on address ranges are desirable on non-cache-coherent architectures. However, to the best of our knowledge, our work is the first to explore an actual hardware implementation in the context of such an architecture.

VII. Conclusion

In this paper, we proposed a new technique to transfer pointered data structures on non-cache-coherent shared memory systems. Our novel cloning approach avoids serialization by managing cache coherence in software at object granularity. We presented a compiler-assisted implementation for PGAS languages that is fully automatic, safe, and has zero overhead. Our experimental results using a distributed-kernel benchmark suite show that using our technique reduces communication time by up to 39.8%. Additionally, we demonstrated that cache operations on address ranges are desirable on non-cache-coherent architectures. We showed that for 15% additional hardware resources, an existing cache controller can be extended with an efficient non-blocking implementation of range operations. Our approach forms a new point in the design space of non-cache-coherent shared memory systems: the PGAS model hides shared memory from the user (as it is in general unsafe to use due to missing cache coherence) but data transfers of pointered data structures are accelerated through a compiler-assisted approach that exploits shared memory.

Acknowledgments: Thanks to the whole InvasIC team, especially those who helped to build the platform used in the evaluation, and to Michael Mechler for implementing the range operations. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

References

[1] B. Choi *et al.*, "Denovo: Rethinking the memory hierarchy for disciplined parallelism," in *PACT*, 2011, pp. 155–166.

[2] S. Kaxiras *et al.*, "SARC coherence: Scaling directory cache coherence in performance and power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, Sept 2010.

[3] J. Howard *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *ISSCC*, Feb 2010, pp. 108–109.

[4] N. P. Carter *et al.*, "Runnemede: An architecture for ubiquitous high-performance computing," in *HPCA*, 2013, pp. 198–209.

[5] S. Lyberis *et al.*, "Formic: Cost-efficient and scalable prototyping of manycore architectures," in *FCCM*, 2012, pp. 61–64.

[6] Y. Durand *et al.*, "Euroserver: Energy efficient node for european micro-servers," in *DSD*, August 2014, pp. 206–213.

[7] J. H. Kelm *et al.*, "Cohesion: A hybrid memory model for accelerators," in *ISCA*, 2010, pp. 429–440.

[8] T. J. Ashby *et al.*, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE TC*, vol. 60, no. 4, pp. 472–483, Apr. 2011.

[9] X. Zhou *et al.*, "A case for software managed coherence in manycore processors," in *USENIX*, 2010.

[10] S. V. Adve *et al.*, "Comparison of hardware and software cache coherence schemes," in *ISCA*, 1991, pp. 298–308.

[11] I. A. C. Ureña *et al.*, "RCKMPI – lightweight MPI implementation for Intel's single-chip cloud computer (SCC)," in *EuroMPI*, 2011, pp. 208–217.

[12] S. Christgau *et al.*, "Software-managed cache coherence for fast one-sided communication," in *PMAM*, 2016, pp. 69–77.

[13] J. Heisswolf *et al.*, "The invasive network on chip - a multi-objective many-core communication infrastructure," in *ARCS*, Feb. 2014, pp. 1–8.

[14] Cobham Gaisler, "LEON 3," http://gaisler.com/index.php/products/processors/leon3, retrieved on 2016-09-09.

[15] SPARC Inc., "The SPARC architecture manual, version 8."

[16] V. Saraswat *et al.*, "X10 language specification," IBM, Tech. Rep., June 2015.

[17] M. Braun *et al.*, "An X10 compiler for invasive architectures," Karlsruhe Institute of Technology, Tech. Rep. 9, 2012.

[18] B. Oechslein *et al.*, "OctoPOS: A parallel operating system for invasive computing," in *SFMA*, 2011, pp. 9–14.

[19] Cobham Gaisler, "LEON bare-C cross compilation system."

[20] Cobham Gaisler, "GRLIB IP Core User's Manual."

[21] S. Gupta and V. K. Nandivada, "IMSuite: A benchmark suite for simulating distributed algorithms," *Journal of Parallel and Distributed Computing*, vol. 75, pp. 1–19, 2015.

[22] K. Chapman *et al.*, "X10 on the single-chip cloud computer," in *X10*, 2011, pp. 7:1–7:8.

[23] T. Prescher, R. Rotta, and J. Nolte, "Flexible sharing and replication mechanisms for hybrid memory architectures," in *MARC*, vol. 55, 2011, pp. 67–72.

[24] R. Rotta, T. Prescher, J. Traue, and J. Nolte, "Data sharing mechanisms for parallel graph algorithms on the Intel SCC," in *MARC*, 2012, pp. 13–18.

[25] ARM, *ARM1136J-S technical reference manual*, r1p5 ed., 2009.