Self-aware Memory: Managing Distributed Memory in an Autonomous Multi-Master Environment

Rainer Buchty, Oliver Mattes, Wolfgang Karl

Universität Karlsruhe (TH) – Institut für Technische Informatik (ITEC)
Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung
Zirkel 2, 76131 Karlsruhe, Germany
{buchty|mattes|karl}@ira.uka.de

Abstract. A major problem considering parallel computing is maintaining memory consistency and coherency, and ensuring ownership and access rights. These problems mainly arise from the fact that memory in parallel and distributed systems is still managed locally, e.g. using a combination of shared-bus- and directory-based approaches. As a result, such setups do not scale well with system size and are especially unsuitable for systems where such centralized management instances cannot or must not be employed. As a potential solution to this problem we present SaM, the Self-aware Memory architecture. By using self-awareness, our approach provides a novel memory architecture concept targeting multi-master systems with special focus on autonomic, self-managing systems. Unlike previous attempts, the approach delivers a holistic, yet scalable and cost-friendly solution to several memory-related problems including maintaining coherency, consistency, and access rights.

1 Introduction and Motivation

An increasing problem in parallel, distributed system is how to assign and manage memory resources. Traditionally, this is done through a layered approach where local memory is managed per node. Locally, memory is typically attached using a shared bus, where appropriate coherency protocols (such as MESI) are applied; above node level, directory-based methods are employed to enable coherency and consistency. Virtualization of the memory subsystem, i.e. forming a system-wide, eventually shared, distributed memory resource from the individual local memory entities, and securing access rights in such systems require further assistance, typically realized through additional, OS-assisting software layers, or underlying virtualization and abstraction layers.

This strategy becomes increasingly performance-hampering: bus-based methods are hardly applicable beyond dual-core systems as they require bus systems running at a multiple of the required access speed, hitting technology boundaries. Likewise, in directory-based systems the directory itself and its connection, i.e. network, speed, and latency, become a bottleneck.

With the increasing distribution of multicore and upcoming manycore architectures, we already observe a shift in computer architecture: instead of traditional bus-based approaches, these systems feature NoC-based interconnection between the individual cores and memory. Also, not necessarily each processor has explicit local access to an

attached memory. Examples for such architectures are e.g. MIT RAW [20] and the late Intel Polaris being part of Intel's Tera-scale computing program [9].

Our work is motivated by a special kind of NoC-connected SoC architecture, the Digital On-demand Computing Architecture for Real-time Systems (DodOrg) [2]. DodOrg comprises a heterogeneous grid of processing elements, memory, and I/O cells. The processing cells can be e.g. standard CPU, DSP, or programmable logic (FPGA) cells [16]. It was especially designed as an architecture employing techniques derived from so-called organic or autonomic computing. Hence, no central management instance is used, likewise no supervising distributed OS can be used. Instead, a lightweight control loop for autonomous task-to-cell assignment and system surveillance is employed [3, 4].

A memory management system fitting into such a system therefore also has to act autonomously and without external assist by e.g. OS and higher software layers. Since DodOrg was designed with focus on embedded real-time systems, this must not induce significantly raised hardware costs.

The aforementioned memory management problems arise from the fact that also in distributed systems memory is typically exclusively assigned and physically connected to a single processor, becoming the "housekeeper" of its assigned entity. Combination of these individual memory entities into a global shared memory resource requires the described multi-layer approach to ensure consistency, coherency, and uphold access rights. Depending on the target platform, such an approach is, however, not applicable.

In our paper, we describe a novel memory architecture concept, dubbed Self-aware Memory (SaM), targeted at distributed multi-master systems. Unlike previous attempts, the approach delivers a holistic, yet scalable and cost-friendly solution to several memory-related problems including coherency, consistency, access rights. This is achieved by employing so-called self-awareness so that the memory subsystem essentially becomes self-managing. As a beneficial side-effect, this also reduces complexity on hardware and software level as previously required instances for ensuring consistency, coherency, and access rights are no longer necessary in SaM.

This paper is structured as follows: we will first present related work in Section 2, where we shortly discuss their benefits and drawbacks. To further motivate our work, in Section 3 we provide a short introduction into an application scenario, our Digital on-Demand Computing Architecture for Real-time Systems (DodOrg), a so-called organic computing architecture which inspired the development of Self-aware Memory. In Section 4 we will present our SaM architecture concept, architecture implications, and show how such a setup matches the architecture requirements parallel systems in general, and specifically of architectures like DodOrg. The existing prototype, current work, and initial results derived from the prototype are shown in Section 5. The paper is concluded with Section 6.

2 Related Work

In the past multiple concepts for a different usage of the memory in a system were explored. With Intelligent RAM (IRAM) [13], Processing in Memory (PIM) [17], Parallel Processing In Memory (PPIM) [15] and some other related projects, computation

of simple instructions is sourced out into small processing elements integrated in the memory modules. FlexRAM [10] is another PIM-based architecture; it features a programming concept called CFlex [6]. All these approaches share the same concept, i.e. offloading computation into memory and therefore saving expensive transfer time from memory to processor and back. Although these concepts are coined *intelligent memory*, this solely reflects the processing "intelligence". These concepts are all based on a static architecture and do not expose any flexible or autonomous behavior.

Active Pages [12] are another concept for relocating processing of instructions to the memory. In contrast to the aforementioned approaches, Active Pages are based on so-called RADram (Reconfigurable Architecture DRAM) which means, that the logic functions integrated in the memory can be changed during the execution. This gives the possibility to specifically adjust the logic to the requirements of an executed program. The system is flexible, so the same hardware can be used for more varying systems leading to lower costs. Another advantage is that Active Pages integrates in normal systems by using the same interface as conventional memory systems, hence, it is not replacing conventional architectures. Active Page functions are invoked through memory-mapped writes. Synchronization is accomplished through user-defined memory locations.

A special kind of connecting memory to a system of several processors exists in the *parallel sysplex architecture* of IBM mainframe computers. The so-called *coupling facility* [14, 7] is a central memory concurrently used from all subsystems. It ensures the integrity and consistency of the data. This is achieved through a special processor with attached memory, which is connected to all processing elements of the sysplex configuration.

In contrast to this different projects, we introduce *Self-aware Memory* (SaM) – a memory system with autonomous, intelligent, self-aware and self-managing memory components. SaM features an easy to use memory concept in a shared memory system which adopts the four basic Self-X principles of Autonomic Computing [8]. By handing over memory control from the operating system to the self-managing and software-independent hardware, SaM is fully transparent to the programmer, i.e. no dedicated program support is required and is extraordinarily well-suited for parallel and distributed systems. SaM therefore lays the foundation for scalable, flexible on-demand computing systems.

3 The DodOrg Hardware Architecture

SaM was inspired by the requirements of the Digital On-demand Computing Architecture for Real-time Systems (DodOrg). DodOrg comprises a grid of individual nodes called Organic Processing Cells (OPCs) featuring peer-to-peer (P2P) connection of the cells [2]. Using a biologically inspired method, these cells individually announce their suitability for processing tasks leading to a decentralized, flexible, and fault-tolerant task distribution. As an effect of that method, closely collaborating tasks typically are executed on neighboring cells leading to the formation of so-called organs, i.e. clusters of cells performing a meta-function.

Because of the dynamic behavior of this task allocation and organ formation, traditional memory management techniques are not applicable. Instead, a similarly flexible

approach – SaM – was required, which does not only account for the specific DodOrg requirements but also integrates nicely into the hardware and communication infrastructure offered by the DodOrg architecture.

4 Self-aware Memory (SaM)

Key problem of shared memory in parallel, distributed systems is its construction from several local memory entities, requiring several layers of management ranging to enable coherence and consistency, and enforcing access rights. Hence, in SaM local memory no longer exists. Instead, the memory is turned into an active, self-managing component interconnected through a network infrastructure spanning on-chip and off-chip communication.

In traditional systems memory management, consistency, and coherency are maintained by dedicated units embedded into a CPU's memory management unit, assisted by additional software layers. Within the focus of SaM, self-managing means, that these issues are handled by the memory itself.

This is already very beneficial for parallel and distributed systems, because now the management is no longer done by the individual processing units. No additional hardware or software overhead is required on a processor's side. Hence, scalability of a system is effectively decoupled from the number of processors and solely dependent on the capabilities of the used interconnection network and memory entities.

As a side effect of the self-managing aspect, the entire memory subsystem is effectively abstracted and treated as a single virtual memory entity, i.e. a uniform memory view is achieved and individual parameters or access methods of the attached memory are hidden. Because of the self-managing aspect, it is furthermore possible, to alter this memory entity by adding, removing, or replacing individual memory units. Hence, in contrast to similar concepts as outlined in Section 2, SaM is entirely transparent to the programmer, i.e. no dedicated programming technique such as e.g. with FlexRAM [1] is required.

Such behavior is eminent for the creation of dynamic, flexible systems ranging from traditional fault-tolerant systems to currently researched autonomous and organic systems like the aforementioned DodOrg architecture. With the abstraction of the memory hardware, the programmer has only to provide the desired memory capacities and capabilities such as size and access speed. Likewise, the OS only needs to support respective calls, translating the program calls to appropriate SaM protocol messages.

The implications of the SaM concept are that private local memory no longer exists, but only a globally shared memory resource. Hence, a processor does not have private access to an associated local memory as it is the case in traditional computer systems and prior to use, any memory (besides a required bootstrap memory) must be allocated first from the memory subsystem.

Given this basic introduction into SaM, we will in the following describe the design considerations, anatomy, and use of a SaM-based system, and demonstrate how SaM integrates into the DodOrg architecture framework.

4.1 Architectural Considerations

Driving force behind the design of our SaM concept was to provide a scalable, flexible infrastructure at minimal costs and software overheat – not only within the scope of our DodOrg project, but also for current and future parallel & distributed systems. In the following, we'd therefore like to address certain design aspects and their outcome with respect to system design.

Communication and Scalability Memory allocation within SaM requires at least three steps: first, the processor is sending out its request where it specifies size and access parameters. The memory subsystem will then answer with an according offer, leading to one or more responses depending of size and current usage of the memory subsystem. From this choice, a best-fit selection is made and the associated memory region is selected and subsequently gets assigned to the requester.

To save bandwidth and communication efforts, we propose different scenarios for such a request and its appropriate answer: for a first-time request, the request is sent out as a broadcast to the network. Associated time-to-live or number-of-hops can be used to limit the broadcast's range. From the memory subsystem then only positive answers are sent back. A timeout mechanism ensures that this request phase will always terminate, even when no positive answer is received, e.g. in case requested access times and/or latency cannot be met.

Such an allocation process can be viewed best as a brokering system where a processor advertises its requirements and gets one or more offers from the memory subsystem. The processor then evaluates these offers and acknowledges fitting ones; finalizing, the memory subsystem grants these acknowledged memory regions and also stores ownership and access rights. CPU-wise, the only information stored is the translation table, i.e. which (virtual) memory addresses map what memory entities and addresses within those entities.

It is also possible to send direct requests to individual memory entities. This may be used in case when it is already known that a specific entity can provide desired resources, so that the initial broadcast and brokering phase is not required. Freeing memory or updating rights and ownership will typically use direct communication with the affected memory entities.

In addition, a multicast scheme – one message addressing several entities – and the aforementioned servicing, i.e. sending out the message only to the service node which then distributes it to its sub-nodes, are possible.

To further improve scalability and reduce communication, SaM specifically supports servicing: for instance, in a tree-like memory structure the root node will represent the entire tree and act as the sole communication partner between the requesting processor element and the underlying memory hierarchy.

The SaM approach therefore scales well with the overall communication capacity within a parallel computer system. Because the memory subsystem is fully decoupled from the system's processing elements, it is especially well suited for systems where processing capabilities are dynamically added, removed, or replaced. Using a metrics-based classification and a unified memory protocol provides an abstract, uniform view of the memory subsystem.

Processor Impact Although – by design of SaM – a processor does know neither size nor structure of the attached SaM memory subsystem, memory allocation and rights management works similar to conventional memory: the processor needs to allocate a memory region of desired size and assign desired access parameters (code or data segment, access rights). In conventional systems this is typically handled by OS functions and, on hardware level, assisted by the processor's MMU.

Within SaM, the task of memory allocation and management becomes a function of the memory itself. To support this allocation process, we define a set of processor instructions supporting the new allocation and management process such as memory allocation and deallocation, or rights assertion. These solely interact with the SaM-Requester, anything else is handled transparently within the SaM infrastructure. In conventional systems, similar processor support exists, e.g. using machine status registers and supervisor-level instructions.

For a minimal setup, at least two instructions for allocating and freeing memory regions are required, possibly complemented by an additional instruction supporting resizing and re-allocation. To enable a shared memory system, another instruction for later access right change is required. Additional instructions providing additional guidance to the memory subsystem might be introduced, but are not mandatory.

These instructions carry required parameters such as size and mandatory access speed for a requested memory region in case of memory allocation.

4.2 Composition of a SaM-enabled System

With SaM, the memory-modules are no longer directly associated to a single processor but are part of a network. This network may be an exclusive memory network or embedded into existing connection resources, forming a virtual memory network.

To achieve this, every memory entity is connected to a component called *SaM-Memory* managing the connected memory and serving as an interface to the memory network. Several different memories may coexist in the system, each of them connected through an own SaM-Memory component to the network.

Processors (or any other memory-accessing entity) are likewise attached to the memory network through a similar component called *SaM-Requester*. SaM-Requester manages the memory accesses of a processor and handles the overall access protocol of SaM. It also generates request messages to the memory subsystem and processes the answers from the affected *SaM-Memory* components.

Both, SaM-Memory and SaM-Requester, employ dedicated units called *SaM-Table* for storing internal information regarding memory allocation and ownership, and hold information about their corresponding address space. A SaM-Table therefore is basically a list of entries for address translation plus additional housekeeping data for memory management and access security.

The network used for communication within the SaM infrastructure is not specified; this was done deliberately to not restrict the usage of SaM to a special type of network. While this leaves the most freedom for the implementer, of course performance issues must be taken into account so that no performance bottleneck arises from choosing the wrong network infrastructure.

An exemplary setup of a SaM-enabled system consisting of two processors and three memory entities is provided in Figure 1. In the following subsections we will present the individual components, their integration into existing systems, and how memory access in a SaM-enabled system takes place.

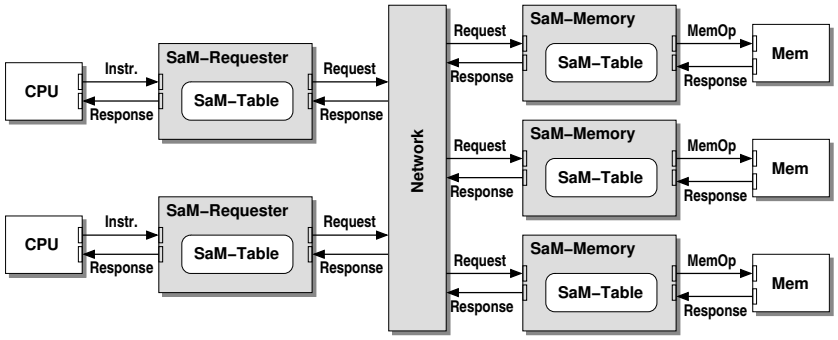

Fig. 1. Structure of SaM

4.3 SaM-Table

As mentioned before, the SaM-Table component is used to store the management data for individual memory segments accessed via SaM-Requester and SaM-Memory. Per used (i.e. allocated and assigned) memory segment, it contains an entry consisting of several parameters which are used to associate memory regions and access rights.

In our current setup, each SaM-Table entry consists of ID, Segment, Address, Length, Security, and Usage. Depending on whether the SaM-Table is used in a SaM-Requester or a SaM-Memory, the parameters have slightly different meanings, but for ease of development the overall structure remains identical.

For *SaM-Memory*, the ID field contains the network address of the owner, i.e. the originally requesting processor or, more precisely, its associated SaM-Requester. The Segment field contains an assigned identifier value; the Address and Length fields store start address and length of the assigned memory region, whereas the Security field stores the requested access rights such as e.g. exclusive or shared access, read-only or read/write access. The Usage field denotes whether an entry is valid or not.

When used within a *SaM-Requester*, the ID field contains the network address of the assigned memory entity (SaM-Memory). The Segment field will hold the aforementioned identifier value which is returned during the memory allocation process. In the Address field, the start address of the assigned memory segment in the processor's logical address space is saved. Length and Access right fields stay identical.

Take notice of the fact that the effective physical memory address is neither stored in the SaM-Table of the SaM-Requester. The actual place of the segments are defined by the network address of the SaM-Memory and the segment number. For the communication between SaM-Requester and SaM-Memory, solely the addresses of the underlying network are used. The translation between these different address spaces is done by the SaM-Requester and SaM-Memory with the data of the SaM-Tables.

4.4 SaM-Requester

From a processor's point of view the whole memory is abstracted and perceived as a single memory entity; neither size of existing memory resources nor the actual hardware structure of the memory subsystem is visible to the processor. To achieve that, every processor has its own logical address space which is managed by the *SaM-Requester*.

This unit basically performs address translation from local, logical address space of the connected processor into the distributed SaM memory space, i.e. which chunk of of the local address space maps to what physical memory location. Hence, SaM-Requester may be considered a (partial) replacement for a CPU's memory management unit. The management data is stored within the SaM-Requester's associated SaM-Table.

For allocating new memory space, it generates a request and sends it to the network. After that, it processes the received answers and reacts on them in a way which is defined in its implemented algorithms. For managing the logical address space and the allocated segments it uses a SaM-Table to store the data in it. In different implementations there could be various algorithms to manage the SaM-Table and optimize the usage of the address-space [19]. Which one of them is the best often depends on the underlying structure of the system.

Since this allocation process takes a certain time, a defined time window is opened during which answers to an allocation request are accepted. Upon positive answer by the memory subsystem, SaM-Requester will assign the offered memory region to a new segment of the processor's logical address space. If no suitable memory area is offered or if the request times out, this is signalled to the processor which then enters an appropriate handler.

4.5 SaM-Memory

SaM-Memory is the vital component for creating an abstract and uniform memory access regardless of the attached memory's type or access parameters and therefore providing additional memory management functionality. This is done by actively managing the attached memory, hence, SaM-Memory requires detailed knowledge about that memory, like size, usage, access parameters, and physical condition. To keep track of these parameters, monitoring capabilities are required.

SaM-Memory answers incoming requests generated by a SaM-Requester on the basis of the current memory status as stored in the associated SaM-Table. If an incoming allocation request can be fulfilled, SaM-Memory sends back a positive answer and assigns the new segment to the requesting processor's corresponding SaM-Requester. Likewise, read/write accesses, access right updates, or deallocation are processed. Typically, SaM-Memory will not respond to requests it cannot fulfill; however, this behavior is configurable so that it e.g. may offer insufficient resources for an incoming request, or sending out negative answers to requests instead of leaving them unanswered.

For a more detailed description of memory access we would like to refer the reader to the following section.

4.6 Memory Access in SaM

As said before, the SaM memory infrastructure may well consist of a variety of individual memory entities scattered among one or more memory interconnection networks. Naturally, these individual entities expose different size and access parameters, such as access time and latency. Hence, memory allocation requires additional guidance to account for the described memory subsystem heterogeneity. For memory allocation therefore not only the requested amount of memory must be provided, but also additional requirements such as maximum tolerated latency, or minimal block-size of individual memory chunks.

This request is directed to the memory subsystem, where it is evaluated and answered appropriately. This answer is then processed by the requesting entity and acknowledged accordingly so that the memory subsystem will then reserve a selected memory region and apply the specified access rights, so that for subsequent memory accesses an automatic access right checking can be performed and only rightful accesses take place.

In a system containing more than one memory units, a single request will therefore result in several answers. From these answers a best-fitting choice is selected which may lead to a requested memory size being assembled from individual offered chunks.

This memory layout is then stored on the requester's side and acknowledged to the offering memory entities, which in term mark the corresponding memory regions as used, and assign ownership and access rights accordingly.

Subsequent memory accesses then directly take place between the accessing entity and the addressed memory. If allocated memory is no longer used, it can be freed by the owner, the corresponding memory unit(s) will then unlock the affected memory regions so that they can be offered to further allocation requests.

Supervision of Memory Accesses One big advantage of SaM is the implicit enforcement of memory access rights. On each memory access, the addresses will be checked two times: first, a processor's request is checked by its associated SaM-Requester. This unit will already reject accesses which do not comply with data stored in its SaM-Table, e.g. in case the memory is not allocated at all or if accesses beyond an allocated segment occur. Hence, it is ensured that spurious, maybe malicious accesses of a single processor are already stopped at the source and do not even enter the memory network.

A second check takes place at the addressed memory unit where the access is matched against boundaries of the accessed segment, ownership, and access rights stored in the corresponding SaM-Memory's SaM-Table. This ensures that only rightful accesses are actually performed.

So far, this does not protect against devices which generate malicious access messages; because of the high level of abstraction, SaM can be easily enhanced by additional cryptographic methods such as hashing and signing to enable proper identification and authorization of incoming requests. Because this extra functionality will be embedded into SaM-Memory and SaM-Requester, no software overhead is required.

Shared Memory and Rights Management To enable shared memory in parallel systems, fine-grained rights management is required to control the memory accesses. Traditionally, this is done by the combination of software handlers (shared access over a network) and hardware (MMU managing local memory accesses). While this scheme can be used with SaM, it is unnecessarily complex and does not make use of the specific features of the SaM concept.

As mentioned before, within SaM a processor first allocates desired memory regions through its associated SaM-Requester. If that memory region is to be shared with other processors, then the Security fields of the affected memory units, i.e. their corresponding SaM-Memory component, have to be adjusted.

As of now, the simple case was implemented where a memory is either private to a single processor or completely shared. To fully accommodate for the requirements of parallel systems, however, a more fine-grained method is required where individual access rights can be assigned to each processor and memory region. We are currently evaluating the options, how these increased functionality will be implemented into SaM.

4.7 DodOrg Integration

The SaM extensions, i.e. SaM-Requester and SaM-Memory with their according SaM-Tables, will be unique to the cell-specific part as they are only required for memory cells and cells with memory-access capabilities such as processor or DSP cells.

SaM does not require any alteration of the cell-uniform communication infrastructure of DodOrg cells. Virtual links, already provided by that infrastructure, greatly enhance SaM's performance as they enable direct communication between corresponding SaM-Requesters and Memories over the P2P network at guaranteed communication times.

5 Results

To show the basic functionality of SaM, we designed and implemented a prototype using the United Simulation Environment (UNISIM) [5]. UNISIM consists of a modular simulator and a library of predefined modules. Using these predefined and additional user-programmed modules, a simulator framework is programmed which is then compiled into an executable simulation application by the UNISIM compiler.

The simulated hardware is divided into several components to ease development of complex simulation engines, hence, each implemented by an own module with well-defined interfaces enabling to simply replace modules or make further use of them in other simulations.

5.1 Prototype Implementation

Our prototype basically resembles the system layout shown in Figure 1 consisting of processor and memory cells connected through a network. The prototype therefore also reflects the basic structure of the DodOrg architecture, i.e. the connection of cells using a communication infrastructure. The P2P nature of DodOrg's communication infrastructure is not visible to the running application and abstracted on hardware level,

hence, restricting our SaM prototype to a globally shared communication resource is a valid simplification for evaluating our SaM concept.

As a processor core we chose the DLX core which is provided together with UNISIM. This core was extended in two ways: first, the processor model was extended by dedicated functions for memory allocation and freeing. Second, the SaM requester component was implemented, providing the required interfacing to the communication infrastructure, most notably the memory allocation and access protocol. Likewise, we extended a UNISIM-provided memory model by the SaM memory component.

All SaM components are interconnected via a simple full-duplex communication model. The prototype offers the possibility to simply test and typical conflicts of concurrent memory accesses from different cores. The progress of requests and accesses of segments can be visualized and the limitation of the network can be made visible.

The prototype is set up in a parameterizable and scalable way, so that any number of processor and memory cores can be generated.

5.2 Simulation Setup and Results

Aim of the described prototype is to properly evaluate the basic concept, i.e. to not only demonstrate the basic functionality, but also determine the overhead introduced by the new memory protocol and also addressing the question of real-time behavior. We therefore concentrate on the aspects of memory access, i.e. the eventual amount of overhead introduced by allocation and read/write accesses.

In order to test different scenarios of memory accesses, we developed a set of test programs as shown in Table 1, each demonstrating certain features and testing the behavior of SaM. Because of the additional, SaM-specific commands used for memory allocation, the test programs were written in assembly language and translated into their binary representation using a patched DLX assembler.

Depending on the test scenario (function test vs. performance evaluation), these test programs were either executed on a single or all processors of the simulation model and the simulation output was logged and analyzed, proving the theoretically predicted behavior.

Only during the allocation process exists an overhead which directly corresponds with the used allocation procedure. Two basic approaches exist, which are round-robin and broadcast. For our prototype, we implemented the round-robin allocation procedure and were able to observe the expected behavior: in best case, the first addressed SaM-Memory will be able to fulfill the request and sending back an acknowledgement

Test Program	**Performed Test**
`request.s`	Basic request to two memory segments
`read_write.s`	Simple read and write to and from a memory segment
`mult_req.s`	Multiple requests to different memory components
`req_eject1.s`	Memory request rejected without reaction of the program
`req_eject2.s`	Abort of program after rejected memory request
`inv_access.s`	Invalid memory accesses
`inv_br_acc.s`	Invalid memory access due to wrongly calculated branch address

Table 1. Memory-related test programs performed on the prototype

message, hence, no subsequent requests are sent. This scenario is comparable to memory allocation within a processor's local memory where a request either succeeds or fails.

If the first addressed memory cannot fulfill the request, the next in the queue of the round-robin mechanism is used. For every try, another message has to be sent to the memory and back to the requesting CPU. In worst case, the allocation request therefore needs to be subsequently re-sent to all other memory cells until the allocation either succeeds or fails. Hence, if n memory entities exist, it takes a maximum amount of time of $2 * n * \lambda_{net}$ for the allocation process to finish, where λ_{net} is the time for sending a message over the network. After that the request is rejected. In general, a request can be completed in $i * 2 * \lambda_{net}$, where $1 \leq i \leq m$ is the first memory which can fulfill the request. Simulation showed this expected behavior.

Therefore, request time is limited by the time for the transfer of the request over the network and the number of components. Bandwidth problems might arise when multiple users access a shared communication infrastructure, e.g. a shared bus. To account for that, the used communication infrastructure was a simple shared bus model. If no other component wants to access the bus, it can be used directly, otherwise the bus can be accessed by the components in a round-robin order.

Under most pessimistic assumptions, i.e. a fully occupied bus, the maximum waiting time for an individual access is $2 * k * \lambda_{net}$, with $0 \leq k < n$ being the number of components with pending bus requests and having higher access priority, i.e. their requests being performed before the actual request.

This leads to a worst-case time of $(2 * n * \lambda_{net}) * m$ for an allocation request to finish – as also attested by the simulation infrastructure – in case the bus is fully used with $k = n - 1$ pending requests being executed prior to the current request, and none or the very last memory element, i.e. $i = m$, in the round-robin queue being able to fulfill the request. This extreme case, however, should not occur very often in practice as it would indicate a severe mismatch between required and provided bus bandwidth.

Once, memory is already assigned, subsequent accesses do not impose any overhead, because the read/write message can be directly sent to the corresponding memory component. To find the memory assigned to a given memory section, the SaM-Requester performs a lookup in the corresponding SaM-Table; likewise, the addressed memory will match the request against its own SaM-Table to ensure access rights. This will not contribute significantly to access latency, as such mapping and checking is conventionally done by the CPU's memory management unit.

No external overhead takes place, hence, like in conventional systems, data access time is solely limited by the communication network's bandwidth and the addressed memory's access latency.

The tests performed did not only cover individual allocation and access scenarios, but we also addressed interference issues resulting from several simultaneous or colliding requests, and parallel accesses from different processors. We could demonstrate that not only the spreading or interleaving of allocated segments over different memory components works well, but we furthermore could verify the proper functioning of the SaM protocol with regard to memory allocation and access.

Allocation requests can be accomplished up until no more space is left in the memory subsystem, otherwise a defined error code is sent back to the processor in case the desired amount of memory is not available. Like in conventional approaches, this error code is then processed by the respective application.

Memory accesses may only be performed by the CPU holding appropriate access rights. Neither should accesses to un-allocated memory enter the network, nor should the memory itself answer illegitimate accesses. The first case is handled by the SaM-Requester, which will not forward accesses for which no corresponding entry exists in its SaM-Table. Likewise, SaM memory will match incoming read/write accesses against its SaM-Table to either fulfill this request, or send out a negative acknowledge.

5.3 Simulation Conclusion

The simulation prototype clearly confirmed our theoretical assumptions regarding the SaM protocol overhead, worst-case timing behavior, and proved the overall functionality of the SaM concept.

Overhead only takes place for allocation and deallocation (freeing) of memory. In our prototype, we chose a simple-to-implement Round-Robin allocation procedure where a SaM-Requester will sequentially address all present memory modules with the current allocation request and directly receive the individual module's answer. Once the request is satisfied, no more allocation messages are generated.

The drawback of this method is that it only works on enumerated systems where all memory modules are known such as e.g. on a single DodOrg hardware chip. We are currently investigating broadcast-based methods which do not rely on an upfront enumeration.

Once the allocation process is finished, subsequent memory accesses do not carry additional overhead and are also independent of the chosen allocation strategy; regarding memory read/write accesses, SaM is therefore not introducing any more overhead.

Already this simple prototype shows the beneficial effects of introducing self-managing features into the memory subsystem: any number of memories can be shared by any number of processors without requiring an additional software level to ensure access rights in this distributed system. SaM is completely software- and OS-agnostic and therefore especially suitable for heterogeneous systems employing several, or – in the case of dedicated hardware accelerators – no operating system at all.

We further can show that any individual access to the SaM network has a guaranteed upper bound dictated by the network traversal time and distance (number of hops) between requesting and memory unit. This effect is not SaM-specific but a general effect of any communication. However, this information is vital for further work on the communication protocol as we can safely introduce time-out intervals for allocation and access messages when switching to a more sophisticated communication infrastructure model, and also use such information for answering allocation requests and autonomous optimization processes.

5.4 Current Development

Ongoing development of the simulation prototype targets protocol refinements, extended monitoring capabilities for improved measurements, and presentation and visualization of the simulation process.

The protocol is currently extended beyond simple allocation and access to also include assignment of access rights. With the introduction of more sophisticated monitoring and detailed communication infrastructure models we then will be able to perform detailed simulations of life-like systems where network-induced side-effects such as race situations between different messages might occur.

We then will target the introduction of high-level self-managing aspects such as autonomous memory layout optimization (defragmentation), access optimization through autonomous migration and replication of data, which in term require to address consistency and coherency aspects.

6 Conclusion & Outlook

SaM provides a promising way to deal with memory resources in dynamic parallel systems: within SaM, no central memory management unit is required. Instead, the memory itself manages allocation, ownership, and access rights easing the construction of scalable parallel computing systems. Single memory entities therefore are treated in a uniform way, regardless of their type: any memory entity is only classified by its core parameters such as memory capacity, access time, and access latency. The used communication protocol was specifically designed with respect to scalability and minimal communication overhead.

So far, we successfully implemented memory allocation and access rights enforcement. Ongoing and future work specifically addresses the exploration of further self-organizing features such as autonomous de-fragmentation, swapping, or reaction to changed memory capacities and access parameters.

With a refined simulation prototype, more detailed simulations of network infrastructures systems, also including cascading structures, will be possible. The memory components may be connected to more than one network and build a hierarchical structure. With autonomous extension of services – comparable to the self-extending services in JINI [11, 18] – services of not directly connected components could be used by the processors.

References

1. B. Fraguela and J. Renau and P. Feautrier and D. Padua and J. Torrellas. Programming the FlexRAM parallel intelligent memory system. In *Proceedeings of the 2003 ACM SIGPLAN Symposium on Principles of Parallel Programming (PPoPP-03)*, pages 49–60, June 2003.
2. Jürgen Becker, Kurt Brändle, Uwe Brinkschulte, Jörg Henkel, Wolfgang Karl, Thorsten Köster, Michael Wenz, and Heinz Wörn. Digital On-Demand Computing Organism for Real-Time Systems. In Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, and Erik Maehle, editors, *Workshop Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS'06)*, volume P81 of *GI-Edition Lcture Notes in Informatics (LNI)*, pages 230–245, March 2006.

3. Uwe Brinkschulte, Mathias Pacher, and Alexander von Renteln. Towards an Artificial Hormone System for Self-Organizing Real-Time Task Allocation. In *5th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems (SEUS 2007)*, 2007.
4. Rainer Buchty and Wolfgang Karl. A Monitoring Infrastructure for the Digital on-demand Computing Organism (DodOrg). In Hermann de Meer and James P. G. Sterbenz, editors, *Self-Organising Systems: First International Workshop (IWSOS2006), LNCS4124*, page 258. Springer Verlag, Berlin–Heidelberg, Sep 2006. ISBN 3-540-37658-5.
5. European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). UNISIM: UNIted SIMulation Environment.
6. Basilio B. Fraguela, Jose Renau, Paul Feautrier, David Padua, and Josep Torrellas. Programming the flexRAM parallel intelligent memory system. pages 49–60, New York, June 11–13 2003. ACM Press.
7. Wolfram Greis. *Die IBM-Mainframe-Architektur*. Open Source Press, 2005.
8. Paul Horn. Autonomic computing manifesto - ibm's perspective on the state of information technology. October 2001. IBM Research.
9. Intel Corp. Intel Tera-scale Computing. 2007. http://techresearch.intel.com/articles/Tera-Scale/1421.htm.
10. Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD'99)*, pages 192–201, Washington - Brussels - Tokyo, October 1999. IEEE.
11. S. Ilango Kumaran. *JINI technology*. Prentice Hall PTR, 2002.
12. M. Oskin, F. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 192–203, New York, June 27–July 1 1998. ACM Press.
13. David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.
14. David Raften. System-managed cf structure duplexing. *IBM e-server zSeries*, June 2004.
15. K. Rangan, N. Abu-Ghazaleh, and P. Wilsey. A distributed multiple-SIMD processor in memory. In *2001 International Conference on Parallel Processing (ICPP '01*, pages 507–516, Washington - Brussels - Tokyo, September 2001. IEEE.
16. Christian Schuck, Stefan Lamparth, and Jürgen Becker. artNoC - A novel multi-functional router architecture for Organic Computing. In *17th International Conference On Field Programmable Logic and Applications*, August 2007.
17. Thomas Sterling, Jay Brockman, and Ed Upchurch. Analysis and modeling of advanced PIM architecture design tradeoffs. In *SC'2004 Conference CD*, Pittsburgh, PA, November 2004. IEEE/ACM SIGARCH.
18. Inc Sun Microsystems. Jini architectural overview. *Technical White Paper*, 1999.
19. Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2. ed. edition, 2001.
20. Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. In *IEEE Micro*, March/April 2002.