Memory Architectures for NoC-Based Real-Time Mixed Criticality Systems

Neil C. Audsley
Department of Computer Science,
University of York, UK
email: neil.audsley@york.ac.uk

Abstract

Mixed criticality systems (MCS) allow software components of differing criticalities to use the same physical resources (ie. CPU, memory). MCS highlight the trade-off between partitioning components of different criticalities and efficient resource usage. Components are partitioned due to safety concerns, but physical partitioning requires more resources than if components are unpartitioned and share resources.

Influential recent work in scheduling of MCS shows the benefits of sharing resources between criticality levels. One issue with this work is that allowing resource sharing within MCS requires that sufficient partitioning and separation can be provided by the architecture to ensure mixed criticality components can share resources without compromising system safety at the highest criticality levels.

This paper examines this issue from the perspective of the memory hierarchy for Network-on-Chip architectures, considering memory system design for partitioning to support MCS scheduling approaches for multiprocessor systems – without the need for the resource expensive approaches of totally separated (ie. federated) approaches.

I. Introduction

Mixed Criticality Systems (MCS) are becoming increasingly important within the embedded real-time domain. Such systems contain components with different levels of criticality on a single platform. Domain motivation for MCS comes from mainly from aerospace and automotive. The former embodies the concept of components with different criticality levels within domain safety standards (eg. IEC 61508, DO-178B). Interestingly, the aerospace industry has also been developing mixed criticality infrastructure (ie. platforms, OSs), allowing components of different criticality levels to reside on the same physical platform, for several over 20 years via IMA [1], [2], [3], [4]. The automotive industry appears to be moving towards MCS as the number and complexity of in-vehicle software components increases.

In both aerospace and automotive domains, the main motivation for considering MCS designs is to meet non-functional requirements related to, for example, space, cost and weight – and for aerospace especially, fault-tolerance[1]. By allowing components of different criticalities to share physical resources, potentially fewer resources are required. In contrast, traditional (federated) systems architectures dictate that components of differing criticalities are allocated physically separate resources.

Mixed criticality systems highlight the fundamental trade-off between partitioning or separating components of different criticalities and efficient resource usage. From a safety perspective, partitioning is required: high criticality components must be protected from the interference and potential failure of lower criticality components. To achieve adequate partitioning, federated architectures can be used. These help when there is a need to show that a single failure cannot lead to a complete system failure – a typical requirement for safety-critical (fault-tolerant) systems. However, the approach is expensive in terms of physical resources needed.

In contrast, influential recent work in mixed criticality scheduling has shown the benefits of sharing resources between criticality levels [5]. The work is motivated primarily by resource efficiency, allowing software components of different criticalities to be scheduled on the same CPU (either in a single or multiprocessor system).

This paper examines MCS from the perspective of the memory hierarchy, showing how memory systems can be designed so that partitioning can be achieved to support MCS scheduling approaches – without the need for the resource expensive approaches of totally separated (ie. federated) approaches. The approach taken is to use a memory hierarchy designed for a predictable multiprocessor Network-on-Chip (NoC) system and enable support for multiple criticality levels within that architecture. We note that this is in contrast to conventional critical system design which seeks to use unpredictable (commodity) components and constrain them to be (almost) predictable.

The remainder of this paper is structured as follows. Section II reviews memory architectures. In section III the predictable memory architecture developed within the EU T-CREST project is introduced, which is extended into an appropriate architecture for MCS systems in section IV. Conclusions are offered in section V.

II. Memory Architecture Review

Memory provides storage for state (data) and code (instructions) which must be delivered to the CPU when required, within the time and resource constraints of the system. For real-time systems, constraints are focussed upon time – increasing memory latencies will increase worst-case execution times (WCET) and reduce the overall schedulability of the system. There are a number of issues for memory architectures

[1] In an IMA system in aerospace if a lane fails (e.g. one processor board from three) functionality from the failed board can be re-assigned to one of the other boards. This results in the possibility that software components of mixed criticality levels would eventually be resident on the same board, even if the original configuration placed functions of differing criticalities on physically separate boards.

for realistic mixed criticality systems, including[2]:

1) *Performance* – the increasing gap between CPU performance and memory system performance (ie. the "memory wall" [6]).
2) *Scalability* – increasing numbers of CPUs sharing the same memory hierarchy.
3) *Physical Separation* – a memory architecture must also provide sufficient physical separation between software components using memory to meet system safety requirements.

Performance and scalability requirements suggest moving towards higher performance architectures – which is in direct conflict towards the provision of physical separation needed within safety-related systems. At the extreme, in safety-critical systems a single failure cannot lead to a total system failure. When applied to the system architecture this implies that degrees of fault-tolerance are used (eg. redundancy). When applied to the memory architecture it implies protection between memory used by different components (particularly if they are of differing criticality levels). The simple method to achieve protection is physically separated system components; or at least to physically separate components of different criticality levels. Otherwise, the method by which memory protection is achieved becomes an important part of the safety argument for the system. Unfortunately, conventional (commodity) CPUs do not provide simple memory protection, but rather memory protection based upon complex memory management units (eg. virtual memory) which can be viewed as too complex to be free of errors (and hence usable within safety-critical systems).

The conventional memory hierarchy for safety-critical systems is illustrated in Figure 1. Storage increases in volume, but decreases in performance and cost (per byte), as the hierarchy is descended [7]. The degree of potential sharing between software components also increases as the hierarchy is descended.

The remainder of this section discusses the memory hierarchy, together with memory architectures for basic systems, multicore systems and many core systems. Throughout, issues in mixed criticality are highlighted.

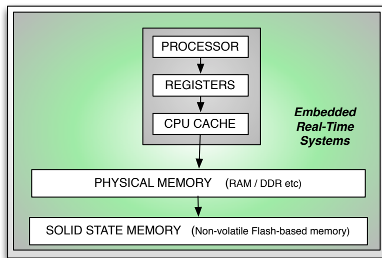

Fig. 1. Memory Hierarchy for Embedded Real-Time Systems

[2]*Application Complexity* is also important in terms of the memory hierarchy, as increasing complexity is being seen in terms of the amount of data being input, accessed, processed and stored by a system, e.g. due to the use of high bandwidth I/O devices within systems, and large scale persistent data storage (eg. for storing maps in navigation). These issues are not discussed further in this paper.

A. Basic Memory Architecture

Many systems have used simplistic memory architectures – a single CPU is connected directly to a memory structure consisting of some form of RAM, together with persistent program / data storage, the latter used mainly during bootstrap.

Within this architecture, memory latencies consist of the latency of the memory device itself; together with the bus between CPU and memory. In simplistic architectures the latter is minimal (a few cycles). Burst mode memory accesses together with DMA (potentially from I/O devices) can increase the latency of memory access as viewed by an application executing upon the CPU (eg. as their memory access may be delayed by the DMA burst instigated by an I/O device).

More complex CPUs may include caches to help performance, particularly as CPU speeds increase compared to available memory bandwidth. Worst-case execution time (WCET) calculation becomes more complex in the presence of cache [8], [9] as it is difficult to predict during analysis the state of cache at any time. Other memory variants include scratchpad memory (SPM), which make WCET calculation easier and less pessimistic [10], [11].

Clearly, basic memory architectures can be used with more complex high performance CPUs, with multi-level caches for performance, and that support memory protection via Memory Management Units (MMUs). We note the issue regarding the complexity of MMUs raised above.

B. Multicore Memory Architecture

Multicore CPUs (within the context of this paper) refer to the incorporation of several CPUs on the same chip, biased towards a shared memory architecture [7]. The development from the basic memory architecture to the multicore architecture is merely the presence of several (commonly upto 4) accessing memory via the same shared bus.

One of the largest effects of CPUs sharing the bus is upon memory latencies – a CPU may have to wait for the memory transactions of other CPUs to finish before it is able to access the bus. Similarly, a memory request reaching the physical memory may be delayed by the completion of requests for other CPUs (eg. when using DDR).

Within multicore architectures, there are many memory design choices, two important ones being whether one or more cache levels are shared between CPUs; whether to use symmetric or non-symmetric memory architecture. Shared caches are often used in high performance (eg. desktop) architectures, needing hardware support for cache coherence for obtain adequate performance [7]. In terms of real-time systems, it is not clear that any performance advantage is gained, given that execution times are extremely hard to model and bound accurately with shared coherent caches, resulting in pessimistic WCETs [8] (potentially removing any performance gained). From a safety-critical perspective, it is unclear whether the complexity of cache-coherence implementations (at the hardware level within the CPU) are appropriate.

Non-Uniform Memory Architectures (NUMA) allow different physical views of memory for each CPU. Thus, the addition of private memories to the CPUs within a symmetric architecture would make the architecture NUMA. We

can consider the non-uniform nature of the architecture at a number of levels. At a physical level, access latencies will be different depending upon whether the CPU addresses the shared memory, or its own private memory. Access to the other CPU's private memory may well be provided at the physical level (ie. within the address map of the CPU), or could require OS, and/or application support.

NUMA architectures where CPUs have private memories as well as access to shared (global) memory are an interesting option for safety-critical systems. Private memories support requirements for physical separation – assuming that components of different criticalities were placed upon different physical CPUs. If components changed criticality level however, there is a requirement to then move to a different CPU, which raises issues of migration overhead as well as schedulability of the changed system configuration.

C. Many-Core Memory Architecture

Many-core architectures (within the context of this paper) refer to approaches to scaling the number of CPUs within a chip in a manner far exceeding conventional multicore. A key difference in approach is to move away from a shared bus approach to connecting CPUs to shared memory as this approach is not scalable from the perspective of contention delays [12]. In contrast, many-core architectures adopt a form of communication mesh to connect CPUs and memories [13]. This results in many routes between a CPU and memory, so that contention on the connections between CPUs and memory can be reduced. We note that when requests actually arrive at the memory, contention will still occur at the physical level.

The manycore approach is exemplified by the Network-on-Chip (NoC) approach [14]. Typically a packet switched communications mesh of regular Manhattan topology is used, with arbiters at junctions routing traffic, and CPU tiles connected to arbiters via a local link. Essentially, the NoC appears as a distributed system, with separate CPUs connected by a communications network. In terms of memory architecture, external shared memory is attached to an arbiter at the edge of the mesh (as is any I/O); CPU tiles can also contain local memory. Alternatively, a second network can be used to connect CPUs to shared external memory to remove memory traffic from the mesh [15], [16].

This is essentially a non-symmetric approach. Each CPU has a different view of the memory depending upon how many hops are between it and the external memory, and other CPUs (if a CPU is allowed to indirectly access other CPUs local memory [17], [18]).

In terms of achieving adequate performance from the memory architecture, the mapping of application components to CPUs and memories within the many-core becomes crucial. Code needs to be close to required data; application threads that communicate should be placed on CPUs that are close. Mapping approaches are an active research area [19], noting that optimal solutions are NP-complete.

D. Summary: Memory Requirements for MCS

Mixed criticality systems highlight the fundamental trade-off between partitioning or separating components of different criticalities and efficient resource usage. Recent MCS research (based upon [5]) considers scheduling CPUs between processes of differing criticality levels, largely ignoring other resources. For memory, available separation is defined by the physical architecture. To facilitate recent MCS scheduling approaches, memory architectures require software processes of differing criticality levels to share physical memory, with two areas of sharing to consider:

- *Shared route / connection between CPU and memory*: Shared connections imply the potential for interference between competing memory transactions. This can lead to race conditions between memory transactions, and difficulties in bounding transaction latency.
- *Shared physical memory*:
  Where memory (and memory controller) are physically shared between many CPUs, the memory is effectively multiplexed between the CPUs – hence memory requests from one CPU may be delayed by those of another.

One of the characteristics of MCS scheduling approaches is that Worst-Case Execution Times (WCET) are dependent on criticality level – more conservative (higher) values are used if the process is assigned a higher criticality level. In terms of memory access latencies (a part of WCET), there is a fundamental requirement for predictability. However, different assumptions regarding competing memory transactions can be made to allow less pessimistic memory latency times to be derived (see sections III and IV).

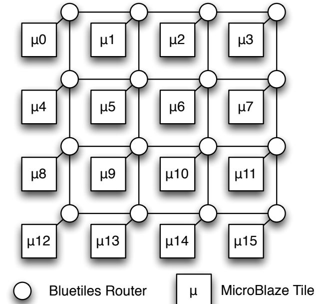

Fig. 2. Blueshell Network-on-Chip with 16 CPU Tiles

III. Predicitable Memory Architecture

The EU T-CREST project[3] is developing NoC architectures suitable for safety-critical systems requiring the highest levels of predictability. In this section we discuss a shared memory tree architecture (Bluetree) that has been developed within T-CREST. Subsequently we show how this architecture supports mixed criticality systems.

We note that the memory hierarchy that is discussed in this section was designed to support timing predictability. This paper contends that this is a more appropriate starting point for a MCS memory architecture than adopting commodity hardware approaches and attempting to restrict their behaviour to being (nearly) predictable.

[3]T-CREST – Time Predictable Multicore Architecture for Embedded Systems: http://www.t-crest.org/

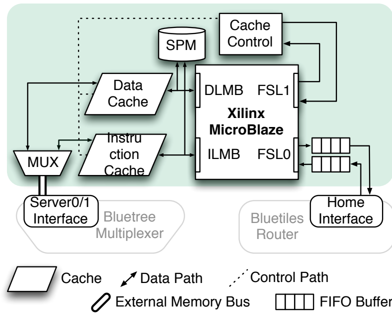

Fig. 3. Internal Structure of Microblaze CPU Tile

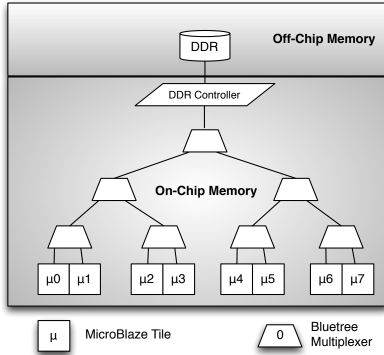

Fig. 4. Bluetree Shared Memory Tree for 8 CPU Tiles
Each CPU tile in the Bluetiles NoC (see figure 2) also connects directly (via cache) to the Bluetree shared memory (binary) tree to access external DDR memory, shown above. There is no interference between CPU to CPU messages across the NoC and CPU to memory transactions across the tree.

A. Bluetree Shared Memory Tree Architecture

Bluetree is a shared memory tree architecture developed for NoC architectures [15], [16], [20] CPUs are arranged in a mesh (Figure 2) to allow CPU to CPU communication. Each CPU tile within the architecture has two main interfaces (Figure 3), one to access the mesh (*Home Interface*) and one to access shared memory (*Server Interface*). The separation of inter-CPU communications from shared memory access (eg. for cache misses) ensures that memory accesses do not interfere with CPU to CPU communications – thus aiding predictability. The shared memory tree architecture is illustrated in Figure 4:[4]

- *Routers*: Routers are 32-bit bi-directional with X-Y routing used (destination is contained in the first word). We note that the choice of routing policy does not impact upon the research presented in this paper, since this research focuses on the communication over Bluetree; Bluetiles is only used for simple synchronisation.
- *Shared Memory Tree (Bluetree)*: 2-to-1 multiplexors form a tree connecting CPUs to memory – CPUs are the leaves of the tree, memory being at the root. High-bandwidth memory requests do not impact the performance of other CPUs – there is no interference between CPU to CPU messages across the NoC and CPU to memory transactions across the tree. Each multiplexor port allows 128 bits of data (corresponding to the cache line size).
- *CPU Tiles* [15], [16]: CPU tiles are built using the Microblaze CPU. CPU configuration is 8kB split data and instruction caches, and a 8kB shared scratchpad used for fast local storage. The CPU accesses the cache via Microblaze LMB interfaces; cache misses being issued to external memory via Bluetree. The CPU tile contains custom cache control is configured to allow selective invalidation of cache lines and to record prefetch related data on a per cache line basis (the cache control unit also serves as the Microblaze's interrupt controller and provides a clock-cycle counter facility). Cache control is accessed via Microblaze Fast Simplex Links (FSL), utilising single-cycle FIFOs. Further details of the cache design are given in [15], [16].

[4]Bluetree has also been implemented with a pre-fetch unit between off-chip and on-chip memory[15], [16], [20].

B. Off-chip Memory

The Bluetree shared memory tree is connected to off-chip memory as shown in Figure 4. The key component is the memory controller which interfaces between requests originating from the CPU and passing over the shared memory tree, and the external (off-chip) SDRAM (ie. DDR). The controller is based upon the architecture in [21], but has been developed with additional configuration infrastructure to allow, effectively, several distinct channels. The motivation for this is to allow, for example, separate memory access bandwidths to be specified for each channel, thus providing a degree of separation in the memory system – essentially each channel forms a separate queue of memory requests which are then multiplexed (according to remaining bandwidth) onto the single SDRAM.

IV. Predictable Memory Architecture for MCS

This section discusses the predictable memory architecture described in section III in the context of MCS. Initially an appropriate arbitration scheme is outlined for Bluetree. This is then taken, together with definition of the worst-case latency across the memory tree, to provide a worst-case timing of the memory architecture. Finally, this is assessed to see whether it supports MCS and MCS scheduling approaches.

A. Arbitration within Bluetree

The shared memory tree requires arbitration at each of the multiplexors. A simple approach would be to adopt a first-come-first-served approach. It could be argued that this could lead to starvation, as one input (CPU) to a multiplexor could monopolise – although this would require memory requests to be generated on *every* clock cycle, which is unlikely. Another approach would be to always favour one input over another, but this could lead to, eg. priority inversion (in a priority scheduled system). Another approach could be to ensure that turns were taken at each multiplexor to eliminate starvation, but would partially suffer from effects like priority inversion.

The approach taken within Bluetree is to allow run-time programming of the arbiters. Essentially, each memory request from a CPU is accompanied by some measure of importance (eg. priority) so that the most important request always wins

arbitration at a multiplexor. We note that this easily maps onto criticalities – at each multiplexor, if two requests arrive at the same time (ie. same clock cycle) the request with the highest criticality would be forwarded. In the event of a tie (i.e. equal criticality) a secondary mechanism can be used, eg. turns.

The Bluetree arbitration approach provides separation between streams of memory requests originating from software components of different memory criticalities. From a safety-critical perspective it would be relatively straightforward to show that there can be no interference between input channels to the multiplexor (as they have separate latches) – and therefore that separation is maintained (multiplexor logic is simple, cf. MMUs).

B. Worst-Case Latency

The worst-case time for a memory request issued Bluetree is the sum of three components:

1) *Latency to cross Bluetree from CPU to memory controller*: The basic latency to cross a shared memory tree multiplexor is two cycles (Bluetree uses a buffer for both input and output) [16], hence total latency is 2 + (2 * the memory tree depth) – noting that there is an extra latch stage used to link shared memory tree to the memory controller.
2) *Latency to cross the memory controller and access the SDRAM*: This varies according to the exact configuration and build of the controller, but typical figures are around 25 cycles for a read (and 1 cycle for subsequent reads in a burst), and 1 cycle for a write. These figures refer the time needed *before* return can be made to the CPU.
3) *Latency to cross Bluetree from memory controller to CPU*: This is 1 + the memory tree depth (as no latching occurs within the tree on the return path). For a burst, the first word returned would suffer the full return path latency, successive bytes are delivered in successive clock cycles.

Note that a burst read, which can be encapsulated within a single 128 bit request to the memory controller (and is therefore non-preemptable within the tree) is actually broken into a series of successive memory requests to the SDRAM – hence there is potential for pre-emption within the memory controller if required.

C. Worst-Case Timing of Bluetree for MCS

The worst-case timing across the shared memory tree requires the worst-case latency and MCS arbitration approaches (above) to be combined.

Consider a CPU executing with a software component of criticality level X. All memory requests from that CPU (whilst that software component is executing) have the criticality of the issuing software component. At each multiplexor a memory request can be delayed by at most one request of equal criticality; and by the maximum number of successive requests of higher criticality memory requests.

When a memory request exits the multiplexor connected to the memory controller and At the memory controller, there

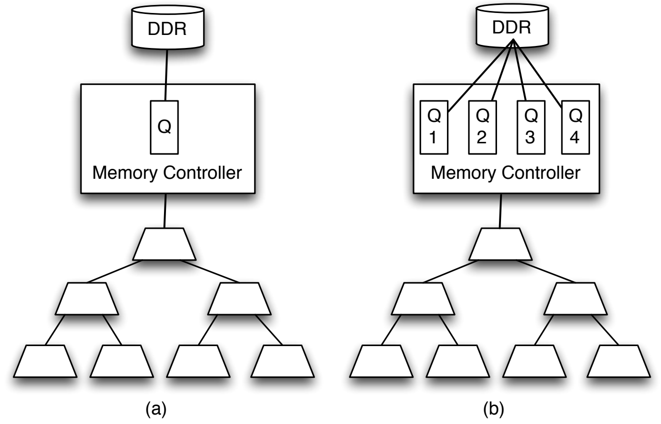

Fig. 5. Memory tree with (a) Single Memory Queue and (b) Memory Queue per Criticality Level

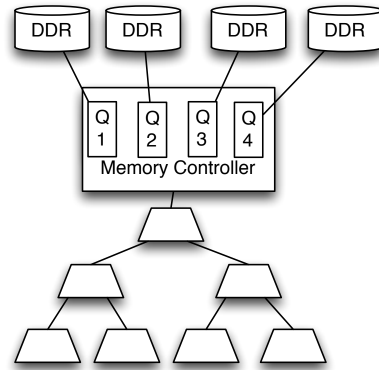

Fig. 6. Memory tree with Memory Queue and Memory per Criticality Level

are three possibilities when considering memory request m of criticality X:

1) *Single Memory Queue* (Figure 5(a)) – account must be taken for all other memory requests that arrive ahead of m that are therefore ahead within the memory controller queue.
2) *Multiple Memory Queue* (Figure 5(b)) – where there are separate queues for each criticality level. Here account must be taken for all other memory requests of the same criticality that arrive ahead of m that are therefore ahead within the memory controller queue; and of all other memory requests in higher criticality queues.
3) *Multiple Memory Queue and Memory* (Figure 6) – where there are separate queues and physical memories for each criticality level. Here account must be taken for all other memory requests of the same criticality that arrive ahead of m that are therefore ahead within the memory controller queue.

One issue remains, that of the effect of bursts. If they are not pre-emptable (see discussion above) then allowance has to be made for the total burst length when calculating maximum delay at the memory controller. If this becomes dominant, it may be appropriate to adopt more bandwidth oriented memory

controller approaches (eg. see [21]).

D. Mixed Criticality Timing Analysis Principles and Memory

The essential principle of MCS is that a software tasks' WCET is dependent upon its criticality level. At a high level of criticality code will have a higher WCET than if it is assigned a lower criticality level. This reflects the degree of pessimism that is required at different criticality levels – if exceeding the WCET is deemed a failure, then at the highest level of criticality (in aerospace systems) this failure has a probability of no more than 1 in 10^9 occurrences. At lower levels of criticality, the same code can be assigned lower (less pessimistic) WCET bounds. Although the chances of exceeding the WCET has increased, the effect on the system is not as great – eg. system integrity is not compromised by multiple failures of components at the lowest criticality level.

The essential principles outlined above applies equally well to the analysis for the memory architecture defined within this section. As the criticality level of a software component increases, a more pessimistic view of the potential interference of other tasks upon memory requests must be included in the analysis. If assigned the highest criticality level, each memory access of a task would assume worst-case interference upon its memory requests as given above – potentially large. At lower levels of criticality, la ess pessimistic view can be taken: that accesses will suffer less interference. This is achieved by considering the number of cycles assumed between successive memory transactions from other CPUs – for high levels of criticality the minimal number of cycles can be assumed; for low levels of criticality longer intervals can be assumed (more realistic in the average case).

V. Conclusions

This paper has considered the role of the memory hierarchy within many core architectures (specifically Network-on-Chip) proposing an appropriate memory hierarchy for MCS based upon a predictable shared memory tree memory hierarchy. The paper has shown that sufficient partitioning and separation can be provided by the architecture to ensure mixed criticality components can share resources without compromising system safety at the highest criticality levels. Thus the approach supports the MCS scheduling work within the real-time community which allows components of mixed criticality to share resources.

The architecture includes an arbitration approach within the memory tree that directly supports criticality levels. The additional benefit of this is that if stricter memory separation was needed to support safety-critical requirements, separate memories can be included, one per criticality level. This is less than normally required for federated architectures which would dictate one memory per component (not criticality level).

Acknowledgements

This work is supported in part by EU FP7 project T-CREST (288008).

References

[1] R. A. Edwards, "ASAAC Phase I Harmonized Concept Summary," in *Proceedings ERA Avionics Conference and Exhibition*, UK, 1994.

[2] *ARINC 651: Design Guidance for Integrated Modular Avionics*. Airlines Electronic Engineering Committee (AEEC), 9th November 1991.

[3] *ARINC 653: Avionics Application Software Standard Interface*. Airlines Electronic Engineering Committee (AEEC), June 17th, 1996.

[4] N. Audsley and A. Wellings, "Analysing apex applications," in *Real-Time Systems Symposium, 1996., 17th IEEE*. IEEE, 1996, pp. 39–44.

[5] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance." in *RTSS*. IEEE Computer Society, 2007, pp. 239–243.

[6] 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.

[7] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2011.

[8] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[9] J. Whitham, R. I. Davis, N. C. Audsley, S. Altmeyer, and C. Maiza, "Investigation of scratchpad memory for preemptive multitasking," in *Proceedings of the 2012 IEEE 33rd Real-Time Systems Symposium*, ser. RTSS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 3–13.

[10] J. Whitham and N. Audsley, "Investigating average versus worst-case timing behavior of data caches and data scratchpads," in *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ser. ECRTS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 165–174.

[11] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*, ser. Embedded systems. Springer, 2010.

[12] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration," *ACM Trans on Design Automation of Electronic Systems*, vol. 12, no. 3, 2007.

[13] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pp. 684–689, 2001.

[14] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*, ser. Systems on Silicon. Elsevier Science, 2006.

[15] G. Plumbridge, J. Whitham, and N. C. Audsley, "Blueshell : A Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators," in *Proceedings HEART Workshop*, 2013.

[16] J. Garside and N. C. Audsley, "Prefetching Across a Shared Memory Tree within a Network-on-Chip Architecture," in *Proceedings International Symposium on System-on-Chip*, 2013.

[17] I. Loi and L. Benini, "An efficient distributed memory interface for many-core platform with 3d stacked dram," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 99–104.

[18] B. C. Lam, A. D. George, and H. Lam, "Tshmem: Shared-memory parallel computing on tilera many-core processors," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 325–334.

[19] P. K. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *Journal of Systems Architecture*, vol. 59, no. 1, pp. 60 – 76, 2013.

[20] J. Garside and N. C. Audsley, "Investigating Shared Memory Tree Prefetching within NoC Architectures," in *Proceedings Memory Architecture and Organization Workshop MEAOW*, 2013.

[21] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration." in *DATE*. IEEE, 2011, pp. 851–856.