Thread Creation for Self-aware Parallel Systems

Martin Schindewolf, Oliver Mattes, and Wolfgang Karl

Institute of Computer Science & Engineering
KIT – Karlsruhe Institute of Technology
Haid-und-Neu-Straße 7
76131 Karlsruhe, Germany
{schindewolf,mattes,karl}@kit.edu

Abstract. Upcoming computer architectures will be built out of hundreds of heterogeneous components, posing an obstacle for efficient central management of system resources. Thus, distributed management schemes, such as Self-aware Memory, gain importance. The goal of this work is to establish a POSIX-like thread model in a distributed system, to enable a smooth upgrade path for legacy software. Throughout this paper, design and implementation of protocol enhancements of the SaM protocol are outlined. This paper studies the overhead of thread creation and presents first performance numbers.

1 Introduction

The rising integration level in chip manufacturing enables to combine more logic on a single chip every year. By now building multiprocessor systems on chips (MPSoCs) or manycore processors is feasible, as demonstrated by the Intel Polaris or its successor called Intel Singlechip Cloud Computer (SCC) which contains 48 x86-cores in one processor. In the future processor architectures are expected to change from homogeneous to heterogeneous processor designs, consisting of different types of cores on one chip.

Efficient usage of these complex processors largely depends on the availability of thread management, memory management and synchronization mechanisms. Managing resources – such as the memory or the CPU – efficiently, gets more and more complicated with every new processor generation. Up to now the management tasks are commonly handled by the operating system (OS). The OS handles the access of all different user programs to system-level entities. Thus, a centralized management scheme results in the bottleneck of the system. So a universally applicable and scalable method for system management with direct support of heterogeneous parallel systems is required.

In addition to the complex management tasks, the reliability gains importance as a major topic in future systems. Due to the increasing integration level and the complex structures, the probability of hardware failures, during the execution of a program, rises. Executing the operation system on the failing component leads to a breakdown of the whole system although unaffected components could continue to run.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 42–53, 2010.

Therefore we propose a decentral system in which several independent operating system instances are executed on a pool of different processor cores. If one operating system instance fails, other parts of the system proceed. A new way for managing this kind of decentral system is required. Our approach relies on self-managing components by integrating self-x capabilities, especially self-awareness. As an example for such a decentralistic system we approach Self-aware Memory (SaM). Coming along with that, memory accesses are processed by self-managing system components. Especially, one hardware component called SaM-Requester manages its assigned resource – the main processing unit – and handles the logical to physical memory mapping. A more detailed introduction to Self-aware Memory is given in Section 2.

However, disruptive hardware changes which break backwards compatibility are unwanted, as programmers must be able to readapt their software without rewriting. Therefore, we propose a smooth upgrade path for hardware and software. In the proposed design software and hardware operate interactively enabling a legacy code base to be executed. Thus, parallel programs written using the POSIX-thread model – which is a well established parallel programming model for shared memory architectures – continues to run. The goal of our work is to keep the POSIX-compatibility so that only minimal software changes are required. In this paper we present a way of enabling the spawning of multiple threads without a central instance.

In the section 2 we present the background and related work. Section 3 holds the design and implementation, followed by the results and the conclusion in section 4 and 5.

2 Background and Related Work

2.1 Background

Self-aware Memory (SaM) [3], [7] represents a memory architecture, which enables the self-management of system components to build up a loosely coupled decentral system architecture without a central instance. Traditionally, the memory management tasks are handled software-based such as operating system and program libraries assisted by dedicated hardware components e.g. the memory management unit or a cache controller. The main goal of SaM is the development of an autonomous memory subsystem, which increases the reliability and flexibility of the whole system which is crucial in upcoming computer architectures.

First SaM controls memory allocation, access rights and ownership. Hence, a central memory management unit is obsolete and the operating system is effectively relieved from the task of managing memory. The individual memory modules act as independent units, and are no longer directly assigned to a specific processor. Figure 1 depicts the structure of SaM [3].

Due to this concept SaM acts 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 component called SaM-Memory and a part of the

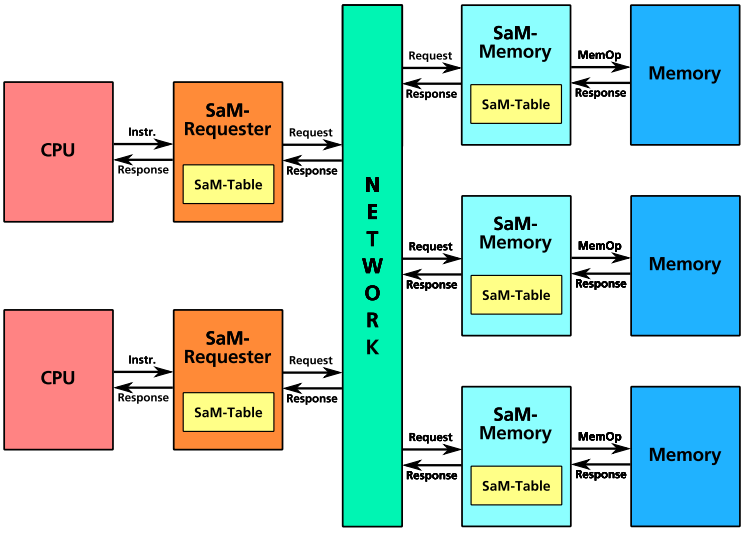

Fig. 1. Structure of SaM

physical memory. The SaM-Memory is responsible for handling the 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 the SaM-Memory, the so called SaM-Requester augments the processor with self-management functionality. Dedicated instructions for memory allocation and management (e.g., free memory, modify access rights) enhance the processor. 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. However, the SaM simulation prototype has been extended with support for basic synchronization constructs. These, also known as atomic instructions, enable lock-based synchronization of parallel threads. However, until now the hardware prototype only ran single-threaded applications, using the SaM protocol mechanisms to allocate/deallocate and access memory. Additionally, the CPU node provides local memory to store the program which is executed from the local memory. Dynamically allocated memory is handled by the SaM mechanisms without operating system support. Until now, the hardware prototype has no operating system at all.

2.2 Related Work

Previous related work proposes Organic Computing as a paradigm for the design of complex systems. Cited from [9]: “The objective of Organic Computing is the technical usage of principles observed in natural systems”. These technical systems interact and are capable of adapting autonomously to a changing environment [10]. The key properties to realize complex systems are the so called

self-x properties [6]. Originally intended to realize autonomic computing systems, these properties, namely self-configuration, self-optimization, self-healing, and self-protection, are also researched in the context of Organic Computing for embedded systems. To achieve self-organization, previous related work addresses task allocation in organic middleware [2]. Task allocation is bio-inspired and relies on artificial hormones to find the best processing element. DodOrg is a practical example for a grid of processing elements organized by organic middleware. In contrast, our approach targets the programmability of distributed systems at the thread level, (instead of the task level) and favors a simplistic selection scheme over a rather complex one.

Rthreads (remote threads) is a framework to allow the distribution of shared variables across a clusters of computers employing software distributed shared memory. Synchronization primitives are derived from the POSIX thread model. The Rthreads implementation utilizes PVM, MPI, or DCE and introduces little overhead [4]. DSM-Threads supports distributed threads on top of POSIX threads employing distributed virtual shared memory [8]. DSM-Threads feature various consistency models and largely rely on a POSIX-compliant operating system. Related work combining Pthreads and distributed systems relies on operating system support combined with a library implementation. However, our proposed design relies on neither of those as an operating system is currently not used. The SaM concept greatly simplifies to build a distributed shared memory system and for the prototype, operating system support is not needed either.

3 Design and Implementation

This section presents design and implementation of the envisioned SaM protocol enhancements. The protocol enhancements over the single-threaded case are three-fold: first, the communication between CPU and SaM-Requester is presented, communication between multiple SaM-Requesters is highlighted and finally, a use case spawning an additional thread involving two SaM-Requesters and one SaM-Memory node is shown.

3.1 Protocol Design

To allow the SaM-Requester to manage the processor as a resource, it needs additional information about the state of the processor. Further, information instruction set architecture, frequency, bus width, caches, special operational modes (e.g. barrel shifter, floating point unit, etc.) of the processor are of importance. In order to reliably answer requests of other SaM-Requesters, CPU and dedicated SaM-Requester have to obey the following rules:

1. The CPU signals the SaM-Requester once it finished its work (idle)
2. The SaM-Requester tracks the current state of the CPU (idle or working)

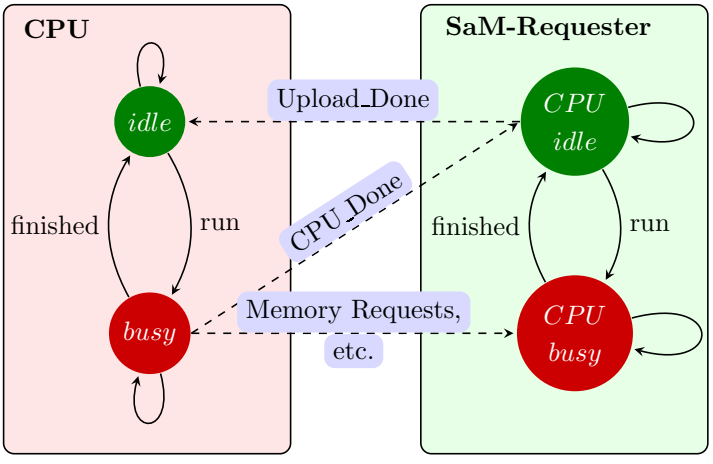

Fig. 2. Schematic protocol overview between SaM-Requester and CPU

3. If the CPU is idle its assigned SaM-Requester competes with other SaM-Requesters for work
4. If the SaM-Requester has been chosen to work, it retrieves the program and signals the CPU to start.

The protocol states and transitions are shown in Figure 2 – messages are marked with dashed arrows. The CPU may either be in an idle state, waiting for a program to arrive or currently executing (busy state). Once the CPU finishes executing a program, a CPU_Done message is sent to the SaM-Requester. By this means the SaM-Requester can track the state of the CPU. Subsequently, the SaM-Requester starts answering to SaM-Requesters demanding CPU resources. If a program is transferred to the SaM-Requester, it sends an Upload_Done message to the CPU, which immediately starts executing. During program execution the SaM-Requester serves memory requests from the CPU as illustrated in [3].

Figure 3 illustrates the negotiation of SaM-Req0 and SaM-Req1 designed to spawn a new thread running in a shared address space. First, SaM-Req0 broadcasts a request for a processing unit into the SaM space (CPU Req). If no answer is received within a predefined time span, an error is returned to the CPU. In case a SaM-Requester answers with a CPU Ack message if the corresponding CPU is idle and fulfills the requirements. Now SaM-Req0 collects answers for a predefined time span t_1 and selects the processor which fits best. CPU Grant messages are exchanged in turn between the Requesters to acknowledge the match. This second round of messages enables the SaM-Req1 to answer subsequent CPU requests while not having granted the CPU. Then, SaM-Req0 sends the CPU state and all SaM-Table entries. As already mentioned the CPU state is needed to start the thread. The SaM-Table entries define the memory space of the first thread. Until now this memory was uniquely assigned to the first thread – creating a new thread requires to share this memory. Thus, sending the SaM

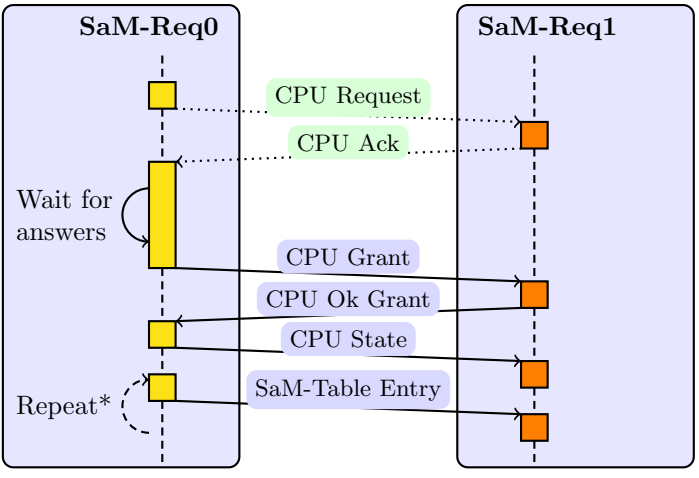

Fig. 3. SaM-Requester: protocol to spawn thread (on CPU1)

Table entries makes them shared. Since memory modules are scattered throughout the system and the first thread may have reserved memory in many of these modules, a distributed shared memory space results from creating the second thread. For additional threads, the memory space already exists and copying the SaM-Table entries suffices.

The example protocol to create a thread on CPU1 is shown in Figure 4: initiated by a call to thread_init, CPU0 sends a Thread_Init message to its SaM-Req0. On behalf of that message, SaM-Req0 finds a SaM-Memory node, which is capable of storing the program. After copying the program from the local memory to the SaM-Memory node, the SaM-Req0 returns. CPU0 continues executing and reaches the point to create a new thread (via Thread Create). SaM-Req0 reacts by finding a suitable CPU and setting up a shared memory space (for more details refer to the section above). After SaM-Req1 received and inserted the SaM-Table entries, it requests the program from the SaM-Memory node. SaM-Mem now sends the program to the SaM-Req1, which on successful completion signal Create_Ok to the SaM-Req0. Further, the newly created thread is executed on CPU1. SaM-Req0 forwards the successful creation of the thread to CPU0 which continues to execute.

3.2 Implementation

A schematic overview of the implementation of SaM-Requester and CPU node is shown in Figure 5. The picture is an excerpt obtained from Xilinx Platform Studio [1]. On the left hand side the microblaze processor with 64 KByte of local block random access memory (BRAM) connected through a local memory bus, divided into separate paths for instruction and data, is shown. The right hand side shows the identical design for the SaM-Requester, which is implemented as

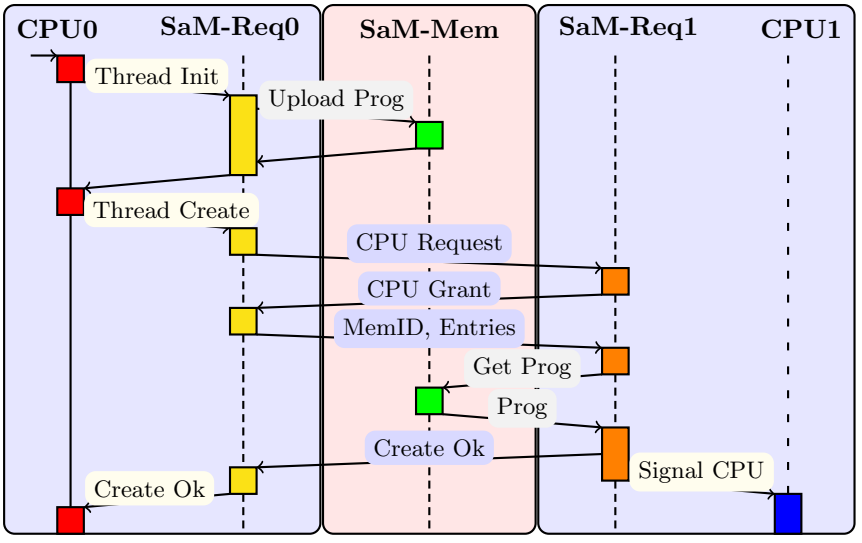

Fig. 4. Protocol overview creating a thread on CPU1

a program running on the microblaze processor. Both microblazes – representing CPU and SaM-Requester – communicate by sending messages over a fast simplex link (FSL) bus. The processor local bus (PLB) connects peripherals as well as the DRAM to the processors.

The implementation of the *idle* CPU state works as follows: the BRAM of the CPU microblaze contains a small program called *tiny bootloop*. This program polls the FSL and waits for the Upload_Done message from the SaM-Requester. The message contains the following information: instruction pointer, pointer to thread local data, return address, and stack pointer. Assigning these information to the architectural registers is done inside the tiny bootloop before branching to the new instruction pointer and executing the newly created thread.

While communication between CPU and Requester is bus based, Requester and Memory nodes communicate over ethernet. As reliable communication is of key importance, we utilize lwip4, a light weight TCP/IP stack for embedded systems [5]. The implementation of lwip4 comprises IP, TCP, ICMP, and UDP protocols. The hardware needs to be interrupt driven and deliver timer as well as ethernet interrupts to the processor. The lwip library takes care of the interrupt handling, exposing callback functions to the programmer (as low level API). By Registering and customizing these callbacks, the communication between SaM-Memory and SaM-Requester is implemented. However, utilizing only one of the available transport layer protocols (TCP or UDP) is not sufficient. While TCP does not support broadcast messages, UDP is connectionless and does not provide reliable communication. Hence, a combination of UDP (for mulitcasts and broadcasts) and TCP (for reliable communication) is employed. Figure 3

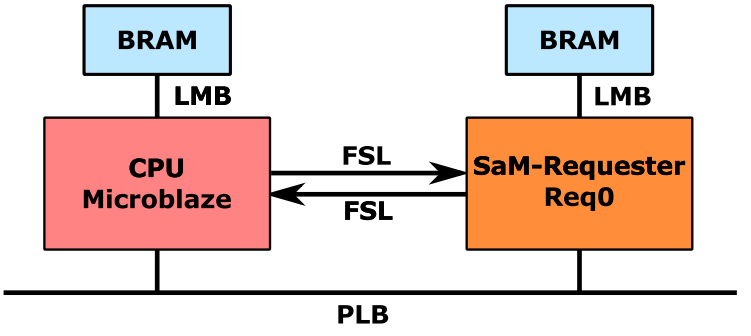

Fig. 5. Implementation of SaM prototype on Xilinx FPGA boards

illustrates this interaction: the messages with dotted lines (CPU Request and CPU Ack) are sent using the UDP protocol. The CPU Request is broadcasted whereas the answer (e.g. CPU Ack) is sent directly to the requesting node. Then, SaM-Req0 initiates a hand-over from UDP to TCP and establishes a connection to SaM-Req1. In order to establish a connection, SaM-Req0 to listen on a specific port and accept the incoming request. Once established, the connection serves as bidirectional communication channel. Now, the connection-oriented TCP protocol enables the reliable transmission of subsequent packets (CPU Grant, etc.).

To show the big picture of the SaM prototype, we would like to draw your attention to Figure 5. The components shown in this figure, CPU and SaM-Requester are implemented on one FPGA. The fast simplex link (FSL) enables message passing between these two components. The design of the SaM-Requester comprises a dedicated interrupt controller, prioritizing and delivering requests to the microblaze. Since FSL messages are of key importance, the priority of these messages is high. In order to service an FSL interrupt request a dedicated FSL interrupt handler was written: it copies messages to a buffer and defers processing of the packets. Thus, the interrupt handler occupies the processor only for a small amount of time. This is a critical aspect as interrupt handlers execute in a different context than user programs. If the interrupt controller raises an interrupt, masks all interrupts are masked (that is held back) until the interrupt handler acknowledges the current one. Thus, processing data packets in an interrupt handler not only degrades reactivity of the whole system but also may lead to a complete system lock up (e.g. because interrupts are masked while trying to send a TCP/IP packet).

4 Results

This section presents first results obtained from the SaM prototype introduced in this paper. Three Spartan-3A DSP 1800A FPGA boards from Xilinx where connected through an 8 port 10/100 Mbps ethernet switch. Each FPGA board holds one SaM component – in total two SaM-Requesters and one SaM-Memory.

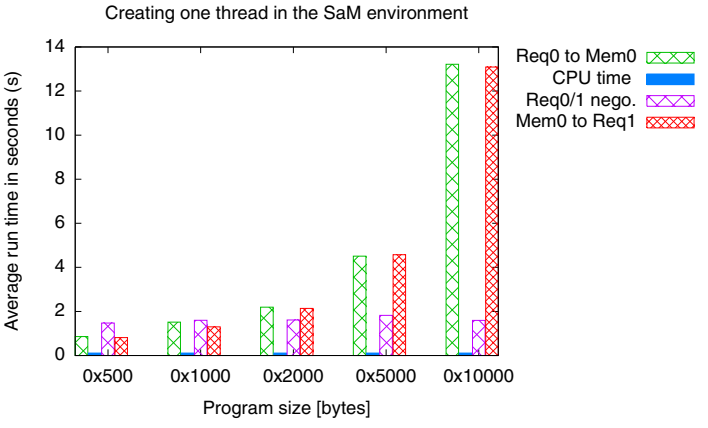

Fig. 6. Average time for spawning one thread while varying program size from 0x500 to 0x10000 Bytes

In our test case the CPU bundled with the first SaM-Requester (denoted Req0) starts the program, indirectly initiates a program transfer to the SaM-Memory (called Mem0) and later calls thread create. During the create call the second SaM-Requester (Req1) positively acknowledges a CPU request from Req0. This section studies the overhead of the process and improvements of the prototype. All times reported in this section are the average of 5 runs. By repeating the process, the effect of outliers on the reported numbers is mitigated.

Figure 6 depicts the various phases of the thread creation process (also confer to figure 4): the first bar shows how long a transfer of the program to a SaM-Memory takes, second the CPU time between `Thread_init` and `Thread_create`, third the negotiation process of the two SaM-Requesters is timed and last transferring the program from Memory to Req1 is shown. Figure 6 shows that only the duration of a program transfer is influenced by the program size. Further, as the program size rises (above 0x1000 Bytes) the time for the program transfer is the most prominent in the whole process. In addition selecting a particular CPU, takes a constant amount of time.

The second scenario simulates the case where a second CPU is not available immediately. Thus, an artificial delay was introduced before Req1 answers the request. The delay time is varied between 0 and 10000 milliseconds as shown in Figure 7. From the reported times we conclude that delays between 0 and 500 milliseconds will go unnoticed by the user, whereas larger delays (5 seconds and above) let the delay contribute the largest individual time to the overall process. The program size used in this scenario is fixed (0x5000 Bytes).

From the first two overhead studies presented before, we concluded that the implementation of the SaM-Memory is crucial for the overall performance. Especially, the program transfer time needs to be reduced. As a program has to be written to DRAM by the SaM-Memory, we decided to speed up the process

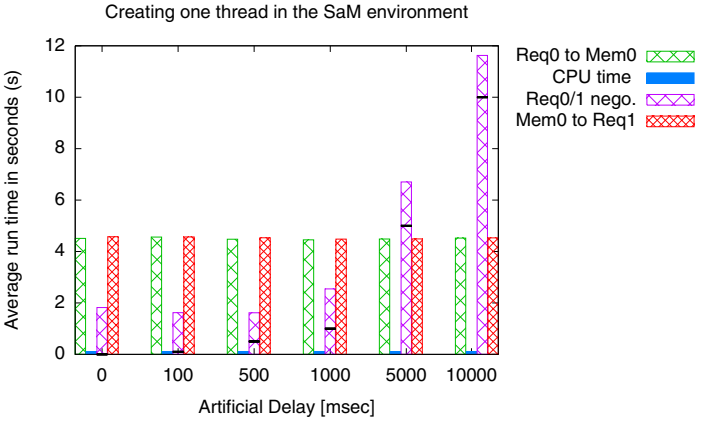

Fig. 7. Average time for spawning one thread while varying an artificial time delay from 0 to 10000 milliseconds

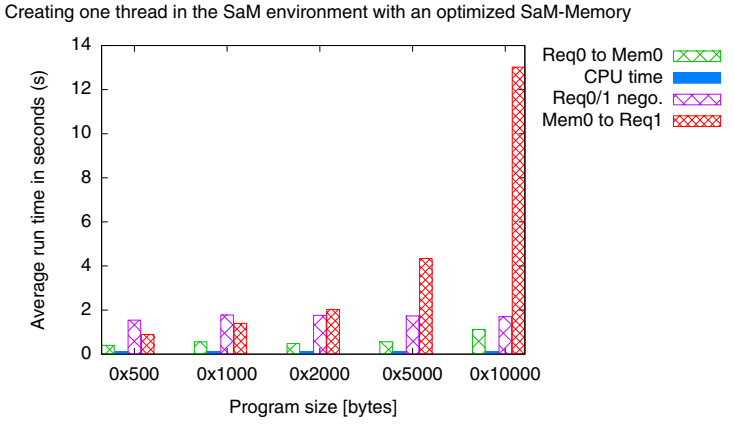

Fig. 8. Average time for spawning one thread while varying program size from 0x500 to 0x10000 Bytes employing an optimized SaM-Memory

by adding Cache Links for instruction and data to the microblaze CPU. The impact of these cache links is studied in the following.

Figure 8 shows a significantly reduced program transfer time from Req0 to Mem0. Thus, copying a program to DRAM memory from the ethernet interface is sped up significantly by adding the cache links.

The same becomes apparent in Figure 9. As writing the program to memory seems to be a crucial factor, equipping the SaM-Requester with cache links could lead to a reduced transfer time from Mem0 to Req1, further reducing the impact of the program transfer time.

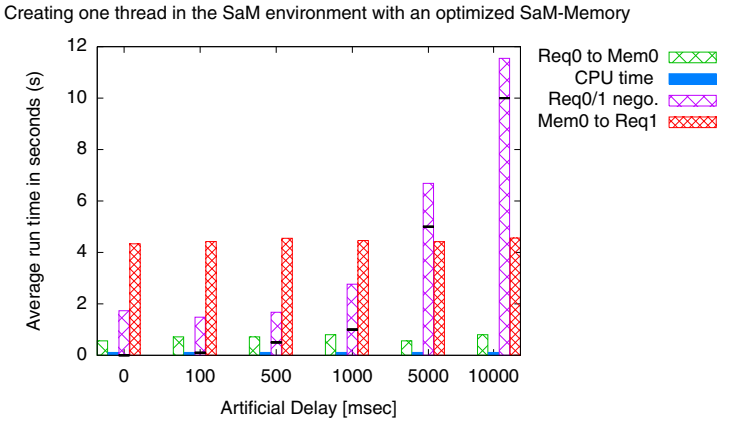

Fig. 9. Average time for spawning one thread while varying an artificial time delay from 0 to 10000 milliseconds with an optimized SaM-Memory

5 Conclusion

In this paper we present an approach towards the flexible use and management of computing resources in a distributed environment. Besides design and implementation of the POSIX-like thread concept, we also showed first performance numbers measured on a SaM prototype utilizing several FPGA boards.

Concluding from the results presented in Section 4, it became apparent that with larger programs (size $> 0x2000$ Bytes) the transfer time of the program becomes the most prominent factor in the protocol. This effect can be mitigated by adding cache links to respective components, as demonstrated with the optimized SaM-Memory design. Hence, future work will consider the optimization of the SaM-Requester.

Further, the response time of the SaM-Requester should not exceed 1 second. Otherwise, the relation between response time and transfer time is disproportional. However, with a optimized design this might change in the future - reducing the tolerable response time.

These are important insights and fundamentals that will help us to advance our work and experiment with real applications and more complex scenarios. In particular we would like to take the next step and design and implement the *join* operation to complement the creation of threads.

References

1. Asokan, V.: Designing multiprocessor systems in platform studio. In: White Paper: Xilinx Platform Studio (XPS), pp. 1–18 (November 2007)
2. Brinkschulte, U., Pacher, M., von Renteln, A.: An artificial hormone system for self-organizing real-time task allocation in organic middleware. In: Würtz, R.P. (ed.) Organic Computing, pp. 261–284. Springer, Heidelberg (March 2008)

3. Buchty, R., Mattes, O., Karl, W.: Self-aware Memory: Managing Distributed Memory in an Autonomous Multi-master Environment. In: Brinkschulte, U., Ungerer, T., Hochberger, C., Spallek, R.G. (eds.) ARCS 2008. LNCS, vol. 4934, pp. 98–113. Springer, Heidelberg (2008)
4. Dreier, B., Zahn, M., Ungerer, T.: The rthreads distributed shared memory system. In: Proc. 3rd Int. Conf. on Massively Parallel Computing Systems (1998)
5. Dunkels, A.: Full tcp/ip for 8-bit architectures. In: MobiSys 2003: Proceedings of the 1st International Conference on Mobile Systems, Applications and Services, pp. 85–98. ACM, New York (2003)
6. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
7. Mattes, O.: Entwicklung einer dezentralen Speicherverwaltung für verteilte Systeme. University of Karlsruhe (TH), Diploma thesis (May 2007)
8. Mueller, F.: Distributed shared-memory threads: Dsm-threads. In: Workshop on Run-Time Systems for Parallel Programming, pp. 31–40 (1997)
9. Müller-Schloer, C.: Organic computing: on the feasibility of controlled emergence. In: CODES+ISSS 2004: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 2–5. ACM, New York (2004)
10. Schmeck, H.: Organic computing - a new vision for distributed embedded systems. In: ISORC 2005: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, USA, pp. 201–203. IEEE Computer Society, Los Alamitos (2005)