Evaluation of Adaptive Memory Management Techniques on the Tilera TILE-Gx Platform

Tobias Fleig, Oliver Mattes and Wolfgang Karl
Institute of Computer Science & Engineering (ITEC)
Karlsruhe Institute of Technology (KIT)
tobias.fleig@student.kit.edu,
mattes@kit.edu, karl@kit.edu

***Abstract*—Manycore processor systems are likely to be the future system structure, and even within range for usage in desktop or mobile systems. Up to now, manycore processors like Intel SCC, Tilera TILE or KALRAY's MPPA are primarily intended to use for high performance applications, utilizing several cores with direct inter-core communication to avoid access to external memory. The spreading of these manycore systems brings up new application scenarios with multiple concurrently running high-dynamic applications, changing I/O characteristics and a not predictable memory usage. Highly dynamic workloads with varying memory usage have to be utilized.**

In this paper the memory management of various manycore platforms is addressed. In more detail the Tilera TILE-Gx platform will be explained, presenting results of own evaluations accessing its memory system. Based on that, the concept of the autonomous self-optimizing memory architecture Self-aware Memory (SaM) exemplarily was implemented as a software layer on the Tilera platform. The results show that adaptive memory management techniques can be realized without much management overhead, in return achieving higher flexibility and and simple usage of memory in future system architectures.

I. Introduction

The continuously increasing integration level of CMOS devices and the limited increase of the CPU frequencies is leading to manycore systems. Up to now, these systems are mainly designed for executing high performance applications on several cores by using direct inter-core communication over shared on-chip caches or small per-core memories. So far, the connection to the system memory commonly is realized over only a small number of controllers to just one or a few external memory components. This lack in the memory system leads to inefficient memory assignment and causes congestion [1] getting worse scaling the core count or integrating heterogeneous cores. Furthermore, with the increasing number of cores, the so called memory wall [2], the difference between the uprising CPU speed and the slow external memory, also gets more and more important.

In addition, concurrently running multiple applications calls for a dynamic memory management. In most cases, an initial optimal dynamic assignment of memory regions to tasks is often not feasible, caused by data locality issues, placement restrictions and memory regions, which are already occupied by other tasks. To be able to scale the memory with the rising core count, we propose Self-aware Memory (SaM) [3] in order to approach the optimization problem of managing and assigning memory to tasks in high-dynamic application scenarios.

In this paper the memory management of different manycore platforms is addressed. In more detail the architecture and usage of the Tilera TILE-Gx platform will be explained. We present results of own measurements accessing the memory system. The first two address the access to private and shared memory. A third evaluation shows the results of message passing based direct communications between the cores. The experiences made within these evaluations confirm the restrictions of the memory access and management of current manycore architectures as well as their thereby restricted application scenario.

Based on these evaluations, the concept of the autonomous self-optimizing memory architecture Self-aware Memory (SaM) exemplary was adapted to the Tilera platform. Since modifications in the system structure and memory management in hardware are not possible for existing systems, we implemented SaM as a software abstraction layer running on top of the existing memory management and system structure. Aside from the fact that an additional software layer cannot be faster than the original HW-based memory accesses and management, the results show that adaptive memory management techniques can be realized without much management overhead, in return achieving higher flexibility and simple usage of memory in future system architectures.

The paper is organized as follows. In Section II the architecture of the Tilera TILE-Gx platform and the results of our measurements are presented. Additionally we present other current manycore systems in Section III. A short introduction to Self-aware Memory is given in Section IV. Section V and VI cover the adaptation and implementation of SaM to the Tilera TILE-Gx platform and first results of evaluations with the self-optimization mechanism on this machine. The paper concludes in Section 6 and gives an outlook regarding ongoing research.

II. Tilera TILE

For this paper the measurements of the existing memory system as well as the exemplary adaptation of SaM was done on a system using a TILE-Gx 8036 processor. Because of that and in contrast to other manycore systems, which are presented in Section III, the Tilera TILE platform is explained in more detail in this section.

The Tilera TILE-Gx processors are a group of commercially available manycore processors and a follow-up of the MIT RAW project [4]. As well as the Intel SCC [5] the Tilera systems use a tiled architecture and are designed to execute parallel applications like streaming applications, which mainly communicate directly between the cores, therefore using a shared cache [6]. Access to the external memory, depending on the core count of the processor, is achieved over one to four memory controllers. In 2008 the first Tilera processor TILE64 was published. Since then the manycore architecture was extended with the TILEPro and the TILE-Gx which is used in the following evaluations. There are TILE-Gx processors available with 9, 16, 36 and 72 cores up to now.

Eponymous for this architecture are the processor cores, called tiles. By using several small VLIW processors to build up identical cores, arranging them in a regular grid and connecting them over a network-on-chip (NoC), a high amount of these similar tiles could easy be combined to one processor.

The NoC connects all tiles of the system among themselves as well as to the external memory modules and further I/O. In order to achieve a high parallelization, the NoC consists out of 5 independent nets. Each tile is directly connected to its 4 surrounding adjacent tiles over a dedicated switch, which handles the redirection of the data packages over one of the 5 networks. Each of these networks has a special purpose - 3 of them are reserved for memory accesses, the cache management and coherency and their usage is transparent for the programmer. The 2 other ones are used for data transfer to the external I/O and direct inter-core connections. In the following evaluation and implementation the User Data Network (UDN) was used for the inter-core communication, because it is freely usable by the programmer.

A. Memory Architecture and Management

The memory of the TILE-Gx architecture is made both for shared memory and message passing, providing the programmer more flexibility and covering a bigger field of applications. In the standard operation mode a Linux operating system is used to provide an abstract system access with shared memory programming. In addition it is possible to pin tasks to distinct cores, directly exchanging message over the UDN. The Bare Metal Environment (BME) mode is available as a third mode, in which no normal operating system (and its libraries) runs any longer and the user has exclusive control over the whole system.

The compute cores are arranged in a grid and connected with a regular mesh network. Each tile has its own L1 and L2 cache. The memory controller are connected at the border of the grid to some individual tiles, depending on the specific processor version. For this paper we had access to a version with 36 cores and 2 memory controllers, each connected to the network on 2 positions. As can be seen in Figure 1 the left one is connected to the tiles with the numbers 6 and 24, the right one to the ones with the number 11 and 29. Thus, the next memory controller is accessible over a maximum of 3 tiles, a distinct controller with 6 hops.

A memory page is always administrated by one single tile, the so called home tile. Memory accesses of other cores to this page are first routed to the responsible home tile and answered using its L2 cache. In case of a miss the data is first updated in the L2 cache of the home tile and then sent to the requesting core. Accessing an already cached value therefore can be answered without an external memory access. But in case of cache miss the data is not directly transfered to the requesting core. This behavior is described as L3 cache by Tilera and is realized in a transparent way in hardware and the hypervisor.

Cache coherency for the pages is also guaranteed by the home tile, which keeps a list of all tiles which may have local copies in their caches. In case of a modification, invalidation messages are sent to all listed tiles.

B. Measure Methodology

The measured time periods are very short, about some hundreds CPU cycles. With that a high-level timing method is not usable, due to their limited resolution, unclear processing times or a context switch to the kernel. A method with constant run-time, high resolution and as low as possible influence on the run-time behavior was needed for our evaluations.

We achieved this with two preprocessor macros. In the first, which is called at the start of the measurement, two variables are declared, which are only used in the second macro. Then in a single machine code instruction, which is always processed in a single CPU cycle, a value of the special purpose register containing the program counter is saved in the normal register (r30). We block the register r30 using a compiler flag. This is done to abstain it from the clobber list, which holds all multiply used registers, due to efficiency reasons, so that the register value can be changed without saving/restoring it before. This is usable for our measurement, but maybe not within real systems.

After the following program code which should be measured, the second macro is called. In this, the program counter is saved into another register again with a single machine code instruction. After moving the register values to the previously declared variables the duration can be calculated. The second register doesn't have to be blocked, because the measured program code is finished at that time and an influence on the run-time is not given any more. With these two macros the measurement method has a constant run-time and no context switches or jumps are needed.

In the normal mode of the Tilera system, the operating system often disturbs the measurements e.g. by the scheduling. In addition the network is used by other tasks. To ensure that the measurement is not disturbed we used the so called dataplane modes. With it, running applications on a core are no longer interrupted, and only 1 core has to run Linux for the management and ssh connections to the systems. We used core 0 for that, because it is neighbor to a tile, which is connected to the memory, and can access the left and in this evaluation unused external memory without disturbing the rest of the network.

In the following the results of evaluations using 3 different memory usage scenarios are presented.

C. Private Memory

The first scenario covers access times to private memory. We reserved a segment as private memory in the right memory module and then access it from all cores. The segment is set

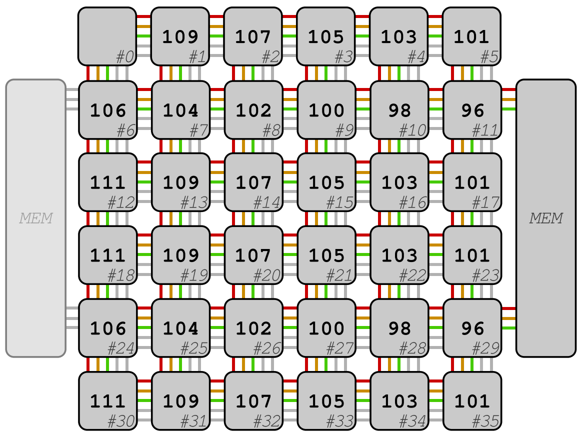


Fig. 1. Minimal access times in cycles for read accesses to private memory

as uncachable, so the data is not buffered on core side and repeated measurements always get the same results.

After the memory assignment we accessed from each core 1,000,000 times a defined memory value. Every time the duration is measured and out of all the minimum, maximum and the mean calculated. For core 0 no results are available because this core is used to run the operating system. Of special interest are the minimal access times as can be seen in Figure 1, because they represent the access times without any disturbance e.g. by other messages in the network. The repeated measurement allows us to increase the probability that for each core the minimal time was measured at least once. This minimal access time is determined by the hardware and cannot be undercut by a software based solution. The influence of the distance between the core and the memory can be seen in the printed values in Figure 1. Routing in horizontal direction always needs 2, in vertical direction 5 cycles. This forms a contrast to the official documentation, in which the transfer cost should be similar independent of the direction the data is routed to.

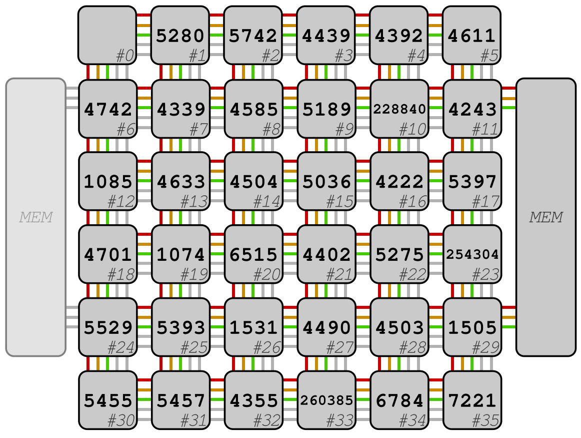


Fig. 2. Maximal access times in cycles for read accesses to private memory

The results also show that the distance to the data source is not negligible – for the core with the highest distance, 13,5 % of the access time are due to the network transfer. This percentage will increase with an increasing core count. In addition these times are only minimal access times, collected while a single core had exclusive access to the network and memory without any disturbance. Figure 2 presents the values for the maximal access times of the same evaluation. These values are collected also with only the test application running on only one core, maybe disturbed by some message of core 0. With applications running concurrently on all cores, the maximal values will be still higher. In real systems with a high load, the minimal values are not reachable and even the average and maximal values will be much higher.

D. Shared Memory

The second scenario covers access times to shared memory. This is the recommended mode of Tilera for the most jobs [7]. On these grounds the NoC is designed for high performance, providing more bandwidth to the automatically managed memory network than to the network which is freely usable by the programmer.

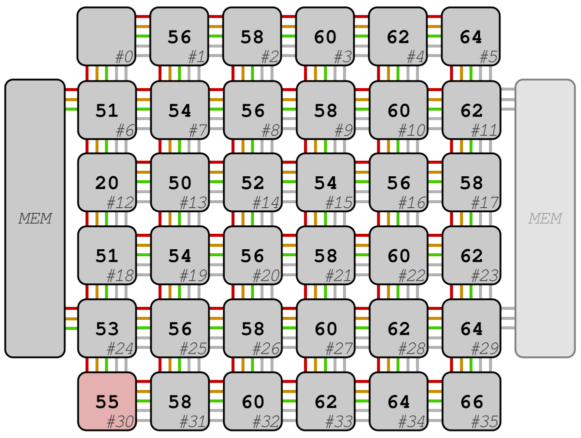


Fig. 3. Minimal access times in cycles for read accesses to shared memory

For the evaluation we used common memory, a shared memory variant of Tilera. First we assigned shared memory from a predefined tile (tile #30). Then the process forks processes to all other tiles – except core 0 with the operating system. In each measurement round the home tile changes the value in the shared memory, which invalidates the caches of the other tiles. Then all other tiles access the memory and measure their access time. To not disturb each other the tiles access the memory in a sequential order. With that, we always get comparable results, because the values cannot be taken out of the local caches but have to be received over the network. The minimal access times are shown in Figure 3, obtained from repeated measurements, in which also the maximum and mean was calculated.

As can be seen the access times to shared memory are shorter than at private memory. A reason for that is that the usage of caches cannot be completely prevented, some buffers like the TLB for the address translation are still used. According to the documentation, the data values are always loaded from the cache of the home tile of the memory segment (here tile 30). The measured values suggest, that the home tile is involved at all. In fact the tiles directly access the used left memory controller. This behavior could not be declared, accessing the value over the cache of the home tile would be much faster.

In Figure 3 tile #12 seems to be an aberration. This access time is only possible in case of a cache usage. Also the results have been the same for minimal and maximal values. A repeated evaluation showed, that sometimes some other tiles or not a single one could also have these values in a non reproducibly way.

E. Message Passing

The third scenario covers times for message passing between tiles. Such messages are sent over the previously described User Data Network (UDN). Neither the operating system nor any standard program make use of this network. Therefore, it is freely usable by the programmer and one has absolute control over every single message. This measurement is of special interest because the SaM implementation presented in Section V primarily relies on the UDN.

For these measurements, tile #1 acts as a server. Whenever a message is received, it is returned to the sender immediately. We measured the time between sending a message to tile #1 and receiving the answer on all other tiles. Again, the measurement is repeated multiple times (1,000,000) to increase the probability of finding the hardware inherent minimum. Figure 4 presents the results.

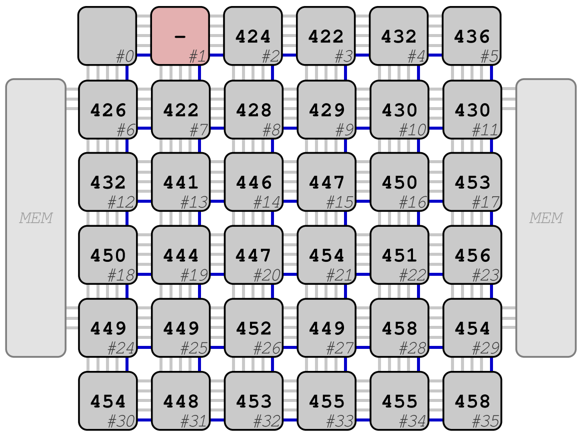


Fig. 4. Minimal round-trip-time in cycles, echo from tile 1

In general, the measured times do increase as expected with the distance to tile #1. However, there are some minor fluctuations. We expect these to be caused by the implementation of the API used to access the network. An example is active waiting for incoming messages in a loop.

F. Measurement Validation

As declared in the former sections, the measured memory access times strongly depend on the position of the tile in the grid and its assigned and accessed memory modules. As expected for currently available manycore systems, the access times are at a minimum 13,5 % higher for the farthest compared to the nearest tile. Moreover, the value for this slowdown can only achieved, when the whole system and network can be exclusively used by a single core. In real application scenarios, this actually will never be the case. With rising core count the slowdown instead will rise to a much higher level due to mutual interference using the same network or in accessing the same memory component.

III. Related Work

In addition to the already presented Tilera TILE architecture, we outline the architecture of some current manycore systems and their memory management in this section. As depicted in the following current manycore systems are not used as a scaled general purpose processor for flexible use with dynamic application scenarios. Most often they are designed to fulfill special needs and are used as co-processors or within the optimized and specialized high performance computing area using distinct parallel programming models. So, too, the memory management and accessibility often is provided in a restricted way. In the following we point out these specialization and accompanying restrictions.

A. RAMP Blue

RAMP Blue [8] is a FPGA-based multicore processor, developed at the University of California. Based on 84 Xilinx FPGAs, 1008 simulated compute cores could be achieved, running 12 MicroBlaze CPUs on each FPGA. Each core has 1 GB of exclusive memory on external memory modules, for usage with message passing. As shown in Figure 5 the system

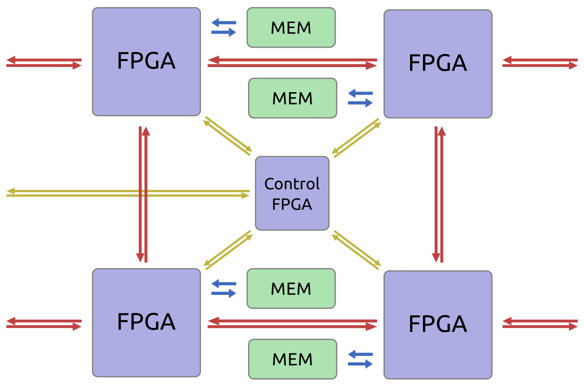


Fig. 5. Structure of a RAMP Blue board (simplified)

contains a multistage network. A control FPGA and 4 FPGAs are aggregated on a board, each connected over a 5 Gb/s ring bus. Multiple boards are connected via 10 Gb/s Ethernet in a 3D mesh network, the system uses processors without MMU running an adapted Linux (uClinux). Evaluations using benchmarks pointed out, that the overall system performance is limited by the high communication cost due to a software-based manual send and receive mechanism.

B. KALRAY MPPA

KALRAY's MPPA (Multi-Purpose Processor Array) manycore [9] is designed mainly as a dataflow architecture. The VLIW cores of the processor are grouped to clusters, containing a system core and integrated memory. The MPPA can be programmed by a c-based parallel dataflow model or with posix C/C++, which enables threading within a single cluster. The external main memory is connected over only two memory controllers, which also leads to differing memory access times.

C. Intel SCC

The Intel SCC (Single-chip Cloud Computer) [5] is the result of a project to evaluate different challenges of future manycore architectures, like a tiled architecture, network-on-chips, communication structures and programming models. Its

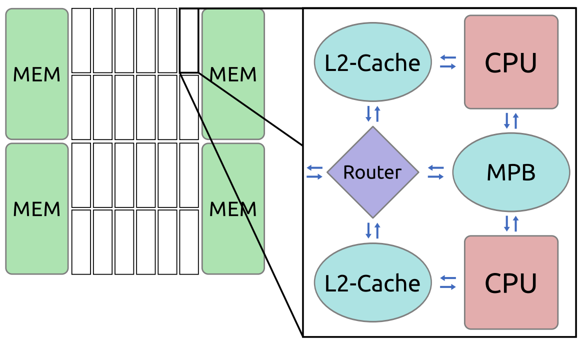


Fig. 6. Intel SCC Structure

main purpose lies in executing message passing applications, which communicate over the Message Passing Buffer (MPB), a distinct per-core cache-like memory. Access to the external connected memory is realized over four memory controllers. An initial memory assignment has to be manually done in advance of starting the system and executing applications. Depending on the locality of the used compute core, the access times to the external memory modules strongly differ. As shown in Figure 6, it consist out of 48 small Pentium 54C based cores, grouped on tiles, consisting of 2 cores, caches, a Message Passing buffer (MPB) and a router. The four external memory modules are each pre-assigned to 6 tiles and can be accessed over a connection. In addition to its main message passing purpose, an operation mode with shared memory is designated, but with no hardware cache coherency.

D. Intel Larrabee

Intel developed Larrabee as an architecture for high performance graphics cards. As Figure 7 shows, x86 processor cores were interconnected with a ring bus, recreating a pipeline structure commonly found in graphics cards. Each core has access to a part of a shared L2 cache that implements cache-coherency in hardware [10]. Larrabee was designed as a consumer-grade graphics card for computer games. Prototypes showed a near-linear speedup in frame rate with the number of cores. By using

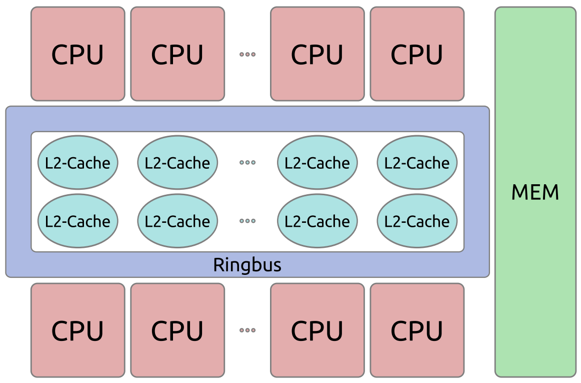


Fig. 7. Structure of Intel Larrabee

general-purpose CPUs instead of specialized vector-processing cores, Intel attempted to provide more flexibility to developers, ultimately resulting in higher frame rates. Because competitors performance could not be achieved [11], Intel stopped the graphics card project and released a Larrabee-based coprocessor card for high performance computing, named Xeon Phi.

IV. Self-aware Memory

Self-aware Memory (SaM) [3] mainly represents a memory architecture, which enables self-management of system components to build up a decentralized system architecture without a central management instance as a single point of failure. The main goal of SaM is to develop an autonomous memory subsystem for increasing the overall system reliability, flexibility and adaptability. This is crucial in upcoming computer architectures.

With SaM the individual memory modules act as independent units and are no longer directly assigned to a specific processor. SaM acts as an distributed and extended memory management unit and controls memory allocation, access rights, and ownership in a distributed manner. Figure

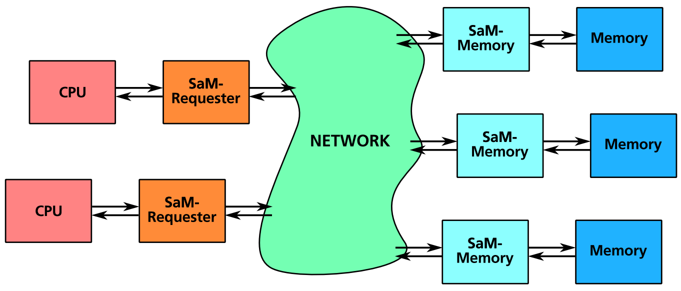


Fig. 8. Distributed SaM structure with assigned management components

8 depicts the structure of SaM. Due to this concept, SaM is constructed as a client-server architecture in which the memory modules offer their services (i.e., store and retrieve data) to client processors. The memory is divided into several autonomous self-managing memory modules, each consisting of a management component called SaM-Memory and a part of the physical memory. The SaM-Memory is responsible for handling access to its attached physical memory, administration of free and reserved space, as well as mapping to the physical address space. As a counterpart of SaM-Memory, the so called SaM-Requester augments the processor with self-management functionality. The SaM-Requester is responsible for handling memory requests, performing access rights checks, and mapping of virtual address space of the connected processor into the distributed SaM memory space. To enable and accelerate the management, tables and small caching structures are used.

The SaM components transparently realize virtual to physical addresses translation. Access to shared memory is enabled by integrating efficient synchronization techniques [12]. Memory coherency accessing shared memory regions is handled by the memory system guaranteeing the TM principles atomicity, consistency and isolation in a combined HW and SW approach. This provides the programmer an easy way for accessing shared memory and an abstract view of the memory resource.

In addition to that, research was done to establish a POSIX-like thread model allowing thread creation and management as well as allocation of compute nodes without a central management instance [13].

To increase the adaptivity of SaM to high-dynamic application scenarios, a decentralized self-optimization mechanism was integrated [14], [15]. The concurrently ongoing self-optimization is achieved with a five-stage cycle, containing the stages decentralized monitoring and data preprocessing, data analysis, optimization algorithms, decentralized consensus building, and the actual optimization. With this a decentralzed optimization approach for the memory assignment can be employed with only a small overhead of additional messages over the on-chip network.

V. Implementation

Up to now there are 3 different evaluation prototypes for Self-aware Memory. The most common prototype is a SystemC-based simulation [3], [14], [15], which easily can be parameterized and adapted to several test scenarios and system structures. With this, the memory management mechanism and the self-optimization process was evaluated and developed. In addition a coarse-grained implementation using several FPGA boards [13], [12], each representing a CPU or memory component, connected over Ethernet is available. The third prototype exists as a SW daemon, running on normal PC and redirecting memory access to other nodes, which also can be connected to the FPGA-based version via Ethernet. The goal of this work was to adapt and implement the concept of the Self-aware Memory to the Tilera platform to exemplarily demonstrate a high-dynamic and adaptive memory management on a existing manycore system.

A. Tracing

This work is motivated to enable a flexible memory management for high-dynamic application scenarios. Up to now there are no predefined benchmark scenarios for manycore systems available. To simulate these dynamic application scenario, we use a collection of memory traces we got from several benchmarks. This allows us to replay exactly the same scenario several times with changed parameters. Along with that it enables us, to easily run different evaluations with variable program and memory access phases. For each tile representing a CPU, an application scenario using a sequence of traces is provided.

B. Scenarios

The implementation on Tilera is also parameterizable to enable an easy evaluation of different application scenarios. The scenario definition is provided by a included file in which all important parameters are defined. Every tile can be configured as SamRequester or SamMemory component. The size of the memory nodes can be configured as well as the size of the used management tables. For each tile representing a CPU, an application scenario using a sequence of traces is provided. For the optimization cycle, the parameters of the decentralized monitoring are provided, e.g. the radius of the broadcast in which the status messages are exchanged. A bigger radius leads to a more global optimization knowledge, but also results in a higher overhead and additional messages. Associative counter arrays are used to arrange and pre-validate the collected information. The threshold of these arrays, which is used to launch the analysis stage of the cycle, as well as the used optimization algorithm can also be configured.

C. System Configuration

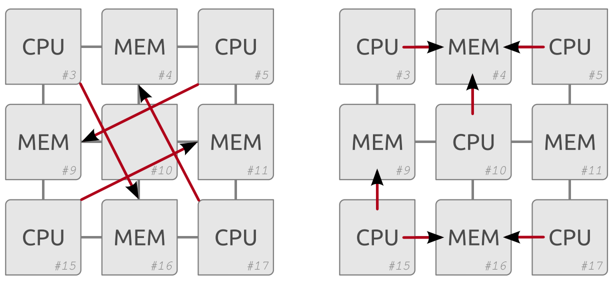
For the communication between the tiles we choose the previously addressed User Data Network (UDN). In the decentralized design of Self-aware Memory the nodes do not have prior knowledge of the system structure and are independent of a distinct network structure. Because of that, for the ongoing decentralized monitoring the status messages are exchanged using broadcasts with a distinct radius. The UDN doesn't support broadcasts, so this behavior is simulated in sending out messages to all neighbors in the regular grid. The implementation stays as software abstraction layer on top of the existing system. Therefor it adapts the existing memory management. In the presented evaluation we implemented the SaM layer as single tasks, each running exclusively on a tile. The memory accesses are blocking, so only one access can be done at one time. This restriction also affects the memory side in which only one message can be handled. So no concurrent memory accesses and monitoring exchange with neighbors are supported up to know.

D. Optimization

Every tile periodically sends out status messages to its neighbors. The content of these messages is depending on the chosen optimization goal. In the simplest way these messages are used to discover the system structure and the distance between the neighbors. The thereby discovered system view can also be resent to other neighbors, so that the total view of the system grows gradually. If a threshold of an associative counter array is exceeded, the optimization algorithm is called, which then calculates an optimization advice. An optimization advice contains a list of possibly exchanged memory segments. This advice is sent to the involved nodes which evaluate the advice and agree or reject the advice. As the last step the actual optimization, normally an exchange of memory segments, is executed.

VI. Evaluation

In this paper we present first results of the optimization cycle on the Tilera system, deploying as algorithms a locality optimization and a load compensation. Two scenarios are shown in Figure 9. In the following tables the node number correspond to the ones in these figures. During the evaluation we encountered some bigger problems with the Tilera system, which froze the total system while using the implementation on big scenarios or big memory segment sizes. This behavior could be isolated as a possible problem in the operating system.



(a) Locality Optimization (b) Load Compensation

Fig. 9. Evaluation Scenarios

Due to this problem in the following we present first results using a reduced system of 3×3 tiles and application scenarios with reduced segment sizes and run-time. Using patched operating system versions, results of additional evaluations with increasing sizes and scenarios will be subsequently provided.

A. *Locality Optimization*

As a first evaluation the locality optimization as optimization algorithm is presented. With this, memory segments are moved to minimize access times to different memory components. A scenario for that can be seen in Figure 9(a) in which arrows point to the initial suboptimal memory assignment.

In our implementation the threshold overflow of the associative counter array, counting the access rate to a segment, triggers the optimization algorithm. The algorithms determines if another known memory node is more narrow to the accessing CPU node. Following this, an advice is generated and sent to the possible participants. After a positive answer the segment is directly exchanged.

TABLE I. Results of the locality optimization, memory nodes

Optimization	activated				deactivated			
Tiles	4	9	11	16	4	9	11	16
Messages out	27512	30156	26886	26343	27623	27623	27623	27623
Words out	55271	60514	54026	52911	55263	55263	55263	55263
I/O segments	3/4	4/2	3/4	4/4	0/0	0/0	0/0	0/0

Tables I and II present the results of the evaluation with and without optimization for memory and CPU nodes. For this quicksort benchmarks are used, concurrently accessing 17 memory segments. The monitoring sends out a status message every 20 ms and uses a radius of 2 for broadcasting these messages to its neighbors.

TABLE II. Results of the locality optimization, CPU nodes

Optimization	activated				deactivated			
Tiles	3	5	15	17	3	5	15	17
Duration (10^6 cycles)	154	154	155	152	129	130	131	130
Moved Segments	4	2	4	4	0	0	0	0
Access time (cycles)	5249	5260	5298	5188	4287	4303	4308	4348

The higher values with activated optimization are due to the periodical exchange of status messages. As previously mentioned, the memory nodes cannot process these status messages and memory accesses at the same time in our implementation. This overhead can be reduced in adapting the exchange frequency. As can be seen in Table III, most of the additional costs are caused by the status messages and the transfer of the moved segments. The slightly changed values compared to Table I are due a different evaluation run. With the operating system problems we had to reduce the size of the transfered data segments. Without that, the transfer time is increased while the number of status messages is kept constant. But as can be seen in Table II the access to the new location is faster after the optimization. So with more realistic application scenario sizes and without the given limitations of the Tilera system, the optimization would be profitable and outbalance the additional management costs.

TABLE III. Message in the stages of the optimization cycle

Tiles	4	9	11	16
regular	15129	53901	37259	4173
busy	0	8	2	0
moved	2	2	3	3
transfer	20	20	21	21
suggestion	2	2	3	3
answer	1	6	3	0
status	70	110	70	110
Messages out	15254	54049	37361	4310
Words out	30732	108269	74951	8814
I/O segments	1/2	6/2	3/3	0/3

B. *Load Compensation*

As second presented evaluation, a load compensation optimization was evaluated. Segments of highly occupied memory nodes are moved to less charged ones. A scenario for that can be seen in Figure 9(b) in which arrows also point to the initial suboptimal memory assignment. Here the segments of 3 CPU nodes are initially assigned to one single memory node.

TABLE IV. Results of the load compensation, memory nodes

Optimization	activated				deactivated			
Tiles	4	9	11	16	4	9	11	16
Msg. out	33459	34596	33389	37246	82869	14280	0	40966
Words out	67911	69990	67314	75305	165789	28569	0	81957
I/IO segments	5/9	7/8	9/4	7/7	0/0	0/0	0/0	0/0

As in the first example, a threshold overflow of the associative counter array is used to trigger the optimization algorithm which sends out an optimization advice. In this example most of the work of the optimization algorithm is done to evaluate the advice by the participants. The algorithms therefor compares the load of its own node and the load of the initiator. In case of a higher load an exchange is done.

TABLE V. Results of the load compensation, CPU nodes

Optimization	activated					deactivated				
Tiles	3	5	10	15	17	3	5	10	15	17
Duration (10^6)	222	223	224	190	215	209	209	209	139	138
Moved Segments	3	7	4	3	11	0	0	0	0	0
Access time	7456	7440	7512	6395	7317	6984	6974	6970	4558	4581

Duration & Access time in cycles

Tables IV and V present the results of the evaluation with and without optimization for memory and CPU nodes. During the evaluation the monitoring sends out a status message every 40 ms and also uses a radius of 2 for broadcasting these message to its neighbors.

TABLE VI. Message in the stages of the optimization cycle

Tiles	4	9	11	16
regular	29799	34554	36428	37334
busy	0	6	3	7
moved	9	6	7	5
transfer	72	42	61	41
suggestion	12	15	15	13
answer	14	14	13	14
status	42	66	42	66
msg. out	29948	34703	36569	37480
words out	60891	69920	73942	75482
segm I/O	4/9	8/6	7/7	8/5

As can be seen in these two tables, the CPU cycle count for the execution is more balanced with activated optimization. Instead of even unused nodes in the execution with deactivated optimization, the number of messages on memory node side are good balanced.

The values in Table VI show, just as in the first evaluation, that most of the additional costs are caused by the fine-granular transfer of the moved segments. But the optimization goal of good balanced memory loads could be achieved nevertheless. So again as in the previous example in scaling the scenarios and without the given limitations of the Tilera system, the optimization would be more profitable.

VII. Conclusion and Outlook

This paper presents evaluation of the memory management of existing manycore systems. In more detail the architecture and usage of the Tilera TILE-Gx platform was addressed. First we measured the existing memory management of the available Tilera platform, second we adapted the the concept of the autonomous self-optimizing memory architecture Self-aware Memory to it.

Our measurements on the Tilera TILE-Gx system pointed out some weak spots in current manycore architectures. As expected, the measured access times to external connected memory modules strongly depend on the position of the tile in the grid. This holds true for access to private and shared memory. The access times distinguish themselves from e.g. a minimum of 13,5 % for the farthest compared to the nearest tile accessing private memory, when the whole system and network can be exclusively used by a single core. In reality, this will never be the case in using manycore systems, even less with the promised high-dynamic application scenarios. Moreover, scaling up the number of cores will significantly rise the slowdown due to mutual interference using the same network or in accessing the same memory component.

With the adaptation and implementation of the Self-aware Memory concept to the Tilera system, we demonstrated that high-dynamic and adaptive memory management techniques are feasible in combination with manycore architectures. Modifications in the system structure and memory management of the existing hardware system were not possible for the evaluations. Therefore we implemented SaM as a software abstraction layer running on top of the existing memory management and system structure. The results of the evaluation showed that adaptive memory management techniques can be realized without much additional messages, in return achieving higher flexibility and and simple usage of memory in future system architectures. To take advantage of these mechanisms, the results and experiences have to be adapted to future manycore architectures and directly integrated in the hardware. As presented, current systems often contain multiple different networks. Moving the monitoring and management messages to an additional or currently unused network, would separate the ubiquitous management from the data transfers and make the optimization profitable furthermore.

In this paper we presented first results with reduced scenario sizes, due to the repeated operating system crashes on the Tilera system. Currently, we work on fixing these problems. Additional evaluations, scaling the number of tiles, applying bigger application scenarios and increased test run-times are the next steps on our agenda.

To further improve the results of the optimization process, aspects from machine learning in combination with program phases could be examined in the future. Adapting and evaluating the presented mechanisms to new and upcoming memory connections like 3D-stacked memory or optical connections, accompanied with a change in the system structure, will be another challenging step.

References

[1] J. Duato, "Beyond the power and memory walls: The role of HyperTransport in future system architectures," in *First International Workshop on HyperTransport Research and Applications (WHTRA)*, February 2009.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[3] R. Buchty, O. Mattes, and W. Karl, "Self-aware Memory: Managing Distributed Memory in an Autonomous Multi-master Environment," in *Architecture of Computing Systems – ARCS 2008*, ser. Lecture Notes in Computer Science, vol. 4934, February 2008, pp. 98–113.

[4] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," *Micro, IEEE*, vol. 22, no. 2, pp. 25–35, 2002.

[5] M. Gries, U. Hoffmann, M. Konow, and M. Riepen, "SCC: A Flexible Architecture for Many-Core Platform Research," *Computing in Science Engineering*, vol. 13, no. 6, pp. 79–83, 2011.

[6] T. Corporation, "TILE-Gx Processor Specification Brief," May 2012.

[7] Tilera Corporation, *Architecture Overview for the TILE-Gx Series, UG130*, San Jose, CA 95134 USA, 2012.

[8] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, "RAMP Blue: A message-passing manycore system in FPGAs," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on.* IEEE, 2007, pp. 54–61.

[9] Kalray, "KALRAYs MPPA (Multi-Purpose Processor Array)," http://www.kalray.eu/products/mppa-manycore/, April 2013.

[10] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin *et al.*, "Larrabee: a many-core x86 architecture for visual computing," in *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3. ACM, 2008, p. 18.

[11] Intel Corporation, "An Update On Our Graphics-related Programs." [Online]. Available: http://blogs.intel.com/technology/2010/05/an_update_on_our_graphics-rela/

[12] O. Mattes, M. Schindewolf, R. Sedler, R. Buchty, and W. Karl, "Efficient Synchronization Techniques in a Decentralized Memory Management System Enabling Shared Memory," ser. PARS Mitteilungen GI, vol. 28, no. ISSN 0177-0454. Rüschlikon, Switzerland: Gesellschaft für Informatik e.V., May 2011.

[13] M. Schindewolf, O. Mattes, and W. Karl, "Thread creation for self-aware parallel systems," in *Facing the Multicore-Challenge*, ser. Lecture Notes in Computer Science, R. Keller, D. Kramer, and J.-P. Weiss, Eds. Springer Berlin / Heidelberg, 2011, vol. 6310, pp. 42–53.

[14] O. Mattes, "An Autonomous Self-Optimizing Memory System for Upcoming Manycore Architectures," in *Proceedings of the First Organic Computing Doctoral Dissertation Colloquium (OC-DDC'13)*, ser. Reports / Technische Berichte - Herausgeber: Fakultät für Angewandte Informatik der Universität Augsburg, no. 2013-06, 2013, pp. 4–7.

[15] O. Mattes and W. Karl, "Self-aware Memory - An Autonomous Self-Optimizing Memory System for Upcoming Manycore Architectures," in *Memory Architecture and Organization Workshop (MeAOW'13)*, ser. Embedded Systems Week 2013, Montreal, Canada, 2013.