# Hardware-Software Partitioning of Soft Multi-Core Cyber-Physical Systems

By

Benjamin Babjak

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

December, 2014

Nashville, Tennessee

Approved:

Akos Ledeczi, Ph.D.

Theodore Bapty, Ph.D.

Aniruddha S. Gokhale, Ph.D.

Janos Sallai, Ph.D.

Pietro Valdastri, Ph.D.

## TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

## LIST OF ABBREVIATIONS

**ADC** Analog–to–Digital Converter 1, 23, 27, 39, 64, 82

**AE** Acoustic Emission v, vii, viii, 3, 4, 69–78, 80, 83–85, 89

**AIC** Akaike's Information Criterion vii, 72, 74, 77–82

**ALU** Arithmetic Logic Unit 11

**AMP** Asymmetric MultiProcessing 11, 12, 29

**AP** Assignment Problem 14

**API** Application Programming Interface 19, 65

**AR** AutoRegressive 72

**ASIP** Application–Specific Instruction–set Processor 11

**BIC** Bayesian Information Criterion 74

**CDF** Cumulative Distribution Function 83, 84, 87

**CMOS** Complementary Metal–Oxide–Semiconductor 10, 85

**COMA** Cache Only Memory Architecture 12, 30

**CORDIC** COordinate Rotation DIgital Computer 11

**CPLD** Complex Programmable Logic Device 17

**CPS** Cyber-Physical System 1–3, 8, 42, 89, 92

**DAC** Digital–to–Analog Converter 1, 23

**DFG** Data–Flow Graph 18

**DFS** Depth–First Search 54

**DMA** Direct Memory Access 12, 82

**DSP** Digital Signal Processor 2, 11, 21

**EEPROM** Electrically Erasable Programmable Read-Only Memory 17, 24

**EM** Expectation–Maximization iv, vii, 74, 79, 83

**EMI** ElectroMagnetic Interference 22

**FIFO** First In, First Out 29

**FIR** Finite Impulse Response 70

**FPGA** Field–Programmable Gate Array 2, 3, 7, 11, 18, 19, 21–23, 25–27, 29, 66, 92

**FSM** Finite–State Machine 61

**FTL** FreeMarker Template Language 51

**GA** Genetic Algorithm 14, 74

**GALS** Globally Asynchronous Locally Synchronous 21, 22, 90

**GMM** Gaussian Mixture Model iv, 74, 79

**HPF** High–Pass Filter 70

**$I^2C$** Inter-Integrated Circuit 13, 17, 39

**IAT** Inter-Arrival Time 83, 84, 87, 89

**IC** Integrated Circuit 6, 10, 13, 16, 17, 20, 21, 23, 62, 64, 85, 92

**IDE** Integrated Development Environment 61

**ILP** Instruction–Level Parallelism 10

**IMU** Inertial Measurement Unit 20

**IO** Input/Output 16, 23, 24, 27, 32–34, 39, 61, 62

**IP core** Intellectual Property core 2, 3, 5, 6, 11, 26, 30, 83

**IQ** In-phase and Quadrature 23

**ISA** Instruction Set Architecture 2, 11, 24, 32

**ISM** Industrial, Scientific, and Medical 23

**KP** Knapsack Problem 14

**MAP(2)** Markovian Arrival Process with two states 84, 88–90

**MCMC** Markov Chain Monte Carlo 14

**MCU** MicroController Unit v, 1, 2, 4, 6, 9, 16–18, 20, 21, 24–26, 32, 61–64, 68, 82, 85, 92

**MIMO** Multiple-Input and Multiple-Output 23

**ML** Maximum Likelihood 74

**MMCMC** Metropolis Markov Chain Monte Carlo 74

**MoC** Model of Computation 3, 18

**nesC** network embedded systems C iii, vi, 5–7, 32, 36–40, 42–44, 46, 47, 51, 92, 93

**NUMA** Non–Uniform Memory Access 12, 30

**OFDM** Orthogonal Frequency–Division Multiplexing 23

**OPTICS** Ordering Points To Identify the Clustering Structure 74, 82, 83

**OS** Operating System 17, 19, 37, 40

**PC** Personal Computer 13, 14, 42

**PLD** Programmable Logic Device 2, 8, 10, 11, 18, 21

**PSO** Particle Swarm Optimization 14

**RAM** Random–Access Memory 11, 16, 18

**RF** Radio Frequency 1, 17, 20–24, 32, 62, 64, 83

**RISC** Reduced Instruction Set Computing 18, 24

**ROM** Read–Only Memory 18, 24

**RTL** Register–Transfer Level 3, 11

**SA** Simulated Annealing 14

**SHM** Structural Health Monitoring v, 3, 7, 69–71, 82, 83, 85, 86, 93

**SHP** System Hyper Pipelining 11, 25

**SMP** Symmetric MultiProcessing 11–13

**SNR** Signal–to–Noise Ratio 23

**SoC** System on Chip 11, 23, 25, 66

**SoE** System of Elements 66, 68

**SPI** Serial Peripheral Interface Bus 64, 66

**SRAM** Static Random–Access Memory 21, 24, 25, 34, 92

**TDMA** Time Division Multiple Access 17

**TDoA** Time Difference of Arrival v, 69, 74, 79, 81, 83, 86, 87

**TM** Transactional Memory 19

**UMA** Uniform Memory Access 12

**USB** Universal Serial Bus 22, 23

**USRP** Universal Software Radio Peripheral 62

**VHDL** Very high speed integrated circuits Hardware Description Language 3, 18

**VM** Virtual Machine 17

**WPD** Wavelet Packet Decomposition iv, v, 70, 77, 78, 83, 87

**WSN** Wireless Sensor Network 5, 20, 22, 26, 32, 37, 50, 62–64, 69, 71, 78

**XML** Extensible Markup Language 51, 52

**XSLT** Extensible Stylesheet Language Transformations 51, 54, 57

# CHAPTER I

# INTRODUCTION

Embedded systems have become an ubiquitous part of our contemporary environment. This is made possible mainly by the continuous decrease in power consumption and size of newer generations of semiconductor devices in accordance with Moore's law. Complete embedded systems are designed with only a handful of components. The central piece of these platforms is a MicroController Unit (MCU) with a set of integrated peripherals, such as Analog–to–Digital Converters (ADCs), Digital–to–Analog Converters (DACs), hardware timers, etc. For additional functionality, external peripheral devices are added to the platform. These devices are themselves highly-integrated, and may provide various services, such as Radio Frequency (RF) communication, optical sensing, motion sensing, etc. By employing efficient duty-cycling, these platforms can operate at low power consumption levels unachievable by other processing solutions, like general purpose processors found in desktop computing.

However, with the proliferation of embedded technology, new use cases started to emerge putting forward processing requirements not achievable by these conventional architectures. New application type, referred to as Cyber-Physical Systems (CPSs), are defined by the close interaction of physical and computing systems. A subset of CPSs observe physical phenomenon and instantly process recordings, which can be best accomplished if close to the source. Hence, these systems are tightly integrated with physical structures or complex machinery, and form an essential part of the whole. These new CPS applications have more complex computational requirements. More precisely, a subset of CPSs are multi-channel high-throughput applications. They usually incorporate several sensors, for which the recordings have to be processed concurrently at a sufficiently high rate in order to efficiently control the underlying system. These requirements of low power operation and high computational throughput may only be met by reconfigurable parallel computing and most recent silicon technology as opposed to the widespread single-core design philosophy.

Traditionally, the single-core design process for all kinds of embedded architectures revolves around finding the right MCU for the task [32]. This may be difficult and unlike other system design processes of other engineering fields. For instance, in case of engines, it is fairly well-known how the end product is used, and requirements will not change over time. CPSs – being in effect computer-based systems – are expected to adapt and provide

more functionality over time with updated software. Improvement needs might originate from an ever changing physical environment, or an update could become available because a better algorithm was developed.

Thus, embedded systems in general usually resort to excessive general-purpose computing solutions to deliver the necessary performance and adaptability. However, this is inevitably suboptimal and inherently inefficient – if feasible at all. For instance, to achieve hard-real time requirements and deterministic timing, the auto industry employs polling and lookup tables, which contain pre-calculated system responses. This way computation is avoided as every response is a memory load operation. Timing can be handled with relative ease, but the cost is a much bigger memory with higher power consumption. This is inherent in and a response to the fact that a general-purpose processor cannot easily guarantee the execution of complicated algorithms in a timely and deterministic manner.

New high-throughput CPSs have basically two main alternatives to MCUs. On the one hand, Digital Signal Processors (DSPs) represent a specialized form of embedded processors that have support for certain types of computations at the Instruction Set Architecture (ISA) level. On the other hand, Field–Programmable Gate Arrays (FPGAs), the most common form of Programmable Logic Devices (PLDs), offer the possibility to implement arbitrary digital circuits. For high-throughput computation-intensive tasks, both outperform general-purpose MCUs. Unfortunately, for severely resource-constrained battery-operated devices, they consume too much power, and are thus inapplicable.

Conventional FPGAs store configuration in an external memory. Hence, every time they start up, the contents of said memory have to be read, resulting in an initial phase when the fabric is already powered on, but not yet configured. It is during this short time period that conventional FPGAs draw high inrush currents, thereby consuming power unnecessarily. However, with the introduction of flash-based process technology, it became possible to store FPGA configuration directly on chip. This had the advantage that the most important power saving feature, duty-cycling became viable. Hence, flash-based FPGAs represent an interesting new direction of research for the embedded field. They provide a unique opportunity to implement parts of the algorithm directly in hardware. This translates to lower power consumption and better timing.

Also, a combination of these devices can be employed, for example, one or several MCUs and an FPGA. Even with just an FPGA, there are several options to explore, e.g., one soft-core instantiated in fabric along Intellectual Property cores (IP cores) for application-specific processing. The main issue of this new FPGA technology is the fundamentally different programming paradigm. MCUs prominently employ simple imperative languages, like C,

describing algorithms in a sequential manner. FPGAs, on the other hand, follow a Register–Transfer Level (RTL) programming concept with naturally more concurrent languages, like Very high speed integrated circuits Hardware Description Language (VHDL) [70]. The two Models of Computation (MoCs) are incompatible. Certain classes of calculations can be conveniently expressed with the latter, resulting in efficient IP cores implementations. Yet, in most cases of high-level algorithm development, the RTL abstraction is cumbersome. Any switch between these models effectively means manual (re)implementation. Once the assignment of MoC components to hardware and software is complete, it is exceedingly tedious to rearrange the setup [84].

Testing the performance under real load will almost certainly lead to rewrites, as system parameters cannot be safely estimated before the whole implementation process finished. Also, since development is iterative, the above steps have to be repeated [24, 74, 83]. Thus, development turns into a labor-intensive trial and error process, without the possibility to rapidly converge towards an optimum solution. Opportunities for reuse of hardware and software modules are limited, and proper trade-off exploration is missing. Hence, the development process will likely yield suboptimal results for the multi-channel high-throughput application domain [81].

Thus, instead of utilizing the hardware directly, soft processing cores should be instantiated in the fabric, which can then be programmed the conventional way. Consequently, developers can use familiar languages and development environments. This necessitates a software development framework that supports multi-core platforms.

In this new co-design philosophy, not only software but – due to soft-core processors – the hardware itself becomes an adaptable, application-oriented part of the design. Hardware and software are co-designed in a spiral development cycle [70, 84]. A prerequisite for this integration are hardware and software modules with well-defined interfaces hiding actual implementation details and encouraging trade-off exploration. These abstractions enable mapping that supports systematic refinement of models into optimized implementations on parallel architectures [58].

Our research focuses on this direction, but with the main idea of instantiating not one, but several soft cores, in order to have a single chip multi-core embedded architecture. The central motivation for our approach stems from the observation that contemporary embedded CPSs have to perform many loosely connected high-throughput tasks in a tightly timed parallel manner.

Consider the example of Structural Health Monitoring (SHM) employing Acoustic Emission (AE) signals. This application is going to be the comprehensive use case of this new concept, and is described in great detail in chapter VI. In this example, cracking events in

metal structures form ultrasonic stress waves, which can be detected. Evaluation of these signals provides deep insight into the structure's condition. A tightly integrated embedded system has the advantage that it can perform the analysis in real-time. The main tasks of the system include onset time detection of AE signals, the classification, and the radio transmission of important events. All of these tasks are fairly independent with individual timing requirements, which can be more easily met by individual cores.

The contributions of this thesis include an end-to-end design approach that yields multi-core applications with event-driven inter-core communication, a detailed simulation environment, and a comprehensive case study.

**Solution strategy**

Our concept involves the functional decomposition of complex applications with the goal to identify the autonomous modular components of the design. Components associated with certain functionality are assigned to individual soft processing cores, which only have limited responsibility. Our approach for parallel embedded computing focuses on a subtype of problems with messaging-based loose connection among cores. Also, computing nodes perform different tasks, and hence have different codes running.

From the programmers point of view, this approach yields simpler per core programs and reduced complexity. From an architectural point of view, the different cores may have different parameters, for example, lower clock rates, which may reduce power consumption. But, as the cores only serve certain limited purposes, latency and response time can still be lower than in the single MCU case. As embedded systems have to perform an increasing number of critical tasks, fault tolerance is becoming an ever more important issue. Another advantage of this multi-core approach and functional decomposition is that it allows the identification of crucial components, which may be duplicated for redundancy. For example, voting systems may be built in the design to guarantee that failure of a single component does not effect the rest of the system.

The outlined system concept has many associated challenges. Parallel architectures have already been extensively researched, but never widely employed in the resource-constrained embedded application field. The described reconfigurable computing-based architecture will have to support integration of hardware and software components. For the developer, the architecture has to provide a method to conveniently move functionality from soft cores to the hardware and back, if required. The assignment of these components to cores also has to be simple, with automated tools helping the developers finding feasible solutions.

*Soft-Core architecture*

As a first step towards a soft multi-core system, a general architecture concept was conceived. The key question is how the soft-cores and IP cores will communicate and synchronize. Communication of systems made up of modular components can be categorized either as loosely or tightly coupled. Systems with tightly coupled communication have their modules integrated in a very interdependent manner. For instance, in case of shared memory with blocking mutually exclusive access, the participating cores have to wait for resource access without being able to perform any useful computations in the meantime. Hence, code execution of one part is very much dependent on what other system segments are currently doing. This has the advantage that the whole system is inherently synchronized to a certain degree at all times. But this also means that certain processing steps cannot be instantly executed because of unrelated processes. For this reason, the tightly coupled approach is less suitable in our case. In order to be able to serve strictly timed peripherals, the loosely coupled concept with its independent and autonomous components seems more favorable.

According to this, an event-driven queue-based messaging architecture was developed, where each processing unit has dedicated data memory and dedicated program memory. As each core can be regarded as a separate individual unit, programming can be performed completely independently without affecting other cores. This independence is a crucial feature of the design, as separate cores are meant to handle individual hardware resources. The goal was to provide exclusive access to the soft core for the resource handling task and avoid costly context switches. To that end, communication with other cores is non blocking. To achieve this, a dedicated message queue and message delivery framework was necessary. Since communication involves a rather complex messaging procedure, delivery times may not be fully deterministic. Time-critical operations should be performed within the execution thread of a single dedicated core. Although this architecture concept imposes some restrictions on the design, the benefits outweigh the disadvantages as the clear partitioning of complex tasks helps the programmer to keep the overview of time-critical segments.

*Programming paradigm*

The network embedded systems C (nesC) language – widely employed in the Wireless Sensor Network (WSN) community – encapsulates sequential blocks of algorithms in autonomous components that have inner states and only interact through interfaces, which are well-defined sets of functions. This approach, along with the basic TinyOS framework, provides a development method using interchangeable components, thus, furthering modularity. This component-based approach fits the proposed multi-core architecture quite well, even though the concept was never meant for multi-core architectures and parallel execution.

The advantage of nesC is its highly modular approach that effectively hides the complexity of the underlying framework and hardware access. The developer only has to deal with top level components. This complicates the restructuring of existing single-core programs to multi-core architectures, as high-level components may have complex interdependencies not apparent at first sight. The goal is to support developers in designating top-level components for different processing units, and have an automated process check the feasibility of the assignment. If it is indeed a feasible solution, the assignment of all the components will be automatically generated by the proposed development tool.

The process can rely on the component containment and component interconnection information extracted by the nesC compiler. This is the key feature to provide rapid iteration through various designs, and as such, is tightly integrated with the development environment.

*Development and simulation*

In the early phases of the application-specific design, any choice of actual hardware may impose inherent limitations on overall system capabilities not immediately evident to the developer (especially in case of highly complicated systems), thereby reducing the solution space for the given problem and yielding suboptimal results. It is thus preferable to be able to test ideas in the least restrictive way but at high enough detail using a sophisticated simulation environment.

Typical simulation approaches either represent the system at the transaction level only or give detailed insight into only some severely limited parts. In order to be able to truly iterate towards an optimum, the developer has to be able to test concepts at various levels and at various points of the system, such as at the instruction level, hardware resource access, networking, etc. Also, to ease the migration of ideas from concepts to actual code running on hardware, the simulation framework has to support the testing of regular compiled binaries. One tool for embedded system simulation that satisfies all of the above described criteria, is the Avrora cycle-accurate embedded platform and sensor network simulator. However, the original software package lacks some crucial features necessary for the simulation of reconfigurable multi-core architectures. For instance, only a limited set of microcontrollers and architectures was supported, meaning that simulated platforms could only consist of a single central MCU and some connected Integrated Circuits (ICs). The framework has been extended to capture every aspect of multi-core system design with special focus on custom IP cores and the proposed queue-based messaging architecture.

**The structure of this document**

This thesis is organized as follows. Chapter II gives an overview on parallel and reconfigurable computing. In chapter III the message-queue based architecture for the novel embedded FPGA platforms is presented. Chapter IV presents nesC and the programming paradigms employed for the multi-core partitioning of applications. Chapter V discusses the simulation environment meant to enable rapid design iterations and in-depth program analysis. Chapter VI shows how the above described ideas were utilized on a SHM example. Finally, chapter VII concludes the thesis.

# CHAPTER II

# BACKGROUND

The purpose of this chapter is to give an overall view on parallel embedded development, the resource allocation problem, and examples for exiting systems. The discussion below is by no means exhaustive. These topics represent several decades of research in computer engineering, and a thorough study is well beyond the scope of this chapter. The goal here is to look at the essential aspects and concepts to convey an intuitive understanding. Also, the aim is to find out where some of the issues lie, so we can start coming up with well-founded hardware and software design choices.

## Concurrent and parallel architectures

This section discusses parallel embedded development with a focus on soft multi-core architectures and PLD utilization for CPSs [85, 56]. The feasibility of a concurrent design is summed up by Amdahl's law for parallel speedup:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

where $P$ is the proportion of the runtime that can be parallel executed on $N$ computational units resulting in a speedup of $S(N)$. The equation clarifies that parallel execution does not always yield performance increase. For example, depending on the problem, instead of many homogeneous processing units, it may be better to have less with a few high performance processors [8]. This is only with regards to speed, and the equation does not deal with the many other issues associated with actual implementation. For example, power consumption of the whole system was shown to have a complicated, non-linear relation with the number of units and clock rates. Slower and consequently less power hungry processors in parallel do not always require less energy than one high-power, fast processor [31].

*Parallel concepts*

There are many ways of approaching parallelism, which can be reduced to assignment of data to computing units and assignment of tasks to computing units [65]. No single assignment is optimal in all cases. Consider the following examples [78].

Fractal calculations, like the Mandelbrot set, can be trivially distributed among homogeneous computing units. The same code has to run on each computing unit with no data shared or exchanged during the entire operation. Only the boundaries of the space have to be specified, and results have to be gathered.

The N-body problem in astrophysics investigates how a system of point masses influence each other's motion. The difficulty is that gravitational forces act between any two objects at any time. Thus, if we divide the objects equally between units, we end up with a solution requiring a lot of data exchange. Computing units have to constantly query the position of the other units' objects. Here mutually exclusive shared memory can mitigate data transfer penalties, while all units can run the same code.

The Barnes-Hut N-body model is a simplification of the above problem. The effect of several objects in the far distance is modeled with a single combined mass [20]. The same or similar code may run on all computing unit, but space segments are associated with processors, not objects. Effects of combined masses in the distance mean low data transfers that allows message passing, as opposed to shared memory with its race condition issues.

The N-queens problem is a simple way to demonstrate unintentional, but unavoidable unbalanced workload partitioning. Find all the possible ways of placing N queens on a chessboard without any of them being able to capture any of the others. A solution is to recursively place queens on possible positions, and pass these chessboards to other processors to perform the same task. This goes on until N queens are placed or none can be. When distributing the task, the number of operations on a processing unit is not known a priori. The same code is executed, with minimal data exchange, but resulting in very inconsistent workload.

### Computing units

In essence, all forms of computations boil down to executions of individual operations. The instruction-stream-based traditional von Neumann general-purpose machine paradigm – and to a certain degree actual implementation of simple MCUs – follows the main steps of (i) (optionally) reading from memory to registers, (ii) performing operations on register contents writing results to registers, (iii) and finally (optionally) storing data in memory. All in order, sequentially executing one operation at a time, controlled by a program counter [70, 37]. This concept suffers severe limitations, which are overcome by smart techniques in actual products. Thus, implementations are by now completely detached from the programmer's model. Timing behavior is unpredictable making real-time computations complicated.

For instance, out-of-order execution and superscalar (multiple instructions issued simultaneously) techniques completely go against fundamental assumptions on the programmer's

side to exploit Instruction–Level Parallelism (ILP) in hopes of achieving some performance gains. These techniques require redundant rename registers in the order of hundreds for contemporary machines. This significantly increases the number of operation input sources and operation output sinks, which is a huge burden on wiring and multiplexing. These housekeeping, switching, administrative circuits consume more power, real estate, and time in contemporary high-performance processors than parts performing operations.

Current process technology is the most efficient – in terms of performance per joule and silicon area – for pipelined processors of five-to-eight stages [8]. Any effort to further increase performance will likely hit one or more of the following "walls" [7]:

- ILP wall: the lack of increase in discoverable parallelism in sequential program code to maintain high utilization of pipelined processors with multiple execution units.
- Power wall: the exponential increase of power consumption of Complementary Metal–Oxide–Semiconductor (CMOS) ICs with increasing clock rates. Decreasing device size may help, but eventually increases power consumption due to leakage currents.
- Memory wall: the growing gap between processor and memory speeds. Whereas memory access used to have no penalties and instruction execution took considerable amount of time, it is quite the other way around for contemporary devices.

The incentive to turn to multi-core solutions stemmed from this constantly and eventually sharply declining performance benefit associated with clock rate maximization and increased hardware complexity. [62] shows an example of a multi-core architecture with four simple cores being more efficient in terms of speed on the same die area than a wide-issue superscalar processor. Thus, even for personal computing, parallel concepts originally conceived in the '60s become a viable alternative [8].

Parallelism is very different for resource-constrained embedded systems. To meet strict requirements, high-throughput embedded systems rely on application-specific and adaptable computing. Speedup factors and power consumption reductions of up to four orders of magnitude for certain tasks my be achieved this way. This is an apparent paradox since the clock frequency is substantially lower, and for reconfigurable PLDs even other parameters (i.e., area, number of transistors for a single functional unit, etc.) are behind that of microprocessors – again by orders of magnitude [70].

*Reconfigurable computing*

The idea to have application-specific circuitry alongside more general-purpose computational units to form a heterogeneous parallel computer was first published in 1960 [30, 28, 27].

However, at the time of its inception, the technology was not ready and able to deliver useful prototypes. Those have only emerged in the '80s thanks to the constant progress of silicon technology. Contemporary off-the-shelf System on Chips (SoCs) architectures commonly include PLDs, which provide the necessary versatility for application-specific purposes [38, 61].

State of the art SoCs feature enough logic gates and block Random-Access Memories (RAMs) to implement complex operations and to support various IP cores and soft processing units. Some FPGAs come with complete DSP blocks, which are beneficial when forming the Arithmetic Logic Units (ALUs) of soft processors [19]. There are plenty of soft-core RTL designs available. Some are vendor specific and locked to their hardware, others are open-source and cross-platform. Most cores implement a well-known ISA, some cores strive to be fully compatible with existing hardware. Others extend existing ISAs with precision timing capabilities [49, 17, 16].

The RTL description of soft cores makes it possible to easily add additional register levels to the design. This is referred to as the C-slowing technique. It can improve throughput of digital circuits, if used in conjunction with retiming. The concept can also be utilized to increase the perceived number of independent digital circuits (instead of throughput), which is called System Hyper Pipelining (SHP) [77]. Reported examples show that the number of instantiated cores on the same fabric can be five times more. This makes the method exceptionally useful for soft multi-core SoC projects.

The software-based nature of cores also provides unique reconfigurability. The first main approach is the customization of instructions [73]. Using Application–Specific Instruction–set Processors (ASIPs) means the technique of adding and removing instructions as needed [54]. For example, application code can be first compiled with every possible instruction in mind. But subsequent scanning of the compiled binary will show what instructions are really used. Unused instructions can be discarded completely from the hardware [56]. If certain operations and algorithms are found to be used extensively, it might be worth implementing them in PLD hardware directly. For instance, a logarithm, using the COordinate Rotation DIgital Computer (CORDIC) iterative method, is far more efficient than a pure software equivalent.

The other approach is customization of processor structure. Conjoined architectures have been investigated, where some execution units were shared between cores. It was found that the technique saved fabric space and power, without having significant adverse effects on execution times [73]. Also, multiple versions of execution units (like multipliers) exist to chose from.

When distribution of tasks is considered, Asymmetric MultiProcessing (AMP) and Symmetric MultiProcessing (SMP) systems can be distinguished. The former has processors

dedicated to a limited set of tasks, while the latter treats processors as equal and allows any task on any processor [54]. AMP was typical of early systems. For instance, one computing unit was solely dedicated to the operating system and the other to user programs. Contemporary computers are dominantly SMP machines.

### Memory sharing

Having several computing units working together on shared problems necessitates some form of inter-core communication. If the main goal of communication is to signal an event, it is sufficient to simply change the state of an input pin. However, communication usually revolves around the transfer of more data than the processing unit can handle at once, thus processor accessible memory has to be involved in some form.

Architectures can be distributed or shared memory systems. With the former, each processor possesses exclusive memory. The contents of this memory is only accessible by others indirectly, i.e., by sending messages to the memory owner and explicitly asking. To transfer large chunks of data from one memory to the other, the Direct Memory Access (DMA) method is the straightforward well-known solution.

The second option of shared memory is effectively the opposite concept. Earlier architectures used shared memory with Uniform Memory Access (UMA), which means that memory address space had uniform access times for all units. Individual processors very likely used private caches, some systems even had extra memory dedicated to that single purpose, which is called a Cache Only Memory Architecture (COMA). On the contrary, newer approaches tend to be Non–Uniform Memory Access (NUMA) machines. The address space is still shared, but each unit has segments that it can access faster. This has the advantage of faster execution times due to code locality, but suffers from memory coherency and consistency issues, which are usually overcome with hardware mechanisms like the snooping protocols or directory-based protocols [37].

### Interconnections

Systems can be categorized either as loosely-coupled or tightly-coupled [70, 37]. Loosely-coupled multiprocessor systems refer to standalone computing units connected via a high speed communication layer, e.g., a fast local network. Contrarily, a tightly-coupled architecture is more integrated and interdependent. Processors often coexist in the same package, likely share memory, and are connected and synchronized at a low level. Tightly-coupled systems outperform their loosely-coupled counterparts in terms of energy efficiency, power consumption, size, and inter-core communication speed. However, loosely-coupled systems

are more flexible and support gradual replacements and upgrades of computing units. Hence, they do not require an initial high design investment and long development times. During normal operation, loosely-coupled parts also have less direct effect on each other, so the system is more resilient towards the failure of single parts.

The most straightforward connection type or topology is when every unit is directly connected to every other unit it needs to communicate with. The overhead of growing edge numbers and the number of wire crosses renders this approach only feasible for a small number of units. Instead, traditionally, bus type connections are utilized with a shared transmission medium. This has less wiring and is very useful for broadcasting, but presents the bus as a bottleneck. For instance, the Inter-Integrated Circuit ($I^2C$) is a very popular bus specification for embedded systems for interconnecting ICs, while AMBA and Wishbone are examples for on-chip buses [70].

Given a certain level of complexity, it makes sense to talk about a network of computing units. The assumption is that inter-core communication reaches a complexity that justifies the addition of a network layer. The designer has many degrees of freedom to form a network [34], with well-documented topologies like the Fat Tree [14, 48]. The additional feature requirements of networks (like routing) can introduce a significant burden on overall system power and area usage. The routing and buffers can occupy as much as 75% of total chip area, which can be reduced – without sacrificing throughput and delay – by employing bufferless routing [57].

Also, if connection types are examined with regards to throughput and delay, two distinctive approaches emerge. Circuit switching, as known from early wired telephone networking, provides a channel with fixed parameters between two nodes. Connectionless and connection-oriented packet switching are more flexible. But they do not provide a dedicated channel, do not guarantee parameters, and can even change data arrival order.

### Resource allocation and deployment

This section provides an overview of the fundamental aspects of resource allocation. The main question is that given a set of processing unit types, tasks, and constraints (e.g., certain components have to communicate with each other), how can tasks be mapped to computing nodes?

First some definitions. Regarding the time of the allocation, static and dynamic methods can be distinguished [70]. Dynamic allocation at run-time [74] is now very common with multi-core SMP Personal Computers (PCs). Static, compile time assignment is more typical of resource-constrained embedded machines [2, 85, 56, 60, 64]. In terms of who is responsible for the allocation, the two main approaches are centralized and distributed techniques.

Centralized means some sort of a manager-worker relationship, where the manager is most likely also responsible for load balancing. State of the art PCs operate this way. Distributed solutions are necessary, when no central management is possible. Here the worker nodes themselves divide the tasks among each other, as seen with the N-queens problem.

### The nature of the problem and solution strategies

The branch of mathematics that is dealing with finding (in some sense) optimal subsets of objects within finite sets is called combinatorial optimization. Resource allocation is a typical use case for this field, and consequently shares the same fundamental issue. The optimum very likely can not be expressed analytically, and an exhaustive search is practically impossible due to the size of the solution space. The way these problems are solved, is by employing iterative algorithms that converge (maybe only probabilistically) towards the optimum [71].

Efficient algorithms exist for certain set of well-known problems, like the Knapsack Problem (KP) and the Assignment Problem (AP). However, if researchers can not formulate their resource allocation tasks such that they resemble one of the well-known (and solved) problems, only suboptimal algorithms may be available. In most cases these problems tend to be NP-complete [47, 26, 63, 18] or even NP-hard [15, 6]. Thus, for quick (sub-optimal) results some form of general or problem-specific heuristics are employed, e.g., Markov Chain Monte Carlo (MCMC) random walks [4], Simulated Annealing (SA) [46], Genetic Algorithm (GA) [68, 6], and Particle Swarm Optimization (PSO) [63, 13]. A crucial assumption here is that it is possible to define a multiple input single output utility or cost function that is indicative of the quality of a configuration. Such a function can be employed to select the (in some sense) optimal point among all others [68].

#### *The goal of optimization*

Although single output cost functions are a very popular concept, not all optimization problems can or should be approached this way. For instance, a fundamentally different approach is called global $c$-approximation. Contrarily to a single scalar value, it is based on $n$ distinct objectives (each equally important) represented by a vector [43]. Different resource allocations are compared by comparing their vectors, which can be interpreted as points in the design space. Vector coordinates may be reordered and multiplied with a constant for certain problems, e.g., scheduling. Two allocations are considered equally good, if their respective coordinates are within a $c$ multiple of each other.

Also, it is not trivial to define single output cost functions. Domain-specific variations could reward shorter execution times or lower power consumption, but for certain problems more abstract notions have to be introduced [44]. Fairness and efficiency are among these ideas proposed for the mapping of tasks with heterogeneous requirements to cores with heterogeneous capabilities [41]. The authors of this paper present a generally applicable family of functions with only two parameters, which can favor fairness over efficiency, or the other way around. Furthermore, constraining said parameters will yield even more beneficial properties of the function and consequently the allocation itself [44].

In the unlikely case that the utility function is of a special form, like a convex or a linear expression, fast converging optimization algorithms can be applied. These are described in the field of convex optimization and linear programming, respectively [71, 6]. However, this usually requires a strong simplification of the problem [6].

If the design space can be depicted in such a way that the axes – corresponding to the dimensions of the space – indicate favorable properties of the system in a consistent manner (e.g., for a car the speed of the vehicle, where a higher value is always considered better), the notion of Pareto points can be introduced [72, 68]. This concept originates from economics, and is a useful tool to distinguish system configurations that represent meaningful trade-offs. A Pareto point is a configuration in the design space that is in some respect (that is to say along at least one axis) better than any other. When dealing with optimization, we are probably exclusively interested in these Pareto points – forming the so called Pareto front. This is because for any other regular point, there exists at least one configuration that is better in every way. Pareto points can be found using heuristics mentioned above, for instance.

With the Pareto front at hand, the question remains which Pareto point to choose? One approach is the above mentioned utility function method. But probably, the very reason, why Pareto points were calculated in the first place, was that no such function exists, or it is inconclusive when making this final decision. So, some other criterion is required, like Pareto dominance. Here the superiority of every Pareto point is judged based on how many out of all the regular points are in every respect worse.

### Related system and architecture examples

In this section, examples of existing systems, architectures are presented. These examples are meant to showcase (i) practical solutions applicable for the embedded domain and (ii) how software and hardware issues of constrained parallel architectures were tackled in the past.

### Early architectures

Parallel reconfigurable computing has been investigated for many decades. The early attempts were typically aimed at the improvement of mainframe level computers. The goal of research was to improve scientific calculations or data-intensive processing. As such, these architectures can not be considered embedded systems, but because of the immature technology, they had to face similar challenges, e.g., power constraints, low clock rates, and limited memory.

The reported earliest concept and architecture for reconfigurable parallel computing is the UCLA fixed-plus-variable (F+V) structure computer from the early '60s [30, 27, 28, 29]. The concept was to have a main processor control the reconfigurable hardware, which would then be tailored to perform a specific task, such as image processing or pattern matching. With the task executed, the hardware would be rearranged to perform some other task. This resulted in a hybrid computer structure combining the flexibility of software with the speed of hardware.

A parallel computing architecture from the '80s was the Connection Machine that had processors as many as 65,536 [22]. The original prototype had processors connected in a hypercubic arrangement, each had 4 kbit of RAM. Programming used a version of LISP and subsequentially C. It was mainly used for physical simulations.

Other early setup showcasing a different but very adaptable parallel platform concept built on the advancements of silicon technology was the Transputer [78, 1, 65] from the '80s. It was basically a set of expansion cards in a personal computer, with each transputer unit consisting of one processor, small memory, and four bidirectional channels, which could be used to hook up to other transputer units. Every unit had a unified 32 bit address space, containing Input/Output (IO), local, and global memory, latter much slower to access, with communication (through the four bidirectional channels) being memory mapped as well. Multiple threads were allowed to run on transputer units with hardware support for fast context switching. Software development was done with imperative languages, the same compiled code was loaded on every unit, each transputer selected appropriate execution branch according to its hard-coded node ID.

### Multi-core embedded architectures

Advancements in silicon process technology have reached the tipping point, where the small size of MCUs makes it possible to add additional cores to even the most resource-constrained designs. In fact, even if it was not originally intended, given a design with sophisticated contemporary ICs, it is very likely to end up with a multi-core architecture. Any kind of state of the art ICs probably already includes an MCU of some sort. For instance,

the CC2520 is advertised merely as an RF transceiver, but readily includes a computing core. Under these circumstances, it is a logical to partition the design into autonomous modules with a well-defined purpose. Hence, many projects result in stackable platforms with units dedicated to power handling, complex processing, sensing, or communication. The biggest difference is the number and interconnections of cores.

The straightforward approaches employ the available MCUs and interfaces, like PIC cores and $I^2C$ respectively [67].

Other research focuses on the design cycle of prototype, pilot, and production, with a reusable module consisting of CC2420 RF transceiver, MCU, and flash memory ICs. The parts are placed on a 1 inch by 1 inch board, with vias on the perimeter cut in half for easy integration. No communication interface, method, or protocol is specified, IC pins are directly wired to the perimeter connectors [24].

More complicated approaches have networks of stacked sensor nodes connected through busses, where individual nodes are running uC/OS-II real-time Operating System (OS) with a complete network stack (phy, link, net). In the traditional flexible node architecture resource control is centralized on a single high-power processor of some sort. But here, each resource in a MASS node is built into physically separable modules with supporting resource-specific processors. This makes it straightforward for developers to extend both the hardware and software architecture while preserving efficiency goals. Adapting to complicated requirements is a matter of connecting additional modules to the system [25].

Some architectures dedicate application-specific hardware to handle communication. That way the main processing unit is free to perform useful computation. An example for this would be the mPlatform, which has the usual stackable modules (CC2420 RF, MCU, power). Communication uses busses in a Time Division Multiple Access (TDMA) way with dedicated bus drivers implemented in Complex Programmable Logic Device (CPLD) providing a serial interface towards the processors [52].

Stackable sensor platforms, however, have inherent size constraints imposed by the packaging technology. Hence, in an effort to miniaturize, research is heading towards platform stacks built by only the layers of silicon and completely omitting the package. This results in complex architectures occupying space as small as a couple $mm^3$ [61].

As for the programming of such embedded systems, usually low-level solutions are employed, but higher-level languages were investigated as well. [66] shows an example of a complete JAVA Virtual Machine (VM) on a sensor platform. An Electrically Erasable Programmable Read-Only Memory (EEPROM) memory at a predefined external memory address stores software components that are loadable at run-time. If the overhead associated with the framework is acceptable this system offers unique plug'n'play capabilities.

**Soft multi-core architectures**

The application of soft-cores within reconfigurable PLDs is not entirely new. Many researchers have recognized the potential of such systems and have made significant contributions. In this section, FPGA-based architectures are discussed that combine MCUs and soft-cores. These are the multi-core systems that are the most similar to our proposed concept.

[85] is an example for a dataflow approach on a system on chip, with parallelism analysis at compile time, and scheduling based on worst case execution times. Resource allocation will have some components running in a single Altera Nios Reduced Instruction Set Computing (RISC) type core, while others in FPGA fabric. Connection of said components and the core are realized with a bus, the efficiency of which is assured with compile time bus arbitration analysis.

The main difference between this idea and the proposed approach is that in this case processing has to (i) strictly follow the Data–Flow Graph (DFG) Model of Computation (MoC) and uses static precalculated scheduling, and (ii) all inter-component communications use the same bus. Our approach envisions no such restrictions, as communication can be direct connection of components, and scheduling is not enforced (it is the responsibility of the application developer) allowing data drops and graceful degradation.

[56] again is an example for compile time analysis and resource allocation. Soft cores are generated for DFG nodes as needed, code dedicated to every core is analyzed, and superfluous processor subunits are removed. RAM and Read–Only Memory (ROM) contents are also generated.

The key differences from the work presented here are the development process, where the cycle accuracy of cores is guaranteed by simulating VHDL code with hardware acceleration, as opposed to the fast software only solution discussed in later sections.

Other works advocating the favorable properties of reconfigurable fabric interconnected with several processors include the ReMAP architecture [82], where fabric shared among cores is employed for flexible inter-process communication and processing resulting in a graph like program structure, once more reminiscent of the dataflow idea. The architecture showed improved computational performance in simulation.

Difference compared to the proposed work is the fundamental concept of fabric utilization. The ReMAP architecture uses fabric primarily as a means for communication among physical cores with additional simple computations. Our approach uses the fabric much more freely, with no such limitations.

Other research used FPGA fabric exclusively to implement a soft multi-core system of four SPARC processors connected with a 64 bit bus, with each core running a complete

OS. Shared memory model was used for communication using the OpenMP Application Programming Interface (API), and platform capabilities were tested with GSM and ADPCM protocol implementations [50]. In contrast, our work does not limit core interconnections to a bus, and assumes a much less resource-intensive OS more applicable for lightweight embedded controller systems.

In [40], the authors argue in favor of standardized virtual fabric on top of real FPGA fabric in conjunction with a hardware microcontroller that is capable of dynamically reconfiguring it during run-time for application-specific processing tasks. The paper focuses on the efficient run-time reconfiguration of virtual fabric for different algorithms, and thus envisions a more hardware like approach for computation, as opposed to our work, where hardware and software components are treated as equal.

[76] is a research focused on the fault tolerant connection of multiple physical cores and fabric components within the FPGA utilizing a bus structure with built in cache memories. The cache-based communication is similar to the concept presented in this document. But the authors focus on the fault tolerant aspect of communication, and only consider bus type connection, which due to arbitration, may in fact be suboptimal in some cases.

[75] shows an example for a more complicated solution where the processor runs a complete Linux-based operating system called BORPH. It can compile, synthesize, and upload configurations into the fabric on-the-fly, which are then treated like regular processes. Command line tools can be used to stream data into these hardware processes, and their output can be piped to the input of other software or hardware processes. If two such processes are connected directly, the operating system takes care that data is streamed only directly between the two components in hardware.

Of course, seriously power constrained systems are not capable of running an OS this sophisticated performing the above mentioned complicated tasks. Also, the pipelining approach limits the type of topologies that can be efficiently created.

Also Linux based is the ATLAS platform [83] developed mainly to research the Transactional Memory (TM) concept, with a main processor running the OS and several soft cores on several FPGAs running parallel threads of the system. The system is meant to be a testbed and includes frameworks for profiling and detection of violations, overflows, and bottlenecks of multi-threaded programs.

This system is very unlike the previously described ones, as it aims to be a hardware simulator. It is thus inapplicable for embedded systems, although the concept of in-fabric instantiated interconnected cores is also present here.

# CHAPTER III

# HARDWARE ARCHITECTURE

The goal of this chapter is to discuss how recent hardware development results can be utilized to build more efficient soft multi-core parallel embedded systems with a focus on WSNs. A state-of-the-art soft multi-core approach with the appropriate programming model seems a promising, new alternative to the well established MCU architectures. The design problem is shifted towards a combined hardware and software development, where the hardware is not firmly specified, but instead the needs of a particular application determine the configuration.

## Analysis

Simple embedded systems are typically implemented on single-board microcontroller-based hardware. These systems may also include application-specific highly-integrated ICs, e.g., RF transceivers, Inertial Measurement Units (IMUs), optical sensors, actuator drivers, or other complex peripherals. There are many different commercial MCUs and they all support this architecture with readily available reference boards, evaluation kits, and corresponding development environments. The abundance of available solutions gave rise to the rapid development of a large variety of embedded devices. However, the conventional approach with a single simple MCU has unavoidable shortcomings.

For instance, very high clock rates may be necessary if all peripherals are to be served in a timely manner. A failure to increase the clock rate sufficiently can lead to deadline misses and interrupt misses. From a programming point of view, the growing number of peripherals and tasks increase the overall code complexity as well. The single execution thread, which has to deal with several peripherals and consequently interrupts, will have to go through several context switches while serving nested interrupts. This will add significant jitter to interrupt handling times. Also, developers have to be aware of the effects of several interrupt handling routines being executed in a single thread – assuming that the necessary clock rate could be properly determined beforehand.

Tendencies for embedded computing and sensor nodes point to more peripherals and more processing power requirements. Applications, which require higher computational power than state-of-the-art MCUs can provide, need to choose another direction. It has been long known that application-specific processors have better computational properties than

general purpose processors. The two main alternatives are DSPs (or other high performance microprocessors) and PLDs predominantly in the form of FPGAs. The latter category represents the more suitable choice for embedded systems, as it supports the instantiation of a variety of arbitrary digital circuits and easy reprogramming by the users. This could mean various soft MCU cores and interconnection networks within a single IC.

However, FPGA platforms face serious issues for duty cycling – one of the most important power saving feature – as every restart of their circuitry is penalized by high inrush currents and thus high power consumption. This is due to the unconfigured state of the fabric at startup, which only gets setup once the circuit configuration is loaded from an external Static Random–Access Memory (SRAM). On the other hand, flash-based FPGA technology is a feasible solution even from the power consumption point of view. As opposed to conventional devices, it stores the configuration on chip, so that it is always available right form the startup. Thus, resets and duty cycling are no cause for increased power demand, and novel flash FPGA platforms can be designed.

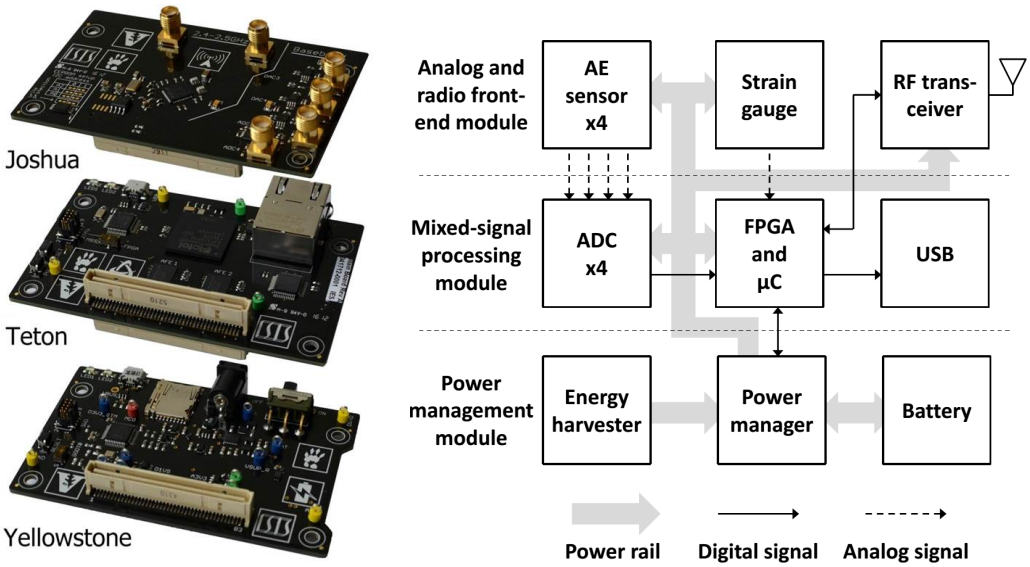
By utilizing flash FPGAs a true parallel, soft multi-core approach can be embraced. The fundamental idea is to have largely independent, parallel running cores dedicated to a single or a limited set of peripherals and tasks – possibly with real-time requirements. High-throughput, time sensitive processing steps can be taken care of by application-specific digital circuits with shorter, more deterministic run-times. This setup is especially beneficial for dataflow type computations, like DSP, however, this is not merely a dataflow problem. The architecture addresses the more general field of high-throughput applications with partial real-time requirements. For example, a system with a RF IC very likely has strict deadlines to meet in order to successfully communicate, while at the same time other aspects of the operation may not have such constraints. It is important to emphasize that end-to-end real-time requirements of the whole system are not addressed here. The focus is on the real-time requirements of certain independent, limited tasks.

The benefits of this architecture are reduced complexity and simpler programs. The independent parts are much more predictable and exhibit a more easily understandable behavior, while still providing lower latency and shorter response times. Also, cores can run at their own optimal speeds forming a Globally Asynchronous Locally Synchronous (GALS) system. In fact, above a certain complexity and clock rate this is a necessity, as global synchrony is not possible. The GALS design can achieve power savings due to two main factors. Firstly, computing cores can choose lower clock rates, secondly, power dissipation due to routing a global clock signal across IC does not exist. The clock distribution network is significantly easier to design at the local level. Global clock skew and slew rate issues are also alleviated since the design complexity of each synchronous block is reduced and easier

to manage. Furthermore, the presence of several independent clocks can reduce switching noise, and GALS systems also tend to be more resilient towards ElectroMagnetic Interference (EMI) [69]. In addition, malfunctioning of a single core does not directly influence the rest of the system, and critical functionality can even be redundantly distributed on multiple cores, providing fault tolerance.

### MarmotE platform example

In the following, our custom-designed, universal, low-power, multi-channel, wireless sensor node called MarmotE is briefly presented [12], see Figure 1a, to serve as an example for the above described novel flash FPGA platform concept. The primary driving force behind the hardware design was to create a general and flexible WSN research platform. Our most important goal was to enable experimentation with power saving techniques (such as energy harvesting), various analog sensor and radio front-ends, reconfigurable processing (such as cross-layer optimization for RF communication), and embedded multi-core computing approaches.




**(a)** Modular layers. **(b)** Simplified block diagram.

**Figure 1:** MarmotE sensor platform.

The platform follows a modular layered approach, and can be physically and logically divided into three parts, see Figure 1b. The bottom layer manages energy, featuring power monitoring and interfaces for batteries, wall power, and other sources, e.g., energy harvesting units. The middle layer is responsible for domain conversion, digital processing based on the SmartFusion flash FPGA, and high-speed connectivity such as a Universal Serial Bus

(USB) or Ethernet. The application-specific front-end layer has baseband amplifiers and carries a RF chip for wireless communication. The stacked architecture makes it possible to seamlessly replace the top-layer radio front-end and the bottom-layer power supply modules, while keeping the same mixed-signal processing module intact.

The current top-layer module, named Joshua, is an analog front-end designed to operate in baseband, as well as in the 2.4 to 2.5 GHz Industrial, Scientific, and Medical (ISM) frequency band. It interfaces with the middle-layer module through pairs of analog baseband In-phase and Quadrature (IQ) signals, both for transmission and reception. For RF research, the board allows experimentation with various types of channel access methods and modulation techniques. It is built primarily around the integrated Maxim MAX2830 RF transceiver supporting both single and dual-antenna setups. The RF transceiver has outstanding properties. For example, on the receiver side, the IC is capable of amplifying, downconverting, and filtering with a noise figure of only 2.6 dB, and, based on the Signal–to–Noise Ratio (SNR) requirements of a 54 $\frac{\text{Mbit}}{\text{s}}$ Orthogonal Frequency–Division Multiplexing (OFDM) WiFi signal, the receiver sensitivity is as low as -76 dBm.

The middle-layer, called Teton, is a mixed-signal processing module. It controls the top-layer and provides computational resources for rudimentary baseband signal processing. The basis of the module is a flash FPGA-based Microsemi A2F500 SmartFusion SoC and two external Maxim MAX19706 domain converters that can simultaneously process two sets of analog baseband IQ signal pairs. Each IQ Rx and Tx pair is connected to the 10 bit ADCs and DACs, respectively. Interfacing with two sets of baseband signals renders the Teton board suitable for even Multiple-Input and Multiple-Output (MIMO) application development. The most important guiding design concept here was the lack of space for sharp analog anti-aliasing and reconstruction filters, which meant oversampling and subsequent digital filtering for proper channel selection. This is well supported by the 22 MHz sampling rate of the ICs.

The bottom-layer module, named Yellowstone, is a power manager designed around a low-power microcontroller to regulate and monitor the power rails of the MarmotE platform. It powers the entire stack and measures and logs current draw, along with battery status. The module has three possible sources of power, a 5 V wall adapter, a USB connector, and a Li-Ion battery. The former two are used both to power the voltage regulators and to charge the battery. A step-down regulator controls the 1.5 V rail, while a low-dropout regulator is used on the 3.3 V rail, primarily supplying the core and the IO blocks of the SmartFusion SoC, respectively.

**AVR HP Soft core example**

The AVR HP is an example for a soft processor that may be employed in a soft multi-core architecture. It is a hyper pipelined (thus the HP extension in the name) version of an AVR type processor written in Verilog. The AVR platform in general is a Harvard architecture and RISC type processor. The AVR HP was chosen from a wide variety of contemporary available soft cores with vastly different capabilities and properties. The parameters of the AVR HP provide a good insight into the requirements of soft cores.

This soft processor has several benefits. It has readily available compilers, like avr-gcc that works on linux and windows alike. It is an actual hardware microcontroller family with many embedded systems and wireless sensors using it. Also, a simulator, called Avrora [80], is available for the platform with support for peripherals (like RF chips) and communication simulation.

The AVR is an 8 bit architecture with a two stage, single level pipeline design. In other words, the next instruction fetching and the current instruction execution takes place simultaneously. Most instructions take just one or two clock cycles, and usually clock rates of up to 20 MHz are supported. The AVR ISA is considered more orthogonal than the available alternatives. The AVR HP soft-core is compatible with this instruction set meant for classic cores.

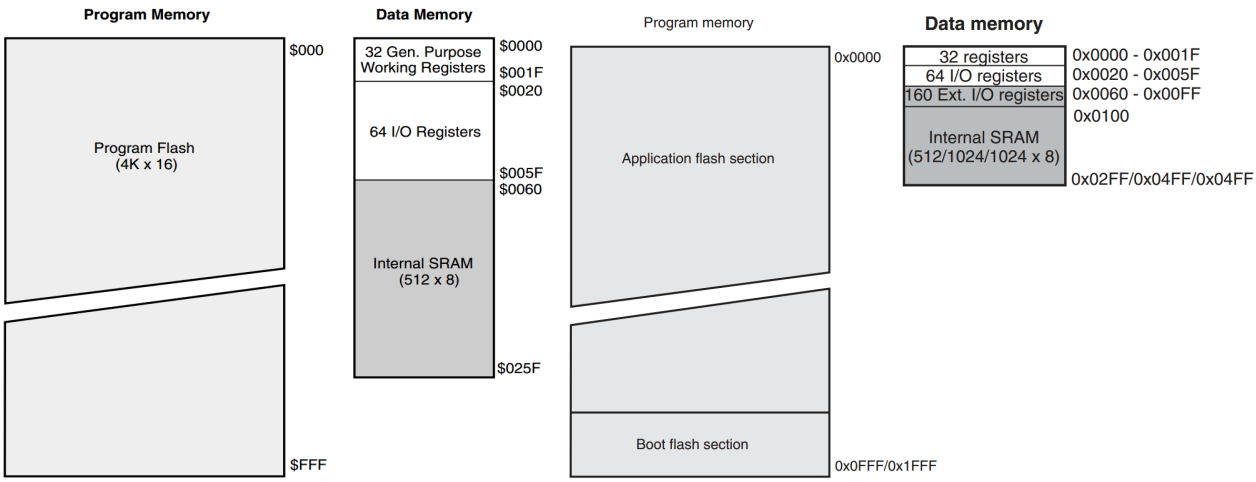
Program memory (flash ROM) is typically less than 64 kB for the smallest MCU types. Data memory (SRAM) is also severely limited. Table 1 shows the maximum memory sizes (based on address widths) compared to actual usual memory sizes.

| | Address width [bit] | Data width [bit] | Size [kB] | Max size [kB] |
|---|---|---|---|---|
| Program memory | 12 | 16 | 8 | 8 |
| Data memory | 16 | 8 | 2 | 64 |

**Table 1:** Usual and possible total memory sizes for small AVR MCUs.

The data address space is further restricted as it also incorporates the memory mapped register file and IO registers. Figure 2 shows the memory maps and layout of typical AVR MCUs with separate data and program memory.

AVRs have 32 single-byte registers. In most cases, the working registers are memory mapped to the first 32 memory addresses followed by 64 IO registers. The actual SRAM starts after these sections, however, for devices with more IO capabilities more than 64 addresses are reserved at the lower addresses. Most chips have a third type of memory as well, a built-in EEPROM for persistent data storage, but that is usually not completely memory mapped, and is treated like a peripheral.




**(a)** AT90s8535 **(b)** ATmega88

**Figure 2:** Memory maps of two AVR microcontrollers copied from their respective datasheets, [9] and [11].

*Synthesis bottlenecks*

Synthesis of an AVR compatible MCU on a FPGA has to address two main issues. As opposed to their large counterparts, FPGAs viable in embedded systems offer limited amounts of resources. Still, both the digital logic and the memory requirements of the MCU have to fit the FPGA. Obviously, the memory requirements should be preferably satisfied by on-chip block memory. Employing fabric for this purpose would be wasteful, and external memory may not be an option on small embedded systems. The Microsemi A2F500 SmartFusion SoC (utilized in our MarmotE platform) features 24 4 kB block memories. An alternative, the Spartan-6 XC6SLX16, is equipped with 32 block memories each 2 kB in size, supporting true dual port access. To be precise, the Spartan device is a conventional SRAM device, mostly used in the embedded environment. As such, it does not represent the flash FPGA technology, nevertheless, it is an informative example in terms of FPGA parameters.

Digital logic is implemented in fabric, and thus, the key question here is how many of the MCUs may be instantiated. If the cores are the same, it is easy to see that certain parts are instantiated redundantly several times without actually being fully utilized. It is this realization that sparked interest in conjoined processor architectures, where certain operational units, i.e., multiplier, divider, etc., may be shared among many concurrent cores. In a sense, the most advanced form of this sharing is SHP, where every part of the processor is shared, except for core registers, which store the current state of the processor. The advantage of such an approach is of-course a better fabric utilization. The AVR HP soft-core under discussion is designed along these lines. Synthesis and place-and-route results indicate that up to around 10 HP AVR cores could fit in the fabric of a Spartan-6 XC6SLX16.

In an actual deployment, this number may vary due to fabric requirements of other parts of the system, e.g., application-specific IP cores, messaging framework, sensor drivers, bus adapters, etc.

The conclusion of this comparison is that for small FPGAs in embedded systems both the available block memory and fabric present a hard limit on the number of soft-cores.

## Soft multi-core architecture

In this section, the architecture is introduced that was designed to address the above stated issues. The guiding design philosophy was to be as non-intrusive and transparent from the developer's point of view as possible. Ideally, this multi-core architecture can seamlessly run code developed for single MCU systems.

*Sense and Forward application example*

In order to design a useful architecture, however, first the application field has to be understood, and the type of parallelism at hand has to be determined. A well-known example in the field of WSNs is examined to help with the design considerations for the proposed architecture concept.

The "Sense and Forward" application illustrates some of the major tasks of WSNs. The embedded system has to measure some value in a periodic manner, it then has to send a radio message to the rest of the network based on the measurement results. Since the communication is expected to be cooperative, nodes are also required to intercept each other's radio messages and process and retransmit them as needed, thus forming a multi-hop network.

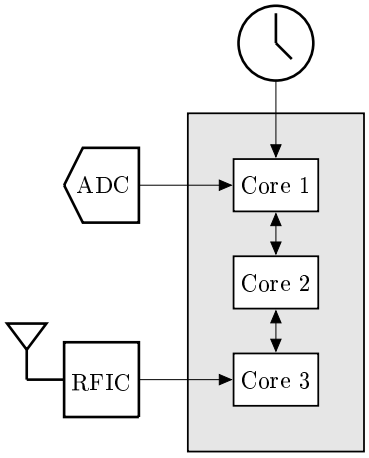



**Figure 3:** Multi-core sense and forward application concept.

Figure 3 shows a possible multi-core partitioning of the problem. A sensor's value is read with an ADC at periodic time intervals guaranteed by a hardware timer on Core 1. After some preprocessing and evaluation, it is sent to Core 2, which is then responsible for further processing. Core 2 sends the result, which is going to be radio transmitted, to Core 3. Core 3 takes care of all aspect of the radio communication. If a radio message is received, that will also be processed on Core 3, and subsequently sent to Core 2 where after some more processing a response is generated. Finally, the response is sent back to Core 3, and gets transmitted. Core 1 and Core 3 are independent and have no direct influence on each other; Core 2 depends on the input of the other two cores.

The example resembles the parallelism described at the Barnes-Hut N-body model example. Similar code may (but not necessarily) run on multiple cores. No raw unprocessed data streams are sent between cores; low data transfer rates allow message passing – as opposed to shared memory with its race condition issues.

*Assumptions and constraints*

Some basic assumptions and constraints have to be made. I will assume that the embedded system, built using the above outlined FPGA platform, can have a set of (i) peripheral soft cores, dedicated for IO handling, (ii) and internal soft cores, alongside components in fabric, for processing.

The number of cores is limited to a fairly low number – on the order of ten. Thus, every core can be directly connected to those it has to communicate with. Alternatively a bus system may be employed [76], but is very likely not necessary. Resource allocation and deployment, scheduling, and optional load balancing can be centralized to a manager, there is no need for distributed solutions.

As discussed above, very limited amount of memory is available, typically only a few kB, which has far reaching consequences. In a 32 bit or more likely 16 bit data address space, there is plenty of room to memory map peripherals. As for the code, only small programs may fit into the individual program memory of each core.

**Evolution of ideas**

The guiding principle is that the user should be able to start out from a single-core project, and employ it in a multi-core environment. The architecture has to support this transition as efficiently as possible. The main issue is the access to the limited memory available. Memory is going to be used for data and program code storage. The program code is less important, because the Harvard architecture separates it from the data memory

space, and hence it can be in a read-only block. This immediately eliminates the issues of mutually exclusive access. A core's data memory space, on the other hand, is different, as it may be altered by the owner and other cores – directly or indirectly during communication.

*I.*

The simplest approach is if communicating cores are directly connected, with no low-level buffering. One core transmits data, while the receiver stores and processes.

**Problem**: This results in a tightly coupled system, with all its drawbacks. Cores have to be synchronized, and they have to be simultaneously in communication mode using up clock cycles to handle the transmission. No meaningful processing or task execution can take place during this time.

*II.*

A better approach for our purposes is a more loosely coupled connection, where communication does not require tight synchronization. This may be achieved by dedicating a buffer for incoming data at every core. This solution has hardware and software aspects to it. On the hardware side, memory is required, which is going to be the primary limiting resource. Also, some form of hardware supported, mutually exclusive access mechanism has to be in place to resolve coinciding resource usage. On the software side, the resource access may be blocking or non-blocking. The latter is preferred, otherwise parallelism can not be exploited to its fullest. In this case, the outcome of the access attempt has to be propagated up to the software level, and has to be handled there.

**Problem**: The question is where the line is drawn between the hardware and software. Which parts of the mutually exclusive access mechanism are implemented in hardware and software?

*III.*

The concept can be to minimize hardware complexity, and only have minimal hardware support (e.g., with semaphores). This way the burden of proper resource access would be carried by the software.

**Problem**: Although this approach is more flexible, it requires severe changes in the software architecture. These changes are not just a couple lines indicating the locking of a resource, but a whole conceptual change moving from a single-threaded to a multi-threaded paradigm.

*IV.*

A solution where the whole of the data transfer process is supported by hardware seems more viable. In this architecture the software is only responsible for initiating a transfer and handling the response. In this context the response indicates whether the transfer was successful or not, which still has to be handled by the program.

**Problem**: Depending on the type of resource, this concept can still result in complex software. For example, a shared data memory abstraction means that the software would have to determine how to find and allocate empty spaces. On the receiver side, the software would have to subsequently determine the order and priority of data placed there. Thus, the final solution is simple First In, First Out (FIFO) message queue-based concept.

### Final architecture

The proposed final hardware architecture can be categorized as a loosely coupled, circuit switched, AMP system made up of Harvard type cores, see Figure 4. Cores can be connected to hardware peripherals or connected to special purpose processing blocks implemented within fabric. The key part of the architecture is the introduction of a queue-based messaging framework for inter-core communication, which utilizes both FPGA fabric and block memories, and has associated components in software as well. The latter are introduced in chapter IV.

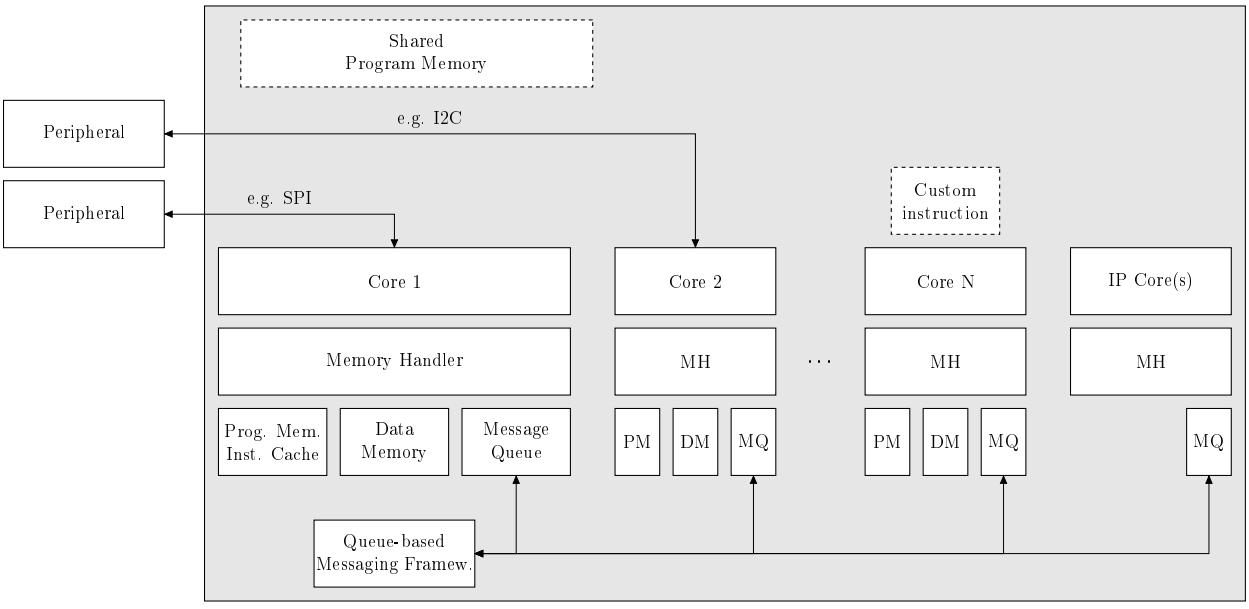



**Figure 4:** Soft multi-core architecture. Inter-core communication is exclusively done using Queue-Based Messaging Framework.

Cores are assigned individual block memories, which are divided into three parts:

- Program memory or – if the code is to big to fit – instruction cache (completely transparent for the core)
- Data memory (including static and dynamic data)
- Message queue (one memory mapped message queue for every core)

For example, a theoretical allocation of block memories for eight cores could designate three pieces of 2 kB block memories to each. One block for instruction cache or program memory, data memory, and message queue, respectively, which would use altogether 24 memory blocks. Remaining blocks, for instance, $6 \times 2\,\text{kB} = 12\,\text{kB}$ in case of the Spartan device, can be used for miscellaneous purposes. Table 2 shows the memory address sizes for this arrangement.

| | Address width [bit] | Data width [bit] |
|---|---|---|
| Instruction Cache/Program memory | (5+)7 | 16 |
| Data Memory | (8+)8 | 8 |

**Table 2:** Possible address space and data size when utilizing 2 kB block memories.

Data memory is the simplest to describe in this setup. It is distributed and not shared among cores, and (in order to avoid race conditions) other processors can only influence the content of it indirectly through messages.

As for the program memory, depending on the compiled code size, two approaches are possible. The first method has a single (or very few programs) compiled and loaded into a shared, read-only memory area created from the remaining miscellaneous block memories. The cores then load the relevant parts into their respective instruction caches. In other words, this is a COMA approach in terms of program memory. Alternatively, if individual program memories are large enough to hold the whole code, the binaries may be loaded directly into individual program memories.

The main difference between this and desktop multi-core systems is that latter have the shared memory in a NUMA arrangement. NUMA has the usual problems of memory coherence and consistency, but here these issues are non-existent, since the shared program memory is read-only.

Within this framework each core and IP core has a dedicated hardware message queue also utilizing block memory. Inter-core communication is exclusively done by sending short messages to these queues. The queues can hold messages of arbitrary format, but in its simplest form, function pointers (because of their uniqueness) along with parameters can be used.

With such a memory setup, cores likely do not directly connect to block memories, but indirectly through some form of a multi-purpose memory handler. It takes care of loading the instruction cache (if shared program memory is employed). It could pre-cache program code from shared memory while the cores are in reset. Also, the memory handler maps the message queue seamlessly into the data memory address space.

The clock of the memory handler and block memories can be much faster than the clock of the cores. This way memory transactions do not stall core execution. Also, the handler and memory blocks preferably have to support dual port access, meaning that processes can modify memory content independently on two ports in one clock edge – assuming they do not work on exactly the same memory cell. This is especially critical for the message queues, because with dual port access it becomes possible to read the next message from the queue and insert a new message at the same time.

Asynchronous or parallel execution is achieved by putting messages in the core's message queue, with a process running on the core solely dedicated to sequentially removing said messages, and running associated functions. The queue write operation is a cooperation of different memory handlers.

The queue-based messaging framework provides rudimentary routing as well, but because communicating cores are expected to be hard wired in a direct point-to-point manner to each others' queues, the routing problem is reduced to mere switching. This has the advantage that the hardware parts of the framework become simple logic functions, and hence, less fabric is required when synthesized.

In-fact the limiting factor of the framework is the available block memory and not the fabric. For instance, message queues have to be big enough to not overflow and hold all the messages throughout the normal operation of device. In depth analysis is unavoidable for obtaining the exact required minimum queue and memory sizes for a given application. The Avrora simulator, described in chapter V, can be employed for this step. Table 3 shows examples for possible message sizes in a 2 kB block memory queue.

| Max messages in queue [pcs] | Message size = header + payload [B] |
|---|---|
| 256 | 8=4+4 |
| 128 | 16=4+12 |
| 64 | 32=4+28 |
| 32 | 64=4+60 |
| 16 | 128=4+124 |

**Table 3:** Examples for maximum message numbers in a 2 kB queue.

*Further possibilities*

There are many optional ways to extend the capabilities of this architecture depending on the requirements. For instance, peripheral soft-cores can optionally be mapped to several IO pins handling critical events. The advantage is that for such pins, parameters like reliability can be improved.

Also, soft-cores may be extended with application-specific instructions in the fabric. The AVR ISA already has an instruction hierarchy in which only the most fundamental instructions are supported on every AVR MCU; more complex operations are only provided by the more complex MCUs.

A further possibility is that memory handlers may be sophisticated enough to handle certain interrupts autonomously. They could be extended to generate and push messages into the queue alleviating the real-time interrupt handling burden of processors.

**Multi-core MicaZ**

The MicaZ is a 2.4 GHz IEEE 802.15.4 compliant RF mote module used for enabling low-power WSNs. It is a well-known platform widely used for various purposes. Also, it is well supported; both the TinyOS framework and the Avrora simulator provide support for the device out of the box. Thus, when devising a multi-core capable platform, the MicaZ mote was chosen as the starting point.

The goal was to create a platform that is backward compatible, yet can be easily extended for multi-core applications. The multi-core MicaZ is a virtual sensor platform in the Avrora simulator that features all the components of a normal MicaZ. But, instead of a single ATMega128L AVR MCU, it has several ATMega128L MCUs interconnected using message queues. The first core has the exclusive access to the platform's regular peripherals. All other cores only have individual, exclusive timer peripherals. Due to the backward compatibility, development for the platform is very simple; existing frameworks and tools can be used.

The platform was meant to be as non-restrictive as possible to support design space exploration. Hence, parameter upper limits can go beyond the above outlined restrictions. For instance, no shared program memory is required, as MCUs have sufficient program memory to hold code. So, using the nesC compiler, binaries can be generated that can be directly loaded onto any of the cores of the platform. Also, it can have an arbitrary number of cores, up to 256, and by default every core is connected to every other core, and thus, has access to all message queues of the system. The queue-based messaging framework within the simulator is simulated at the transaction level, and so message transmission times can be arbitrarily chosen, non-zero values.

*Message queues*

To maintain backward compatibility, the queue-based messaging framework was added in such a way that it does not interfere with already existing system parts.

Using the memory handler, the cores are able to push messages into the queue of any other core, even into their own. The queue size is limited to ten entries for the multi-core MicaZ platform and can be extended if need be. The messaging passing process employs a request-reply approach. The transmitter first requests an empty slot in the message queue, to which the receiver replies whether the request was granted or not. If granted, the transmitter sends the actual message.

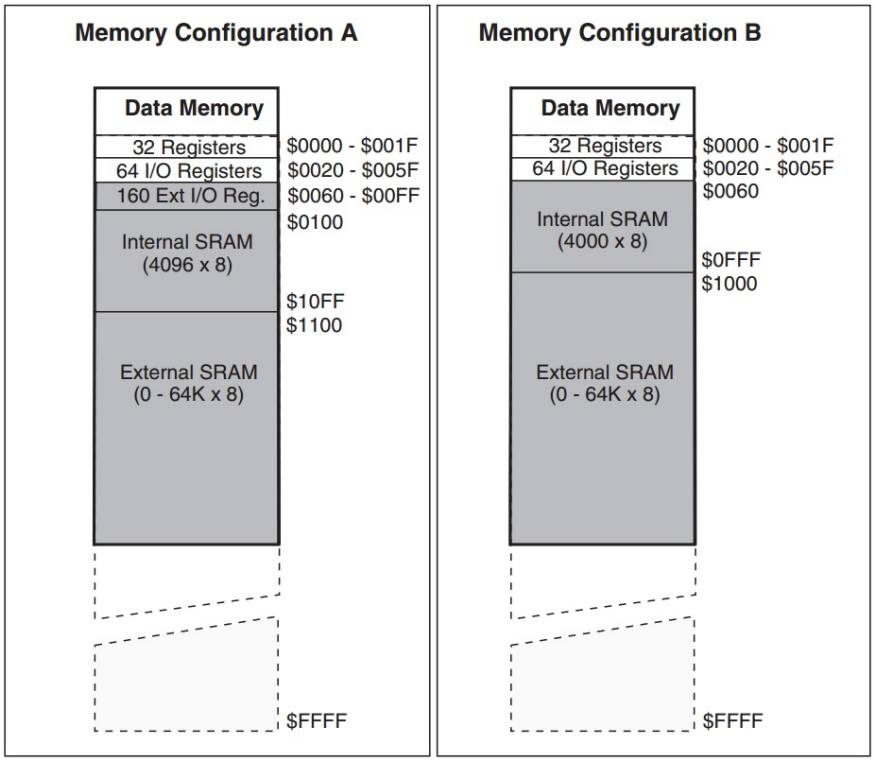



**Figure 5:** The two memory modes of the ATMega128L. Copied from the datasheet [10].

The receivers and transmitters are memory mapped for the ATMega128L cores. The ATMega128L has two memory modes [10], see Figure 5. The main difference is the lack of 160 IO registers in memory configuration B – which is the so called ATmega 103 compatibility mode.

An obvious choice to map the message queue would be the 64 IO registers, which can be found in both memory configuration modes at the same location, but unfortunately that region is too small to carry all the necessary registers. Queues could be mapped as external memory as well, but there the interface is pin-based, thus simulation would have to be performed at bit/pin level. Internal memory could be utilized for the memory mapping, but

there is not too much of that available, and it is not known how it is used in some programs. There may be even hard-coded addresses in a program code, which might interfere with the memory mapped registers, that would instantly break backward compatibility. Also internal SRAM is used for other purposes, like stack pointer, which can be set in code to point anywhere in SRAM.

| | Reg Name | | Address | Flags Name | Flags Bit # | Type | Description |
|---|---|---|---|---|---|---|---|
| RX | STATUS | | 0x009E | RDY | 0 | R | 1 indicates one or several new messages in the buffer, resets to 0 if message queue empty. |
| | CTRL | | 0x009F | POP | 0 | W | Any write to this bit pops the first message (and source ID) form the message queue. |
| | SOURCE* | | 0x00A0 | NA | | R | Source ID/number of the transmitter the message originated from. |
| | MSG | _0 | 0x00A1 | NA | | R | 1st byte of RX message buffer. |
| | | ⋮ | ⋮ | ⋮ | | ⋮ | ⋮ |
| | | _31 | 0x00C0 | NA | | R | 32nd byte of RX message buffer. |
| TX | STATUS** | | 0x00C1 | BUSY | 0 | R | 1 indicates that the message buffer is already in use. |
| | | | | TRNSF | 1 | R | 1 indicates that the contents of TX buffer are being transmitted. |
| | DEST* | | 0x00C2 | NA | | W | By writing a RX queue's ID here the buffer content transfer is initiated. |
| | ALLOC | | 0x00C3 | SCS | 0 | R | 1 indicates a successfull allocation of the message buffer, and sets the "BUSY" bit. |
| | MSG | _0 | 0x00C4 | NA | | W | 1st byte of TX message buffer. |
| | | ⋮ | ⋮ | ⋮ | | ⋮ | ⋮ |
| | | _31 | 0x00E3 | NA | | W | 32nd byte of TX message buffer. |

**Table 4:** Description of message queue registers.
* Destination/source ID are NOT globally unique, but separately definied for every core.
** Initiating the transfer will automatically reset these bits to 0.

Hence the 160 IO registers in memory configuration B are the only feasible location. A significant part of that is unused, but still not enough to memory map the whole message queue. Only the first entry in the incoming queue is available, and only one entry for outgoing messages is memory mapped. Also, this introduces some limitations on the message sizes,

the upper limit was set to 32 bytes. The memory mapped registers with description can be found in Table 4.

The usage of these registers is rather simple, an example pseudo code for a minimalistic message transmission and reception application can be seen on Figure 6 and 7 respectively.

```
1 if TX ALLOC register SCS bit == 0 then
2 |  return, message buffer allocation failed
3 end
4 write message bytes into TX MSG registers
5 write message destination ID into TX DEST register (starts transfer)
```

**Figure 6:** Pseudocode example for a simple transmission.

```
1 if RX STATUS register RDY bit == 0 then
2 |  return, no new message
3 end
4 read message bytes from RX MSG registers
5 read message source ID from RX SOURCE register
6 write RX CTRL register POP bit (removes this message)
```

**Figure 7:** Pseudocode example for a simple reception.

# CHAPTER IV

# PROGRAMMING PARADIGM

This chapter introduces the nesC programming language and the related TinyOS framework, and shows how they may be employed in the soft multi-core embedded environment.

The soft multi-core architecture outlined in chapter III was not designed with any particular programming language or paradigm in mind. It is language-agnostic, and from the architecture's point of view, the only criterion for the language is that it should support message passing. Hence, the choice of the language is rather decided based on development related issues.

The ultimate goal is to augment the single core programming approach so that it enables programmers to easily distribute applications on many cores. Ideally, existing single-core software projects should be fairly simple to migrate to the new multi-core architecture. The resulting new multi-core application then should have improved properties compared to the old one, particularly with regards to reduced event misses. It is this aspect of the development process that is going to serve as the main criterion when assessing the feasibility of different programming concepts.

Practically speaking, this means requirements for structural composability and modularity. This is essential in order to be able to distribute the different parts of the project on different cores. Also, concurrency and asynchronous function execution have to be supported due to the parallel nature of the new architecture.

Currently, there are several languages employed in the design of embedded systems, which fall into a few categories in terms of programming paradigm. The lowest level languages, i.e., assembly, and mid-level languages, like C, are following the procedural paradigm, which provides composability by defining procedures. This is only a rudimentary form of modularity, hence for more sophisticated cases the object-oriented paradigm is employed instead. Object-oriented languages, like C++ or JAVA, are not as common in embedded development, but they are still used.

The ideal programming paradigm is one that combines the simplicity and support of C and the modular capabilities of object-oriented languages. There are several languages that could be considered to fulfill these criteria. For instance, the Virgil III language was developed specifically with the embedded environment in mind balancing object-oriented, functional, and procedural programming features. However, the nesC programming language

is more widely accepted than its counterparts and more mature. It provides the necessary modularity, and fits the multi-core concept, thus, it was chosen as the base language.

### The TinyOS framework and nesC

The nesC language – widely employed in the WSN community – addresses the lack of modularity and reusable components in C [33]. It is a component-based event-driven extension of C meant for a framework called the TinyOS platform [61]. TinyOS is an OS designed to run on resource-constrained hardware platforms. The core OS requires only 400 bytes of code and data memory, combined.

#### Basic concepts

The language and the OS are based on some elaborate concepts. In order to develop a clear understanding, two main viewpoints are presented first, which will enable subsequent parallelization as well. Details are then described later on.

*Code structure*

From a code structuring point of view, nesC enables composability and furthers modularity. Applications for the framework can be described in terms of graphs. The nodes are components, which encapsulate functionality and state, and expose a subset of them through interfaces. Edges are bi-directional connections between components via these interfaces. For these connections, or wirings in nesC terminology, the interfaces need to be specified only, not the associated components.

*Execution model*

From an execution model point of view, the OS does not support thread-based concurrency in which thread stacks would consume precious memory. Instead, it schedules and provides asynchronous, deferred execution of non-time-critical and computationally intensive operations referred to as tasks. Tasks are independent, but do not run truly concurrently in the single execution thread of a single core, as there is no pre-emption. Tasks run to completion, so they can be considered atomic with respect to each other, but not with respect to interrupts. A task can be thought of as a chain of (subsequent and branching) function calls entering components through their interfaces. These task call trees are rooted in the scheduler. Task executions are requested for either a hardware interrupt or a previous task having posted a deferred task.

Due to the lack of pre-emption, an approach was needed to break up long tasks into smaller, manageable pieces. This is achieved by using split-phase interfaces. These interfaces provide a way to initiate an operation in a non-blocking manner, and have a callback that signals the completion of the operation later on within the context of a different task.

**Detailed description**

*Language design*

A few basic principles underlie the design of nesC a direct extension of C. Developers are already familiar with C, and it is widely supported on many platforms. In fact, even during nesC executable generation, an intermediary C code project is created first based on the nesC source. However, the new features of nesC are not directly dependent on any C features.

nesC is a static language, so there is no dynamic allocation of components or graph restructuring during run time. The main advantage is that (with the intermediary C code) whole program analysis and optimization can be performed at compile time. nesC is based on the concept of components, and directly supports TinyOSâĂŹs event-based model. TinyOS itself was written in nesC.

*Interfaces*

nesC applications are built by writing and assembling components, which provide and use interfaces. Provided interfaces represent functionality that the component offers; the used interfaces represent functionality the component needs to perform its job. Interfaces specify a set of callable functions to be implemented by the interface's provider (commands) and a set of callable functions to be implemented by the interface's user (events). This allows a single interface to contain the functions necessary for sophisticated interactions between components. It is impossible to connect two components unless the correct number of command and event handlers are implemented.

Component interfaces may be wired zero, one, or more times, which makes it possible to fan-in and fan-out. For example, many client components can be wired to the same interface of a serving component, and so in the client components many call expressions may access the same provided command, which is the fan-in case. Conversely, a single command call expression can be connected to a number of command implementations, which is the fan-out case.

A regular command of a normal interface in nesC will usually return some value, which can be, for example, the value read from an ADC. In fact, the command will block until it has this value. A split-phase type interface, on the other hand, has a command that only initiates the operation. The execution of this command is non-blocking and does not return a result directly. Progress is signaled at a later time through an event within the context of a different task. For instance, the initialization of a timer and subsequent deferred callbacks for when the timer fired is a common example.

*Components*

Because interface definitions are separated from component definitions, components can cleanly abstract away underlying differences in implementation. For example, on one platform the temperature sensors may be a memory mapped internal device, while on other platforms the sensor access may require external $I^2C$ communication.

Components may be modules or configurations, showing a strong separation of construction and composition: Modules exclusively contain the actual application code and expose interfaces; configurations support hierarchy by containing other graphs of wired (connected) components. At the topmost level, a single configuration is defined, referred to as the top component (configuration), that includes all other components.

Most components in TinyOS represent services or hardware and therefore exist only in a single instance. A very important fact is that these components are allowed to simultaneously exist at many different levels and points in the hierarchy. In other words, one single instance of a component may be found in many different configurations at the same time. It is important to stress that such components are not copies of an object sharing some states, but are truly one single object, see Figure 8.

Some components have to be instantiated multiple times, with slightly different working parameters. In nesC, this can be done using abstract components, which have optional initialization parameters. For example, many different timer components may be instantiated, which provide different time resolutions specified as an input parameter.

*Concurrency and atomicity*

The TinyOS execution model is single-threaded, but interrupts can suspend the main thread to access shared resources. Resources, in this case, mean peripherals (accessed via IO commands or memory addresses) or state variables (memory). This leads inherently to race conditions that have to be dealt with.

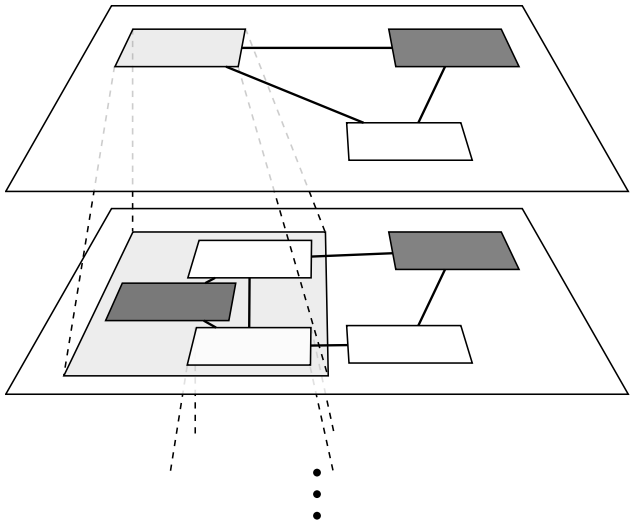


**Figure 8:** Structure of a nesC application showing how a configuration can be expanded to reveal other components within yielding hierarchical containment. The darker object is one single instance of a component appearing simultaneously within different configurations at different levels of the hierarchy.

nesC distinguishes synchronous and asynchronous code. Synchronous code is only reachable from tasks, while asynchronous code can be called from at least one interrupt handler. Due to the non-preemptive nature of the OS, synchronous code can be regarded atomic with respect to other synchronous code. However, there can be potential race conditions whenever asynchronous code is involved.

nesC employs atomic sections to prevent race conditions. An atomic section is a small code sequence (handling shared resources) that nesC ensures will run atomically. The underlying mechanism of atomic sections is disabling and enabling interrupts. This requires only a few cycles, however, if the code within the atomic section is long, interrupt losses can occur. To minimize this effect, atomic statements are not allowed to call commands or signal events, either directly or in a called function.

### Sense and Forward application example

Figure 9 shows one implementation of the sense and forward example on a single core. Gray, dashed boxes represent configurations, white, dotted boxes are interfaces, and light brown boxes depict modules. Each of these have names, which are always inside the box in the middle at the top. Some components and interfaces are referred to by aliases that are written above them. Modules can have state variables, which can be found under their names and are marked (circle, star). If the same markings within that module appear at certain functions, it is a hint that that particular function accesses that particular variable. Arrows

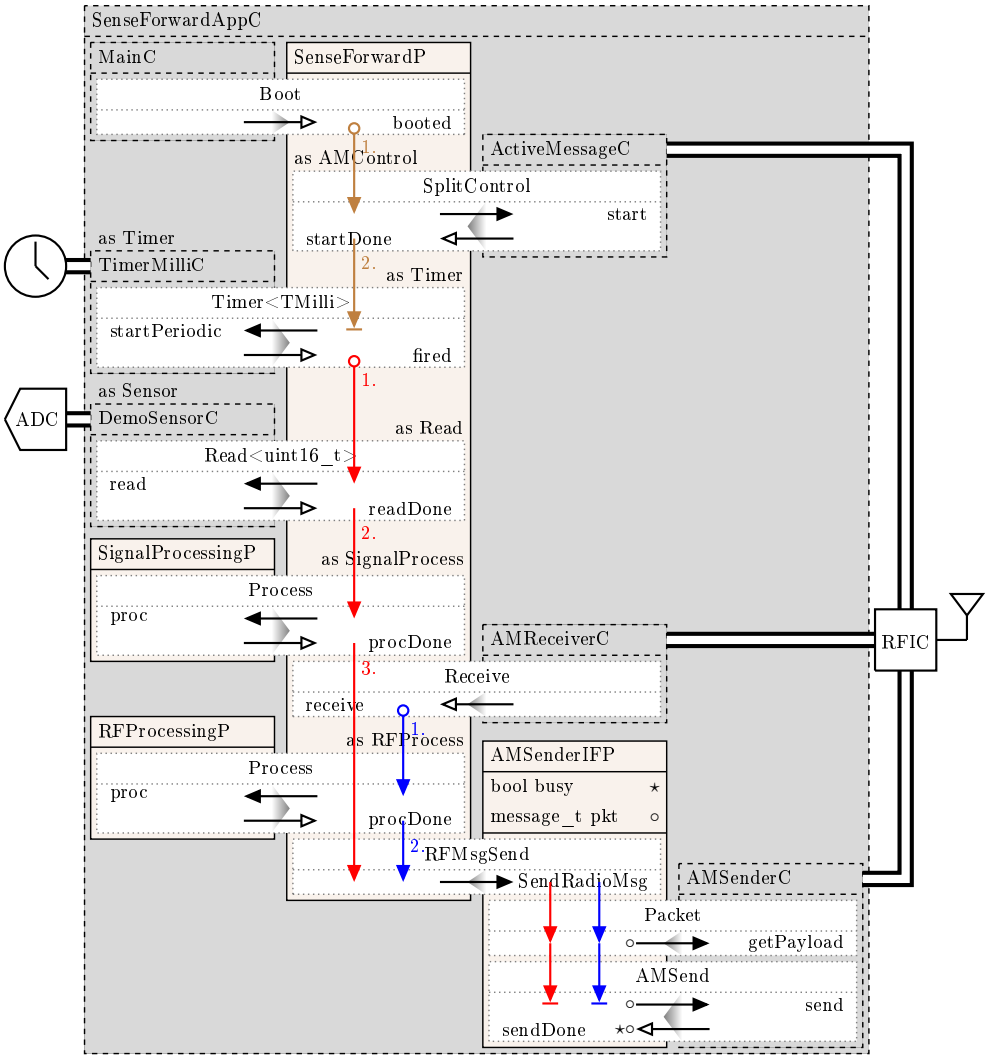



**Figure 9:** Sense and forward example on a single core.

indicate control flow; inside interfaces solid arrows are commands, hollow arrows are events. Within modules colored arrows show the sequence in which commands and events are called. There are three main colors standing for the three main flows of the program. Brown arrows show what functions are called during initialization, red arrows show the process of taking a sample and eventually sending it via the radio interface, blue arrows show what happens when the device intercepts a radio message and retransmits it.

The actual working of the program is straightforward. The "MainC" configuration signals the "booted" event on start up, which starts the radio interface using the "ActiveMessageC" configuration. When the radio interface is up and running, the hardware timer is initialized using the "Timer" configuration. From that point on the hardware timer will periodically

signal the "fired" event, which will call the "Sensor" and read a value. The "SignalProcessingP" module will then take this value do some sort of processing, and call "SendRadioMsg" command in "AMSenderIFP" module. This latter is really nothing more than a module dedicated to hold the state variables (i.e., message buffers and flags) associated with the radio stack, and prepare and send a message. The "Packet" interface provides a simple way to handle message buffers.

### Multi-core programming

Parallel execution of nesC makes it possible to address temporal issues not handled by single-threaded single-core solutions. The language fits the proposed multi-core architecture quite well, even though the concept was never meant for such platforms.

First, the goal of parallel execution has to be clarified. For PCs, parallel execution serves merely as a way to speed up computation. Users expect deterministic behavior regardless of their application running on several or just a single core. Parallel embedded applications and CPSs are expected to add value – for example, by providing better analysis of environmental observations – thus, potentially resulting in different behavior with additional processing capacity. For instance, an automobile anti-lock braking system in a vehicle can benefit from additional computing power and yield shorter break distances and improved vehicle control.

Our main goal for parallel execution is to be able to serve more interrupts in a timely manner without compromising power constraints, resulting in lower latency, shorter response times, and less event misses.

*Partitioning*

Every nesC application, examined at the topmost level, is a combination of hypergraphs and multigraphs. This is because several components can be wired up using the same interface, and components may have many parallel connections. The partitioning process of such graphs can be viewed as a division along some carefully chosen lines, so that the graph is separated into subsets of nodes and edges. Depending on how the partitioning lines are drawn, some edges or nodes may fall on borders dividing the subsets.

In terms of nesC, this results in two options when considering partitioning: cutting components or cutting along interfaces. Cutting a component would mean that parts of its interfaces, state variables, and functions within can be extracted to form a separate component. While this is conceivable, it is rather unlikely, because this would mean that there are more or less independent parts within the same component. The very idea of a

component is to collect related functionality, so cutting components is improbable. Hence, the basic approach is to cut along interfaces and maintain the integrity of components.

However, for a valid nesC program, all interfaces have to stay connected, so if one of the connections is moved to another core, some other new component has to take over that interface at the old place. That new component has to make sure that any command calls will eventually trigger the appropriate command calls on the other core. This helper component is going to be called the wrapper component, as it wraps up the whole underlying messaging framework in a transparent manner.

*The task call tree interpretation*

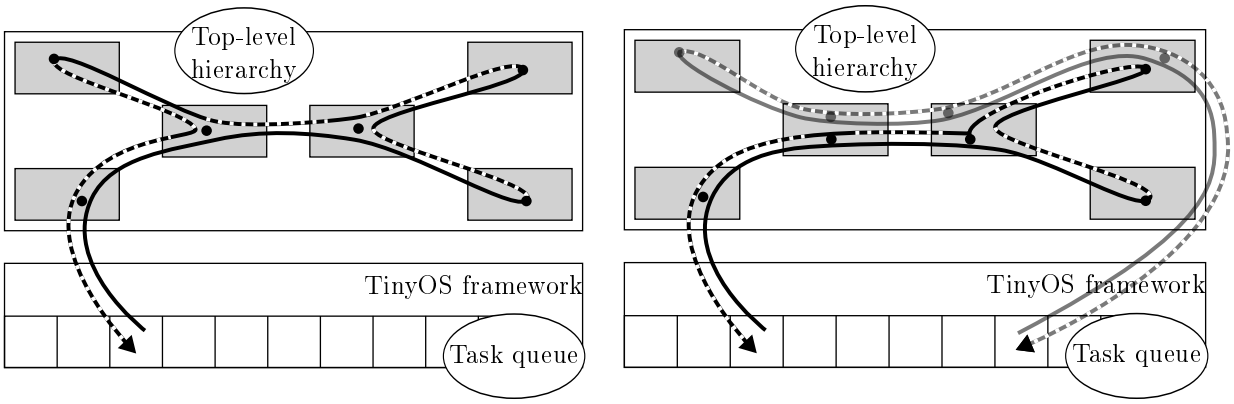
The component to core assignment is not just a graph partitioning problem despite mainly employing the structural model. The graph abstraction omits the time and causality aspects of the execution model. Causality, in this context, means the order in which functions (provided by components) get subsequently called. After all, nesC is still a procedural imperative language, thus, computation is still expressed as nested, branching function calls within the context of tasks. Hence, it is crucial to see how partitioning and mapping (using wrapper components and the queue-based messaging framework) affects not just the component graph but the call trees as well.

Call trees are defined for procedural languages, like C, as trees representing the functions that the program enters and leaves during run time. Call trees are extensively used in computer science to analyze memory requirements and program run time, but they have practical value as well during the debugging process in software development.

However, a call tree that details every function call provides too much detail for the partitioning purposes described here. Several function calls can take place within one component never actually leaving the component, which calls are thus irrelevant for a partitioning where the smallest entity is a whole component. Hence, for all practical purposes, the partitioning described here has a granularity that does not go beyond the component level. Instead of conventional function call trees, here task call trees are discussed that describe how components call each other via their interfaces, see Figure 10.

The structural model is used for partitioning, but the partitioning decisions are based on the task call trees of the execution model. Thus, the main types of task call trees have to be identified. These are crucial patterns that allow the assessment of timing and causality issues thereby aiding successful mapping to multiple cores.

- Disjoint tree: Two task call trees can be regarded completely disjoint, if they use disjoint set of components. Disjoint trees can run in a trivially parallel manner. The




**(a)** Simple function calls. **(b)** Split-phase.

**Figure 10:** Task call tree example.

underlying assumption is that every resource is only accessed by a single component providing a convenient abstraction layer for it, in accordance with the nesC language concept. Resources, in this context, are peripherals and component state variables in memory. A subtype of this class is the inert tree, which is a tree with no access to any resources. These trees can serve benchmarking purposes, e.g., for power consumption.

- Conjoined tree: Two trees sharing a set of components. A shared component is a likely sign of shared resource access, which means that both task call trees have to be executed on the same core. However, there are certain cases, where components do not contain resources, but rather provide services, e.g., mathematical functions. These components have no persistent inner states and affect no peripherals, thus they can be safely copied. Each task call tree can receive its own copy, and the trees become disjoint.

With this notation, the partitioning and assignment problem boils down to finding disjoint trees that may run in parallel. If such trees are not found, conjoined trees may be turned into disjoint trees, if shared components can be safely duplicated. It is very likely that the above described classification alone will still not result in satisfactory partitioning, and the task call tree distribution on separate cores is unbalanced. In order to reduce the number of components within a task call tree and thereby shorten its run time, rerooted cutting (following the tree analogy) can be applied.

The rerooted cutting technique is in effect the splitting of a task call tree at a split-phase interface into two disjoint trees. In other words, a new call tree is created from the branch that is the split-phase interface, see Figure 11. It can shorten the runtime of the original tree, which may help to meet timing constraints.

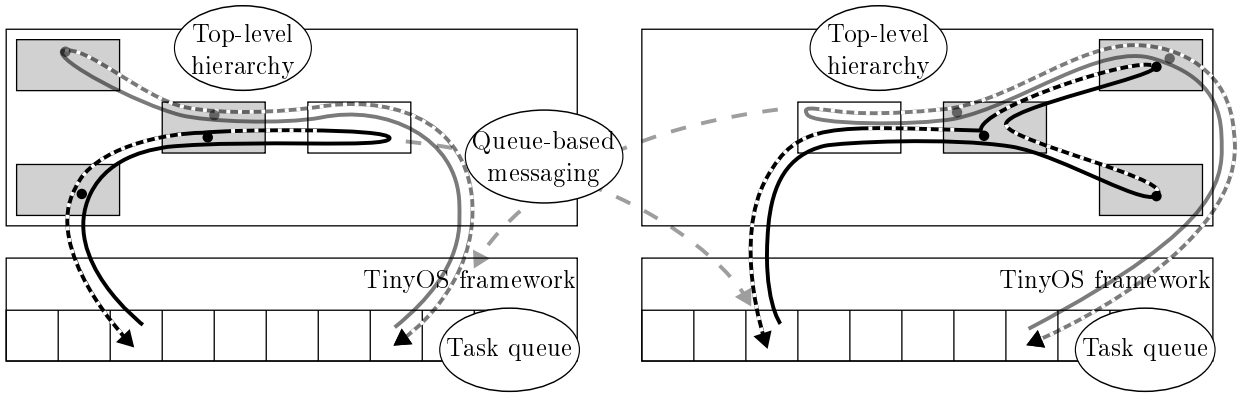



**Figure 11:** Rerooted cutting.

For example, the independent sequential initialization of peripherals can be split up very easily this way. On the conventional single core system, a task call tree may sequentially visit components (dedicated to peripherals), and execute their initialization commands. On a multi-core system some of the peripherals may be assigned to different cores. So, the split-phase call to the peripheral component's initialization command may be redirected through the messaging framework. This way the original task call tree returns immediately once the messaging framework has sent the message. The new call tree on the other core will be initiated by the messaging framework having received the initialization command call message. Eventually the messaging framework will call the same component, and the peripheral gets initialized. But, by that time the original task call tree on the first core is already finished.

*Partitioning abstraction*

For partitioning purposes an abstraction was devised in order to easily describe the process. The goal while designing this abstraction was to give a simple way to model the hypergraph partitioning problem. The simplest and most important features of the problem were extracted, which (i) allow to address a wide variety of situations, and (ii) aid in complexity analysis of algorithms. Chapter V provides two examples for this in the form of partitioning validation algorithms.

See Figure 12 for a visual representation of the abstraction. The abstraction defines cores, so that during the parallel development workflow the developer has to assign some components to these computing cores. There are no restrictions within the abstraction on the assignment process, hence, theoretically arbitrary components may be assigned to arbitrary cores. However, in practice this does not always hold:

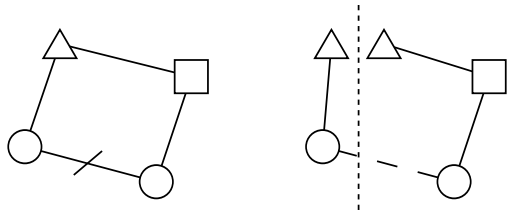


**Figure 12:** Partitioning abstraction on the topmost level. Circle nodes are assigned or dedicated components, triangle nodes are copyable components, square components are non-dedicated and non-copyable. Simple edges can not connect components across cores, while crossed edges can as they are cuttable. Dashed edges indicate connection of components on different cores.

- If a resource or state is associated with a core, then modules directly working with said state or resource have to run on the same core as well.

So, the abstraction defines assigned or dedicated components to specific cores, but it also defines unassigned components, which may be associated with any core. The final assignment of such components is automatically performed by the development tool set. However, it is the developer's responsibility to mark certain components and edges in order to guide this process.

Some components may be marked "copyable", which indicates that they can be safely replicated on different cores. In other words, in case of conjoined trees, these are the shared components which may be duplicated thereby safely separating the call trees. There are no restrictions within the abstraction on the outgoing wires of such components. If a copyable component is a configuration (in nesC terms), then it is assumed that every subcomponent within can be copied as well.

The developer has the responsibility to verify that for the application at hand, a given component can be considered copyable. There are some guidelines the developer may follow to assess this property:

- Components may not access core specific peripherals or resources.
- Components may use their state variables only temporarily. In other words, they can not have a real permanent state that would store information in between consecutive calls affecting overall code execution.

Some interfaces may be defined as "cuttable", meaning they may (but not necessarily) connect components residing on different cores. In other words, a call tree may be cut and rerooted at this branch. Whether an interface has this property or not depends very much on the application, and thus no general method was found so far to identify such interfaces.

Again, this definition is very much application dependent, and certain cuttable interfaces in one case may not be regarded as such in other cases. That being said, certain assumptions can be made that guide the developer:

- If a signal or event of a module returns a value of any kind, the interface is not cuttable. In nesC terminology, the interface has to be split-phase. Otherwise, the issuing core would be blocked waiting for the response of the other core thereby defeating the whole concept of parallel execution.
- The interface must not be timing critical, as the communication through these interfaces is not guaranteed and might involve the queue-based messaging framework.
- No pointer parameters may be passed, as cores have no shared memory. If a signal or event of a module works on the state of another module passed on as a pointer parameter, it has to be executed on the same core as the calling module.
- Transmission data rates can not be too high, as message queue block memories are limited in size.

**Multi-core Sense and Forward application example**

Figure 13 shows how the sense and forward example could be subdivided among three cores employing the multi-core concepts. Also, it introduces some of the software components involved in the messaging process. It may look complicated at first sight, but the multi-core transformation and inner workings are straightforward.

Notice how existing configurations and modules are left intact, and the queue-based messaging framework (enclosed with a dashed, blue box) was added. In between cores, message queues were added that represent a simplified version of the hardware message queues. Each core has one queue, and an additional fabric based processing algorithm (depicted as a simple white box with no markings) has one as well. On the software side, queues have associated modules, which have the sole task of putting messages in and reading messages from the hardware. Message handler blocks use this functionality. These modules are responsible for creating the right message format and routing the messages to their proper destination. Finally, wrapper modules provide the original interfaces for the original components, and manage the transformation of values into messages and the other way around.

The work flow is the same as seen in the single core case, only this time most command calls and event signaling have to go through the messaging framework. The solution

is however transparent, and pre-existing modules and configurations require no significant rewrites. All the new framework related modules can be computer generated. This approach provides an easy way to move single core applications to the multi-core platform. The exact procedure of this migration is discussed in chapter V.

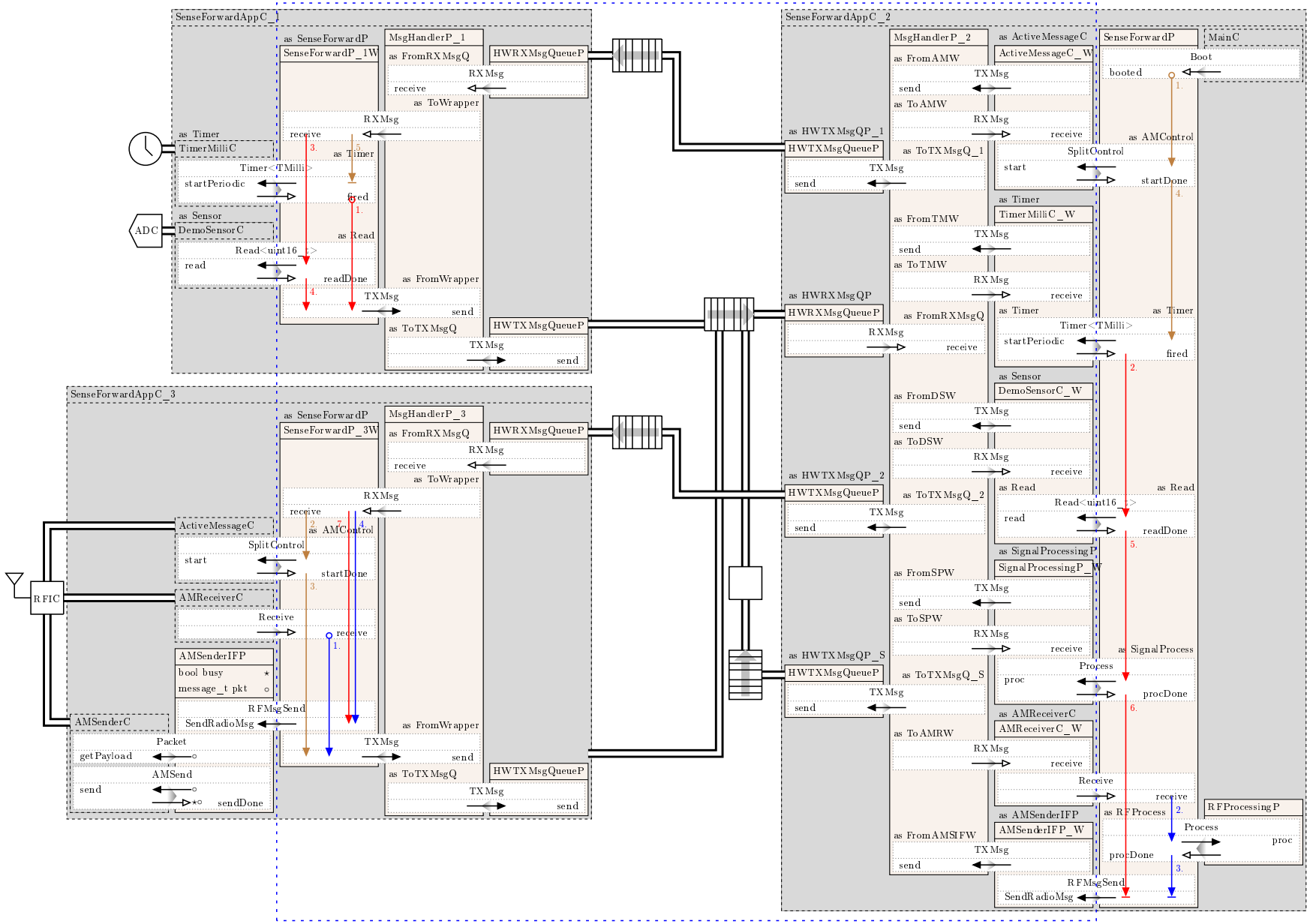



**Figure 13:** Sense and forward example on multiple cores.

# CHAPTER V

# ENVIRONMENT

In this chapter the development framework and the Avrora WSN simulator are introduced. The development framework is meant to help with the transition of single core projects to parallel solutions. To assess the result of this transition before actual deployment the simulator can be employed. It helps answering questions, like how many and what type of cores a certain application needs, and how these shall be connected.

## Framework

The goal of the framework is to provide support such that developers are able to designate top-level components for different processing units. An automated process checks the feasibility of the assignment, and if it is indeed a feasible solution, the assignment is automatically generated.

*Analysis*

There are several conceptual questions and challenges associated with this problem. Chapter III described the underlying multi-core architecture, and chapter IV introduced the programming concept. These are the necessary requirements, and it is possible to develop multi-core applications based on these concepts and resources alone. However, without the framework to automate the steps, the development is a tedious and error-prone process.

A major challenge is the extraction of information at each step of the code generation process. The guiding concept is to find the optimum level of detail and use only as much as necessary. Redundancy and useless information only increase complexity.

One aspect of this problem is the identification of the common part of algorithms and functionality that have to be provided by every core of the architecture. This means predominantly the software components fundamental to the multi-core communication. However, these components cannot be simply copied and distributed among cores. The components have to be tailored to the application and underlying architecture. The common code elements and patterns have to be determined. This is a non-trivial issue, but is crucial in order to create the code template that can form the basis of any multi-core project.

The second aspect of this challenge is directly connected to the previous issue. With the template at hand, the code generation process could create arbitrary projects given

the right information. Suitably, component containment and component interconnection information is extracted by the nesC compiler itself and is readily available. However, the generated description contains too much information, and can be considered too noisy for our purposes. The challenge here is full project generation with minimal information is used.

The third aspect applies to the highest level partitioning description. The previously defined level of detail, which is imperative for code generation, is still too convoluted for human understanding. Thus, it has to be generated from an even simpler description and the information extracted by the nesC compiler. It has to fully describe the partitioning, yet has to be simple enough for human interaction.

Another challenge is the viability assessment of the partitioning. This is the key feature to provide rapid iteration through various designs, and as such, has to be tightly integrated into the framework. Not all partitioning concepts can be realized, but this my not be apparent at first. Feasibility assessment has to take into account hardware resources and component inner states. From a mathematical point of view, the nesC top-level component arrangement can be considered a combination of a hypergraph and multigraph. The former because any number of components can be wired up via a single interface type, and the latter because two components may be simultaneously connected via several interfaces. Thus, the viability assessment means algorithms that traverse this graph structure and try to find connections between components that violate the partitioning assignment.

**Multi-core project generation**

Once the above described key challenges were identified, the structure of the framework followed implicitly. Only widely available, free, preferably open source and standardized tools and technologies were employed. The backbone of the framework became the Extensible Markup Language (XML) file format, the Extensible Stylesheet Language Transformations (XSLT) transformation standard, and the FreeMarker template engine with the corresponding FreeMarker Template Language (FTL). Also, all of the tools working with these formats are written in JAVA, and hence the whole framework is platform agnostic and easily portable. The shell scripts, which glue together the individual tools, were written for the bash shell, and are the most platform specific parts. But these scripts are little more than a collection of subsequent command calls, not complex algorithms, hence, they can be easily rewritten for any platform.

Figure 14 shows the complete framework (utilizing the TinyOS development environment) for code generation. The top part of the figure shows the conventional single core development process. The developer starts out by creating a single-core project, which (compiled by the nesC compiler) results among others in a platform-specific binary and

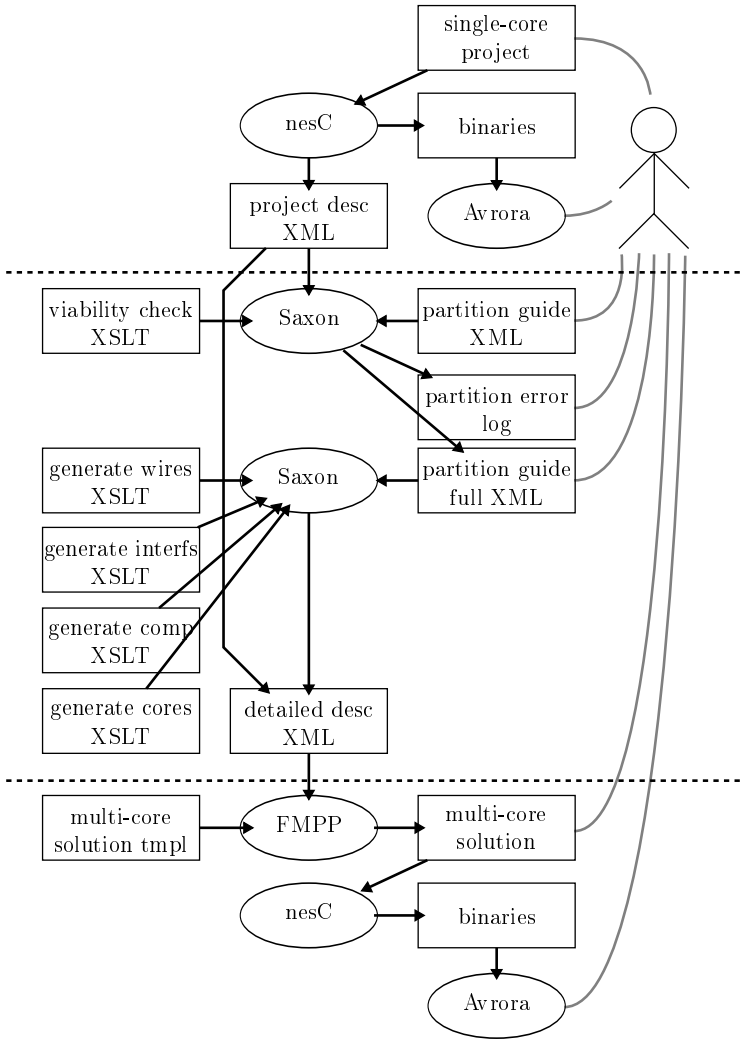



**Figure 14:** Framework for multi core project code generation.

an XML file ("wiring-check.xml") describing the project. This XML contains information regarding the project code (parsed by the compiler) including among others definitions of components, interfaces, variables, event and command parameters, return values, etc. Also, at this point the developer has the option to take the compiled binaries and run them on the Avrora cycle accurate simulator to verify functionality and detect early bugs.

Once satisfied with the results, the developer can move on to the multi-core project generation phase. First and foremost, this means that the developer has to create a partitioning guide. The partitioning abstraction, introduced in chapter IV, is employed here to describe the desired partitioning. The framework was designed such that this step is kept simple and requires minimal information:

- The name of topmost component of the project.

- A list of cores.
- A list of component names within the top component dedicated for a core from the above list.
- A list of "cuttable" interface names, which refers to split-phase interfaces that may be employed to connect components on different cores for the particular application.
- A list of "copyable" component names, which refers to components that do not access resources, like peripherals and persistent state variables in memory, and can thus be copied and instantiated separately on several cores for the particular application.

It is a very real possibility that the developer-defined partitioning is simply not viable, thus, it is crucial to verify the proposed component separation. This is a two-step procedure. The first step only deals with top-level components, hence, it is called top-level partitioning. The second step handles partitioning issues of the whole component hierarchy, and is thus called hierarchical partitioning. The details of the partitioning and feasibility testing are described further down.

The outcome of the feasibility check is a valid full partitioning guide. This file is made up of mostly the same information as the input partitioning guide, but it also lists all the top level components assigned to cores.

In itself this file still does not hold the information necessary to generate actual code. Thus, a subsequent step is necessary during which the full partitioning guide and the single-core project descriptor files are processed, and the actual code generating information is extracted. The extracted detailed description contains the minimal amount of information necessary to create the whole of the multi-core project:

- The name of topmost component of the project.
- A detailed list of wires connecting interfaces at the topmost level.
- A detailed list of interfaces employed including description of commands and events.
- A detailed list of top-level components including references to the interfaces they utilize.
- A detailed list of used cores including references to the components they host.

The final step towards a multi-core solution is the actual code generation, which employs a predefined template for the queue-based messaging framework architecture. The template is general in the sense that it does not assume any particular number or interconnection of

cores. It also does not specify components, except for the components that are part of the queue-based messaging framework itself. The template is universally applicable for any type of top-level arrangement.

***Top-level partitioning***

Top-level partitioning takes care of component partitioning on the topmost level. The feasibility test algorithm serves two purposes. It verifies that the partitioning is feasible given the list of copyable components, cuttable interfaces, and dedicated components. Secondly, the algorithm assigns so far unassigned components.

*Feasibility check*

The devised algorithm for the feasibility check is an application specific variant of the pre-order Depth–First Search (DFS). For all practical purposes, it is safe to assume that the number of components on the topmost level is adequately limited, hence, the run time of this algorithm is not critical.

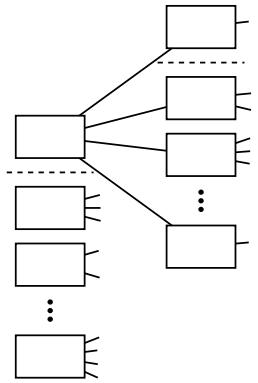


**Figure 15:** Recursive search algorithm working on the first component of a component list.

The algorithm itself is implemented in a recursive manner due to the declarative nature of XSLT (which does not even support variables). At its heart the algorithm is recursively repeating the same steps. In a nutshell,

- take a list of components as input,
- remove the first component,
- check the first component to see if it is assigned to a different core,
- generate a new list from this first component's neighbors (see Figure 15),

- recursively analyze this new list,
- and finally recursively analyze the rest of the input list (without the first component).

Figure 16 shows the simplified pseudo code for the algorithm. For the sake of simplicity, the depicted algorithm only performs the feasibility test. However, it could also return the link (chain of components) between two conflicting components, which is straightforward feature to add.

The recursive function takes two lists of components as input.

- The first list "L" is just a collection of components (assumed to reside on the same core) that we would like the algorithm to check for contradicting assignments. A contradiction, in this case, means that any component in "L" is already assigned to a different core than the given core.
- The second list "potential_L" is the set of components that have been visited once and seem to check out, meaning that they can be potentially dedicated to the given core. However, this is just a temporary state, and these components are not fully verified yet. Their status can change depending on what the algorithm finds during recursive steps.

The algorithm returns a state and another component list. The state simply indicates whether the algorithm has found anything contradicting the assignment or not. This list holds all the (directly or indirectly) connecting components that may be safely assigned to the same core as the original input list.

*Time complexity*

The number of graph nodes, i.e., components, and the number of edges, i.e., connections via interfaces, are represented by $N$ and $M$ respectively. In order to analyze the time complexity of the algorithm, I will assume that complex steps, like set operations, have a simple, naive implementation with a $\mathcal{O}(N^2)$ complexity. So, for example, at line 4 the set operation means the removal of superfluous components. I will assume that the underlying naive approach enumerates the nodes of the input list "L" (which in itself is $\mathcal{O}(N)$), and at every iteration also enumerates the other list "potential_L". This yields the $\mathcal{O}(N^2)$ complexity.

Most operations in the algorithm can be considered set operations on two sets, like the example, and thus complexity has an $\mathcal{O}(N^2)$ component. However, the rest of the operations, like at line 10, enumerate connecting components or interfaces, and thus have to take into account edges, which adds an $\mathcal{O}(M)$ component. So, in itself the recursive function is

```
1  Recursive_top_level( L, potential_L );
2  begin
3      // Only components we don't already have as potential candidates
4      L = L \ potential_L;
5      if E is empty then
6          return NORMAL;

7      // Check first component
8      E = first element from input list L;
9      if ¬(E is on another core) then
10         con_L = list of components connecting to E \ potential_L;
11     if ¬(E is on another core) ∧ ¬(con_L is empty) then
12         rec_state, rec_L = Recursive_top_level( con_L, potential_L ∪ E );
13     if E connects backwards via a cuttable wire then
14         ret_state1 = NORMAL;
15     else if E is on another core then
16         ret_state1 = TAINTED;
17     else if E is copyable ∨ con_L is empty then
18         ret_state1 = NORMAL;
19     else
20         ret_state1 = rec_state;
21     if ¬(E is on another core) ∧ ( rec_state = NORMAL ∨ E is copyable ) then
22         ret_L1 = E ∪ rec_L;

23     // Check rest of the list
24     rest_L = L \ E;
25     if rest_L is empty then
26         ret_state2 = NORMAL;
27     else
28         ret_state2, ret_L2 = Recursive_top_level( rest_L, potential_L ∪ ret_L1 );

29     // Generate return value
30     if ret_state1 = TAINTED ∨ ret_state2 = TAINTED then
31         ret_state = TAINTED;
32     else
33         ret_state = NORMAL;

34     ret_L = ret_L1 ∪ ret_L2;
35     return ret_state, ret_L;
```

**Figure 16:** Pseudocode for the recursive top-level search.

$\mathcal{O}(N^2 + M)$, but as it gets called for every node once, the total time complexity of the algorithm can be estimated as $\mathcal{O}\big(N(N^2 + M)\big)$.

However, in ordinary tinyOS applications $M$, the number of connecting wires, i.e. edges, is small. With certain assumptions based on this fact a simpler time complexity estimation can be derived. In a regular graph the number of edges has to be limited $M \leq \binom{N}{2} = \frac{N!}{2!(N-2)!}$, which corresponds to the case of a complete graph.

In a hypergraph edges can connect arbitrary number of nodes, thus the upper limit changes to $M \leq \sum_{k=2}^{N} \binom{N}{k} = \sum_{k=2}^{N} \frac{N!}{k!(N-k)!}$. If edges are limited to connect at most $\alpha$ components, the limit becomes $M \leq \sum_{k=2}^{\alpha} \binom{N}{k} = \sum_{k=2}^{\alpha} \frac{N!}{k!(N-k)!}$.

In a multigraph nodes can be connected by an arbitrary number of edges, thus the upper limit changes from the simple regular case to $M \leq B \binom{N}{2} = B\frac{N!}{2!(N-2)!}$, where $B$ can be an arbitrary multiplying factor. However, for all practical purposes, it is safe to assume that the number of parallel edges between nodes is very limited, and $B = \beta$ is a constant.

Combining the results of the hypergraph and multigraph case, the upper limit of edges can be expressed as $M \leq \beta \sum_{k=2}^{\alpha} \frac{N!}{k!(N-k)!}$. Examination of this upper limit on $M$ reveals that the highest order of $N$ is $N^{\alpha}$. Hence, the expression for the order of time complexity may be updated to $\mathcal{O}\big(N(N^2 + N^{\alpha})\big) = \mathcal{O}\big(N^{\alpha+1}\big)$.

As a final note, the actual true time complexity of the actual implementation of the algorithm is difficult to estimate. The XSLT interpreter hides the implementation of complex steps (like set operations) and the corresponding time complexity. These operations – depending on the underlying algorithms and data structures – can have vastly different runtime complexity. It is likely the XSLT interpreter is optimized and does not follow the naive approach. Thus, the time complexity for individual set operations is probably better than $\mathcal{O}(N^2)$ as assumed in the beginning. Consequently, the time complexity can be assumed to be better than the derived results. However, as the number of top-level components tends to be low, the exact value of time complexity is not crucial.

*Assigning undedicated components*

The second main objective is the assignment of undedicated components to cores. This is performed in a greedy manner by successively executing the above algorithm with different sets of input components dedicated to different cores. For example, the first run would have components dedicated to the first core as input, and thus would return a list of all the components that can be assigned to the first core. Similarly, the second run would have components dedicated to the second core as input, but all the cores dedicated to the first core

would be on a prohibited list, and hence the algorithm would be forbidden to reassign them to the second core. The procedure continues until all top-level components are assigned, see Figure 17.

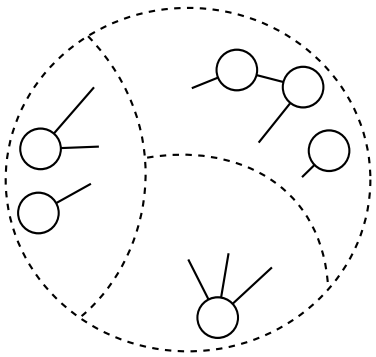


**Figure 17:** Partitioning on the topmost level. Circle nodes are components dedicated to different cores imposing partitioning constraints on the rest of top-level components.

The total time complexity can thus be estimated as $\mathcal{O}\big(C\,N(N^2+M)\big)$, where $C$ is the number of cores, $N$ is the number of components, and $M$ is the number of connections. Once the assignment process is complete, an updated version of the partitioning guide file is generated, which contains all top-level components with their respective cores. This full partitioning guide is used in the following.

***Hierarchical partitioning***

Given a viable top-level partitioning (in an automatically generated full partitioning guide file), the hierarchical partitioning check is performed next. This phase is meant to verify that said partitioning remains viable even if the component hierarchy is taken into account. Hence instead of a top-level connectivity hypergraph, this step analyzes the component containment graph.

*Feasibility check*

The main issue here is that certain components can exist simultaneously at different points (and levels) in the hierarchy. These are not copies of the same component, but are indeed a single instance of one component. In other words, the containment graph is not a tree, and it may very well be cyclic. Given the above abstraction, the shared components may only be assigned to different cores simultaneously if they are copyable, in which case each core receives its own copy.

In this case, the feasibility test algorithm verifies that the partitioning is viable given the list of copyable components and dedicated components. Again a similar algorithm is employed, but for all practical purposes, this simplistic approach proves satisfactory.

The algorithm is implemented in a recursive manner similarly to the top-level partitioning case. The recursive function takes two lists of components as input.

- The first list "L" is just a collection of components (assumed to reside on the same core) that we would like the algorithm to check for contradicting assignments. A contradiction, in this case, means that any component in "L" is already assigned to a different core than the given core.
- The second list "ignore_L" is effectively the set of copyable components, which can be safely disregarded when searching for components instantiated multiple times on different cores.

The algorithm returns a state and another component list. The state simply indicates whether the algorithm has found anything contradicting the assignment or not. This list holds all the (directly or indirectly) connecting components that may be safely assigned to the same core as the original input list. Figure 18 shows the simplified algorithm.

*Time complexity*

The number of containment graph nodes, i.e., components, and the number of edges, i.e., containment relations, are represented by $N$ and $M$ respectively. Again the assumption is made that set operations will contribute a $\mathcal{O}(N^2)$ complexity. However, operation at line 10 is different; it enumerates all contained components. At most $M$ edges have to be examined for this, and because the containment graph is neither a hypegraph nor a multigraph $M < N - 1$. Hence, the complexity of a single execution of the function is $\mathcal{O}(N^2 + N - 1) = \mathcal{O}(N^2)$. However, the function is called for every node once in the worst case, so the total time complexity of the algorithm can be estimated as $\mathcal{O}\big(N(N^2)\big) = \mathcal{O}(N^3)$.

This basic algorithm is repeatedly run for all cores continuously accumulating valid components or stopping if an erroneous component is found. The total time complexity can thus be estimated as $\mathcal{O}(C\,N^3)$, where $C$ is the number of cores, and $N$ is the number of components.

## Simulation

In the following the Avrora discrete time, cycle accurate, embedded systems and network simulator is introduced. Furthermore, the modifications are explained that enabled the

```
1  Recursive_hierarchical( L, ignore_L );
2  begin
3      // Remove components we ignore
4      L = L \ ignore_L;
5      if L is empty then
6          return NORMAL;

7      // Check first component
8      E = first element from input list L;
9      if ¬(E is on another core) then
10         con_L = list of components inside E;
11         rec_state, rec_L = Recursive_hierarchical( con_L, ignore_L );
12     if E is on another core then
13         ret_state1 = TAINTED;
14     else
15         ret_state1 = rec_state;
16     ret_L1 = rec_L;

17     // Check rest of the list
18     rest_L = L \ E;
19     if ret_state1 = NORMAL then
20         ret_state2, ret_L2 = Recursive_hierarchical( rest_L, ignore_L );

21     // Generate return value
22     if ret_state1 = TAINTED ∨ ret_state2 = TAINTED then
23         ret_state = TAINTED;
24     else
25         ret_state = NORMAL;
26     if ret_state1 = TAINTED then
27         ret_L = ret_L1;
28     else if ret_state2 = TAINTED then
29         ret_L = ret_L2;
30     else
31         ret_L = ret_L1 ∪ ret_L2;
32     return ret_state, ret_L;
```

**Figure 18:** Pseudocode for the recursive hierarchical search.

simulator to tackle systems with multiple cores, enabling the analysis of soft multi-core platforms and networks as well.

During any development process, the later fundamental problems are discovered, the exponentially higher the cost of repairs become. Thus, early detection of issues plays a crucial role in the design of contemporary systems. Simulation – whether that of hardware or software – is a prominent method to gain insight into the workings of complex designs.

It is especially important for embedded systems that have sophisticated, real-time analysis requirements, yet cannot themselves run debugging software.

There are plenty of methods to simulate embedded code. Most MCU manufacturers provide some sort of a solution out of the box with their Integrated Development Environment (IDE). The problem is that they either only support the simulation of a single unit, or they provide multi-node simulation only for a certain type of functionality, e.g., radio networking. The Avrora open-source, discrete time simulator [23, 80, 86, 79, 45, 51] addresses this issue. It is cycle accurate and supports the simulation of a network of various nodes inclusive their communication. This makes it possible to directly test binaries developed for various platforms, and, for instance, immediately see how the signal processing and the radio stack hold up for radio networking.

**Analysis**

In Avrora, MCUs are fully simulated (memory, registers, IO, etc.), and peripherals are modeled as Finite–State Machines (FSMs). The main ideas of Avrora stem from two key observations:

- Most nodes spend their time predominantly in sleep or some low power mode where they are not performing any computation but simply wait for some event to occur.
- Even when nodes are not in sleep but working, they mostly perform independent tasks, i.e., inner book keeping and processing, with no influence on any of the other nodes.

These observations have led to a fundamentally different type of cycle accurate simulation approach. Instead of the whole network and all the peripherals running in a synchronous lockstep manner, each MCU core is given a separate thread, which is solely responsible for simulating the core and the connected peripherals. Also, even within a thread, the core and peripherals are not simulated for every cycle. Instead, a single event-queue is employed for every core and connected peripherals, and the simulator is essentially only dealing with events as they happen. Thus, for example, timers do not require clock cycle simulations, they only have to place an event in the queue.

This latter is of course not really an improvement if the MCU is performing some computation intense task, because then practically every cycle means an interpretation of the code's instruction. But, because cores are usually in sleep mode, the simulator can simply jump ahead in the event queue in most cases.

However, this could quickly lead to nodes drifting apart in time to a degree where inter-node communication became impossible. For instance, by the time node A sends a message

to node B, node B may have already advanced so far ahead in time that from its point of view the radio message happened in the past, and thus is not relevant anymore. Hence, the key to the above described ideas is the loose synchronization of nodes, which in essence is a method to stop individual node simulator threads from getting to far ahead.

*Other features*

As the name suggests, the simulator was originally intended for AVR MCUs, but due to its modularity, it can support other architectures as well, e.g., Intel MCS-51 8051. Complete platforms can be simulated, i.e., Mica2, MicaZ, TelosB, inclusive the RF ICs and other peripherals. Also, it is highly modular and can be easily extended with more components.

The simulator supports a network of nodes with accompanying physical topology. The radio communication aspect supports different levels of detail, meaning that by default a simplistic radio channel is employed, but more sophisticated radio models are also available. Theoretically, with GNU Radio and Universal Software Radio Peripheral (USRP) the simulated platforms can actually be tested on real radio channel or sensory data in real time. Existing measurements can be given (with some pre-processing) as input to virtual platforms to see how the WSN manages to handle data.

The simulator is not following a step-wise global approach to perform cycle accurate simulation, but it can fall back to that mode if need be. This way it is fast enough to do real time simulation of a few of cores, but several cores can be simulated as well within a reasonable amount time.

It provides a unique opportunity to perform design space exploration and answer fundamental design questions like:

- Number of nodes needed to cover a WSN application
- Number of processing cores needed on one node
- The association of cores to IO and sensor peripherals
- Number and type of inter-core connections (every core can be connected to every core)
- Interconnecting message queue size (infinite message queue length)
- Transmission time length (can go down to couple cycles, can be asymmetric for cores or for platforms)
- Processing architecture
  - Instruction set (special high level instructions, e.g., log)

– Heavy processing steps placed in fabric

*Disadvantages*

The simulator has been designed with certain assumptions in mind, thus for some scenarios, the original version can be considered suboptimal. The original simulator assumed that platforms are single-core. Although it's easy to extend, it required fundamental rewrites for efficient simulation of multi-core architectures. (There was a simulation possibility for wired networks, but that was limited to certain core connection types and numbers, and only step synchronization was supported, in which case there was no wireless communication.) The WSN simulation only supported nodes of the same type (program code could differ though). Also, the simulator employed time resolution that was equal to the clock rate of the MCUs – assuming that all MCUs had the same clock rate. In order to apply Avrora for multi-core WSN simulation purposes, the code base had to be rewritten.

***Inner workings in a nutshell***

The simulator can run in different modes to analyze different aspects and configurations of nodes. Obviously, the mode that is the most important for WSNs is the full network and node simulation. Within this simulation mode the most important entities – roughly corresponding to JAVA classes as seen in the simulator – are listed below. This is the simplified hierarchical simulator inner structure:

- **Simulation**: Includes everything simulation related. (Much more than listed, but for the sake of simplicity everything else was omitted.)
  - **Synchronizer**: Responsible for the coordination of cores, it acts as a leash restraining individual threads from running to far ahead in time.
    - **Medium**: Shared resource that enables message transfer among participating components running in different threads, e.g. radio channel, digital bus, message queue.
    - **Com device**: Transmitter and receiver associated with a specific medium with corresponding hardware components found in the "Platform".
  - **Node**: Represents a node and acts as an interface towards "Simulation". It does not care about the inner structure and elements of the "Platform", it is only concerned with and contains simulation related objects: the thread(s) and the event queue(s) associated with the core(s) within.

* **Platform**: A container holding the core and every connected peripheral, but being a representation of actual hardware, it does not care about threads and such.
* **Simulator**: Handles the simulation of the core; an interpreter that processes instructions.
* **SimulatorThread**: The thread associated with the particular "Simulator".

Figure 19 shows the structure of a node. The original concept was centered around a single MCU that had a processing core and some inner, memory mapped peripherals e.g. ADCs. Peripherals not integrated into the MCU connected through pins e.g. like the CC2420 RF IC. These peripherals can use sophisticated communication protocols like Serial Peripheral Interface Bus (SPI), in which case the simulation is not pin level but transaction level, i.e., working with bytes as opposed to bits.

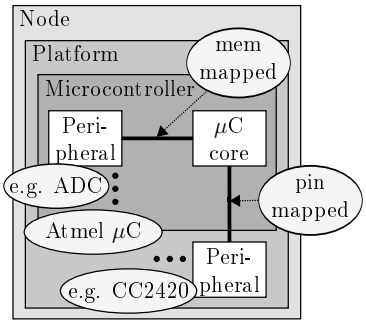



**Figure 19:** The node.

*Medium and synchronizer*

Mediums define message formats and communication protocols. A medium can be anything that allows communication among nodes. Originally, it was meant to represent only a single radio channel, which was the sole way for WSN nodes to interact. Thus, synchronization in essence was the analysis of this medium: determining what messages were broadcasted and when. This way independent threads knew when to wait to not loose possible messages that could potentially influence their further operation. An example of this is shown on Figure 20.

Note that mediums are independent from platform or hardware and thus can reach across nodes, cores, and devices. For every medium, a separate synchronizer is dedicated to take care of the throttling of individual threads. Every core (and thus simulator and simulator thread) of a medium connects to its single synchronizer.

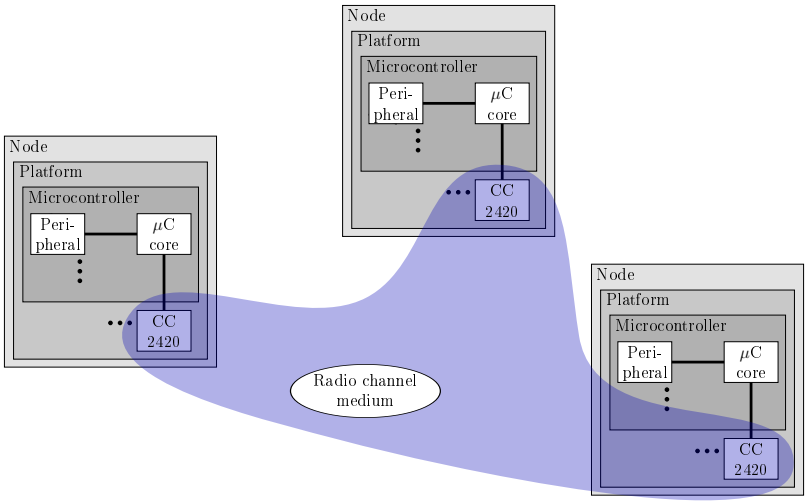



**Figure 20:** Radio channel for the original single core case.

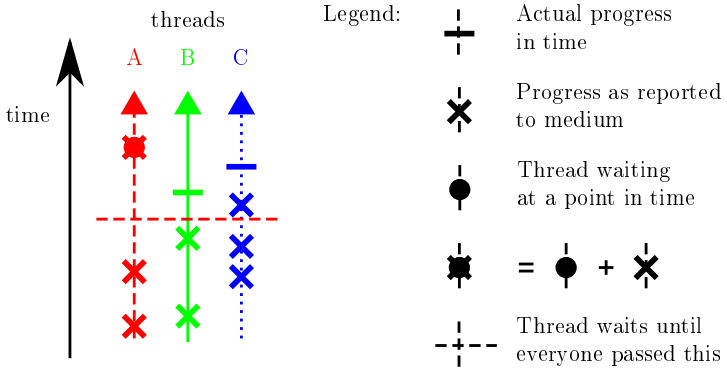



**Figure 21:** Threads, medium, and synchronizer.

The synchronizer keeps track of how far ahead each simulator thread progressed. It does this with the cooperation of the simulator threads themselves, which actively report their progress to each synchronizer they are connected to. Synchronizers provide the API necessary, so that nodes can wait on each other and can report their progress, i.e., there is a command called "waitForNeighbors" that blocks the thread until all other nodes have reached the specified simulation time point. If other nodes have reached or surpassed that time, the wait unblocks, and the thread continues. See Figure 21.

The point in simulation time that a thread can specify for other threads to reach can only be in the past. If threads are allowed to wait until other threads have passed them in simulation time it is possible for a set of threads to get stuck, see Figure 22. Each thread is waiting for the others to reach a given time point in the future, which they never do.

Also, threads can potentially deadlock even if not waiting for others to reach a future event, but just to catch up to the same exact time. This is because it is not guaranteed that

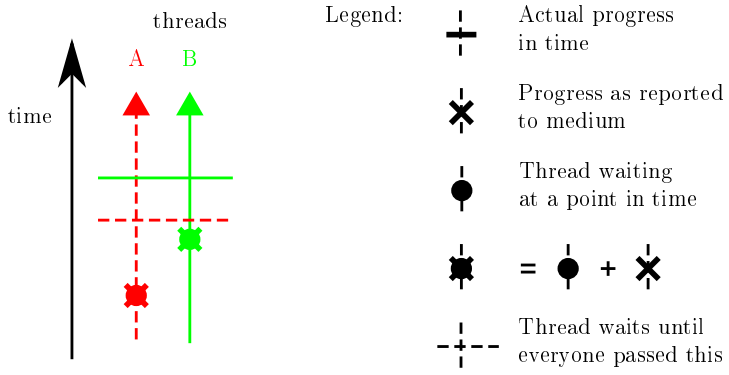



**Figure 22:** Thread A and thread B both waiting for the other thread to reach a point in time in the future.

threads will broadcast their progress before they start waiting. One way to avoid this is to allow only one thread to wait for others passing future or present events, but it is better to allow threads to wait only for points in the past.

### Modifications to support multi-core simulation

For the multi-core simulation, the structure above remained mainly unchanged with the addition of another container entity within the "Platform" called System of Elements (SoE). A SoE contains devices, peripherals, cores, and other SoEs. It has pins as interfaces for outside connections, but high-level interfaces like SPI are also possible. Examples for what a SoE could be used for include the modeling of SoCs, or soft cores instantiated within FPGA fabric.

*Extended medium and synchronizer*

Mediums were extended to incorporate inter-core communication as well. With additional cores the number and type of interconnections and consequently that of the mediums increased. Note how mediums reach across nodes, cores, SoEs, and devices in Figure 23.

For every medium, a separate synchronizer is dedicated to take care of the throttling of individual threads. In the original concept, every core (and thus simulator and simulator thread) of a medium connected to its single synchronizer, but in the improved revision, cores can be connected to several mediums and thus several synchronizers at the same time. This introduced new multi-thread synchronization challenges and situations that could potentially lead to deadlocks.

For example, failure to broadcast thread progress to every synchronizer before entering wait mode can lead to deadlocks. In such a scenario, two threads both associated with the

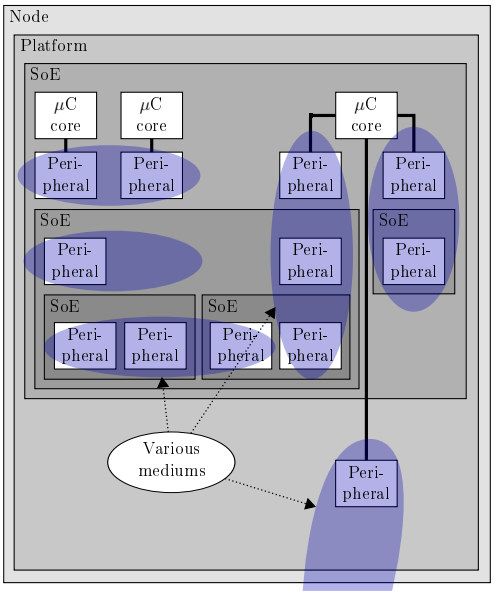



**Figure 23:** Multiple mediums across several components.

two different mediums (and synchronizers) could end up waiting for the other one to reach a certain point in the past, which they already have, but simply did not broadcast, see Figure 24.

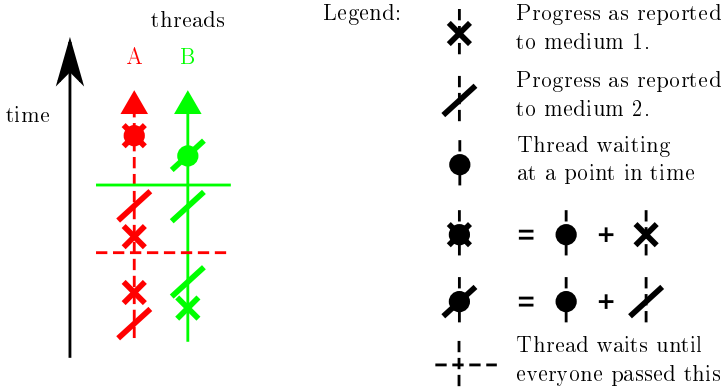



**Figure 24:** Thread A and thread B – both associated with two mediums and hence two synchronizers – waiting for the other thread to reach a point in the past, but because threads did not update every synchronizer about their progress, they never leave waiting state.

Another common form of deadlock comes with the mutually exclusive locking of several resources. The issue emerged if the same set of resources were locked in different order by two threads. The solution was to enforce the locking of resources in the same order.

*Multi-core MicaZ*


**(a)** MicaZ. **(b)** Multi-core MicaZ.

**Figure 25:** The difference between the original and multi-core MicaZ in the simulator.

The MicaZ compatible multi-core platform – introduced in chapter III – was implemented within the enhanced Avrora simulator. Instead of a single ATMega128L MCU the platform employs a SoE with several ATMega128L MCUs tightly interconnected using message queues, see Figure 25.

# CHAPTER VI

# CASE STUDY

In this chapter, a non-trivial, high-throughput, multi-channel application is introduced, and is subsequently transformed to a multi-core project. The application deals with the SHM problem, and utilizes our sensor platform. The application involves analysis techniques (along with measurement results), which are mapped to the soft multi-core architecture. This example is meant to demonstrate the issues associated with high-performance embedded systems.

The goal of SHM is to give insight into the condition and state of structures with emphasis on damage detection. AE-based SHM methods [59] are on-site, non-destructive approaches, which mainly detect ultrasound stress waves caused by sudden, inner structural changes. The sources of AE signals can be damage-related, but alternative causes are also possible introducing false positives and background noise. The nature and location of the damage may be estimated by using one or a combination of multiple parameters, such as the Time Difference of Arrival (TDoA) between different transducers.

The application example focuses on AE measurements to provide signal time-frequency analysis, and to localize cracks with the TDoA method. It is a sensing technology that has been successfully tested in a laboratory environment. However, actual deployment of a WSN based around this technology becomes feasible only if the processing requirements can be met on a power-constrained embedded platform.

### Analysis and processing

The application involves several signal processing steps, which are depicted on Figure 26.

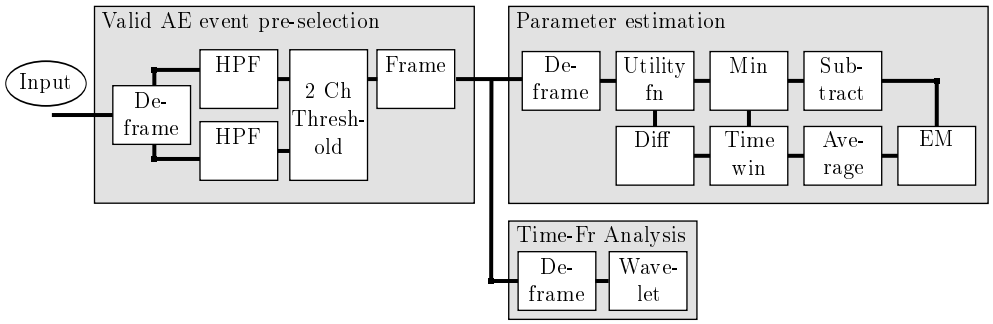



**Figure 26:** Simplified block diagram of the AE signal processing.

The first step of digital signal processing is a simple filtering. Since background noise has significant energy concentrated in the lower frequency ranges, a 15 tap Finite Impulse Response (FIR) High–Pass Filter (HPF) (cutoff frequency 50 kHz, attenuation 50 dB) is employed. For reference, the original bandwidth of the PKWDI AE sensor is 850 kHz. From the data stream, the system collects time windows in which the signal crossed the threshold level.

**Wavelet-based time-frequency analysis**

Because physical phenomena and material properties have such profound effects on measured signals, SHM is considered to be the "science of signatures". For example, sound speed in materials can be frequency dependent due to properties like shape and structure affecting wave propagation. Hence, it is crucial for in-depth analysis to have information on the frequency components present in an AE recording, and their arrival time.

The principal issue here is the inherent time-frequency uncertainty associated with the analysis process. Conventional Fourier analysis involves defining a global time and frequency resolution a priori to performing the actual transformation. This resolution is independent of actual signal content, and the method is only viable if signal parameters, i.e. bandwidth and duration, are also known a priori. Oversampling and other approaches – to cover worst case scenarios – are theoretically possible but infeasible on resource-constrained embedded systems. A careful balance has to be found whether timing accuracy or frequency selectivity are preferred.

The wavelet transform is a widely-used promising alternative for time-frequency analysis that can adjust the transformation process on the fly to adapt to signal contents. It is capable of not only altering the overall time-frequency resolution, but it can also "zoom in" on interesting parts. In other words, the time-frequency resolution does not have to be homogeneous, so within the same analysis some frequency bands may have higher selectivity or more accurate timing. The method achieves this by subsequently dividing the examined signal into an upper and lower frequency band and halving the sampling rate. These steps can be repeated many times until the desired frequency selectivity is reached resulting in a perfect binary tree of band-filtered, reduced sampling rate signals at the nodes. This approach for wavelet analysis is called Wavelet Packet Decomposition (WPD), and it is up to the user to decide on the granularity of the decomposition, i.e. the tree nodes that are utilized to represent the signal. Note that nodes do not necessarily have to be on the same level. Usually, a cost function is employed to select the – in some sense – best partitioning. A common approach aims to minimize the overall Shannon entropy of the decomposition [21], see (2).

$$K = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{2}$$

Where $K$ is the Shannon entropy value that is to be minimized, and $p_i = \frac{E_i}{E_{\text{total}}}$. Here $E_i$ is the energy content of one node (frequency band), $E_{\text{total}}$ is the energy content of all nodes on that same level in the binary tree (all frequency bands with the same bandwidth). The conventional best basis selection (frequency partitioning) process in essence calculates the sum of entropies for two children ($n = 2$) and compares it to the entropy of the parent node ($n = 1$). The lower entropy partitioning is chosen or, in case of equality, the parent.

The wavelet approaches described in the SHM literature are predominantly concerned with the analysis of a single recording of a single channel at a time. Yet in practical applications, an ensemble of related recordings have to be evaluated and compared, i.e. (i) multiple recordings of the same signal from different channels, (ii) multiple signals originating from the same source. This can necessitate a common basis set. Also, by pinpointing common bands of interest and bands that can be safely disregarded, the method provides a way for data reduction, which plays a prominent role in WSN communication.

Our partitioning utilizes not one, but several binary trees corresponding to the ensemble of related recordings. We compute the cost functions for each node for each tree, and subsequently create a sum tree. In the sum tree each node is the sum of all nodes from all the trees at the same level and same position. The sum tree is then evaluated using the same steps as described above. The results section shows examples for our time-frequency analysis.

**Event classification and parameter estimation**

The other fundamental signal evaluation path focused on time domain analysis, where we tried to find the arrival time of valid AE events. The challenge here is that signals are highly dispersive, hence it would be very difficult to define the exact beginning even in a completely noise free recording. Secondly, the system has to (i) identify measurements indicating potential sources, and (ii) recordings stemming from the same source have to be classified as such. Finally, the measurement parameters have to be extracted in order to reason on the damage location.

***Onset time and measurement quality***

Our method for accurate onset time estimates and for separation of valid AE events from false positives was to first provide a short time window around the event, then to calculate a "utility" or "fitness" function that would give a minimum at the exact start of an AE event within the window.

*Utility function 1. – The Akaike's Information Criterion (AIC)-based onset time selector*

Originally, AIC was meant for statistical model identification [3]. It helps to avoid overfitting by finding the simplest model that provides a good enough approximation. It was later applied to model non-stationary, non-overlapping, independent time series with different AutoRegressive (AR) model properties [42]. Because AR model estimation is so resource-consuming, a simpler method was proposed [53], dealing with only two, subsequent time series, see (3).

$$\begin{aligned} \mathrm{AIC}(k) &= k \ln\Big(\mathrm{var}(x[1,k])\Big) + \\ &\quad (N-k-1)\ln\Big(\mathrm{var}(x[k+1,N])\Big) \end{aligned} \tag{3}$$

Where $x[1,k]$ is the time series starting with the first sample and ending with (and inclusive of) the $k$th, $N$ is the number of samples, and $\mathrm{var}() = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x_i})^2 = \widehat{\sigma_2{}^2}$ is a variance estimate. Variations on the calculation of var() can be found in literature. The $k$ value giving the minimum $\mathrm{AIC}(k)$ is the most likely onset time index.

It can be mathematically proven that if both of the time series within the time window have constant but different variances (e.g., Gaussian white noise), the method will point to the onset time of the second series, see appendix A. However, the crucial realization here is that the original AIC method's variance was AR estimation error related, while this latter method is a direct variance of a signal part; thus (without a DC component), the approach boils down to a simple comparison of signal energy in two parts of the time window. Note that for Gaussian white noise, the variance is equal to the noise spectral density times the bandwidth: $\sigma^2 = N_0 B$.

*Utility function 2. – The reciprocal-based onset time selector*

We examined several other utility functions that achieve similar performance to AIC but have lower computational cost, see appendix A. A reciprocal relationship, as seen in (4), stood out in particular.

$$\text{fitness function} = -\frac{n_1}{\widehat{\sigma_1}^2} - \frac{n_2}{\widehat{\sigma_2}^2} \tag{4}$$

Where $n_1$ is the length of the first time series, $n_2$ is the length of the second, $\widehat{\sigma_1}^2$ is the variance estimate of the first block, $\widehat{\sigma_2}^2$ is the same for the second.

The advantages are (i) no logarithm calculation, and (ii) better onset time estimates for some signals. Experience showed that dispersive signals were handled better, see appendix A, and empirical evidence also suggests that in most cases $-\frac{n_1}{\widehat{\sigma_1}^2}$ is a sufficient approximation of the fitness function.

*Quality index for measurements*

To distinguish AE events from noise events, signal energy-based methods are often suggested in the literature, but these approaches are usually unreliable; thus, we devised a different "quality index" indicator. The idea stems from the observation that for valid measurements, utility functions decrease rapidly towards the minimum, then steeply increase, whereas for noise, no such trend is noticeable. Thus, if the fitness function's derivative is taken, the values after the minimum tend to be notably higher than zero for real AE events.

The quality of the measurement is then estimated with (5).

$$q = \frac{1}{M} \sum_{i=i_{min}}^{i_{min}+M} \Big( g_i - g_{i-1} \Big) \tag{5}$$

Where $q$ is the quality index, $g$ is the utility function, M is the number of samples, and $i_{min}$ is the fitness function minimum index. The quality index is hence the mean of a few (e.g. $M = 40$) derivative values right after $i_{min}$.

***Source number and parameter estimation***

AE events form clusters in a two dimensional measurement space (quality index and AE) where the number of random processes – that is the number of AE sources – is unknown. The signal evaluation tries to answer two fundamental questions: What is the number of sources and what are their parameters i.e. time difference and quality? Only very rudimentary assumptions can be made regarding the recordings. We can assume that during the measurement procedure the crack location can be considered fix, resulting in time differences with low variances for valid AE recordings of the same source. Also, we can assume that

these recordings will have a high mean quality index, whereas false positives will generally stay low.

*Expectation–Maximization (EM) for parameter estimation*

With all possible AE events at hand, the TDoA of the crack location may be estimated. In this context, time difference is a random variable, and as such, statistical tools have to be employed to estimate it. For this part we additionally assume a Gaussian Mixture Model (GMM) with at least two mixed independent processes (i.e. the noisy events and valid AE events). No closed formulas exist to estimate a multi-dimensional GMM's parameters, so we utilize the EM algorithm instead, which iteratively converges to the Maximum Likelihood (ML) estimate. The disadvantage of the EM method is that it is very sensitive to numerical representation, and easily finds local maxima. In order for it to converge to the true ML estimate, it has to be initialized relatively close to the right answer with the proper number of processes and plausible parameters.

*Initial estimates and OPTICS clustering*

One way to generate initialization estimates could be to employ search heuristics like the GA or Metropolis Markov Chain Monte Carlo (MMCMC) that could find a near ML estimate, which could then be subsequently refined with the EM algorithm. As for the number of sources, some information criterion like the AIC (in the original sense) or Bayesian Information Criterion (BIC) could be employed to find a model with the right number of independent processes giving a high likelihood estimation without overfitting the data.

However, the problem is that all of these approaches rely on the GMM assumption, which may not be an appropriate model for all cases. Even if the valid AE measurements may be approximated by Gaussian distributions, the set of false positives may not. Also, adhering to the GMM, outliers can adversely affect the estimation procedure. Hence, we ended up utilising a completely different, clustering-based approach, which can provide EM initialization (given the GMM assumption holds), or alternatively can directly provide rough source number and parameter estimates.

Our approach to turn towards density-based clustering algorithms was motivated by the observation that valid recordings from the same source formed dense clusters. The Ordering Points To Identify the Clustering Structure (OPTICS) algorithm proved to be uniquely suitable in our situation as it can too adapt to the analysed data itself and does not require a priori information [5]. The main idea is to order all the measurement points by traversing them sequentially. Starting from an arbitrary point the next available recording is selected

that is (i) in the densest environment and is (ii) in the neighborhood of previous selections. For example, in case of a single Gaussian distribution the algorithm will converge to the densest middle part and subsequently work its way out layer by layer. For each measurement point the so called reachability distance is stored, which tells how far it is from the previous point in the ordered list. Clusters will show up as dents in the ordered reachability distance list. To find the cluster beginning and end the steepness of the curve is evaluated. Clusters start where the distance between consecutive points steeply decreases, and end where the distance steeply increases. The algorithm requires a few tuning parameters, i.e., level of steepness, maximum radius for density calculations, etc. However, we chose very generic non-restrictive values, and the algorithm was not sensitive to these parameters, and would converge to the proper results for a wide range of parameter values.

An important feature of this method is that it is capable of discovering hierarchical clusters. Once again this information on hierarchy can be represented as a tree, and it is up to the user to decide which decomposition is the – in some sense – best. Selecting the right clusters or the right level in the hierarchy is a non trivial problem. On one hand we try to have clusters include as many points as possible, so that statistical estimates are more accurate, but at the same time an overly all-embracing cluster will include several outliers, false positives, and points originating from other sources. There are many possibilities to verify that a cluster sufficiently encapsulates relevant measurements. Again assuming that valid AE events form Gaussian distributions, one tactic is to use statistical indicators, e.g. look for clusters that (i) have certain kurtosis and skewness values, or (ii) perform well at the Anderson-Darling test for normal distribution. The problem – as with all statistical approaches – is that in order to be able to meaningfully reason on sample set distributions, relatively large number of measurement points have to be available. This is clearly undesirable in our case, as we want to estimate as soon as possible, and very likely do not have the luxury of being able to collect hundreds of cracking events before forming a decision. By that time it might be already to late.

| | Aluminium | Steel |
|---|---|---|
| AIC-based | -3.5 | -4 |
| Reciprocal | -2 | -3 |

**Table 5:** Log quality index thresholds for valid AE events.

Instead we chose a simpler approach where we specified constraints corresponding to (i) the two dimensions of the measurement space and (ii) number of points. The first constraint stated that only those clusters are of interest that have a mean log quality index higher than certain threshold values shown in Table 5. For the second constraint, the time difference for a

distance of 30 cm was calculated (based on the sound speed estimate), and any cluster with a time difference standard deviation higher than that value was discarded. The last constraint specified that a cluster has to include at least 4 points. Among hierarchical clusters satisfying these conditions we only kept the parents at the top most level.

## Results

### Measurement procedure

| | Aluminium | | Steel |
|---|---|---|---|
| | 1st beam | 2nd beam | |
| length [m] | 3.35 | 2.44 | 3.35 |
| height [cm] | 7.6 | 7.6 | 7.6 |
| width [cm] | 6.4 | 6.4 | 5.9 |

**Table 6:** Dimensions of the tested metal beams.

To determine our system's capability, two aluminium American Standard 6061-T6 type I-beams and a S3x5.7 section of ASTM A36 steel beam were tested, see Table 6. First, the beams were partially sawn in the middle, so that damages would form in a reasonable amount of time at a known location under a reasonable load. The beams were then mounted to supports on both ends, and an electro-mechanical shaker below the middle of the specimen was connected to the beam center with a tight link that would not impede crack growth. The system of two supports and the shaker-specimen link formed 3-point bending conditions, see Figure 27.

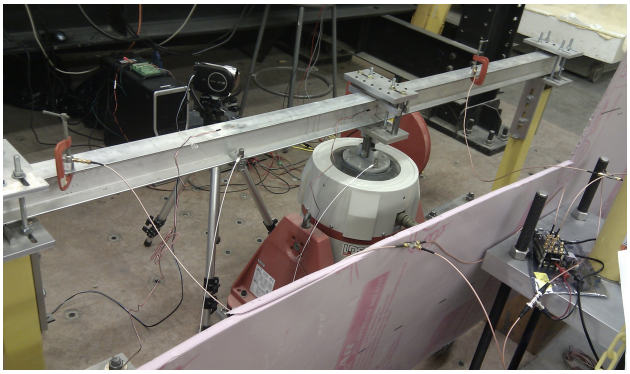
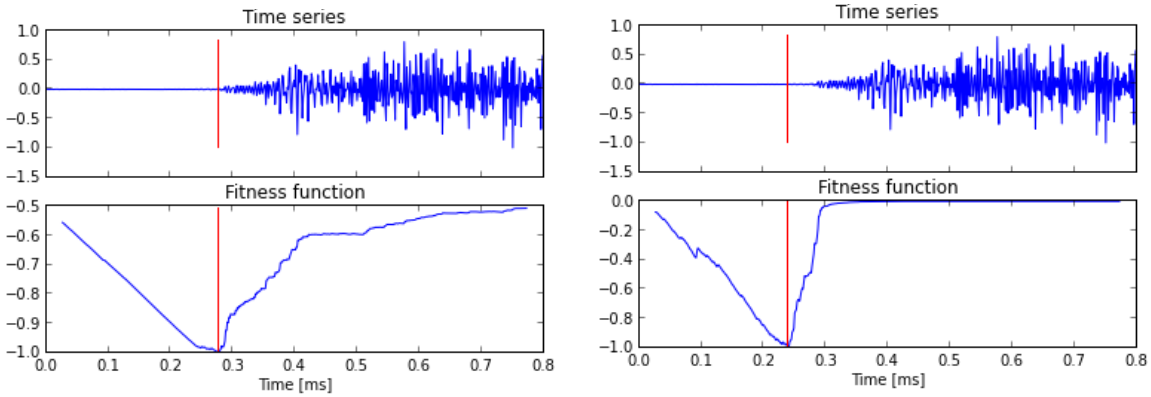


**Figure 27:** Aluminium break test setup, with 2.44 m long beam.

Measurement sessions consisted of several approximately 20 minute long intervals, employing successively increasing shaker amplitudes. Two PKWDI AE microphones were mounted on the beams at different distances from the crack.

### Onset time estimation

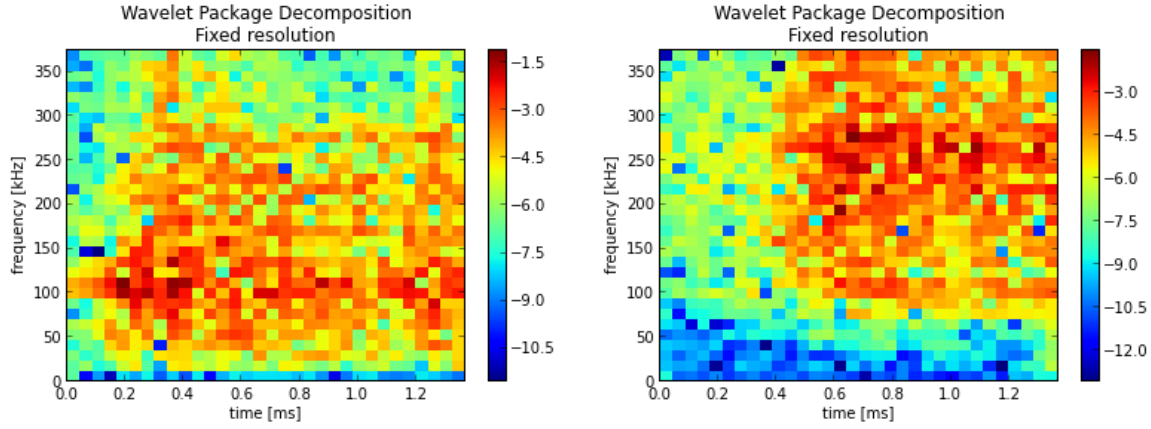
Figure 28a and 28b show the AIC- and reciprocal-based onset time selector results respectively, with the latter seemingly giving an overly early onset time estimate. Proper magnification reveals that the first signal components have indeed arrived at that time, so it has actually provided a better estimate in this case. Generally, for the measured signals at hand we have observed the reciprocal method to yield earlier estimates. As previously stated, this is also a matter of how one interprets the beginning of a highly dispersive signal.




**(a)** AIC-based result. **(b)** Reciprocal selector result.

**Figure 28:** Onset time estimation; red vertical line marks the onset time as detected.

### WPD time-frequency analysis




**(a)** AE event with time difference of around 0.8 ms. **(b)** AE event with time difference of around 0.2 ms.

**Figure 29:** Time-frequency characteristics of AE events from two different sources at the first aluminium break setup.

The WPD of the two different valid measurement groups in the first aluminium break test, see Figure 29a and 29b, revealed fundamental differences in energy distribution in the time-frequency domain. This confirmed the different origin theory, as the 0.8 ms recording, unlike the 0.2 ms, had energy concentrated in predominantly the lower frequency regions.

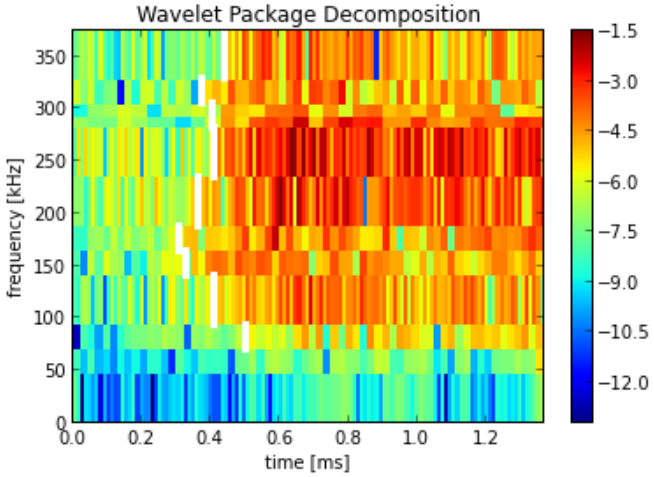
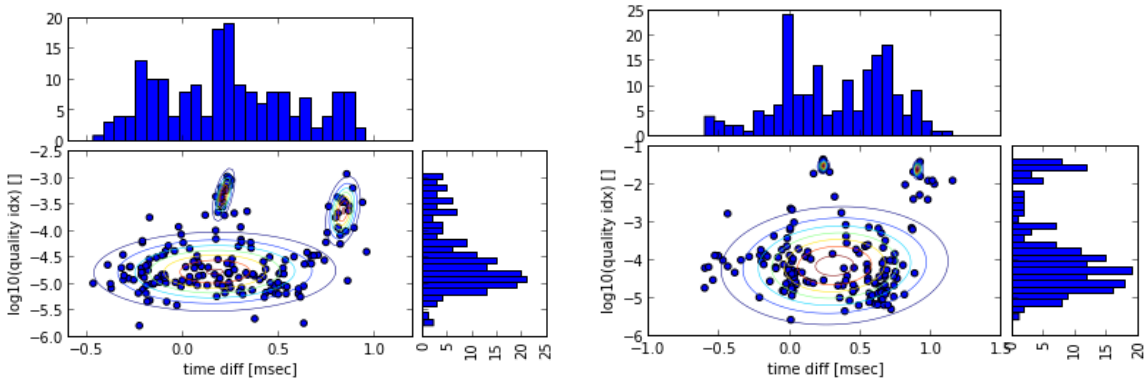



**Figure 30:** Time-frequency characteristics of AE event with time difference of around 0.2 ms at the first aluminium break setup. The white markings indicate where the signal in that band started according to the AIC-based selector.

Figure 30 shows how the above described method adjusted the time-frequency resolution to capture the essence of the signal, which is the same as seen on Figure 29b. In this particular example, compared to the hand chosen fix resolution, the method decided on mainly wider frequency bands with better time resolution. We also applied the AIC-based onset time selector on each signal in the frequency bands to estimate when each component first appeared as marked with white vertical lines. In this case the bottom two bands ended up with no indicators as the algorithm decided there is no significant activity present based on the log quality indices. This shows how in a resource-constrained WSN these bands could be disregarded to save on communication costs. Also, in severely power-constrained setups only the start times within each bands may be transmitted, which would only require a couple of bytes to encode. For this application the AIC-based onset time detection proved to be better because the reciprocal method yielded too early results with much higher variances among bands. Also, this frequency partitioning is not representative for all AE events in the sense that the Shannon entropy-based approach proved to be very sensitive to signal content, and resulted in widely different basis sets for different inputs.

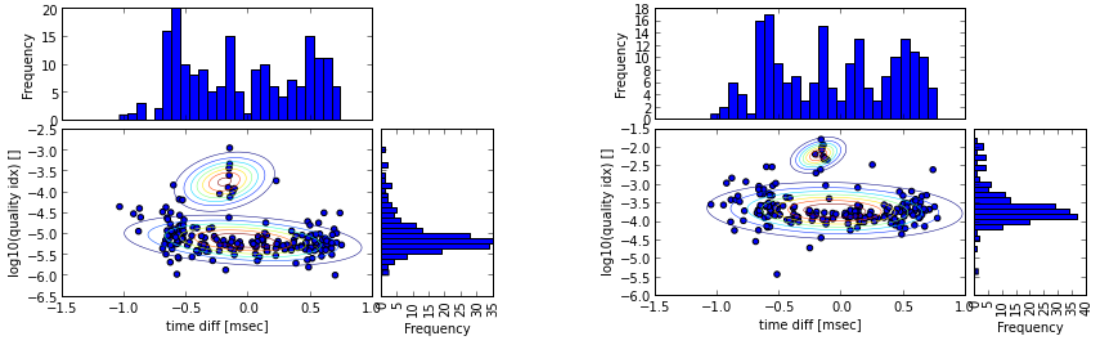
### GMM and EM

Figure 31a and 31b show the results of our EM event grouping and parameter estimation for the first aluminium test. Looking at the AIC results, the events with a log quality index of -3.5 at around 0.2 ms stem from the break in the beam. Points below can be considered useless noise events or false positives. A third cluster unexpectedly appeared as well with high quality indices at 0.8 ms. Closer inspection revealed that it was not caused by reflected waves, but very likely originated from outside the beam (i.e. the supports). Because of the quite different TDoAs, it was simple to categorize the measurements. Note how in this case the reciprocal method provided much tighter grouping of recordings of the same source.




**(a)** AIC-based onset time picker results. **(b)** Reciprocal onset time picker.

**Figure 31:** Gaussian distributions as estimated by the EM algorithm for the first aluminium break test with shaker set to 1.27 cm amplitude.



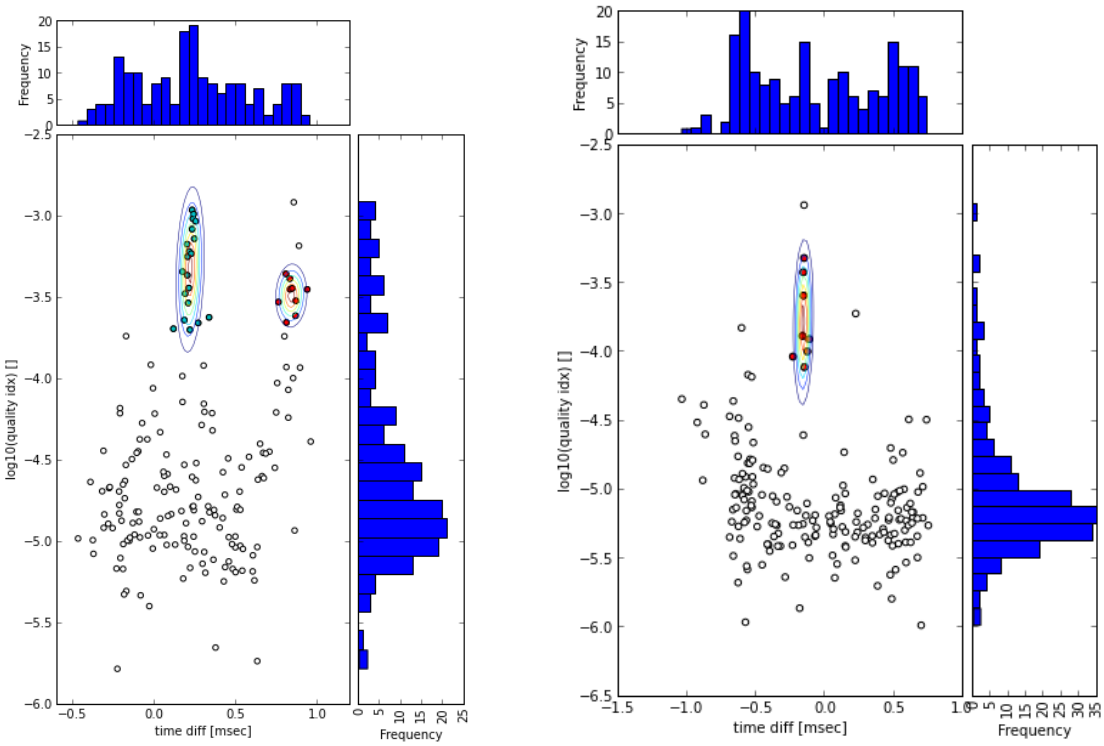

**(a)** AIC-based onset time picker. **(b)** Reciprocal onset time picker.

**Figure 32:** Gaussian distributions as estimated by the EM algorithm for the steel break test with shaker set to 0.51 cm amplitude.

Figure 32a and 32b show the results of our EM event grouping and parameter estimation for the steel test. For the AIC results, the events with a log quality index of -3.5 at around $-0.2$ ms originate from the break in the beam. Points below -4 may be considered useless

noise events or false positives. In this case the two onset time selector methods provided similar performances.

### OPTICS clustering




**(a)** Shaker set to 1.27 cm amplitude.

**(b)** Shaker set to 0.51 cm amplitude.

**Figure 33:** OPTICS clustering results for the AIC-based onset time picker measurements for the steel break test.

Figure 33a and 33b show the OPTICS clustering algorithm filtering out potential clusters for the same measurements as seen previously. Note that some of the recordings classified as outliers by the algorithm are actually valid AE events, however, the method still managed to find the proper number of sources with most of the relevant points.

Figure 34 depicts one of the more challenging clustering problems. In this situation there is no clear separation between false positives and valid points, yet the algorithm managed to find relevant recordings. In fact, out of 18 measurement sets it found the right number of sources 14 times. Out of the remaining 4, in one case it divided the single valid cluster into three adjacent parts at regions, where the measurements where extra dense, and the remaining 3 cases involved sets with very few, altogether less than 35, points.

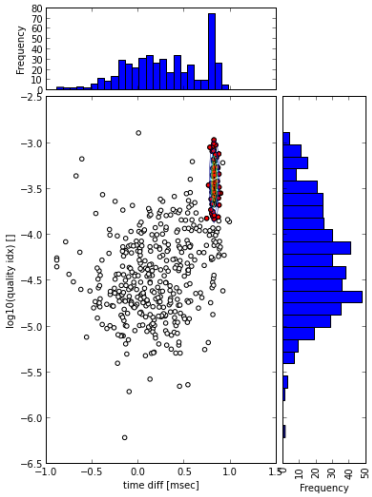



**Figure 34:** OPTICS clustering results for the AIC-based onset time picker measurements for the first aluminium break test with shaker set to 2.54 cm amplitude.

### Localization

Given the simple, one dimensional measurement setup, an accordingly uncomplicated damage localization approach was utilized. The above described onset time selecting methods gave TDoAs, which, in conjunction with accurate sound propagation speed estimates, yielded damage location information. Sound speed was measured and estimated separately, but using the same framework.

The first aluminium beam break test served as proof of concept, and showed that the system worked, but the measurements did not yield accurate results in terms of localization.

| shaker amplitude [cm] | onset time detection | log quality index | time diff [ms] | sound speed $[\frac{\text{m}}{\text{s}}]$ | distance diff [cm] |
|---|---|---|---|---|---|
| 0.25 | AIC based | −2.83 | −0.1475 | 3880 | −57.3 |
| | reciprocal | −2.41 | −0.1276 | 4290 | −54.7 |
| 0.38 | AIC based | −2.41 | −0.1107 | 3880 | −43.0 |
| | reciprocal | −1.64 | −0.1001 | 4290 | −42.9 |
| 0.51 | AIC based | −3.03 | −0.1443 | 3880 | −56.0 |
| | reciprocal | −2.04 | −0.1326 | 4290 | −56.9 |

**Table 7:** The second aluminium break test crack location estimates for two onset time pickers. Actual crack location at −46.2 cm.

Figure 7 gives the overview of the end results for the second break test. Here the sensor positions were accurately measured, the actual distance difference was −46.2 cm. The shaker amplitude was increased gradually and proper cable connections were verified. The results show that the best measurements were recorded at the second measurement run with a shaker amplitude set to 0.38 cm. The quality index is the highest here for both onset time

detections and accordingly the localization is the most accurate here with an error of only around 3 cm. With other measurement runs crack growth location detection suffered a higher error of around 10 cm and had correspondingly lower quality indices.

| shaker amplitude [cm] | onset time detection | log quality index | time diff [ms] | sound speed $[\frac{m}{s}]$ | distance diff [cm] |
|---|---|---|---|---|---|
| 0.38 | AIC based | −3.84 | −0.2134 | 3550 | −75.8 |
| | reciprocal | −2.78 | −0.1913 | 4240 | −81.2 |
| 0.51 | AIC based | −3.74 | −0.1972 | 3550 | −70.0 |
| | reciprocal | −2.19 | −0.1903 | 4240 | −80.8 |

**Table 8:** The steel break test crack location estimates for two onset time pickers. Actual crack location at −78.7 cm.

For the steel break measurement the distance difference ground truth was −78.7 cm. Here the sound speed was measured right before the actual break, which had a very beneficial effect on the reciprocal onset time detection. The error was reduced to around 2 cm. The AIC based method benefited from that as well but still managed to give a worse error of around 9 cm.

### Multi-core transformation

In order to assess the merits of a multi-core device, let's look at an actual SHM system. Assume a structure, for which civil engineering has identified a critical beam that needs to be constantly monitored. Within this context, the above described processing methods are very relevant and applicable.

*Single-core*

Building a system with sensor nodes following the conventional single-core approach is not trivial and may not even be feasible. Taking a MicaZ mote as reference, the clock rate is around 8 MHz (7 372 800 Hz according to the Avrora simulator), while the sampling rate in this application is 750 kHz. This gives only about 10 cycles to fully process every sample, which is not realistic at the level of complexity at hand. Hence, a pure software approach running on a single MCU is not viable. Thus, basic buffering and threshold crossing detection is assumed to be implemented in a separate peripheral device.

The application-specific peripheral takes care of monitoring and recording of the ADC input. It stores measured values in a circular buffer using DMA. Every time a threshold level is crossed, it sends an interrupt to the MCU. The rest of the processing and all other tasks are carried out by software on the MCU. These tasks include the OPTICS clustering,

RF communication, and miscellaneous administrative functions. The EM method and WPD are omitted in this example for the sake of brevity.

*Multi-core*

Figure 35 shows the multi-core version of the SHM application. The thresholding, the onset time detection, the TDoA estimation, and the quality index calculation are moved to IP cores in the fabric. Each channel has these components, so they can provide preprocessed AE events. One core is dedicated to the OPTICS algorithm, which classifies incoming events, and constantly adapts based on previous samples. The result of the classification is then forwarded to the main core that decides on the next step based on the system's overall state. It can examine the state of batteries, the severity of the damage, previous events, received messages. If the event is considered important, a message is generated and forwarded to a third soft-core that is responsible for reliable radio communication and real-time handling of the RF hardware.

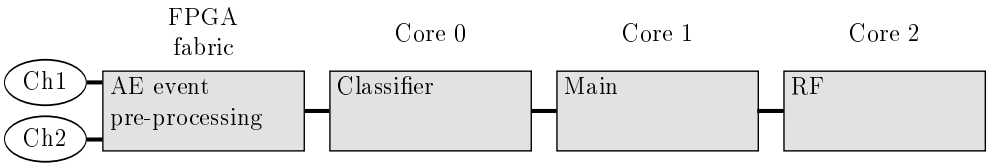



**Figure 35:** Multi-core system architecture.

*Event misses*

Structures can be under constant stress and vibration, e.g., high traffic bridges, airplanes during long distance flights. Thus, in a real life scenario there can be an exceeding number of events, and any event may signal critical structural failure. Hence, a crucial system feature – and the main figure of merit to evaluate the benefits of the architecture – is the probability of event misses.

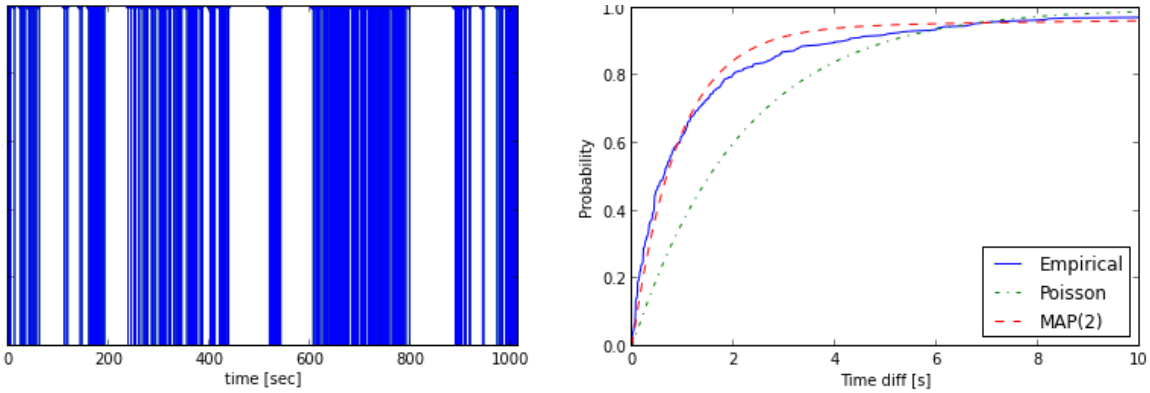
Our shaker experiments are a reasonable starting point, as they simulate the constant stress and vibration that structures are exposed to. The consequence is a series of damage events at different points in time. The success of the system depends on whether it is capable to evaluate an event in real-time before the next one arrives. Hence, the characterization of Inter-Arrival Times (IATs) is the key question.

In queueing theory the most simple way to model the time between the arrival of events is the Poisson or exponential process. The Cumulative Distribution Function (CDF) of such a process is a simple exponential. The Poisson process is a renewal process, so that past events have no influence on current or future arrival times. The simplest example is the

replacement of a malfunctioning device with a new one of the same type. Because after every failure the device is completely renewed, the time to the next failure can be estimated using only the failure rate of the device family. There is no need for any information on previous events.

Renewal processes provide an oversimplified model in our case, because the cracking process is likely influenced by previous cracking events. Hence, a better category of modeling is applied, called non-renewal processes. With these processes, past events have a direct influence on future events. Markov chains are employed as the underlying structure of these models, where event arrivals are associated with certain state transitions. The main issue is determining the number of states and the transitions among them. Increased number of states and transitions provide a better fitting, but at the cost of complexity. Finding the right Markov chain to properly model stochastic processes is a subject of ongoing research, and is beyond the scope of this work. Within this document a simple Markovian model is employed, which can be categorized as a Markovian Arrival Process with two states (MAP(2)) [39, 55, 35]. The most important benefit of the MAP(2) model, compared to more sophisticated models, is that it has analytical formulas for data fitting [36].

Figure 36a shows AE event arrival for the first aluminium break test with a shaker amplitude at 1 inch. Figure 36b shows the CDFs.




**(a)** AE events in time.

**(b)** CDFs of event IAT.

**Figure 36:** AE events for the first aluminium break test with a shaker amplitude set to 1 inch.

The average IAT is 2176 ms, and the MAP(2) model basically alternates between a rarely visited state with a low event generation rate, and an often visited state with a high event generation rate. The results show that the simple exponential model cannot accurately predict IAT. Even though the MAP(2) model has only two states, the CDF estimate provides a much better approximation. A Kolmogorov-Smirnov test would still reveal that the model is not perfect, but it is much closer to the measured values. Also, model fitting could be

improved with additional states, but that is beyond the scope of this work. The goal is not to provide the most accurate characterization of AE event arrivals, but to develop a general understanding of the physical properties and their effects on system design.

**Effects of parallel execution**

One way to mitigate miss rates in the single-core system could be to significantly increase buffer sizes, but, memory is a scarce resource. Also, even if buffer sizes were increased by orders of magnitude, the system could not provide deterministic real-time insight into the current state of the structure. The evaluation delay would be very random and would depend on the current load of the buffers.

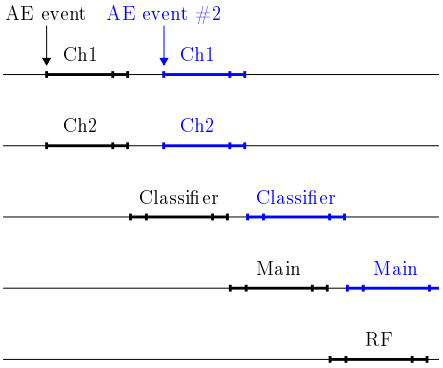
Other way to handle event misses is to increase clock rate. This approach can provide shorter, deterministic response times, and is indeed widely used. However, this method has inherent limitations as well. First, the clock rate has an upper limit, and the higher the clock rate goes, the higher the complexity of associated circuits and consequently their power consumption. Secondly, the dynamic power consumption of CMOS circuits is a linear function of their operating frequency for MCU level ICs. As a side note, this simple linear dependence is becoming less accurate with increasing process resolution. For contemporary complex microprocessors, with transistor counts in the billion range, the static power consumption, due to leakage currents, is significant.

Amdahl's law regarding parallel speedup provides an alternative remedy for the problem. It is well known that the more parts of the algorithm can be run in a parallel manner, the shorter the response time becomes. It is this property that the multi-core system can utilize. This is also heavily exploited in state of the art microprocessors with multi-stage, pipelined architectures.

Figure 37a and 37b show the timing of the single-core and multi-core systems respectively. On the single-core system every task has to be executed on the single execution thread, thus, in order to avoid event losses, all parts of the processing have to be finished before the next event arrives. On the multi-core system, processing steps are executed in a parallel, pipelined manner. This means that the critical time length is reduced to the longest, individual processing time length. However, this is not just the time necessary for the processing, messaging is also included. The bottleneck becomes the longest processing step.

Table 9 shows single-core run time results for an example implementation of the SHM system. Two main observations can be made. Firstly, the total run time of 96.07 ms makes it feasible to duty cycle. Embedded platforms, like the MicaZ, can have multiple clock inputs. Low accuracy, on chip relaxation oscillators are capable of waking up in a matter of µs, while external resonant oscillators require couple ms to wake up in case of an interrupt. Even in


**(a)** Single core system.




**(b)** Multi-core system.

**Figure 37:** Processing time line.

| Processing step | | Δ time [cycle] | [ms] |
|---|---|---|---|
| Onset time, TDoA | Start | | |
| | Finish | 248,832 | 33.75 |
| Event clustering | Start | 187 | 0.03 |
| | Finish | 417,831 | 56.67 |
| Misc management | Start | 261 | 0.04 |
| | Finish | 2,012 | 0.27 |
| RF communication | Start | 10 | 0.00 |
| | Finish | 39,145 | 5.31 |
| | Total | 708,278 | 96.07 |

**Table 9:** Run time results for the single-core SHM system with a clock rate of 7 372 800 Hz. Sampling at 750 kHz for two channels, input buffers 128 samples long, clustering performed based on 10 previous events.

the latter case, the wake up time is negligible compared to the active processing time, so energy can be saved by duty cycling. The extent of this is shown in Table 10. Compared to the active processing state, both idle and power save states consume less energy, with the latter around two orders less.

| State | $P_{dB}$ [dBm] |
|---|---|
| Active | -16.4 |
| Idle | -20.0 |
| Power save | -34.3 |

**Table 10:** Power consumption of a core in different states as measured with the Avrora simulator.

The other main observation is that in the current implementation the majority of time is consumed by the event classification algorithm. A parallel reimplementation here would bring the most benefits. The other significant processing step revolves around the onset time estimation, quality index calculation, and TDoA, detailed in Table 11. Here the run time shows a linear relation with the buffer size, and can quickly become longer than the classification time. Hence, a logical step would be to try to decrease the buffer size. However, the buffer size directly affects system capabilities. A bigger buffer enables the monitoring of longer beams, because larger TDoA values can be detected. Also, the more sophisticated processing steps, e.g. spectrum analysis with WPD, require larger buffers. All these aspects contradict the short response time requirements, which thus may only be achieved employing parallel execution.

| Buffer size | $\Delta$ time | |
|---|---|---|
| [smpls] | [cycle] | [ms] |
| 64 | 127,296 | 17.27 |
| TDoA $\rightarrow$ 128 | 248,832 | 33.75 |
| 256 | 492,928 | 66.86 |
| 512 | 981,118 | 133.07 |
| WPD $\rightarrow$ 1024 | 1,956,670 | 265.39 |
| 2048 | 3,907,582 | 530.00 |

**Table 11:** Run time results of onset time estimation, quality index calculation, and TDoA for different buffer lengths. For TDoA and WPD, the minimum reasonable buffer sizes are marked.

The need for short response times also becomes evident if, based on the CDFs, the number of average lost events is calculated. The average IAT is 2176 ms, so a simplistic approach could assume that a processing time of at most 2 s would be enough to handle events. However, as shown on Figure 38, the actual measurement data reveals four lost events for every processed. The figure shows that for a reliable system the processing time has to be drastically reduced.

Looking at a single-core system, where the processing requires a definitive number of clock cycles, the only way to achieve shorter processing times may be to increase clock frequency, which increases dynamic power consumption. The relation between processing time and active state power consumption is depicted in Figure 39a. The figure demonstrates that below a processing time of 500 ms, which still corresponds to a fairly high average lost event number, the average power consumption can increase by almost three orders of magnitude with decreasing processing time.

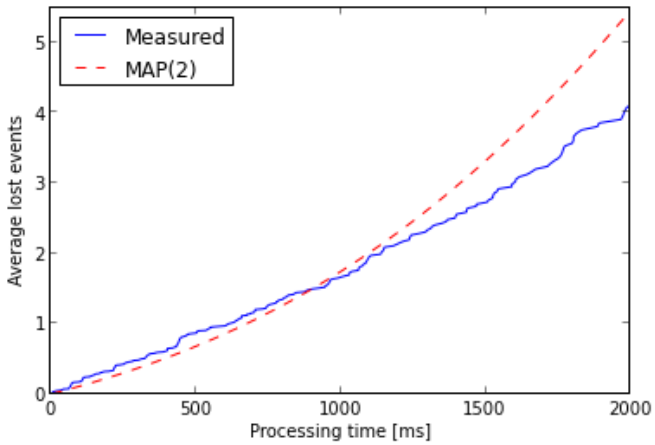
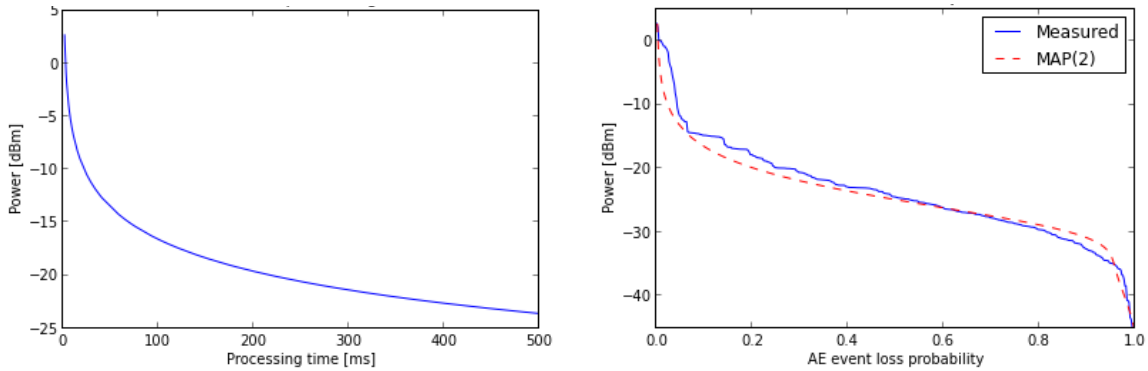



**Figure 38:** Average number of lost events for every processed event.




**(a)** Processing time. **(b)** Event miss probabilities.

**Figure 39:** Power consumption during active processing.

In order to put the power increase in perspective, the processing time can be directly translated into event miss probabilities, shown on Figure 39b. The example single-core system with a total processing time of 96.07 ms may seem a fair compromise based on the previous figure. The average active power consumption is -16.44 dBm, and any decrease in processing time (by increasing the clock rate) would result in steep power increases. However, the new figure shows that this particular power level only corresponds to an event miss probability of around 15 percent, which may be considered unacceptably high in a mission critical situation. Also, the figure shows, that high reliability systems with less than one percent event miss probability require around 40 times more power. As a final note, the figure demonstrates that the MAP(2) model is inadequate to describe event arrival at very low event miss rates.

These active power consumption values are too high for battery operated sensor nodes, so these systems only become feasible once duty cycling is considered. Thus, the length of

the time interval between the finishing of processing an event and the arrival of the next event has to be found. The longer this time interval is, the longer the system may stay in low power mode, thereby saving energy.

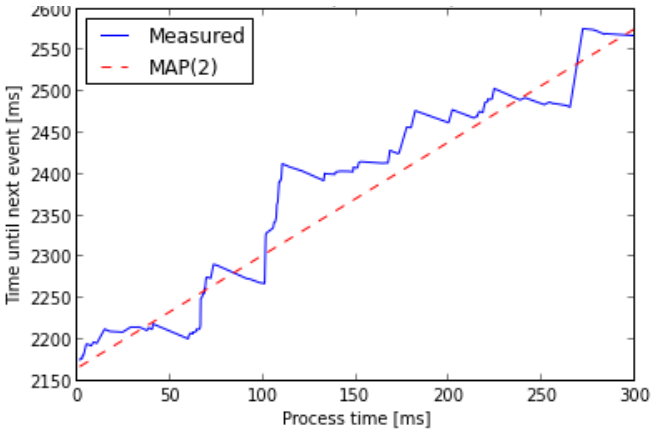



**Figure 40:** Time that can be spent in power save state.

Figure 40 shows the time that can be spent in low power state versus the processing time. The results seem paradoxical, but reveal important properties of underlying physical processes, and highlight why CPSs have to have such tight connections to the physical aspects. The results suggest that the more time the processing takes, the more it has to wait for the next event. The explanation for this stems from the fact that AE events form tight bursts. In other words, if the material gives in and cracks, it is very likely to crack several times in rapid succession. But after this, it will remain stable for a long time, thus increasing the average IAT to over two seconds. The inherent limitations of the Poisson model prevent it from capturing this behaviour altogether, but the MAP(2) captures the essence of actual measurement data.

This has two main consequences. Firstly, if a system takes too much time processing the first event, it will simply miss all the rapidly following events, and will thus miss valuable opportunities to analyze crack growth. Secondly, if the system cannot process events right away, it may have to wait for a long time for the next cracking event in order to gain insight into the structure's state, by which time it may be already too late. Thus, once more the importance of short response times is shown.

With good estimates for low power state times, the effects of duty cycling may be included in the power simulations. Figure 41 depicts the average power consumption of an efficiently duty-cycling single-core system. It can be seen that the overall power consumption dropped by 15 dB, which directly results in improved battery life, but the overall shape of the curve has not changed.

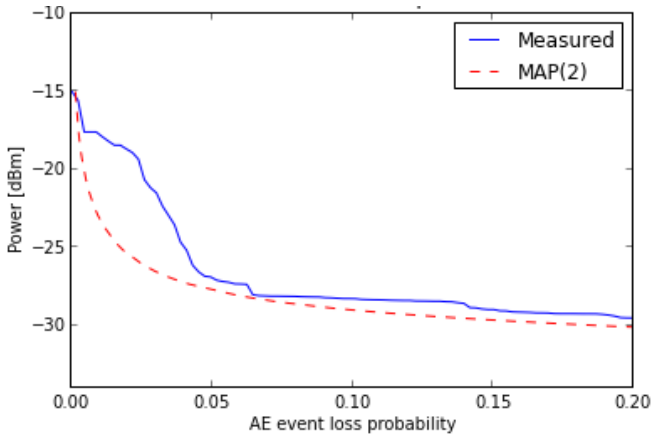



**Figure 41:** Power consumption with efficient duty cycling.

Also, nearly identical curves result if we built the multi-core system, and allow each core (performing a different processing step) to run at its own clock rate. The bottleneck is the longest processing step, and there is no real gain for the other cores to finish faster. So, cores that perform simple processing steps may be slowed down, so that each require the same amount of processing time. With reduced clock rates, significant power savings may be achieved, which balance the cumulative consumption of the several parallel cores.

In that regard, a couple slower cores are equivalent to one fast core, which would make the latter the more preferable solution, as it is much simpler. However, clock rates cannot be increased above a certain level. This is not just a question of increasing power consumption. Large systems are difficult to synchronize due to clock skew and clock slew issues. Above a certain complexity, systems naturally tend to be GALS.

If the clock rate is fixed, and thus the consumed power is fixed, the single-core approach will take longer time to process events, and consequently the event loss probability will increase. From the other point of view, the single-core solution can only achieve the same processing time, and thus event loss probability, if its clock is faster, resulting in a higher power consumption. This relation is depicted on Figure 42.

Due to the nature of these curves, it is obvious, and all models agree on this, that at high event loss probabilities and low power consumptions, the parallel solution is more reliable than the single-core. For example, for the MAP(2) curve at around -26 dBm the event loss probability is 5 percent, while for the parallel system it is less than 3 percent. Also, the measurement data shows a clear advantage at low event loss rates, which is not captured by any of the models. At around -18 dBm power consumption, the single-core solution provides a loss probability of 2.5 percent, while the parallel approach is around 0.5 percent. A five fold improvement.

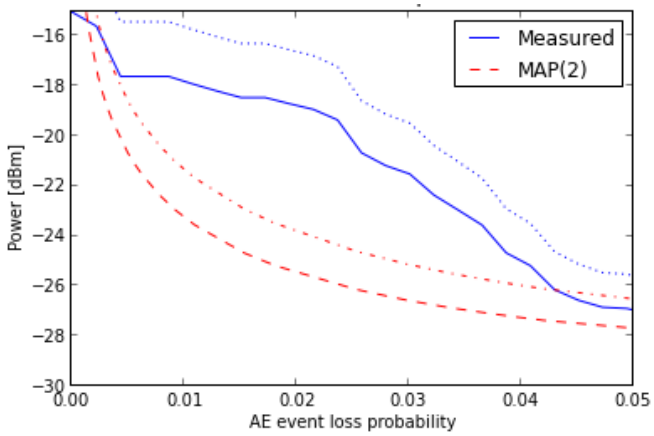
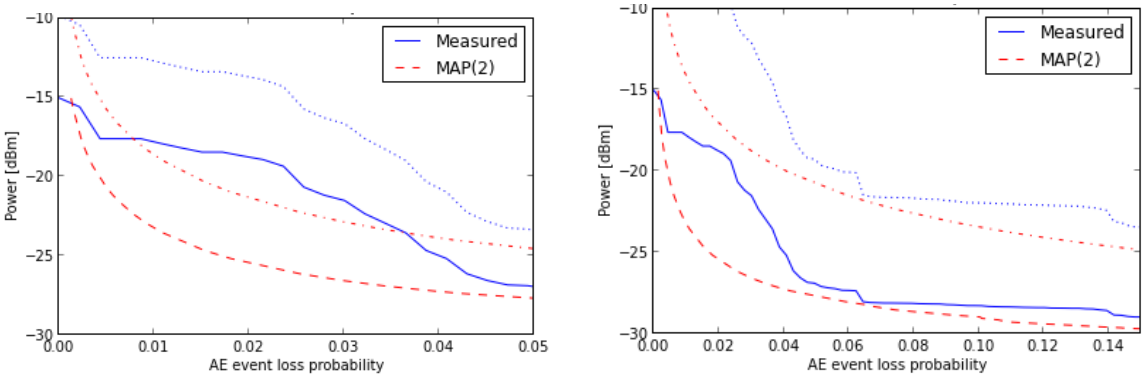



**Figure 42:** Multi-core versus single core (dotted line) event loss probability and power consumption.

However, the clustering algorithm largely dominates the overall run time, and is an obstacle for true parallel execution. With a more balanced distribution, better improvements can be achieved as seen in Figure 43a. For low event loss probabilities the improvement in power is 5 dB. For -18 dBm power consumption the event loss probability shows a seven fold improvement. However, an order of magnitude improvement, 10 dB, can be achieved in power if the 10 cores, maximally provided by the platform, are equally employed, as seen in Figure 43b. The probability improvement for -18 dBm power consumption is about eight fold at more than 4 percent, with even better results at higher probabilities.




**(a)** 5 cores. **(b)** 10 cores.

**Figure 43:** Multi-core versus single core (dotted line) event loss probability and power consumption. Balanced parallel execution.

# CHAPTER VII

# CONCLUSION

Conventional embedded platforms predominantly employ a single microcontroller with additional application-specific ICs. An emerging subset of embedded systems and CPSs have multiple channels and high sampling rates to observe various physical phenomena. The high throughput and computational requirements of these applications render the single MCU approach infeasible, due to the required high clock rates and correspondingly increased power consumption. One approach to overcome the issue is to add an FPGA to the platform in order to implement application-specific processing in configurable hardware. However, the high inrush currents, associated with the duty-cycling of conventional SRAM FPGAs, prevented the application of these devices in the power constrained embedded environment. The application of the novel flash FPGA technology mitigates this problem. Our prototype sensor platform, called MarmotE, is an example of this concept.

However, development of FPGA-based applications is more complex, less flexible and the number of developers familiar with the technologies is limited. Moreover, the conventional approach has a large legacy code base. The key contribution of this thesis stems from this observation, and suggests the instantiation of several soft cores in the configurable hardware fabric. A soft core is basically a fully functional MCUs implemented in a hardware description language, and thus can be instantiated – even multiple times – within most FPGAs. The resulting multi-core architecture provides parallel improvements, in accordance with Amdahl's Law, yet keeps the familiar MCU abstraction for computation. Synthesis results have shown that up to 10 soft cores may fit in the currently available flash FPGAs. We propose an architecture based on a loosely coupled network of cores, because cores can operate largely independently on separate dedicated tasks, each with their own processing and timing requirements. This way cores may run at different clock rates to provide optimum power consumption. To facilitate this architecture, a queue-based messaging framework was developed with corresponding hardware and software abstractions.

The new architecture requires an accompanying application development environment. The nesC was chosen as the programming language, as its programming model enables and enforces modularity that is crucial in partitioning and assigning independent tasks to cores. Regular procedural languages typically yield monolithic programs, which are hard to automatically analyze and partition. The necessary communication components were added

to TinyOS, the modular embedded operating system built on top of nesC, to transparently support the queue-based messaging framework in the hardware. Also, the single core development environment was augmented to help guide the transition of single core projects to the multi-core platform.

A significant addition to this environment has been the improved version of the cycle accurate simulator, called Avrora, which is now capable to fully support multi-core platforms. It is able to simulate and evaluate a network of sensors employing the same binaries that are eventually downloaded into the soft cores.

Finally, a comprehensive case study has been conducted in the field of SHM to demonstrate the requirements and properties of a concrete application. It showcased the level of complexity for contemporary signal processing, and demonstrated an application that can benefit from the computational improvements provided by the parallel platform.

The main advantage of the architecture is that for time critical applications, it can provide better power consumption and response time properties by effectively pipelining tasks. The architecture is especially beneficial, if most of the available cores of the architecture can be equally utilized.

# APPENDIX A

## ANALYSIS AND PROOF OF ONSET TIME PICKER METHODS

Some assumptions:

- time series are upper and lower bounded, more specifically $-1 < x_i < 1$ for $i = 1..N$
- $\gamma_1^2 \ll 1$
- constant zero mean value within any time series
- the biased variance estimation ($\widehat{\sigma}^2$) is accurate enough, thus for our practical purposes:

$$\sigma^2 = \widehat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \frac{1}{N}\Big(\sum_{i=1}^{N} x_i\Big)^2$$

The goal is to come up with a method that, based on the below shown variables, can estimate the start of the signal change. There are many ways to achieve this, the approach taken here is going to be the definition of a "fitness" function $f(n_1, n_2, \sigma_1^2, \sigma_2^2)$ that gives either a minimum or a maximum at the point of change. Many such functions exist, this document focuses primarily on two of them referred to as the "simplified onset time picker following the AIC form" and the "reciprocal onset time picker".

Throughout the proofs the lemma in APPENDIX B is going to be employed.

### Proof for constant variance time series

Some assumptions:

- Constant variance within first time series: $\sigma_1^2 = {\gamma_1}^2 = \beta^2$

*Simplified onset time picker following the AIC form*

The fitness function: $n_1 \ln \sigma_1^2 + n_2 \ln \sigma_2^2$

The statement to prove here is that the right side in the following inequality is always less than the left:

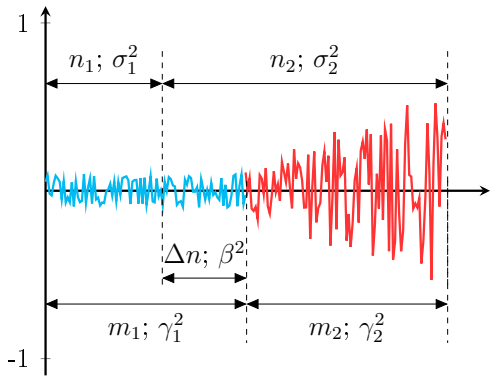



**Figure 44:** Two consecutive time series with different variances. $n_1$, $n_2$, $m_1$, $m_2$, and $\Delta n$ are the length (in samples), $\sigma_1^2$, $\sigma_2^2$, $\gamma_1^2$, $\gamma_2^2$, and $\beta^2$ are the corresponding variances.

$$
\begin{aligned}
n_1 \ln \sigma_1^2 + n_2 \ln \sigma_2^2 &\quad ? \quad m_1 \ln \gamma_1^2 + m_2 \ln \gamma_2^2 \\
\sigma_1^{2n_1} \sigma_2^{2n_2} &\quad ? \quad \gamma_1^{2m_1} \gamma_2^{2m_2} \\
\gamma_1^{2n_1} \sigma_2^{2n_2} &\quad ? \quad \gamma_1^{2m_1} \gamma_2^{2m_2} \\
\sigma_2^{2n_2} &\quad ? \quad \gamma_1^{2m_1 - n_1} \gamma_2^{2m_2} \\
\sigma_2^{2n_2} &\quad ? \quad \gamma_1^{2\Delta n} \gamma_2^{2m_2} \\
\sigma_2^2 = \frac{\Delta n}{n_2}\beta^2 + \frac{m_2}{n_2}\gamma_2^2 = \frac{\Delta n}{n_2}\gamma_1^2 + \frac{m_2}{n_2}\gamma_2^2 &\quad ? \quad \left(\gamma_1^2\right)^{\frac{\Delta n}{n_2}} \left(\gamma_2^2\right)^{\frac{m_2}{n_2}} \\
\left(1 - \frac{m_2}{n_2}\right)\gamma_1^2 + \frac{m_2}{n_2}\gamma_2^2 &\quad ? \quad \left(\gamma_1^2\right)^{1 - \frac{m_2}{n_2}} \left(\gamma_2^2\right)^{\frac{m_2}{n_2}} \\
1 - \frac{m_2}{n_2} + \frac{m_2}{n_2}\frac{{\gamma_2}^2}{\gamma_1^2} &\quad ? \quad \left(\frac{\gamma_2^2}{\gamma_1^2}\right)^{\frac{m_2}{n_2}} \\
1 + \frac{m_2}{n_2}\left(\frac{\gamma_2^2}{\gamma_1^2} - 1\right) &\quad ? \quad \left(\frac{\gamma_2^2}{\gamma_1^2}\right)^{\frac{m_2}{n_2}} \\
1 + r\alpha &\quad ? \quad (1+\alpha)^r
\end{aligned}
$$

Which is Bernoulli's inequality for $\alpha = \frac{\gamma_2^2}{\gamma_1^2} - 1$, $r = \frac{m_2}{n_2}$, and since

- $\alpha => -1$
- $\alpha \neq 0$ because $\gamma_2^2 \neq \gamma_1^2$
- $0 \leq r < 1$

it follows:

$$
\begin{aligned}
1 + r\alpha &> (1+\alpha)^r \\
n_1 \ln \sigma_1^2 + n_2 \ln \sigma_2^2 &> m_1 \ln \gamma_1^2 + m_2 \ln \gamma_2^2
\end{aligned}
$$

*Reciprocal onset time picker*

$$
\begin{aligned}
-\frac{n_1}{\sigma_1^2} - \frac{n_2}{\sigma_2^2} &\;?\; -\frac{m_1}{\gamma_1^2} - \frac{m_2}{\gamma_2^2} \\
-\frac{n_1}{\gamma_1^2} - \frac{n_2}{\sigma_2^2} &\;?\; -\frac{m_1}{\gamma_1^2} - \frac{m_2}{\gamma_2^2} \\
-\frac{n_2}{\sigma_2^2} &\;?\; -\frac{\Delta n}{\gamma_1^2} - \frac{m_2}{\gamma_2^2} \\
-\frac{n_2^2}{\Delta n \beta^2 + m_2\gamma_2^2} = -\frac{n_2^2}{\Delta n \gamma_1^2 + m_2\gamma_2^2} &\;?\; -\frac{\Delta n}{\gamma_1^2} - \frac{m_2}{\gamma_2^2} \\
\frac{\Delta n}{\gamma_1^2} + \frac{m_2}{\gamma_2^2} &\;?\; \frac{n_2^2}{\Delta n \gamma_1^2 + m_2\gamma_2^2} \\
\frac{\Delta n \gamma_2^2 + m_2\gamma_1^2}{\gamma_1^2\gamma_2^2} &\;?\; \frac{n_2^2}{\Delta n \gamma_1^2 + m_2\gamma_2^2} \\
\left(\Delta n \gamma_2^2 + m_2\gamma_1^2\right)\left(\Delta n \gamma_1^2 + m_2\gamma_2^2\right) &\;?\; n_2^2\gamma_1^2\gamma_2^2 \\
(\Delta n)^2\gamma_2^2\gamma_1^2 + \Delta n\, m_2\left(\gamma_2^2\right)^2 + m_2\,\Delta n\left(\gamma_1^2\right)^2 + (m_2)^2\gamma_1^2\gamma_2^2 &\;?\; (m_2 + \Delta n)^2\gamma_1^2\gamma_2^2 \\
\Delta n\, m_2\left(\left(\gamma_2^2\right)^2 + \left(\gamma_1^2\right)^2\right) + \gamma_1^2\gamma_2^2\left((\Delta n)^2 + (m_2)^2\right) &\;?\; \left((\Delta n)^2 + 2\Delta n\, m_2 + (m_2)^2\right)\gamma_1^2\gamma_2^2 \\
\Delta n\, m_2\left(\left(\gamma_2^2\right)^2 + \left(\gamma_1^2\right)^2\right) &\;?\; 2\Delta n\, m_2\gamma_1^2\gamma_2^2 \\
\left(\gamma_2^2\right)^2 + \left(\gamma_1^2\right)^2 &\;?\; 2\gamma_1^2\gamma_2^2 \\
\left(\gamma_2^2 - \gamma_1^2\right)^2 &\;?\; 0
\end{aligned}
$$

Which is a trivial inequality and $\gamma_2^2 \neq \gamma_1^2$, thus:

$$
\begin{aligned}
\left(\gamma_2^2 - \gamma_1^2\right)^2 &> 0 \\
-\frac{n_1}{\sigma_1^2} - \frac{n_2}{\sigma_2^2} &> -\frac{m_1}{\gamma_1^2} - \frac{m_2}{\gamma_2^2}
\end{aligned}
$$

*Power of two onset time picker*

The fitness function: $n_1\left(\sigma_1^2\right)^2 + n_2\left(\sigma_2^2\right)^2$

$$
\begin{aligned}
n_1\left(\sigma_1^2\right)^2 + n_2\left(\sigma_2^2\right)^2 &\;?\; m_1\left(\gamma_1^2\right)^2 + m_2\left(\gamma_2^2\right)^2 \\
n_1\left(\gamma_1^2\right)^2 + n_2\left(\sigma_2^2\right)^2 &\;?\; m_1\left(\gamma_1^2\right)^2 + m_2\left(\gamma_2^2\right)^2 \\
n_2\left(\sigma_2^2\right)^2 &\;?\; \Delta n\left(\gamma_1^2\right)^2 + m_2\left(\gamma_2^2\right)^2 \\
n_2^2\left(\sigma_2^2\right)^2 &\;?\; n_2\Delta n\left(\gamma_1^2\right)^2 + n_2 m_2\left(\gamma_2^2\right)^2 \\
\left(\Delta n\beta^2 + m_2\gamma_2^2\right)^2 = \left(\Delta n\gamma_1^2 + m_2\gamma_2^2\right)^2 &\;?\; n_2\Delta n\left(\gamma_1^2\right)^2 + n_2 m_2\left(\gamma_2^2\right)^2 \\
(\Delta n)^2\left(\gamma_1^2\right)^2 + 2\Delta n\, m_2\gamma_1^2\gamma_2^2 + m_2^2\left(\gamma_2^2\right)^2 &\;?\; n_2\Delta n\left(\gamma_1^2\right)^2 + n_2 m_2\left(\gamma_2^2\right)^2 \\
\Delta n(\Delta n - n_2)\left(\gamma_1^2\right)^2 + 2\Delta n\, m_2\gamma_1^2\gamma_2^2 + m_2(m_2 - n_2)\left(\gamma_2^2\right)^2 &\;?\; 0 \\
-\Delta n\, m_2\left(\gamma_1^2\right)^2 + 2\Delta n\, m_2\gamma_1^2\gamma_2^2 - m_2\Delta n\left(\gamma_2^2\right)^2 &\;?\; 0 \\
-\left(\gamma_1^2\right)^2 + 2\gamma_1^2\gamma_2^2 - \left(\gamma_2^2\right)^2 &\;?\; 0 \\
0 &\;?\; \left(\gamma_1^2 - \gamma_2^2\right)^2
\end{aligned}
$$

Which is a trivial inequality and $\gamma_2^2 \neq \gamma_1^2$, thus:

$$
\begin{aligned}
0 &< \left(\gamma_1^2 - \gamma_2^2\right)^2 \\
n_1\left(\sigma_1^2\right)^2 + n_2\left(\sigma_2^2\right)^2 &< m_1\left(\gamma_1^2\right)^2 + m_2\left(\gamma_2^2\right)^2
\end{aligned}
$$

*Other usable methods without proof*

- ln replaced by its Taylor series: $-n_1\sum_{i=1}^{I}\frac{\left(1-\sigma_1^2\right)^i}{i} - n_2\sum_{i=1}^{I}\frac{\left(1-\sigma_2^2\right)^i}{i}$, which works reasonably well if $I > 64$
- $-n_1\left(1-\sigma_1^2\right)^I - n_2\left(1-\sigma_2^2\right)^I$, which works reasonably well if $I > 64$
- $-\frac{n_1^2}{\sigma_1^2} - \frac{n_2^2}{\sigma_2^2}$

- $-\frac{\sqrt{n_1}}{\sigma_1^2}-\frac{\sqrt{n_2}}{\sigma_2^2}$
- $n_1\big(1-\frac{1}{\sigma_1^2}\big)+n_2\big(1-\frac{1}{\sigma_2^2}\big)$
- $\left(\sum_{i=1}^{n_1} i\right)\left(1-\frac{1}{\sigma_1^2}\right)+\left(\sum_{i=1}^{n_2} i\right)\left(1-\frac{1}{\sigma_2^2}\right)$
- $n_1\left(1-\frac{1}{\sigma_1^{2\,\frac{1}{I}}}\right)+n_2\left(1-\frac{1}{\sigma_2^{2\,\frac{1}{I}}}\right)$

*Unusable methods without proof*

- $n_1\sigma_1^2+n_2\sigma_2^2$
- $\left(\sigma_1^2\right)^{n_1}+\left(\sigma_2^2\right)^{n_2}$
- $n_1\big(\sigma_2^2\big)^2+n_2\big(\sigma_2^2\big)^2$
- $\left(n_1+\sigma_1^2\right)\left(n_2+\sigma_2^2\right)$
- $\frac{-1}{n_1\sigma_2^2}+\frac{-1}{n_2\sigma_1^2}$

**Proof for monotone increasing variance time series**

In this section a dispersive signal is assumed. It is analysed if the above defined fitness functions still hold up under these conditions.

The assumptions are:

- monotone increasing variances: $0<\gamma_1^2<\beta^2<\sigma_2^2\leq 1$

*Simplified onset time picker following the AIC form*

$$
\begin{aligned}
n_1\ln\sigma_1^2+n_2\ln\sigma_2^2 &> m_1\ln\gamma_1^2+m_2\ln\gamma_2^2\\
\sigma_1^{2n_1}\sigma_2^{2n_2} &> \gamma_1^{2m_1}\gamma_2^{2m_2}\\
\left(\frac{m_1\gamma_1^2+\Delta n\beta^2}{m_1+\Delta n}\right)^{n_1}\sigma_2^{2n_2} &> \gamma_1^{2m_1}\left(\frac{\Delta n\beta^2+n_2\sigma_2^2}{\Delta n+n_2}\right)^{m_2}\\
\left(\frac{m_1\gamma_1^2+\Delta n\beta^2}{m_1+\Delta n}\right)^{m_1+\Delta n}\sigma_2^{2n_2} &> \gamma_1^{2m_1}\left(\frac{\Delta n\beta^2+n_2\sigma_2^2}{\Delta n+n_2}\right)^{\Delta n+n_2}
\end{aligned}
$$

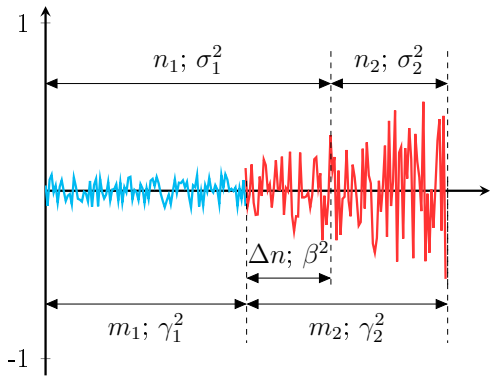



**Figure 45:** Two consecutive time series with different variances. $n_1$, $n_2$, $m_1$, $m_2$, and $\Delta n$ are the length (in samples), $\sigma_1^2$, $\sigma_2^2$, $\gamma_1^2$, $\gamma_2^2$, and $\beta^2$ are the corresponding variances.

It's easy to see this relation, the base is a smaller (and $< 1$) and the exponent is a larger value:

$$\sigma_2^{2n_2} > \left(\frac{\Delta n\beta^2 + n_2\sigma_2^2}{\Delta n + n_2}\right)^{\Delta n + n_2}$$

Thus to have the method working the following has to hold:

$$\begin{aligned}
\left(\frac{m_1\gamma_1^2 + \Delta n\beta^2}{m_1 + \Delta n}\right)^{m_1+\Delta n} &> \gamma_1^{2m_1} \\
\frac{m_1\gamma_1^2 + \Delta n\beta^2}{m_1 + \Delta n} &> \gamma_1^{2\frac{m_1}{m_1+\Delta n}} \\
m_1\gamma_1^2 + \Delta n\beta^2 &> (m_1 + \Delta n)\gamma_1^{2\frac{m_1}{m_1+\Delta n}} \\
\Delta n\beta^2 &> (m_1 + \Delta n)\gamma_1^{2\frac{m_1}{m_1+\Delta n}} - m_1\gamma_1^2 \\
\beta^2 &> \left(\frac{m_1}{\Delta n} + 1\right)\gamma_1^{2\frac{m_1}{m_1+\Delta n}} - \frac{m_1}{\Delta n}\gamma_1^2
\end{aligned}$$

$$\begin{aligned}
-\frac{n_1}{\sigma_1^2} - \frac{n_2}{\sigma_2^2} &> -\frac{m_1}{\gamma_1^2} - \frac{m_2}{\gamma_2^2} \\
\frac{m_1}{\gamma_1^2} + \frac{m_2}{\gamma_2^2} &> \frac{n_1}{\sigma_1^2} + \frac{n_2}{\sigma_2^2} \\
\frac{m_1}{\gamma_1^2} + \frac{(\Delta n + n_2)^2}{\Delta n \beta^2 + n_2 \sigma_2^2} &> \frac{(m_1 + \Delta n)^2}{m_1 \gamma_1^2 + \Delta n \beta^2} + \frac{n_2}{\sigma_2^2}
\end{aligned}$$

It's easy to see this relation, the numerator is a larger and the denominator is a smaller value:

$$\frac{(\Delta n + n_2)^2}{\Delta n \beta^2 + n_2 \sigma_2^2} > \frac{n_2}{\sigma_2^2} = \frac{n_2^2}{n_2 \sigma_2^2}$$

Thus to have the method working the following has to hold:

$$\begin{aligned}
\frac{m_1}{\gamma_1^2} &> \frac{(m_1 + \Delta n)^2}{m_1 \gamma_1^2 + \Delta n \beta^2} \\
m_1^2 \gamma_1^2 + m_1 \Delta n \beta^2 &> \gamma_1^2 m_1^2 + \gamma_1^2 2 m_1 \Delta n + \gamma_1^2 (\Delta n)^2 \\
m_1 \Delta n \beta^2 &> \gamma_1^2 2 m_1 \Delta n + \gamma_1^2 (\Delta n)^2 \\
m_1 \beta^2 &> \gamma_1^2 2 m_1 + \gamma_1^2 \Delta n \\
\beta^2 &> \gamma_1^2 \Big( 2 + \frac{\Delta n}{m_1} \Big)
\end{aligned}$$

*Comparison of $\beta$ for the two methods*

$$
\begin{aligned}
\gamma_1^2\Big(2+\frac{\Delta n}{m_1}\Big) \quad &? \quad \Big(\frac{m_1}{\Delta n}+1\Big)\gamma_1^{2\,\frac{m_1}{m_1+\Delta n}} - \frac{m_1}{\Delta n}\gamma_1^2 \\
2+\frac{\Delta n}{m_1}+\frac{m_1}{\Delta n} \quad &? \quad \Big(\frac{m_1}{\Delta n}+1\Big)\gamma_1^{2\,\frac{-\Delta n}{m_1+\Delta n}} \\
\frac{(m_1+\Delta n)^2}{m_1(m_1+\Delta n)} \quad &? \quad \gamma_1^{2\,\frac{-\Delta n}{m_1+\Delta n}} \\
1+\frac{\Delta n}{m_1} \quad &? \quad \gamma_1^{2\,\frac{-1}{\frac{m_1}{\Delta n}+1}} \\
1+\frac{\Delta n}{m_1} \quad &? \quad \frac{1}{\gamma_1^{2\,\frac{1}{\frac{m_1}{\Delta n}+1}}} \\
\gamma_1^{2\,\frac{1}{\frac{m_1}{\Delta n}+1}} \quad &? \quad \frac{1}{1+\frac{\Delta n}{m_1}} \\
\gamma_1^2 \quad &? \quad \left(\frac{1}{1+\frac{\Delta n}{m_1}}\right)^{\frac{m_1}{\Delta n}+1}
\end{aligned}
$$

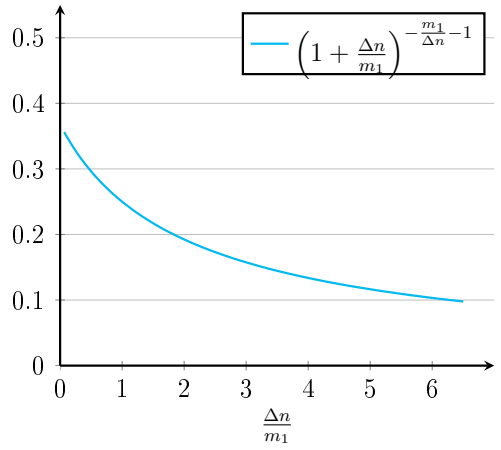



**Figure 46:** Relation of $\beta$ values

Plotting the right side of the equation as seen in Figure 46 reveals that it is mostly greater than 0.1 whereas the left side of the equation (the variance of baseline noise) can be safely considered much less than that. This means that for our practical purposes:

$$
\begin{aligned}
\gamma_1^2 &< \left(\frac{1}{1+\frac{\Delta n}{m_1}}\right)^{\frac{m_1}{\Delta n}+1} \\
\gamma_1^2\Big(2+\frac{\Delta n}{m_1}\Big) &< \Big(\frac{m_1}{\Delta n}+1\Big)\gamma_1^{2\,\frac{m_1}{m_1+\Delta n}} - \frac{m_1}{\Delta n}\gamma_1^2
\end{aligned}
$$

Which means that among all the time series with monotone increasing variance those that satisfy the $\beta^2 > \gamma_1^2\Big(2+\frac{\Delta n}{m_1}\Big)$ condition will necessarily give a minimum for the reciprocal fitness function right at the beginning of the signal, while at the same time no such guarantee exists for the simplified AIC method.

*Break down of the simplified AIC method*

A break down of the simplified AIC method means:

$$
\begin{aligned}
n_1\ln\sigma_1^2 + n_2\ln\sigma_2^2 &< m_1\ln\gamma_1^2 + m_2\ln\gamma_2^2 \\
\sigma_1^{2n_1}\sigma_2^{2n_2} &< \gamma_1^{2m_1}\gamma_2^{2m_2} \\
\left(\frac{m_1\gamma_1^2+\Delta n\beta^2}{m_1+\Delta n}\right)^{n_1}\sigma_2^{2n_2} &< \gamma_1^{2m_1}\left(\frac{\Delta n\beta^2+n_2\sigma_2^2}{\Delta n+n_2}\right)^{m_2} \\
\left(\frac{m_1\gamma_1^2+\Delta n\beta^2}{m_1+\Delta n}\right)^{m_1+\Delta n}\sigma_2^{2n_2} &< \gamma_1^{2m_1}\left(\frac{\Delta n\beta^2+n_2\sigma_2^2}{\Delta n+n_2}\right)^{\Delta n+n_2} \\
\left(\frac{m_1\gamma_1^2+\Delta n\beta^2}{m_1+\Delta n}\right)^{m_1+\Delta n}\frac{\sigma_2^{2\Delta n+n_2}}{\sigma_2^{2\Delta n}} &< \frac{\gamma_1^{2\Delta n+m_1}}{\gamma_1^{2\Delta n}}\left(\frac{\Delta n\beta^2+n_2\sigma_2^2}{\Delta n+n_2}\right)^{\Delta n+n_2} \\
\left(\frac{m_1+\Delta n\frac{\beta^2}{\gamma_1^2}}{m_1+\Delta n}\right)^{m_1+\Delta n}\gamma_1^{2\Delta n} &< \sigma_2^{2\Delta n}\left(\frac{\Delta n\frac{\beta^2}{\sigma_2^2}+n_2}{\Delta n+n_2}\right)^{\Delta n+n_2} \\
\left(\frac{m_1+\Delta n\frac{\beta^2}{\gamma_1^2}}{m_1+\Delta n}\right)^{m_1+\Delta n}\left(\frac{\Delta n+n_2}{\Delta n\frac{\beta^2}{\sigma_2^2}+n_2}\right)^{\Delta n+n_2} &< \left(\frac{\sigma_2^2}{\gamma_1^2}\right)^{\Delta n} \\
\left(\frac{m_1+\Delta n\frac{\beta^2}{\gamma_1^2}}{m_1+\Delta n}\right)^{1+\frac{m_1}{\Delta n}}\left(\frac{\Delta n+n_2}{\Delta n\frac{\beta^2}{\sigma_2^2}+n_2}\right)^{1+\frac{n_2}{\Delta n}} &< \frac{\sigma_2^2}{\gamma_1^2} \\
\left(\frac{m_1+\Delta n\frac{\beta^2}{\gamma_1^2}}{m_1+\Delta n}\right)^{1+\frac{m_1}{\Delta n}}\left(\frac{\Delta n+n_2}{\Delta n\frac{\beta^2}{\sigma_2^2}+n_2}\right)^{1+\frac{n_2}{\Delta n}} &< \left(\frac{m_1+\Delta n\frac{\beta^2}{\gamma_1^2}}{m_1+\Delta n}\right)^{1+\frac{m_1}{\Delta n}}\left(\frac{\Delta n+n_2}{n_2}\right)^{1+\frac{n_2}{\Delta n}} < \frac{\sigma_2^2}{\gamma_1^2}
\end{aligned}
$$

Meaning that for any given $n_2$, $m_1$, $\Delta n$, $\beta^2$, and $\gamma_1^2$ one can come up with a sufficiently large $\sigma_2^2$ that will render the simplified AIC method useless, while as long as $\beta^2 > \gamma_1^2\left(2 + \frac{\Delta n}{m_1}\right)$ condition holds the reciprocal method still remains applicable. This described scenario can very well happen if the variance increases exponentially, which can easily occur with dispersive waves in a complex wave propagation environment.

## APPENDIX B

## CALCULATING VARIANCES

*Lemma*

$$
\begin{aligned}
\widehat{\sigma_2^2} &= \frac{1}{n_2}\sum_{i=n_1+1}^{N} x_i^2 - \frac{1}{n_2}\Big(\sum_{i=n_1+1}^{N} x_i\Big)^2 = \\
&= \frac{1}{n_2}\left[\sum_{i=n_1+1}^{n_1'} x_i^2 + \sum_{i=n_1'+1}^{N} x_i^2 - \Big(\sum_{i=n_1+1}^{n_1'} x_i + \sum_{i=n_1'+1}^{N} x_i\Big)^2\right] = \\
&= \frac{1}{n_2}\left[\sum_{i=n_1+1}^{n_1'} x_i^2 + \sum_{i=n_1'+1}^{N} x_i^2 - \Big(\sum_{i=n_1+1}^{n_1'} x_i\Big)^2 - 2\sum_{i=n_1+1}^{n_1'} x_i \sum_{i=n_1'+1}^{N} x_i - \Big(\sum_{i=n_1'+1}^{N} x_i\Big)^2\right] = \\
&= \frac{1}{n_2}\left[\sum_{i=n_1+1}^{n_1'} x_i^2 + \sum_{i=n_1'+1}^{N} x_i^2 - \Big(\sum_{i=n_1+1}^{n_1'} x_i\Big)^2 - \Big(\sum_{i=n_1'+1}^{N} x_i\Big)^2\right] - \frac{2}{n_2}\sum_{i=n_1+1}^{n_1'} x_i \sum_{i=n_1'+1}^{N} x_i = \\
&= \frac{1}{n_2}\left[\sum_{i=n_1+1}^{n_1'} x_i^2 - \Big(\sum_{i=n_1+1}^{n_1'} x_i\Big)^2 + \sum_{i=n_1'+1}^{N} x_i^2 - \Big(\sum_{i=n_1'+1}^{N} x_i\Big)^2\right] - \frac{2}{n_2}\sum_{i=n_1+1}^{n_1'} x_i \sum_{i=n_1'+1}^{N} x_i = \\
&= \frac{1}{n_2}\left[\Delta n\left(\frac{1}{\Delta n}\sum_{i=n_1+1}^{n_1'} x_i^2 - \frac{1}{\Delta n}\Big(\sum_{i=n_1+1}^{n_1'} x_i\Big)^2\right) + n_2'\left(\frac{1}{n_2'}\sum_{i=n_1'+1}^{N} x_i^2 - \frac{1}{n_2'}\Big(\sum_{i=n_1'+1}^{N} x_i\Big)^2\right)\right] - \\
&- \frac{2}{n_2}\sum_{i=n_1+1}^{n_1'} x_i \sum_{i=n_1'+1}^{N} x_i = \\
&= \frac{1}{n_2}\left[\Delta n\widehat{\sigma_1^2} + n_2'\widehat{\sigma_2'^2}\right] - \frac{2}{n_2}\sum_{i=n_1+1}^{n_1'} x_i \sum_{i=n_1'+1}^{N} x_i = \\
&= \frac{\Delta n}{n_2}\widehat{\sigma_1^2} + \frac{n_2'}{n_2}\widehat{\sigma_2'^2} - \frac{2}{n_2}\sum_{i=n_1+1}^{n_1'} x_i \sum_{i=n_1'+1}^{N} x_i
\end{aligned}
$$

But since mean values are considered to be zero:

$$\widehat{\sigma_2^2} = \frac{\Delta n}{n_2}\widehat{\sigma_1^2} + \frac{n_2'}{n_2}\widehat{\sigma_2'^2}$$

## REFERENCES


[1] B. Abbott, C. Biegl, R. Souder, T. Bapty, and Janos Sztipanovits. Graphical programming for the transputer. In *[1990] Proceedings. The Twenty-Second Southeastern Symposium on System Theory*, pages 86–90. IEEE Comput. Soc. Press, 1990.

[2] I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[3] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, December 1974.

[4] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, June 2010.

[5] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: ordering points to identify the clustering structure. *ACM SIGMOD Record*, 28(2):49–60, June 1999.

[6] P. Arato, S. Jahasz, Z.A. Mann, A. Orban, and D. Papp. Hardware-software partitioning in embedded system design. In *IEEE International Symposium on Intelligent Signal Processing, 2003*, pages 197–202. IEEE, 2003.

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[8] Krste Asanovic, John Wawrzynek, David Wessel, Katherine Yelick, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, and Koushik Sen. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56, October 2009.

[9] Atmel. AT90S/LS8535 Datasheet Mature, 2001.

[10] Atmel. ATmega128 Complete, 2004.

[11] Atmel. ATmega48/88/168 Complete, 2011.

[12] Benjamin Babjak, Sandor Szilvasi, Alex Pedchenko, Mark Hofacker, Eric J. Barth, Peter Volgyesi, and Akos Ledeczi. Experimental Research Platform for Structural Health Monitoring. In *Advancement in Sensing Technology Smart Sensors, Measurement and Instrumentation*, volume 1, pages 43–68. Springer Berlin Heidelberg, 2013.

[13] Alakananda Bhattacharya, Amit Konar, Swagatam Das, Crina Grosan, and Ajith Abraham. Hardware Software Partitioning Problem in Embedded System Design Using Particle Swarm Optimization Algorithm. In *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 171–176. IEEE, 2008.

[14] A. Bouhraoua, O. Diraneyya, and M.E. Elrabaa. A simplified router architecture for the modified Fat Tree Network-on-Chip topology. In *2009 NORCHIP*, pages 1–4. IEEE, November 2009.

[15] Hajo J. Broersma, Daniel Paulusma, Gerard J. M. Smit, Frank Vlaardingerbroek, and Gerhard J. Woeginger. The Computational Complexity of the Minimum Weight Processor Assignment Problem. In Bernhard Hromkovič, Juraj and Nagl, Manfred and Westfechtel, editor, *Graph-Theoretic Concepts in Computer Science*, pages 189–200. Springer Berlin Heidelberg, 2005.

[16] David Broman. High-Confidence Cyber-Physical Co-Design. In *Proceedings of the Work-in-Progress (WiP) session of the 33rd IEEE Real-Time Systems Symposium (RTSS 2012)*, 2012.

[17] David Broman, Michael Zimmer, Yooseong Kim, Hokeun Kim, Jian Cai, Aviral Shrivastava, Stephen A. Edwards, and Edward A. Lee. Precision Timed Infrastructure: Design Challenges. In *Proceedings of the Electronic System Level Synthesis Conference (ESLsyn 2013)*, 2013.

[18] Fangzhe Chang, Jennifer Ren, and Ramesh Viswanathan. Optimal Resource Allocation in Clouds. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 418–425. IEEE, July 2010.

[19] Hui Yan Cheah, Suhaib A. Fahmy, Douglas L. Maskell, and Chidamber Kulkarni. A lean FPGA soft processor built using a DSP block. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA '12*, page 237, New York, New York, USA, 2012. ACM Press.

[20] Barry A. Cipra. The Best of the 20th Century: Editors Name Top 10 Algorithms. *Society for Industrial and Applied Mathematics News*, 33, May 2000.

[21] R.R. Coifman and M.V. Wickerhauser. Entropy-based algorithms for best basis selection. *IEEE Transactions on Information Theory*, 38(2):713–718, March 1992.

[22] W. Daniel Hillis. The connection machine: A computer architecture based on cellular automata. *Physica D: Nonlinear Phenomena*, 10(1-2):213–228, January 1984.

[23] Rodolfo de Paz Alberola and Dirk Pesch. AvroraZ: Extending Avrora with an IEEE 802.15.4 Compliant Radio Chip Model. In *Proceedings of the 3nd ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks - PM2HW2N '08*, pages 43–50, New York, New York, USA, 2008. ACM Press.

[24] Prabal Dutta, Jay Taneja, Jaein Jeong, Xiaofan Jiang, and David Culler. A building block approach to sensornet systems. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, page 267, New York, New York, USA, 2008. ACM Press.

[25] N. Edmonds, D. Stark, and J. Davis. Mass: modular architecture for sensor systems. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 393–397. IEEE, 2005.

[26] Vincent C. Emeakaroha, Ivona Brandic, Michael Maurer, and Ivan Breskovic. SLA-Aware Application Deployment and Resource Allocation in Clouds. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pages 298–303. IEEE, July 2011.

[27] G. Estrin. Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer. *IEEE Annals of the History of Computing*, 24(4):3–9, October 2002.

[28] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. *IEEE Transactions on Electronic Computers*, EC-12(6):747–755, December 1963.

[29] G. Estrin and R. Turn. Automatic Assignment of Computations in a Variable Structure Computer System. *IEEE Transactions on Electronic Computers*, EC-12(6):755–773, December 1963.

[30] Gerald Estrin. Organization of computer systems. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference on - IRE-AIEE-ACM '60 (Western)*, page 33, New York, New York, USA, 1960. ACM Press.

[31] Silvia Figueira. An analysis of the energy efficiency of multi-threading on multi-core machines. In *International Conference on Green Computing*, pages 283–290. IEEE, August 2010.

[32] Jack Ganssle. *The Art of Designing Embedded Systems*. 1999.

[33] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation - PLDI '03*, page 1, New York, New York, USA, 2003. ACM Press.

[34] E. Harrington. Synchronization Techniques for Various Switching Network Topologies. *IEEE Transactions on Communications*, 26(6):925–932, June 1978.

[35] Armin Heindl. Inverse characterization of hyperexponential MAP(2)s. In *11th International Conference on Analytical and Stochastic Modeling Techniques and Applications*, page 183˜189, 2004.

[36] Armin Heindl, Gábor Horváth, and Karsten Gross. Explicit Inverse Characterizations of Acyclic MAPs of Second Order. In *EPEW'06 Proceedings of the Third European conference on Formal Methods and Stochastic Models for Performance Evaluation*, 2006.

[37] John L Hennessy and David A Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Number 0. Morgan Kaufmann, 2006.

[38] Jonathan Hill. The soft-core discrete-time signal processor peripheral. *IEEE Signal Processing Magazine*, 26(2):112–115, March 2009.

[39] András Horváth and Miklós Telek. Markovian Modeling of Real Data Traffic: Heuristic Phase Type and MAP Fitting of Heavy Tailed and Fractal Like Samples. In *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 405–434, 2002.

[40] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker. A Heterogeneous Multicore System on Chip with Run-Time Reconfigurable Virtual FPGA Architecture. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 143–149. IEEE, May 2011.

[41] Carlee Joe-Wong, Soumya Sen, Tian Lan, and Mung Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *2012 Proceedings IEEE INFOCOM*, pages 1206–1214. IEEE, March 2012.

[42] Genshiro Kitagawa and Hirotugu Akaike. A procedure for the modeling of non-stationary time series. *Annals of the Institute of Statistical Mathematics*, 30(1):351–363, December 1978.

[43] A. Kumar and J. Kleinberg. Fairness measures for resource allocation. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 75–85. IEEE Comput. Soc, 2000.

[44] Tian Lan, David Kao, Mung Chiang, and Ashutosh Sabharwal. An Axiomatic Theory of Fairness in Network Resource Allocation. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, March 2010.

[45] Olaf Landsiedel, Klaus Wehrle, Ben L. Titzer, and Jens Palsberg. Enabling detailed modeling and analysis of sensor networks. *Praxis der Informationsverarbeitung und Kommunikation*, 2005.

[46] Akos Ledeczi. *Parallel systems with flexible topology*. PhD thesis, 1995.

[47] Ben Lee and A. R. Hurson. Issues in Dataflow Computing. *Advances in Computers*, 37:285–333, 1993.

[48] E.K.F. Lee and P.G. Gulak. A CMOS Field-programmable Analog Array. In *1991 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 186–314. IEEE, 1991.

[49] Isaac Liu. *Precision Timed Machines*. PhD thesis, University of California, Berkeley, May 2012.

[50] Tao Liu, Zhenzhou Ji, Qing Wang, and Suxia Zhu. Research on Efficiency of Signal Processing on Embedded Multicore System. In *2010 First International Conference on Pervasive Computing, Signal Processing and Applications*, pages 907–911. IEEE, September 2010.

[51] Yu Liu. The Wireless Sensor Network Simulator, 2012.

[52] Dimitrios Lymberopoulos, Nissanka B. Priyantha, and Feng Zhao. mPlatform: A Reconfigurable Architecture and Efficient Data Sharing Mechanism for Modular Sensor Nodes. In *2007 6th International Symposium on Information Processing in Sensor Networks*, pages 128–137. IEEE, April 2007.

[53] N. Maeda. A method for reading and checking phase times in auto-processing system of seismic wave data. *Zisin, Journal of the Seismological Society of Japan*, 38:365–380, 1985.

[54] G. Martin. Overview of the MPSoC design challenge. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 274–279. IEEE, 2006.

[55] William Mary, Giuliano Casale, and Eddy Z. Zhang. Interarrival Times Characterization and Fitting for Markovian Traffic Analysis. In *Dagstuhl Seminar*, 2008.

[56] Oleg Maslennikow, Juri Shevtshenko, and Anatoli Sergyienko. Configurable Microprocessor Array for DSP Applications. In *Lecture Notes in Computer Science*, pages 36–41. 2004.

[57] Thomas Moscibroda and Onur Mutlu. A case for bufferless routing in on-chip networks. *ACM SIGARCH Computer Architecture News*, 37(3):196, June 2009.

[58] Laurent Moss, Hubert Guérard, Gary Dare, and Guy Bois. Recent Experience on an ESL Framework for Rapid Design Exploration Using Hardware-Software Codesign for ARM-based FPGAs. In *SAME 2012*, 2012.

[59] Boris Muravin, Gregory Muravin, and Ludmila Lezvinsky. The Fundamentals of Structural Health Monitoring By The Acoustic Emission Method. In *20th International Acoustic Emission Symposium*, pages 253–258, Kumamoto.

[60] P.K. Murthy and E.A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, August 2002.

[61] B. O'Flynn, S. Bellis, K. Delaney, J. Barton, S.C. O'Mathuna, A.M. Barroso, J. Benson, U. Roedig, and C. Sreenan. The development of a novel minaturized modular platform for wireless sensor networks. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 370–375. IEEE, 2005.

[62] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *ACM SIGPLAN Notices*, 31(9):2–11, September 1996.

[63] Suraj Pandey, Linlin Wu, Siddeswara Mayura Guru, and Rajkumar Buyya. A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 400–407. IEEE, 2010.

[64] Joseph Porter, Zsolt Lattman, Graham Hemingway, Nagabhushan Mahadevan, Sandeep Neema, Harmon Nine, Nicholas Kottenstette, Peter Volgyesi, Gabor Karsai, and Janos Sztipanovits. The ESMoL Modeling Language and Tools for Synthesizing and Simulating Real-Time Embedded Systems. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[65] Hideki John Reekie. *Realtime Signal Processing: Dataflow, Visual, and Functional Programming*. PhD thesis, 1995.

[66] Till Riedel, Philipp Scholl, Christian Decker, Martin Berchtold, and Michael Beigl. Pluggable real world interfaces Physically enabled code deployment for networked sensors. In *2008 5th International Conference on Networked Sensing Systems*, pages 111–114. IEEE, June 2008.

[67] Utz Roedig, Sarah Rutlidge, James Brown, and Andrew Scott. Towards multiprocessor sensor nodes. In *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors - HotEmNets '10*, page 1, New York, New York, USA, 2010. ACM Press.

[68] Thomas W Rondeau, Allen B Mackenzie, Jeffrey H Reed, Scott F Midkiff, and Sheryl B Ball. *Application of Artificial Intelligence to Wireless Communications*. PhD thesis, Virginia Polytechnic Institute, 2007.

[69] Emre Salman and Eby Friedman. *High Performance Integrated Circuit Design*. 2012.

[70] Patrick R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. 2010.

[71] Alexander Schrijver. *Combinatorial Optimization Polyhedra and Efficiency*, volume 24. 2003.

[72] David Sheldon, Rakesh Kumar, Roman Lysecky, Frank Vahid, and Dean Tullsen. Application-Specific Customization of Parameterized FPGA Soft-Core Processors. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 261–268. IEEE, November 2006.

[73] David Sheldon, Rakesh Kumar, Frank Vahid, Dean Tullsen, and Roman Lysecky. Conjoining Soft-Core FPGA Processors. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 694–701. IEEE, November 2006.

[74] L.T. Smit, G.J.M. Smit, J.L. Hurink, H. Broersma, D. Paulusma, and P.T. Wolkotte. Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture. In *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, pages 421–424. IEEE, 2004.

[75] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems*, 7(2):1–28, February 2008.

[76] Martin Straka, Jan Kastil, Jaroslav Novotny, and Zdenek Kotasek. Advanced fault tolerant bus for multicore system implemented in FPGA. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 397–398. IEEE, April 2011.

[77] Tobias Strauch. Hyper pipelining of multicores and SoC interconnects. *EETimes*, 2010.

[78] Dominique Thiebaut. *Parallel Programming in C for the Transputer*. 1994.

[79] Ben L. Titzer and Jens Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *ACM SIGPLAN Notices*, volume 40, page 59, New York, New York, USA, July 2005. ACM Press.

[80] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 477–482. IEEE, 2005.

[81] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer, and Gerben Essink. Design and programming of embedded multiprocessors: An Interface-Centric Approach. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '04*, page 206, New York, New York, USA, 2004. ACM Press.

[82] Matthew A. Watkins and David H. Albonesi. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 497–508. IEEE, December 2010.

[83] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A practical FPGA-based framework for novel CMP research. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays - FPGA '07*, page 116, New York, New York, USA, 2007. ACM Press.

[84] W.H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.

[85] Baifeng Wu and Chenglian Peng. System-on-chip design with dataflow architecture. In *8th International Conference on Computer Supported Cooperative Work in Design*, volume 2, pages 748–752. IEEE, 2004.

[86] Fei Yu. A survey of Wireless Sensor Network simulation tools, 2011.