ABSTRACT

RAMNATH VENUGOPALAN, Improving Energy-efficiency in Sensor Networks by Raising Communication Throughput using STI (Under the direction of Dr. Alexander G Dean).

Energy consumption plays a crucial role in the design of a battery operated embedded system like a sensor node. Current technologies attempt to fulfill this need by optimizing various parts of the sensor node like the software for communication by using energy-aware algorithms that allow the sensor nodes to sleep as often as and as long as possible and by optimizing the hardware to reduce its energy consumption. In addition to these, there are energy-aware compiler techniques, for example, those on optimization of code-generation techniques of compilers that advocate usage of instructions that consume the least energy, usage of compiler optimizations like loop unrolling and cloning, elimination of dead code, those on instruction scheduling for minimum number of switches, optimizations in register allocation, register pipelining, etc. However, optimization of interrupt overhead, which is one of the biggest contributors to energy consumption, is not addressed by any of these technologies.

Our new methods involve the elimination of interrupt overhead using a compiler technique known as Software Thread Integration (STI). STI is a compiler technology which interleaves multiple assembly language threads at a fine-grain level. This method may be used in conjunction with the above methods for improved energy-efficiency. The energy consumption of a sensor node is primarily due to two components its communication (RF) module and its computation (CPU) module. Currently, communication is performed using an interrupt-based scheme that raises an interrupt at frequent intervals. The interrupt overheads mount up to large values for long periods of communication. Using STI to interleave the threads of computation with the threads of communication allows these statically scheduled threads to function without the use of interrupts. This saves a large number of cycles and time in interrupt overhead. These savings can be used in multiple ways to save energy. 1) The cycles that are saved can be used by the CPU to go to sleep when there is no work to be done. 2) The CPU can be run at lower clock frequencies so as to save energy. 3) The code for the communications can be run faster so that the bit-rate for communications can be increased to the highest limit. This allows the RFM to transmit as fast as possible and go to sleep sooner. The energy consumption of the RFM depends on the transmit power if it is transmitting, and the duration for which it has to stay powered up. The reduction in time that it has to stay on decreases the energy consumption of the node.

We demonstrate these methods in this thesis by applying them to a sensor node that runs communication and a DSP application and show significant reduction in energy consumption of the node.

IMPROVING ENERGY-EFFICIENCY IN SENSOR NETWORKS BY RAISING COMMUNICATION THROUGHPUT USING STI

By

Ramnath Venugopalan

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science in **Computer Networking**

Department of Electrical and Computer Engineering

North Carolina State University

2003

Approved by _____
Dr. Alexander G. Dean, Chairperson of Advisory Committee

_____
Dr. Frank Mueller

_____
Dr. Mihail Sichitiu

Date _____

To my Mother,

C.B. Vijayalakshmi

For giving me more caring, love and affection than I could ever hope for, supporting me at every step of the way and teaching me the importance of education early in life.

And my sister,

Reshma Venugopalan

For her love and support always

BIOGRAPHY

Ramnath Venugopalan was born in the once idyllic but now bustling city of Bangalore in India. He pursued most of his education in Bangalore, where he went to High School at Kendriya Vidyalaya ASC Centre (S) (1986-1995), College of Engineering at the University Visvesvaraya College of Engineering for his B.S in Electrical and Electronics Engineering (1995-1999). Following this, he worked at IBM for 2 years, both in Bangalore and in Boulder, CO (1999-2001). At the time of this writing, he is working towards his M.S in Computer Networking at North Carolina State University at Raleigh, North Carolina.

ACKNOWLEDGMENTS

I thank Dr. Alexander Dean, without whose help and insight this work would not have been possible. He has always been a source of inspiration for me. I also thank my committee members, Dr. Frank Mueller and Dr. Mihail Sichitiu, for all their advice and suggestions during the course of my thesis.

I thank my parents for showing me the path to where I am today and providing constant encouragement throughout the way.

Thanks to all the wonderful people here at NC state who helped me with my work and helped make my stay here an enjoyable one.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EQUATIONS

Page

Chapter 1

INTRODUCTION

In recent years, sensor networks have been undergoing a quiet revolution, promising to have a significant impact on a broad range of applications relating to a variety of areas like national security, health care, the environment, energy, food safety, and manufacturing. A sensor network comprises of a large number of sensors that communicate with each other to achieve a common goal. It is desirable to make the sensor nodes as inexpensive and energy-efficient as possible since the sensor networks rely on their large numbers for accurate and high quality results.

Current research in sensor networks is attempting to decrease the size, weight and cost of these sensors by orders of magnitude so that this cost-effectiveness will lead to their deployment in far larger numbers than it is today. Power consumption is a very important aspect of current research in sensor networks due to the fact that the nodes in these sensor networks are typically un-tethered and unattended and hence possess small energy reserves. The reduction in size of the sensor nodes limits the size of the battery, which in turn limits the lifetime of the battery. The lifetime of a sensor network is only as long as that of the batteries that power the nodes and hence every portion of the node including the software must assume some of the responsibility of reducing the energy consumption of the node.

The technique of Software Thread Integration (STI) [19], when applied to the software running on the sensor nodes, will help in reducing their energy consumption. STI is a compiler technology which interleaves multiple assembly language threads at a fine-grain level. The resulting thread offers low-cost concurrency yet executes on a generic processor without fast context switches. This avoids the overhead that interrupts bring along with them. STI is extremely valuable for recovering idle time from real-time threads performing low-level (e.g. MAC and data link layer) network communication as are present in the wireless sensor nodes.

STI is used here to integrate these low-level communication threads on the nodes with other applications that are required to run on the sensor nodes such as those that involve digital signal processing tasks like filtering. This integration provides us with the benefit of the lack of interrupt overhead coupled with excellent utilization of the idle time in the real-time thread. This idle time is used for the DSP tasks and to sleep when the DSP work has been completed. The amount of time that the node will be able to sleep when running STI-based threads is greater than that when it is running ISR-based threads. This enables the node to consume much lesser energy while running STI-based threads when compared to that consumed when it runs ISR-based threads. This saving energy comes from the CPU, which is turned off during the idle mode leaving just the communication running.

Another component of the sensor node that consumes a large percentage of the total amount of energy consumed is the RF module (RFM). The ISR-based thread takes a much longer time to run on the CPU due to the larger number of extra cycles of interrupt overhead that need to be executed in addition to the communication instructions. This limits the highest bit-rate that can be supported by the node to:

Highest bit-rate possible = 1/Time taken by the CPU to execute the instructions in the communication thread to send out 1 bit.

Equation 1-1: Highest bit-rate possible

Since the time taken by the CPU to execute the instructions in the communication thread to send out 1 bit is much greater in the ISR-based thread than in the STI-based one, the highest possible bit-rate for the STI-based thread is much greater than the ISR-based one. The energy consumed by the RFM is dependent only on the duration for which it is actively participating in communications and the power with which it is transmitting. It is not dependent on the bit-rate. By completing the communications soon at a high bit-rate, the RFM can reduce the duration for which it is required to be active mode and can go to sleep after that. The shorter the duration for which it has to actively participate in communication, the lesser amount of energy it is that is consumed by the RFM. Hence, higher bit-rates lead to less energy consumption for the node. As the STI-based thread is capable of transmitting at much higher bit-rates than the ISR-based thread, it can turn the RFM off much earlier than the ISR-based

thread can. This leads to a much lower consumption of energy by the node running the STI-based thread when compared to a node that is running the ISR-based thread.

It is possible with STI to save energy even by transmitting lesser number of bits as compared to the ISR version. This reduction in the number of bits is achieved by reducing the number of start/stop bit pairs to one per packet for the STI version from one per byte for the ISR version. This is possible due to the lack of interrupt introduced timing variance in the STI-based thread due to static scheduling. The STI-based thread meets all the deadlines for the real-time communication thread while also executing the non-real-time application thread during the idle time of the real-time thread with static scheduling of the two threads.

We go on to demonstrate in this thesis that STI can save a considerable amount of energy for the wireless node thus elongating its lifetime. The lifetime is extended to the following:

*Lifetime with STI = (energy consumed with the ISR-based thread/ energy consumed with the STI-based thread) * Lifetime with ISR implementation*

Equation 1-2: Lifetime with STI

The thesis has been organized as follows. Chapter 2 presents the background details and the concepts of STI. Chapter 3 details the new methods used in this thesis. Chapter 4 goes on to explain the hardware and software architecture used in obtaining the results. Chapter 5 discusses the results and analyses them in detail. Chapter 6 presents the conclusion and future work.

Chapter 2

BACKGROUND

2.1 Wireless Sensor Networks

Recent advances in miniaturization using MEMS-based sensor technology and low-cost, low-power design have led to active research in large-scale, highly distributed systems of small, wireless, low-power, unattended sensors and actuators [1][2][3] also commonly known as Wireless Sensor Networks. The idea is to create sensor-rich ``smart environments" through planned or ad-hoc deployment of thousands of sensors, each with a short-range wireless communications channel, and capable of detecting ambient conditions such as temperature, movement, sound, light, or the presence of certain objects.

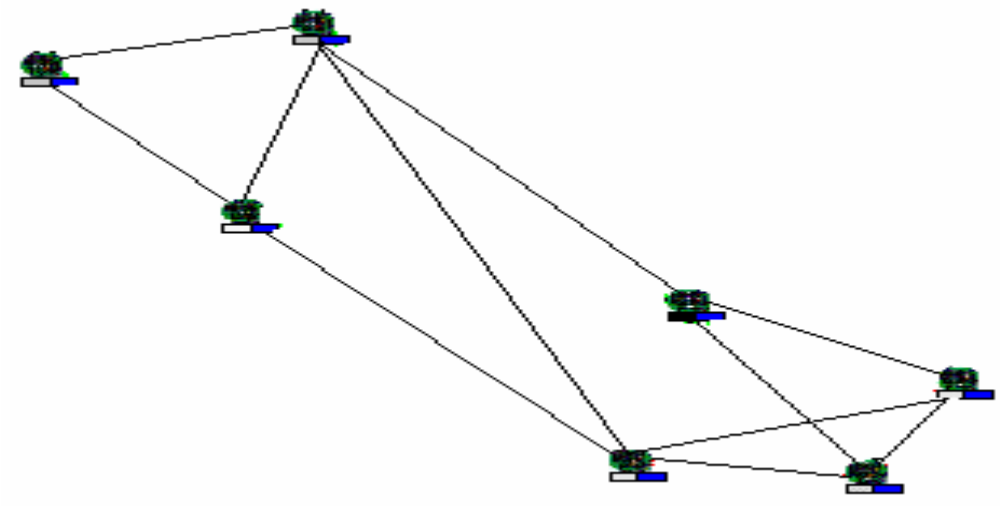

Figure 2-1: A Sensor Network

These sensors are not as reliable or as accurate as their expensive macro-sensor counterparts, but their small size and low cost enable applications to network hundreds or thousands of these micro-sensors in order to achieve high quality, fault-tolerant sensing networks. These

sensor networks are becoming very popular for reasons such as lower cost, ease of deployment, and fault tolerance. Each sensor has wireless communication capability and sufficient intelligence for signal processing and networking of the data. Some examples of smart sensor networks are the following:

- Military sensor networks to detect enemy movements, the presence of hazardous material (such as poison gases or radiation, explosions, etc.)
- Environmental sensor networks (such as in plains or deserts or on mountains or ocean surfaces) to detect and monitor environmental changes.
- Wireless traffic sensor networks to monitor vehicle traffic on a highway or in a congested part of a city.
- Wireless surveillance sensor networks for providing security in a shopping mall, parking garage, or other facility.
- Wireless parking lot sensor networks to determine which spots are occupied and which spots are free.
- Besides offering certain capabilities and enhancements in operational efficiency in these conventional applications, smart sensor networks can assist in the national effort to increase alertness to potential terrorist threats.

2.1.1 Sensor Nodes

Each of the nodes that form the sensor network is made up of the following sub systems:

i. The data and control processing a.k.a computing subsystem consisting of a microprocessor or microcontroller.

ii. The communication subsystem consisting of a short range radio transceiver for wireless communication.

iii. The sensor subsystem consisting of the A/D converter and various sensors including acoustic and seismic sensors.

iv. The power module that consists of the power supply to the node, including the batteries.

The data is continuously sampled and stored in on-board RAM to be processed by the data and control processing module. The other three modules are adaptable depending on the role of the sensor within the network, and all three are controlled by the data and control processing module.

2.1.2 Power Consumption and its Importance to Sensor Networks

Micro-sensor networks can contain hundreds or thousands of sensing nodes. It is desirable to make these nodes as cheap and energy-efficient as possible and rely on their large numbers to obtain high quality results. These nodes in the sensor systems are un-tethered and unattended and therefore have small energy reserves. Small sensor nodes imply limited physical space for batteries, and high density implies that periodic battery replacement will be a great inconvenience and more likely, impossible. A state-of-the-art lithium primary battery offers an energy density of about 2 kJ per cm^3 [16]. Assuming that 1 cm^3 is available within the node for the battery and that the desired device lifetime is one year, the average power dissipation must be less than

(2000 J) * (1 year / 365 Days) * (1 Day / 24 Hours) * (1 Hour / 3600 Seconds) = 63.4 μW

As this value exceeds the standby power of most digital systems, energy dissipation is of paramount concern. Moreover, Moore's law simply does not apply to batteries: the energy density of batteries has only doubled every five to 20 years, depending on the particular chemistry, and prolonged refinement of any chemistry yields diminishing returns [17]. Energy conservation strategies are therefore essential for achieving the lifetimes necessary for viable applications [18].

All communication--even passive listening--will have a significant effect on the energy reserves of the node. The life of the sensor node depends on the life of its battery. To prolong the lifetimes of the wireless sensors, all aspects of the sensor system should be energy efficient, and design should focus on minimizing both computational and communication energy.

2.1.3 Current Research in power optimization in sensor networks

Energy/power awareness [12] being such an important topic to sensor networks, there is a lot of research that has gone into this area in the past and will continue to do so in the future. This research has tried to bring energy-awareness to various facets of the sensor and the sensor-network. There have been studies into the upper-bounds of the lifetime of a sensor network and the factors that prevent them from achieving this theoretical maximum lifetime [4]. The sensors themselves are being designed in as energy efficient a manner as possible [5][6][9][10]. The network layers, like the MAC layer [8], the network layer [7] are being optimized for energy savings since communication is one of the major sources of consumption of energy in a sensor node. Partitioning the task at hand among several nodes is another way of saving power for complex calculations like a 1024 point FFT [15]. A large amount of contemporary research in sensor network is devoted to this area and the above are representative of only a small portion of the same.

The main sources of power consumption in sensor networks are the computation and communication subsystems and thus optimizations in those subsystems derive maximum energy gains for the sensor nodes and most of the current research is concentrated therein. Detailed analyses have performed in the areas of power consumption in a sensor node and various optimizations have been suggested to improve them [11].

2.1.4 Digital Signal Processing in Sensor Networks

Networked micro-sensors enable a variety of new applications such as warehouse inventory tracking, location sensing, machine-mounted sensing, patient monitoring, and building climate control [12][13]. One prime example of a micro-sensor application is the use of acoustic sensors for environmental monitoring. Reliable environment monitoring is important in a variety of commercial and military applications. Acoustic sensors are highly versatile and can be used in a variety of applications, such as speech recognition, traffic monitoring, and medical diagnosis. Multiple sensors can be used to pinpoint the location of an acoustic source (e.g., moving vehicle, speaker) by using a line of bearing estimation technique.

The wireless communication network between sensors nodes facilitates sensor collaboration, and the digital signal processing (DSP) for the analysis of sensor data can be done locally. Common digital signal processing tasks in the analysis of sensor data are those that involve

digital filtering to remove unwanted portions of the signal leaving only those portions that are required for further analysis. Two major forms of such filters are the Finite Impulse Response (FIR) and the Infinite Impulse Response (IIR) filters.

A *Finite Impulse Response (FIR)* filter produces an output, y(n), that is the weighted sum of the current and past inputs, x(n).

$$\begin{aligned} y_n &= b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + \ldots + b_q x_{n-q} \\ &= \sum_{j=0}^{q} b_j x_{n-j} \end{aligned}$$

The weights (b_0, b_1, b_2…) are calculated to change the characteristic of the FIR filter to a high pass, band pass or low pass filter for particular frequency ranges.

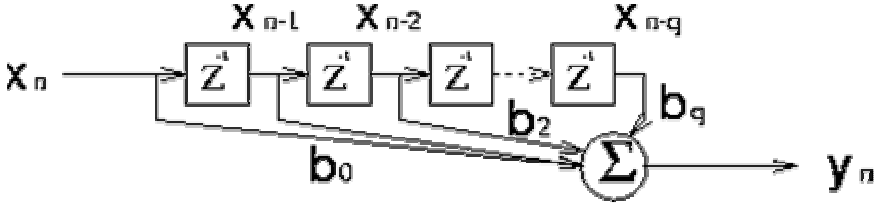

Figure 2-2: An FIR filter

The impulse response of the FIR filter is "finite" because there is no feedback in the filter; if you put in an impulse (that is, a single "1" sample followed by many "0" samples), zeroes will eventually come out after the "1" sample has made its way in the delay line past all the coefficients.

An *Infinite Impulse Response (IIR)* filter produces an output, y(n), that is the weighted sum of the current and past inputs, x(n), and past outputs.

$$y_n = \sum_{i=1}^{p} a_i y_{n-i} + \sum_{j=0}^{q} b_j x_{n-j}$$

The weights (b_0, b_1, b_2… and a_0, a_1, a_2…) are calculated to change the characteristic of the IIR filter to a high pass, band pass or low pass filter for particular frequency ranges.

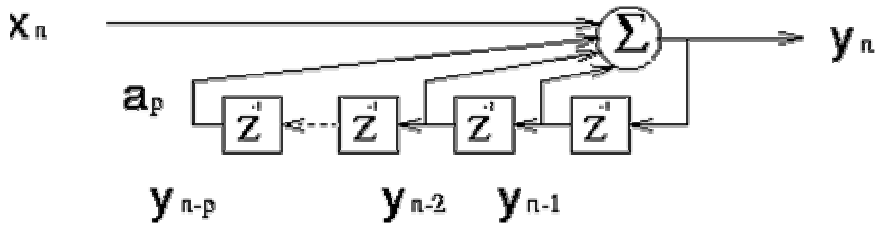

Figure 2-3: An IIR Filter

The impulse response of an IIR filter is "infinite" because there is feedback in the filter; if you put in an impulse (a single "1" sample followed by many "0" samples), an infinite number of non-zero values will come out (theoretically).

Compared to IIR filters, FIR filters offer the following advantages:

i. They can easily be designed to be "linear phase" i.e. linear-phase filters delay the input signal, but don't distort its phase.

ii. They are simple to implement. On most DSP microprocessors, the FIR calculation can be done by looping a single instruction.

iii. They have desirable numeric properties. In practice, all DSP filters must be implemented using "finite-precision" arithmetic, that is, a limited number of bits. The use of finite-precision arithmetic in IIR filters can cause significant problems due to the use of feedback, but FIR filters have no feedback, so they can usually be implemented using fewer bits, and the designer has fewer practical problems to solve related to non-ideal arithmetic.

iv. They can be implemented using fractional arithmetic. Unlike IIR filters, it is always possible to implement a FIR filter using coefficients with magnitude of less than 1.0. (The overall gain of the FIR filter can be adjusted at its output, if desired.) This is an important consideration when using fixed-point Digital Signal Processors, because it makes the implementation much simpler.

Compared to IIR filters, FIR filters sometimes have the disadvantage that they require more memory and/or calculation to achieve a given filter response characteristic. Also, certain

responses are not practical to implement with FIR filters. An FIR filter can be used for the implementation of matched filters that are very useful in signal processing.

IIR filters can achieve a given filtering characteristic using less memory and calculations than a similar FIR filter. However, they are more susceptible to problems of finite-length arithmetic, such as noise generated by calculations, and limit cycles. (This is a direct consequence of feedback: when the output isn't computed perfectly and is fed back, the imperfection can compound.) Also, they are harder (slower) to implement using fixed-point arithmetic.

2.1.5 Fixed Point Arithmetic

Certain types of embedded systems require the handling of real numbers also known as floating point numbers (or at least what appear to be real numbers). Few microprocessors offer real-number support, such as for floating-point data types and operations at the instruction level. Those that do are generally large, complex, expensive, and not intended for embedded applications. Certainly none of the small 4- and 8-bit microcontrollers support floating point, even though these are precisely the processors that are going to be at the heart of many apparently "real-number" applications.

Floating-point numbers allow one to deal with an extremely wide range of numbers: from the very small to the very large. They do this by storing the number as some digits and the position of the decimal point. Without a floating-point co-processor, this sort of arithmetic can be very slow. Things can be speeded up by fixing the position of the decimal point and using integer arithmetic operations. This is called fixed-point arithmetic.

In fixed-point arithmetic, a fixed number of bits are allocated to the decimal portion and another fixed number of bits to the fractional portion. The programmer can treat the numbers as integers and perform integer arithmetic with them so long as he takes care to not overflow the maximum value that the fixed-point number can hold. The number of bits for integer and fractional parts should be chosen such that overflow will not occur during the course of the program. Addition and subtraction can be performed as with integers, but with multiplication and division care needs to be taken to keep the decimal point fixed at the same position. For example, multiplying two numbers that have 8 bits devoted to the fractional part causes the decimal point to shift left by 8 bits, thus the result should now be shifted right by 8 bits before using its value in another arithmetic operation. Thus, from the processors point of view all the

arithmetic is integer-based but the programmer keeps track of the decimal point and views the results as floating point numbers. This speeds up the arithmetic by 6-7 times.

2.2 Software Thread Integration

STI [19] is a compiler technology which interleaves multiple assembly language threads at a fine-grain level. The resulting thread offers low-cost concurrency, but still executes on a generic processor without fast context switches. STI can be used for hardware to software migration (HSM). HSM is the process of moving functions from dedicated hardware components to real-time software. HSM helps to improve system cost, size, weight, power, function availability, time to market, and field upgrades. The main targets of HSM are embedded systems, which cannot afford the luxury of a high performance microprocessor yet require fine-grain thread concurrency.

In HSM with STI, two assembly threads are integrated. One of the threads is a real time thread with real time deadlines to meet and the other thread is a non real time thread associated with the application. The real time thread is the software implementation of the dedicated hardware function, which we want to move to software. The real time thread becomes the guest thread and the non real time thread becomes the host thread. The real time thread does not use up the processor the entire time its running. It has a fine grain idle time between instructions that have real time deadlines to meet. This idle time is either used up in executing NOP instruction (busy-waiting) until the next deadline to be met is reached or is used to process interrupts when the real time deadlines are met using interrupts generated through a timer. STI helps us in extracting this fine grain idle time that was hitherto wasted during the execution of a real time thread using it for executing other threads. The existing thread, which does not have any real time requirements becomes the host thread and the guest is inserted into the host thread at locations which satisfy its real time constraints.

This kind of integration involves a variety of factors and must be done carefully. Static timing analysis is performed on the control dependence graph generated for both the threads. This timing analysis gives us information about how early or how late an instruction would be executed when the threads are run and also the duration of execution of instructions. This type of information helps in identifying insertion places in the host thread, where instructions from the guest thread can be placed so that the real time requirements of the guest thread are

properly met. Control Dependence Graphs (CDG) facilitate easier implementation of static timing analysis and provide excellent support for determining the insertion points that meet the real time constraints in the host thread and the subsequent placement of the guest code in those locations. Register reallocation techniques are also employed to enable sharing of the processor's register set by the two threads, without any conflicts.

STI is extremely valuable for recovering idle time from real-time guest threads performing low-level (e.g. MAC and data link layer) network communication and video refresh functions. Other applications with significant fine-grain idle time can benefit from STI as well. An example of an application in which STI has been implemented is given in [20] in which high temperature (185-225 C) CAN network interface is implemented in software and integrated with buffer management code for the CAN protocol. This system has been built and tested at up to 225 °C.

A special compiler called *thrint* has been developed for this purpose. *Thrint* is a post-pass (back-end) compiler, which reads assembly code, performs control-flow, data-flow and static timing analysis, and integrates threads.

C h a p t e r 3

NEW METHODS – USING STI TO SAVE ENERGY

3.1 Byte-level Protocol controllers and Software Thread Integration

Byte-level protocol controllers are those protocol controllers that use a byte as the unit of communication. A good example of a byte-level controller would be the Serial Peripheral Interface (SPI), which sends and receives no more than 1 byte at a time.

Current implementations of byte-level protocol controllers are implemented in two possible ways: Interrupt based and Polling Based.

The interrupt based protocols recognize the completion of transmission or reception of a byte by the generation of an interrupt. This process has several disadvantages:

- High interrupt overhead. The interrupt process involves the following steps before the ISR begins execution:
    - Finish current instruction: Up to 4 cycles
    - Push program counter onto the stack in Data SRAM (as is pointed to by the SPH:SPL): 4 cycles.
    - Push status register onto the stack: 2 cycles
    - Follow interrupt vector (Jump instruction): 3 cycles.

  This process results in a delay of 9-14 cycles before the ISR starts. The overhead introduced by interrupts cause delays and a sizeable fall in throughput. The context switch time becomes increasingly expensive as communication protocol rates rise relative to processor speeds. This overhead makes higher data rates impossible.

- Variation in execution time. Interrupt processing can begin only after the completion of the current instruction, which can be anywhere between 1 and 4 cycles in length for the ATMEL AVR series of microprocessors that have been used in this dissertation.

- Loss of synchronization between transmitter and receiver unless start and stop bits are introduced for each byte.

- Amount of work that can be done apart from communication during the idle period between transmitting two consecutive bytes of data is reduced.

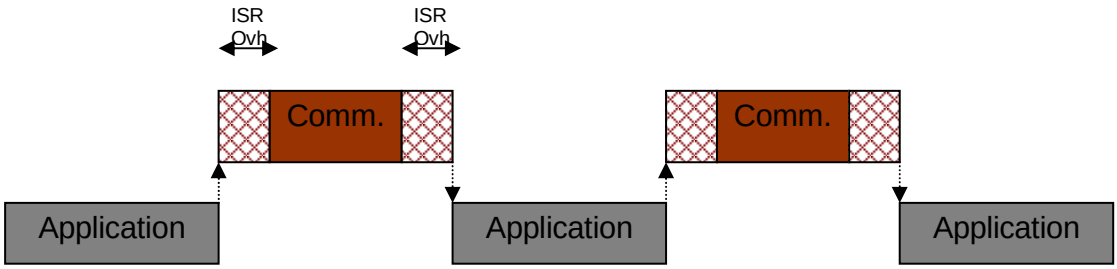

Figure 3-1: Interrupt overhead for ISR implementation

The polling based byte-level protocol controllers busy wait on a certain event that indicates completion of communication, for example, a bit in the SPI status register that is set to indicate that the transmission or reception of a full byte is complete. The software runs in a loop until there is a change in the status of the respective bit in the status register. The processor is now forced to stay awake for the entire duration of communication due to the busy-waiting. This is inevitable as this is the only way the processor can accurately ascertain the exact instant of completion of communication so as to not lose synchronization between the transmitter and receiver. Synchronization can easily be lost due to the accumulation of timing errors over the communication period. For example, if the processor loses 1/4 bit-time at the reception of each byte, it will begin to misread the data after it has received 4 bytes. Busy-waiting has the advantage of maintaining synchronization between transmitter and receiver for the entire duration of communication of a packet if their clocks are reasonably accurate. An obvious disadvantage in busy-waiting is that the microprocessor is wasting its time busy waiting while it could have gone to sleep to conserve energy or performed other useful work so as to maximize its throughput.

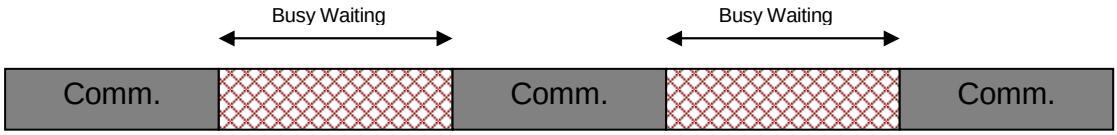

Figure 3-2: Busy wait implementation overheads

3.2 Saving Energy with Software Thread Integration

The technique of Software Thread Integration integrates two assembly threads, the real-time guest code and the host code into a single assembly thread. The real-time guest code is statically scheduled and is inserted into the host code so that it executes at the exact instants of time as necessary to communicate at the required bit rate.

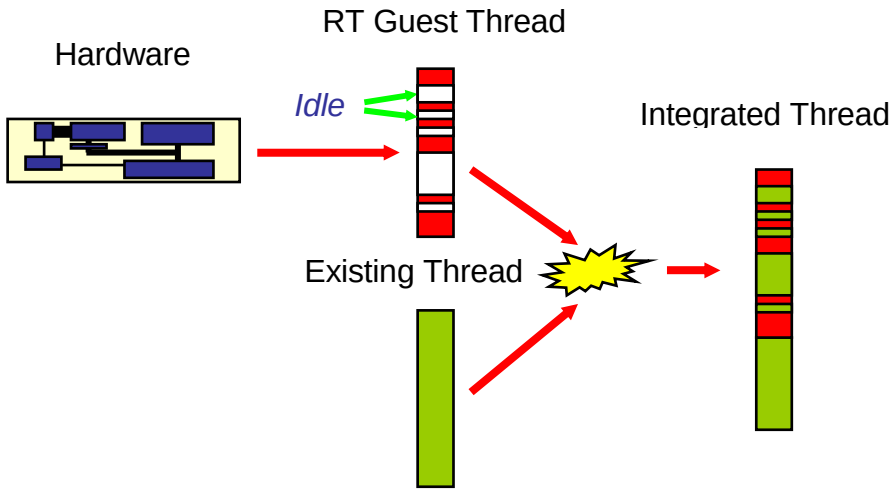

Figure 3-3: STI and elimination of interrupt overhead

In this case the real-time guest code is the communication code that needs to send data and receive data at a certain bit-rate and the host code is an application that performs common digital signal processing tasks like filtering. The filters that have been used here are an 8^{th} order Finite Impulse Response (FIR) filter and a 6^{th} order Infinite Impulse Response (IIR) filter.

This technique completely eliminates the interrupt overhead due to static scheduling. This allows for the availability of a larger number of free cycles for the processor to perform other useful work. These free cycles can be used to sleep, as well as perform other useful work in addition to communication, which allows the processor to conserve energy.

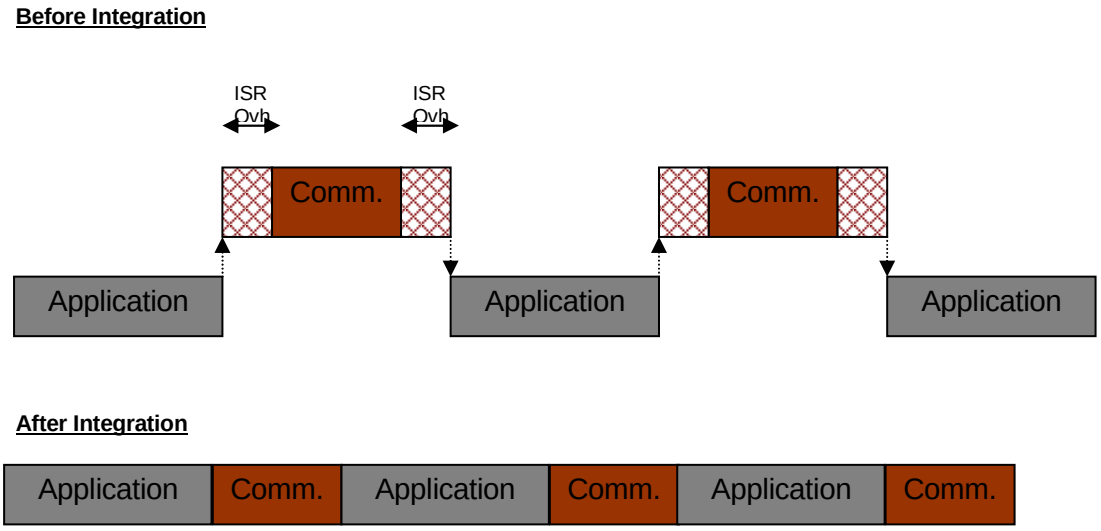

Figure 3-4: Increase in communication bit-rate due to STI

The overhead introduced by interrupts can be so high as to limit the rate of communication. The removal of this overhead enables the node to communicate at much higher rates. As mentioned earlier, most of the energy in a sensor node is consumed by two major components – the radio module and the micro-processor. The radio module's energy consumption is dependent on the power with which it transmits and the duration for which it stays powered on. It does not depend on the bit-rate at which it transmits. Thus, the possible increase in bit-rate introduced by STI can be used by the node to transmit as fast as it can and then put the radio module to sleep after the transmission is complete. Here, the lengthening in the time that the radio module can sleep allows for increase in energy savings. However, this is possible only if the receiver is capable of receiving at the rate at which the transmitter is transmitting. This may mean that both the transmitter and receiver nodes may have to run code that is integrated using STI. In this way, both transmitter and receiver nodes stand to benefit by saving energy due to STI.

Figure 3-4 above shows the increase in communication bit-rate due to STI. We can see that before integration, the interrupt overhead takes up a sizeable portion of the CPU time away from actual work being performed. After integration, the interrupt overhead has been removed and actual work in the form of the communication and application has taken its place, thus increasing throughput and the communication bit-rate of the node.

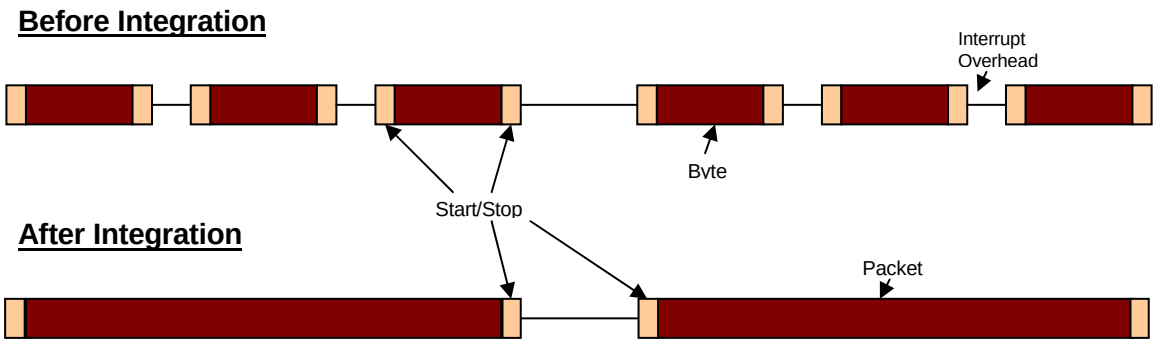

Figure 3-5: Savings from using start and stop bits per packet

Byte-level protocol controllers, which are commonly in use like those for the UART, SPI, etc. typically send one byte at a time and have to attain and maintain synchronization for every byte that is transmitted and received. In case of SPI, a start bit will have to precede and a stop bit will have to be appended to every byte that is communicated so as to attain and maintain synchronization between transmitter and receiver for the period of time that the communication of the byte is in progress. This needs to be done, since, in the wireless world, there is no way for a receiver to know if it has drifted out of synchronization with the transmitter or not if safeguards like this are not implemented. The receiver can continue reading garbage data without recognizing it as such until it tries to run a parity/CRC check on the received packet. If the protocol controller is such that it must read the packet length field in the received packet (J1850, CAN) to determine the length of the packet, then it may wait for a very long period of time while it thinks that the data has been received, if this particular value of packet length has been misread due to loss of synchronization. This may lead to energy losses due to the processor cycles wasted and increase in duration that the radio module needs to stay on. After the recognition of a bad packet, the receiver may request retransmission of the whole packet, which leads to energy loss. Byte-level protocol controllers typically generate an interrupt for every byte of data that is received, which introduces timing variability into the

communication as mentioned earlier. This variability may lead to the loss of synchronization between transmitter and receiver during the communication of the packet thus leading to energy losses through retransmissions.

In general there will be clock differences between any two clocks. Crystals are typically accurate to about +/- 50 ppm (parts per million), which means that they may drift about 50 cycles in every 1 million cycles that they generate. A typical transfer of a byte takes about 32 cycles in the STI implementation that we have. Thus, a packet of 100 bytes that will take 100*32 = 3200 cycles to transmit/receive should be communicated without synchronization problems caused by clock drift. In this situation, the code generated by software thread integration, due to its static scheduling (and hence lack of variability in execution time) ensures that the transmission and reception maintain timing accuracy and happen at exact intervals of time. The basic requirement is that by the end of the packet (since synchronization occurs at the beginning of the communication of each packet) clocks can not have drifted more than 1 bit time totally. If we were to choose a conservative figure of ½ * bit time for the clock drift to be on the safer side, then the maximum frequency error allowed for a crystal so that the nodes do not lose synchronization is

Max. Allowable Frequency Error between crystals in ppm $= 10^6$ **(0.5 * bit time)/ (Duration of the longest packet)*

*Where, duration of longest packet = 8 * length of longest packet in bytes * bit-time*

Maximum allowable frequency error for 1crystal in ppm = (Max. Allowable Frequency Error between crystals in ppm)/2

Equation 3-1: Maximum frequency drift without loss of synchronization

In case we consider a maximum packet length of 250 bytes and a maximum bit-rate of 500 kbps, the maximum allowable frequency error per crystal evaluates to 125 ppm, which is far greater than the maximum frequency drift in currently available crystals. However, with increase in packet size beyond 625 bytes (most common packet sizes in sensor networks are under 100 bytes); the maximum frequency stability begins to fall below 50 ppm. After this, very high frequency stability crystals (+/- 6 ppm) will be required and this can prove expensive.

Thus, if the clock generators for both the transmitter and the receiver modules have high frequency stability and match each others frequency as closely as possible, then this technique ensures that there will be no error in reception of the packet due to loss of synchronization. Thus, after gaining synchronization at the beginning of the packet, it will not lose it till the last bit of the packet has been received. It becomes unnecessary in this scenario to have to have start and stop bits for every byte of data in the packet. Thus, the overhead of having start and stop bits for every byte is reduced to having start and stop bits for one whole packet. This can reduce the energy consumption twofold:

i. The reduction in the overhead of start and stop bits (2 bits for every 10 bits of communication = 20% overhead) is sizeable. The amount of reduction is dependent on the size of the packet i.e. the larger the packet, the higher the percentage of savings that will be observed. The radio module now has to transmit and receive that much less amount of data, thus saving power in both the radio module and the microprocessor module.

ii. The perfect synchronization between the transmitter and receiver during the communication period ensures that there are no losses of packets due to loss of synchronization. This leads to a lower number of retransmissions in reliable protocols like TCP, thus causing reduction in energy consumption in both the radio module and microprocessor.

STI allows a byte-level protocol controller that has to pad every byte with start and stop bits to be used as a packet-level protocol controller, which only has to pad each packet with start and stop bits thus allowing for sizeable energy savings.

Chapter 4

HARDWARE AND SOFTWARE ARCHITECTURE

4.1 Hardware Architecture

The nodes use an Atmega128 microcontroller from Atmel® and a Chipcon® CC1000 for its communication needs.

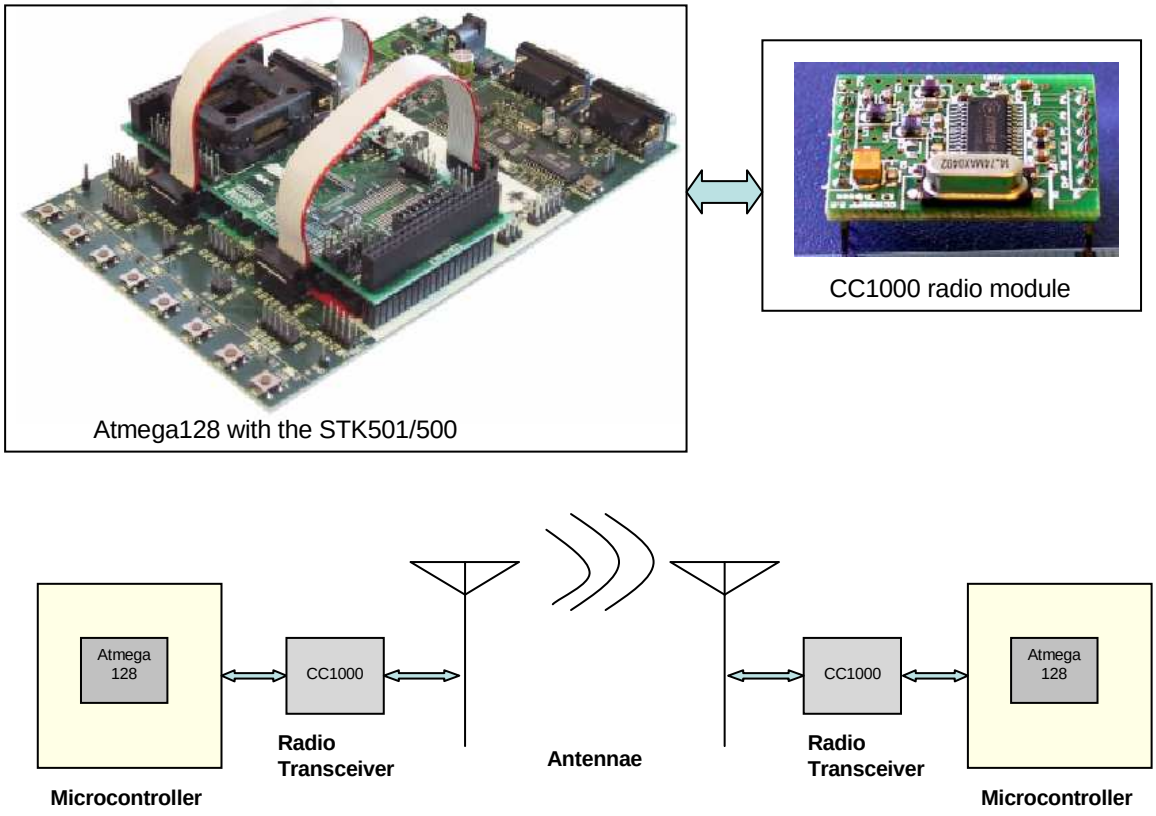

Figure 4-1: Sensor Nodes with an Atmega128 and a CC1000

The architecture described above has not been implemented in its entirety. This, however, is the architecture that the power analysis has been performed with. The Atmega128 microcontrollers were connected to each other in the wired implementation described in Section 4.3.2 for the purpose of actually measuring the effect of interrupt overhead in terms of the number of cycles during communication between the two nodes and also for running the integrated threads to verify the accuracy of the integration. The communication and DSP threads were run on this architecture to verify their correctness before and after integration. We have not used the CC1000 RFM since its energy performance can be completely and accurately characterized based on data gained from the working of the wired implementation of this architecture and transferring it to the wireless domain using numbers from its datasheet as has been done here.

4.1.1 Atmega 128 Microcontroller

The Atmega 128 implements the AVR architecture, a RISC architecture featuring 8 bit native word size, 32 general-purpose registers, and limited support for 16 bit operations. The processor features a two-stage pipeline. This processor has multiply but lacks divide instructions. Data memory is byte-accessible and byte-aligned. The Atmega 128 is currently at the top of the performance spectrum of the AVR device family. We use an Atmel STK500 evaluation board with a 16 MHz clock. On-chip memory consists of 4 kilobytes of SRAM and 128 kilobytes of Flash EEPROM. In addition, 32 kilobytes of external SRAM are used (with a one cycle performance penalty). No cache exists, and no coprocessor is available. The C compiler used is AVR-GCC 3.2. It possesses on-chip debugging support in the form of a JTAG interface to which a JTAG emulator can be plugged in. It also possesses an on-chip 10-bit ADC. The frequency of the Atmega128 can be varied from 0.3 MHz up to 16 MHz.

The Atmega128 also possesses Serial Peripheral Interface (SPI) that allows for high speed synchronous communication between several AVR devices. The SPI can be configured to run at speeds ranging from $f_{cpu}/2^1$ to $f_{cpu}/2^7$. This means that each bit will be transferred via the SPI interface between two devices at speeds ranging from 2 cycles to 128 cycles depending on the speed configured. Higher bit-rates like $f_{cpu}/2^1$ mean that an entire byte is transferred in 2*8=16 cycles. Thus, every 16 cycles a new byte will be transferred. This will however not be possible with interrupt-driven implementations of byte-level controllers since the interrupt overhead is much higher than the 16 cycles that is required to transfer a byte. This prevents

implementations using such high-speed transfers from using ISR based schemes and limits them to using busy-wait schemes.

4.1.2 Power Characteristics of the Atmega 128

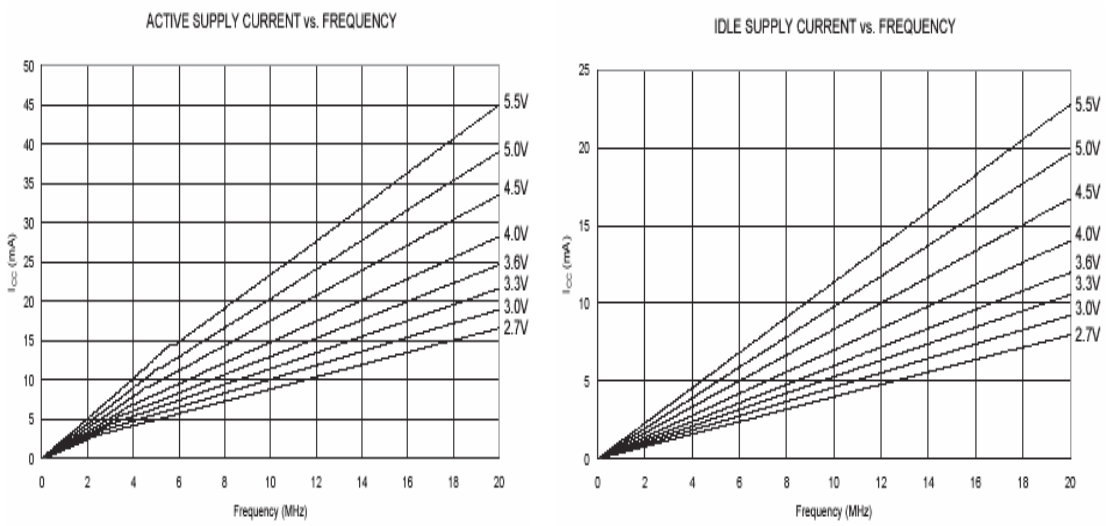

Figure 4-2: Frequency vs. Power Characteristics for the Atmega128 [24]

The Atmega 128 supports voltages from 2.7V to 5.5V. We use an operating voltage for the Atmega 128 of 3.0 V to reduce power consumption. The Chipcon CC1000 is also run at 3.0 V, which is its recommended operation voltage. The power consumption of the Atmega 128 depends on the duration for which it stays in active mode, the duration it stays in idle mode or sleep mode, the frequency at which it is operating and the voltage supply it is operating at. The power characteristics show that the energy consumption increases with increase in operating frequency and the square of the supply voltage. The longer that the microprocessor stays in active mode, the larger is the amount of energy that it consumes. Hence, the microprocessor must attempt to stay in a sleep mode like the idle mode as long as possible. The Atmega 128 has five other sleep modes apart from idle: ADC Noise Reduction, Power-down, Power-Save, Standby and Extended Standby where the supply current reduces to a few 10s of micro-amperes from a few milli-Amperes in Active and Idle modes. The MCU can be sent into any one of these modes by simply writing into the MCU Control Register (MCUCR).

Idle Mode – In this mode, the MCU enters Idle mode, stopping the CPU but allowing the SRAM, Timer/Counters, SPI port, and interrupt system to continue functioning. This sleep

mode basically halts clk_{CPU} and clk_{FLASH}, while allowing the other clocks to run. The MCU can be woken up from this sleep mode by externally triggered interrupts as well as internally triggered ones.

ADC Noise Reduction Mode – In this mode, the CPU is stopped along with all of the I/O modules except Asynchronous Timer and ADC, to minimize switching noise during ADC conversions. This sleep mode basically halts $clk_{I/O}$, clk_{CPU}, and clk_{FLASH}, while allowing the other clocks to run. Apart form the ADC Conversion Complete interrupt, only an External Interrupt can wake up the MCU from ADC Noise Reduction mode.

Power-Down Mode - In this mode, the register contents are saved, the External Oscillator is stopped, disabling all other chip functions until the next interrupt or Hardware Reset. This sleep mode basically halts all generated clocks, allowing operation of asynchronous modules only. When waking up from Power-down mode, there is a delay from the wake-up condition occurs until the wake-up becomes effective. This allows the clock to restart and become stable after having been stopped.

Power-Save Mode - This mode is identical to Power-down, with the exception that the asynchronous timer continues to run, allowing the user to maintain a timer base while the rest of the device is sleeping. This sleep mode basically halts all clocks except clk_{ASY}, allowing operation only of asynchronous modules, including Timer/Counter0 if clocked asynchronously.

Standby Mode - This mode is identical to Power-down with the exception that the Crystal/Resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low power consumption. From Standby mode, the device wakes up in 6 clock cycles. This mode is available only with an external crystal or resonator.

Extended Standby Mode - This mode is identical to Power-save mode with the exception that both the main Oscillator and the Asynchronous Timer continue to run. From Extended Standby mode, the device wakes up in 6 clock cycles. This mode is available only with an external crystal or resonator.

The overhead introduced by power-down, power-save modes can be as high as 18 cycles for an external RC oscillator. To enter any of the six sleep modes, the SE bit in MCUCR must be

written to logic one and a SLEEP instruction must be executed. The SM2, SM1, and SM0 bits in the MCUCR Register select which sleep mode (Idle, ADC Noise Reduction, Power-down, Power-save, Standby, or Extended Standby) will be activated by the SLEEP instruction. If an enabled interrupt occurs while the MCU is in a sleep mode, the MCU wakes up. The MCU is then halted for four cycles in addition to the start-up time; it executes the interrupt routine, and resumes execution from the instruction following SLEEP. The contents of the register file and SRAM are unaltered when the device wakes up from sleep. If a reset occurs during sleep mode, the MCU wakes up and executes from the Reset Vector. Thus, the overhead incurred by going to sleep in any of the sleep modes apart from idle can be between 20-30 cycles, which are not inclusive of the time taken by the ISR for the interrupt that woke the MCU.

Hence, we chose to use the *idle* sleep mode, which takes the least overhead among all of the sleep modes and due to the fact that it only stops the CPU while allowing the SRAM, Timer/Counters, SPI port, and interrupt system to continue functioning. This is suitable for our purposes as we need to allow the SPI port to continue functioning so that it can finish the communication of data while the CPU is off. When the CPU has no work to do, including waiting for communication to finish, it is put into *extended standby* mode due to the energy savings of more than 2 orders of magnitude when compared to active mode and because the overhead incurred is only about 6 cycles. The extended standby mode is chosen since the timer/counter 0 can be used to wake the system up after a certain fixed period of time. This is not possible with the standby mode. The power-down and power-save modes have high start-up times and hence are not preferred.

The SPI interface is used along with the radio module, the CC1000 for wireless communication.

4.1.3 CC1000 RF Transceiver

The CC1000 is a single-chip UHF transceiver designed for very low power and very low voltage wireless applications. The circuit is mainly intended for the ISM (Industrial, Scientific and Medical) and SRD (Short Range Device) frequency bands at 315, 433, 868 and 915 MHz, but can easily be programmed for operation at other frequencies in the 300-1000 MHz range. The CC1000 can switch between Reception and Transmission or between two different frequencies in 200 μs or less, ensuring as little overhead as possible in a two-way application. The frequency synthesizer of the CC1000 can be programmed in steps of 250Hz for any

frequency between 300 MHz and 1 GHz, thus providing a fine-grained accuracy in controlling the communication frequency. The CC1000 supports data rates of up to 76.8 kbits/s. There are other low power RF transceivers like the Chipcon® CC1020 that support higher bit-rates of up to 153.6 kbits/s.

4.1.4 Power Characteristics of the CC1000

The CC1000 has a voltage supply range of 2.1 V to 3.6 V, allowing it to run on two standard AA or AAA batteries. The low current consumption of less than 10 mA at 868 MHz and 915 MHz (the current consumption at 433 MHz is even lower) enables an active time of more than 200 hours (using alkaline AA cells with a capacity of 2600 mAh and allowing for current consumption by the micro controller). In power-down mode, the CC1000 draws less than 1 μA. Ultimately, this means the stand-by time for the node is only limited by the shelf life of the batteries. It offers great flexibility for power management in order to meet strict power consumption requirements in battery operated applications. Power Down mode is controlled through the MAIN register. There are separate bits to control the Receiver part, the Transmitter part, the frequency synthesizer and the crystal oscillator. This individual control can be used to optimize for lowest possible current consumption in a certain application. The characteristics of the CC1000 are as listed below.

Specifications		Min.	Typ. (433 / 868 MHz)	Max.	Unit
General:	RF Frequency Range	300		1000	MHz
	Data Rate	0.6		76.8	kbit/s
TX Mode:	Output Power (programmable)	-20		10/5	dBm
	FSK Separation (programmable)	0		65	kHz
RX Mode:	Receiver Sensitivity, 1.2 kbit/s		-110/-107		dBm
Power Supply:	Supply Voltage	2.1		3.6	V
	Current Consumption, RX:		7.4/9.6		mA
	Current Consumption, TX, -20 dBm		5.3/8.6		mA
	Current Consumption, TX, -5 dBm		8.9/13.8		mA
	Current Consumption, TX, 0 dBm		10.4/16.5		mA
	Current Consumption, TX, 5 dBm		14.8/25.4		mA
	Current Consumption, TX, 10 dBm		26.7/-		mA
	Current Consumption, power down		0.2	1	µA

Figure 4-3: Characteristics of the CC1000 RF Transceiver

4.1.5 Power Characteristics of the node

The overall power usage of the node is as shown in the following graphs:

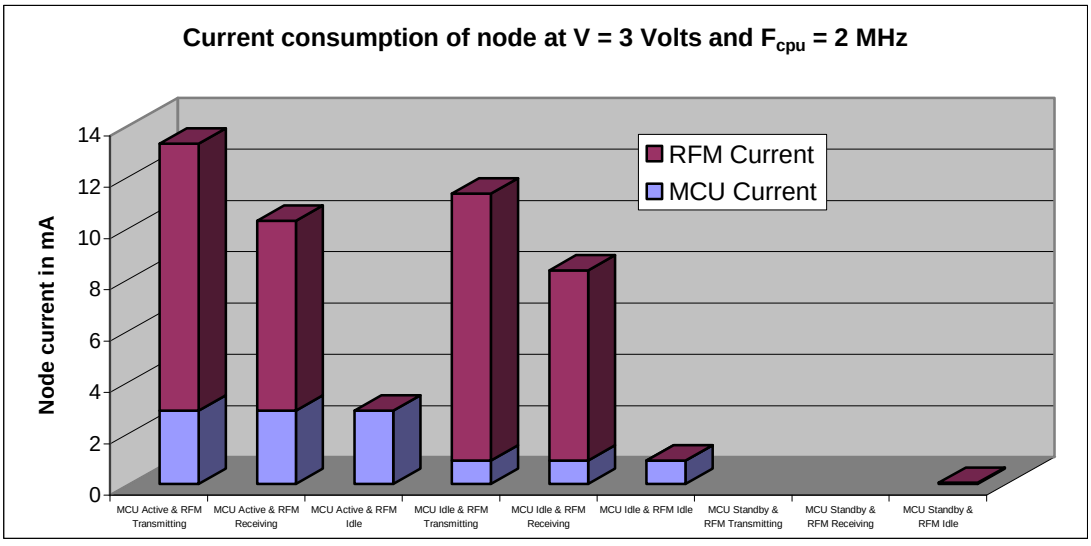

Figure 4-4: Current consumption of node at f_{CPU}=2MHz and V=3 volts

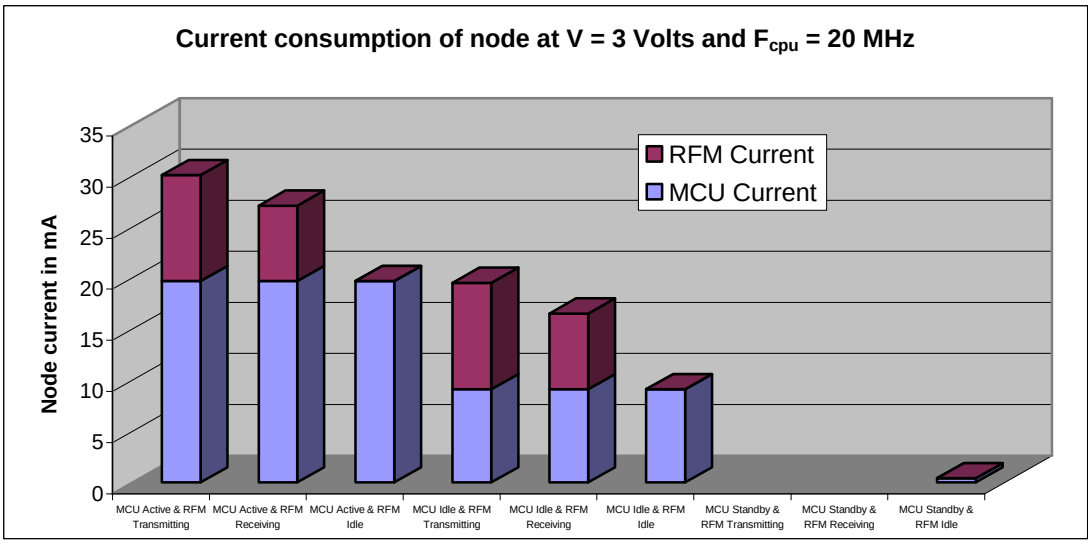

Figure 4-5: Current consumption of node at f_{CPU}=20MHz and V=3 volts

The power consumption of the node in mW can be determined by multiplying the current consumption shown in the above graphs by the supply voltage of 3V. During the period that the MCU is in extended standby mode, the RFM also has to be in standby/idle mode since it is not possible for the node to begin or continue communication while the MCU is in extended standby mode. Hence the bars corresponding to the current consumption of the node while the MCU is in extended standby mode are absent when the RFM is in transmit/receive mode.

4.2 Testing and verification of results

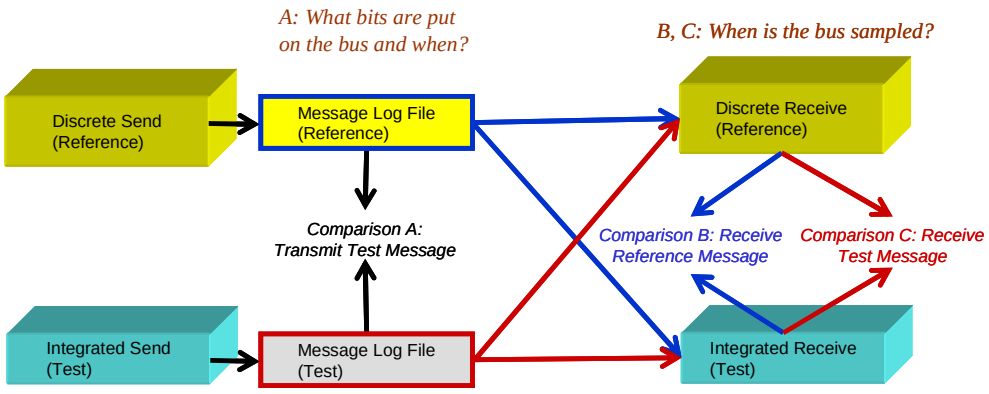

Figure 4-6: Overview of verification approach

In order to verify the correctness of the code after transformation with STI, we simulated the integrated code with AVR Studio and generated log files that record the data output on a particular port along with the time in the number of cycles of CPU time from the start of the program at which this data was output. The microcontroller is run at 4 MHz. These log files were then compared with known good (reference or "gold") log files created from simulations using the original (non-integrated) code. The timing of the code that generated the gold log files had been set to exactly match a certain bit rate, which in our case was 100kbps.

The verification procedure [26] we followed is as shown in Figure 4-6. Comparison A determines if the times and values of the signals transmitted by the STI implementation match the reference. In order to perform comparisons B and C, the log files generated by the transmission of data using the STI implementation and the reference implementation was input to the receive threads of both the STI and reference implementations. A hardware debugging signal on an output pin of another port was used to indicate the bus sampling times. This debugging signal is similar to the bit that is set in the SPI status register when a byte of data has been completely received in the SPI data register. The log files generated by the integrated (using STI) receive thread are now compared to those generated by the reference receive thread (non-STI, discrete-padded) to determine the timing accuracy of the bus sampling instant in comparison B. In comparison C, the data received is also compared (using log files) to that which was transmitted using both the STI and non-STI (discrete-padded) versions of the transmit code and the accuracy of reception is verified.

4.3 Software Architecture and Design

In order to demonstrate and verify the power savings from using software thread integration, a pair of communication threads (Transmit and Receive) have been integrated with a common application like digital filters that are used in sensor nodes as part of various voice/sound recognition schemes such as would be used in the tracking of bird calls.

4.3.1 The Application

The digital filters that have been implemented here are an 8^{th} order Finite Impulse Response (FIR) filter and a 6^{th} order Infinite Impulse Response (IIR) filter. These filters have been

chosen due to the fact that they are part of the most common implementations of sound recognition schemes. These filters can be made to behave like high-pass, band-pass or low-pass filters by merely changing the value of their filter coefficients. Another characteristic of these filters is that, the higher the order of the filter, the better is their ability to filter out the unwanted components of its input signal.

The digital filters are implemented using fixed point arithmetic. There is a large amount of floating point arithmetic involved in the calculation of the filter output values. The atmega128 does not have a Floating Point Unit (FPU) or a floating point co-processor and hence this arithmetic can cause the program to run extremely slowly. Using fixed point arithmetic speeds up the execution of the program by almost an order of magnitude. In this implementation, 32 bits in total are used to represent a real number, 16 of which are for the integer portion and the remaining 16 for the fractional part. The details of such an implementation of digital filters using fixed point arithmetic along with the issues that are required to be considered in such an implementation such as overflow may be found in ATMEL's AVR 223 Application Note [21].

4.3.2 The communication threads

The transmit and receive threads use the Serial Peripheral Interface (SPI) to send and receive data. They are required to send and receive bytes at a certain agreed upon bit rate. The bit rate is fixed by setting the SPI bus frequency as a factor of the clock frequency of the chip as mentioned earlier. The transmit and receive threads run on two different nodes and communicate with each other at a predetermined bit-rate. They are called by an application to transmit/receive a packet to/from another node.

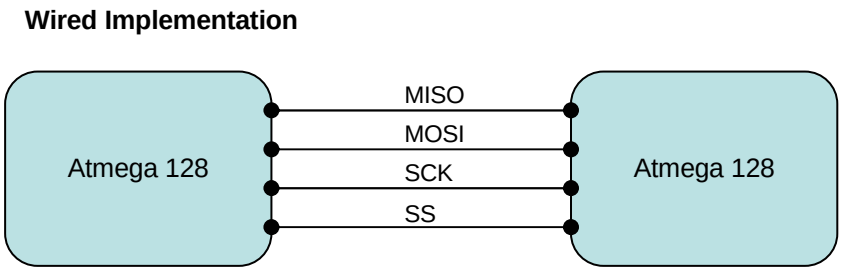

Figure 4-7: Wired Implementation of SPI Communication

The transmit node is configured as a SPI master and the receive node as a SPI slave by means of driving a slave select pin high for a master and low for a slave. This allows high-speed synchronous data transfer between the ATmega128 and peripheral devices or between several AVR devices. The CPU using the SPI operates in either the master or slave mode. As shown in the above Figure 4-7 there are four lines that control the operation of SPI. The SPI Master initiates the communication cycle by pulling low the Slave Select (SS) pin of the desired Slave. Master and Slave prepare the data to be sent in their respective Shift Registers, and the Master generates the required clock pulses on the SCK line to interchange data. Data is always shifted from Master to Slave on the Master Out Slave In (MOSI) line, and from Slave to Master on the Master In Slave Out (MISO) line. After each data packet, the Master will synchronize the Slave by pulling high the Slave Select (SS) line. This operation, where two 8-bit shift registers communicate by shifting data onto and from the bus, is similar in operation to the serializer/deserializer found on some protocol chips. Therefore a study of the SPI scheme provides a generic solution to the operation of many protocol controllers.

Wireless Implementation

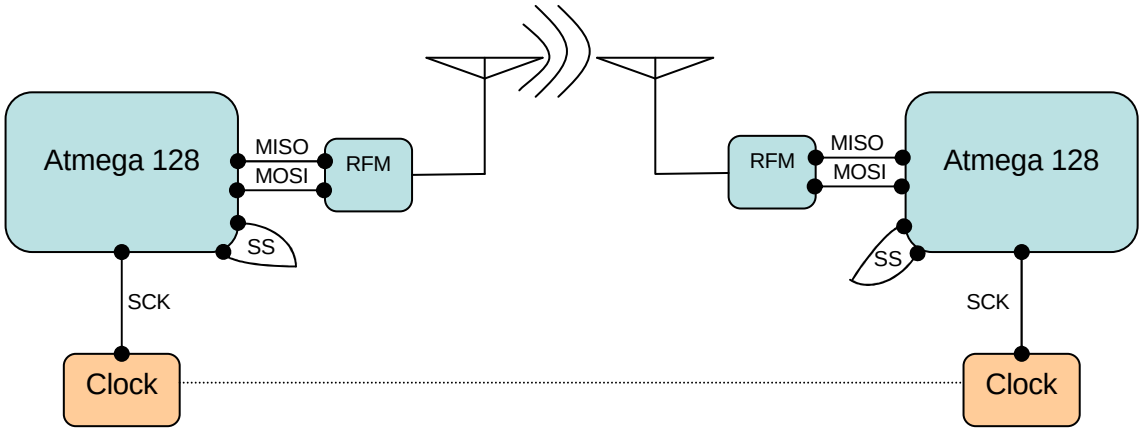

Figure 4-8: Wireless Implementation of SPI Communication

In a distributed environment, as in sensor networks, where there is no physical connection and communication is through the wireless medium, SPI is configured as follows. Separate clock lines drive the SCLK lines of various nodes. These clock signals need to be generated by oscillators that have high frequency stability and match each others frequency as closely as possible. This helps in maintaining synchronization between transmitter and receiver during communication. Any node that transmits becomes the master and transmits on the MOSI line.

The intended recipient in turn pulls its SS line low and operates as the slave and receives on the MISO. The RF Module (RFM) is connected to the MOSI and MISO lines. Control is governed by the SPI Control Register. The status register is used to verify the current status and the byte to be transmitted is written to the SPI data register while the received byte is read from it. An ISR can be setup to execute each time a byte is transmitted or received or the method of polling the bit on the status register that indicates the receipt of a full byte of data can be used.

The transmission begins when the transmitter writes into the SPI Data Register (SPDR) and ends after all the bytes have been shifted out. The Slave may continue to place new data to be sent into SPDR before reading the incoming data. The last incoming byte will be kept in the buffer register for later use. The system is single buffered in the transmit direction and double buffered in the receive direction. This means that bytes to be transmitted cannot be written to the SPI Data Register before the entire shift cycle is completed. When receiving data, however, a received character must be read from the SPI Data Register before the next character has been completely shifted in. Otherwise, the first byte is lost. In SPI Slave mode, the control logic will sample the incoming signal of the SCK pin. To ensure correct sampling of the clock signal, the frequency of the SPI clock should never exceed fosc/4. Here, the timing accuracy is very important to not lose data. The higher the data rate, the more important the accuracy of timing becomes.

As we have discussed earlier, STI provides accuracy in timing in as far as the two clocks of the two nodes generate clock signals that vary in frequency from each other by a very small number. This number needs to be so small as to not cause a drift of a bit-time during the transmission of a packet. This accuracy is provided due to the static scheduling that is brought about by thread integration.

4.3.3 Integration

In order to achieve all the benefits of using STI that were discussed in chapter 4, we need to integrate the communication threads with the application threads so as to optimally use both the idle time and time available due to elimination of interrupt overhead. This integration can be achieved by using our thread integrator tool (Thrint) [19][22]. Thrint is a post-pass (back-end) compiler which reads assembly code, performs control-flow, data-flow and static timing analysis, and integrates threads. It uses a hierarchical thread representation, which simplifies

analysis and transformations. After the initial degenerate integration (combining the code for the two threads into one thread without regard for any timing or other issues), which is done manually, it makes thread control-flows compatible through various code transformations (initial integration, motion into code/conditional/loop, as well as integration of loops of different speeds (faster and slower)). These transformations create an integrated thread, which can be optimized for execution speed, timing accuracy, or code memory size. We use register file partitioning techniques to share the processor's register set with a minimum of spilling and filling. This implementation had little register pressure, and hence did not have to face a performance penalty for this. The integrated thread that was generated in our case was optimized for timing accuracy as that is of prime importance in this case.

Four integrated threads were generated after the completion of thread integration of the communication threads with the FIR filter and the IIR filter applications:

i. Receive thread integrated with the 8th order FIR digital filter application.

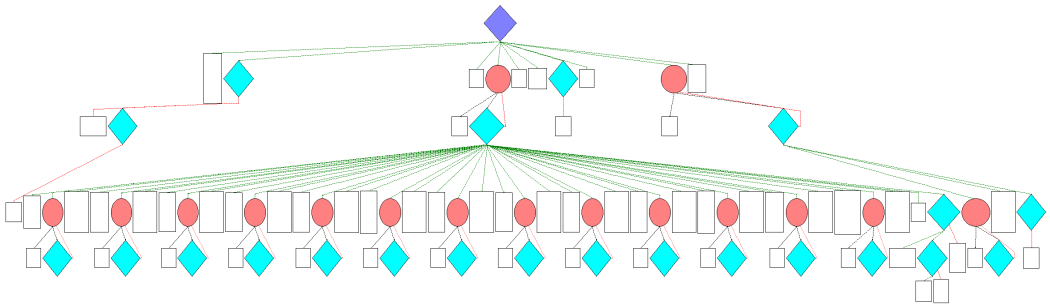

Figure 4-9: Rx thread integrated with 8th order FIR

ii. Transmit thread integrated with the 8th order FIR digital filter application.

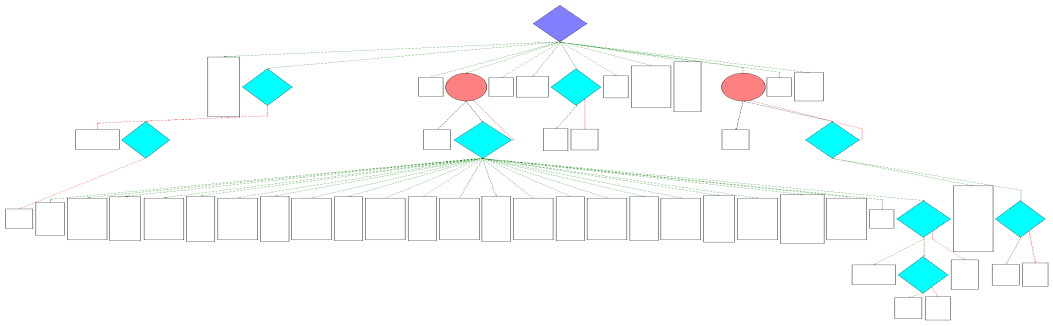

Figure 4-10: Tx thread integrated with 8th order FIR

iii. Receive thread integrated with the 6th order IIR digital filter application.

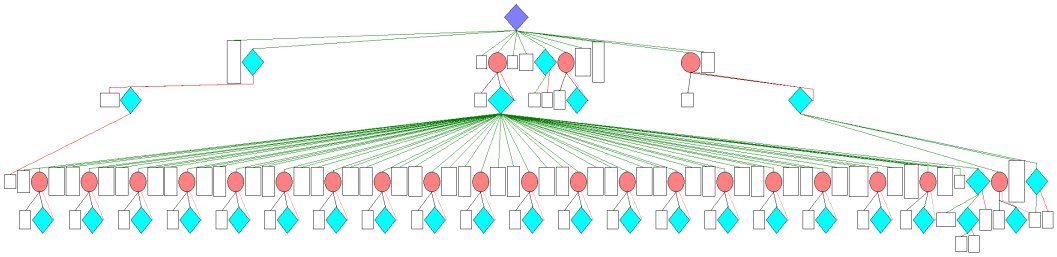

Figure 4-11: Rx thread integrated with 6th order IIR

iv. Transmit thread integrated with the 6th order IIR digital filter application.

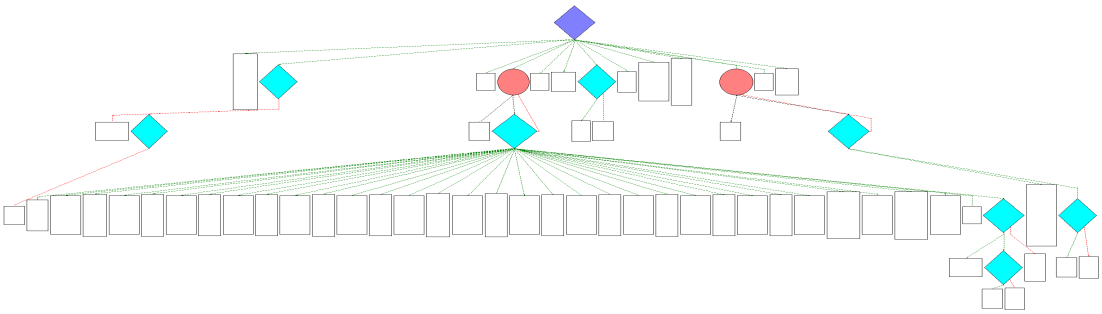

Figure 4-12: Tx thread integrated with 6th order IIR

This results in six threads in the system that can be executed i.e. the four integrated threads and the two un-integrated (discrete-padded) communication threads (Receive and Transmit). The un-integrated (discrete-padded) threads are just the transmit and receive threads that have not been integrated with the application threads. These threads have been passed through Thrint and have been statically scheduled so that they execute in a continuous loop and repeat at exact intervals. They do not use interrupts or busy waiting as the non-STI implementations do. This is done so that the timing accuracy provided by STI is maintained even when there is no other external work to be done. The idle time in these threads is used mainly by the CPU to go to sleep.

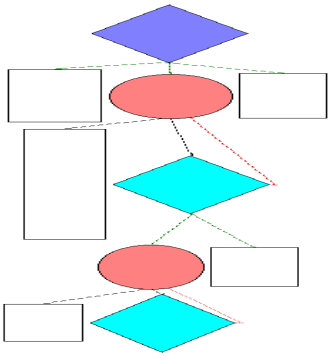

Figure 4-13: Discrete Padded Rx thread

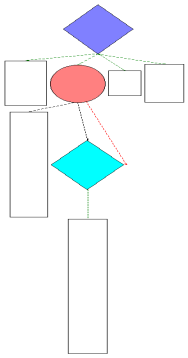

Figure 4-14: Discrete Padded Tx thread

The integration process causes a code size explosion due to padding. The code size of the integrated FIR-Tx and the integrated FIR-Rx threads increases by almost 42% and that of the integrated IIR-Tx and the integrated IIR-Rx threads increases by about 45%.

4.3.4 The System

Any system that can have more than one thread of execution that is ready to run at one time has to have a way of executing them all one by one. This can be achieved either by static or dynamic scheduling. Static scheduling is a scheme wherein the order of execution of the threads is predetermined by the designer of the system. Dynamic scheduling is a scheme wherein the order of execution of the threads depends upon certain scheduling algorithms implemented by the scheduler, which may be based on priority assigned to the threads or as simple as first-in-first-out (FIFO) or as complex as a priority ceiling based scheme.

This system has 6 threads that can potentially be in the ready queue, as mentioned above. The filter samples are collected separately and queued up as work for the digital filters. The filters pick up the work from this queue and execute them when they (the filters) are scheduled. The communication code transmits and receives packets of data as per the needs of the application that is using it. The packets of data can be of variable length as indicated by the length field in the packet. The packets are encapsulated by a start and stop bit to indicate the beginning and

end of the packet. The integrated code needs to be scheduled only when there is work to perform for both the communication and digital filter application code.

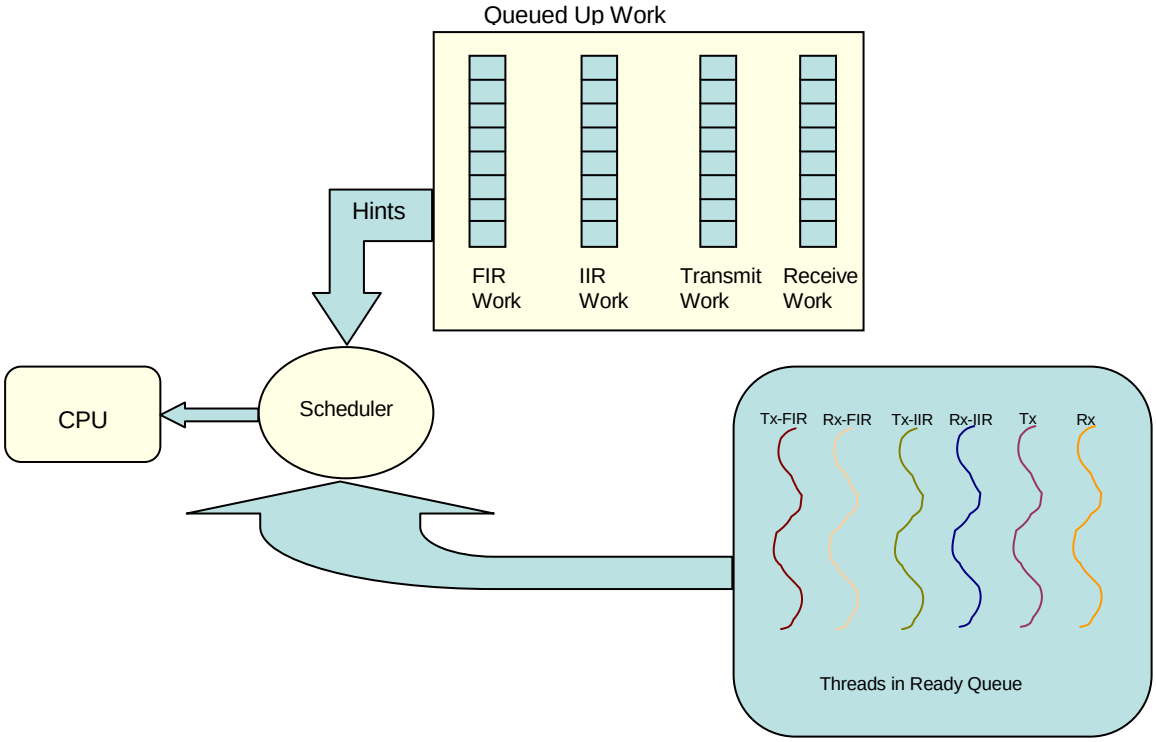

Figure 4-15: Scheduling Threads on CPU with STI

The scheduler in Figure 4-15 checks for which queues have work waiting in them and schedules the threads accordingly. If there is no work to be done for the digital filter applications, the scheduler schedules the non-integrated (discrete-padded) transmit or receive thread in case there is only communication work to be performed. If there is work for either the FIR or the IIR filter along with communication, it will schedule the appropriate integrated thread to run on the CPU. If there is no work to be done, then it, being a power-aware scheduler, will put the CPU in a sleep mode for a short, configurable period, if it will still be able to meet all deadlines by doing so. A point to be noted is that the communication module needs to be in active mode only when there is communication going on and it can go to sleep in case only the application is running. It can go to sleep only if by doing so, the deadline of the next communication period will not be missed i.e. there is so much idle time for the communication module that it has enough time to go to sleep and wake up before the start of the next communication period. In most cases it does not have that much idle time between transmitting bytes of the same packet due to the long startup times for the radio module, so it

will not go to sleep until the whole packet has been transmitted or received. However, the CPU itself has much lower overheads for going to sleep and waking up and so it may go to sleep between bytes of the same packet if possible. We have not implemented the scheduler in this thesis as part of the system as suggested in this architecture as part of this thesis since it is not germane to the ideas demonstrated herein. A similar scheme has been implemented with AVRX in [27].

The Tiny OS [9][23] scheduler is a simple function queue scheduler that by default schedules on a first-in-first-out (FIFO) basis. This can easily be changed so that it looks up the status of the various queues in which work accumulates and schedule the appropriate thread in the correct position in the function queue. It is simple to perform this modification to the Tiny OS scheduler since it is easy to insert and remove functions (threads) from the function queue as and when required and fits in very well with the architecture. These threads can very easily be fitted into the Tiny OS model as modules.

Thus, this architecture allows for performing integrated work so long as there is work to be performed by the integrated thread and if there is not, it finishes its work and goes to sleep.

Chapter 5

RESULTS – ANALYSIS AND DISCUSSION

The theory that the concepts of Software Thread Integration can bring about large savings in energy was evaluated by applying those concepts to the system discussed in the previous chapter on software and hardware architecture. We have attempted to verify the theory by modeling herein the following two design criteria for both the integrated and the non-integrated thread (ISR-based implementation):

1. Change in the energy consumption of the node as the clock frequency changes.

2. Change in the energy consumption of the node as the communication rate of the node changes.

We assume herein that:

- The MCU can go to sleep (extended standby mode) only when it is not doing any work i.e. when the CPU is not performing any secondary application work or executing instructions for communication and when the RFM is not performing communication with another node.

- The MCU can go to an idle mode only when it is not executing any instructions, however, it may still continue to perform any ongoing communication.

- The RF transceiver can go to sleep only when the communication work is finished i.e. a packet has been completely transmitted.

The results have been obtained for the energy consumption of a packet size of 35 bytes (chosen so as to match the packet format of Tiny OS with 23 bytes of data). The execution of the secondary thread occurs between the transmission/reception of each byte and continues

after the communication period if necessary. The CC1000 is assumed to be transmitting at 0.1 mW. A packet rate of 200 packets per second has been assumed. The values for the active, idle and extended standby currents for the Atmega128 have been ascertained from its datasheet [24]. The values for the transmit, receive and power-down currents of the CC1000 RF transceiver have been ascertained from its datasheet [25].

5.1 Experimental Method

We began with the ISR-based communication code, which was run at various bit-rates on the wired implementation of communication to verify the accuracy of the communication between two nodes. The number of CPU cycles used by the thread including interrupt overhead was measured using the information about the execution time given by the JTAG emulator and multiplying it by the clock frequency of the MCU. The average of these measurements was taken over several runs to minimize error and was duly recorded. Then, the code for the implementation of the digital filters was written and tested. The code in the filters was written with loops unrolled to increase its performance. The filters coefficients were designed to perform a low-pass, high-pass or band-pass filtering on the input waveform The testing of the filters was performed by feeding the node with (square and impulse) waveforms and comparing the input and output waveforms with a digital oscilloscope to verify the correctness of the design and implementation of the filter.

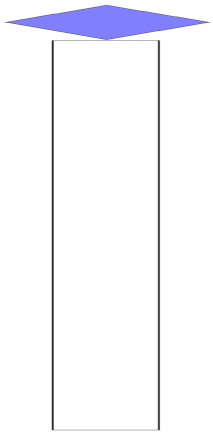

Figure 5-1: Control Dependence graph of FIR\IIR threads

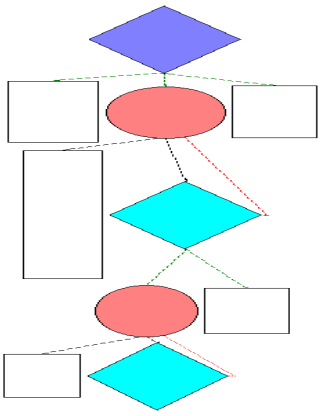

Figure 5-2: Control Dependence graph of the Rx thread

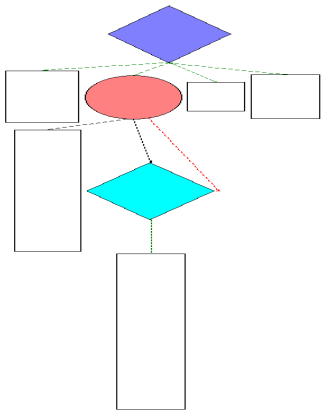

Figure 5-3: Control Dependence graph of the Tx thread

The digital filters were then integrated with the communication threads using STI to generate the integrated communication threads as shown in Figure 4-9, Figure 4-10, Figure 4-11 and Figure 4-12.

The correctness of the integrated thread was verified by simulations on a cycle accurate simulator as described in Section 4.2. The actual energy consumption of the node at various values of MCU clock frequencies and communication bit-rates was determined using the following items:

- The actual number of cycles of computation and communication code after integration as measured by summing up the number of cycles for the individual assembly instructions using thrint
- Data of active, idle and extended standby currents of the Atmega128 from its datasheet and the active and power-down currents of the CC1000 at 0.1 mW transmit power from its datasheet
- An assumed packet rate of 200 packets/sec and an assumed packet size of 35 bytes

These values were then inserted into a spreadsheet that we created to perform the necessary calculations of the total energy consumed by the node during the communication of a packet. The results are as shown in the following section.

5.2 Experiments and Results

The first experiment is to observe the variation in the energy consumption of a node with variation in the clock frequency of the MCU.

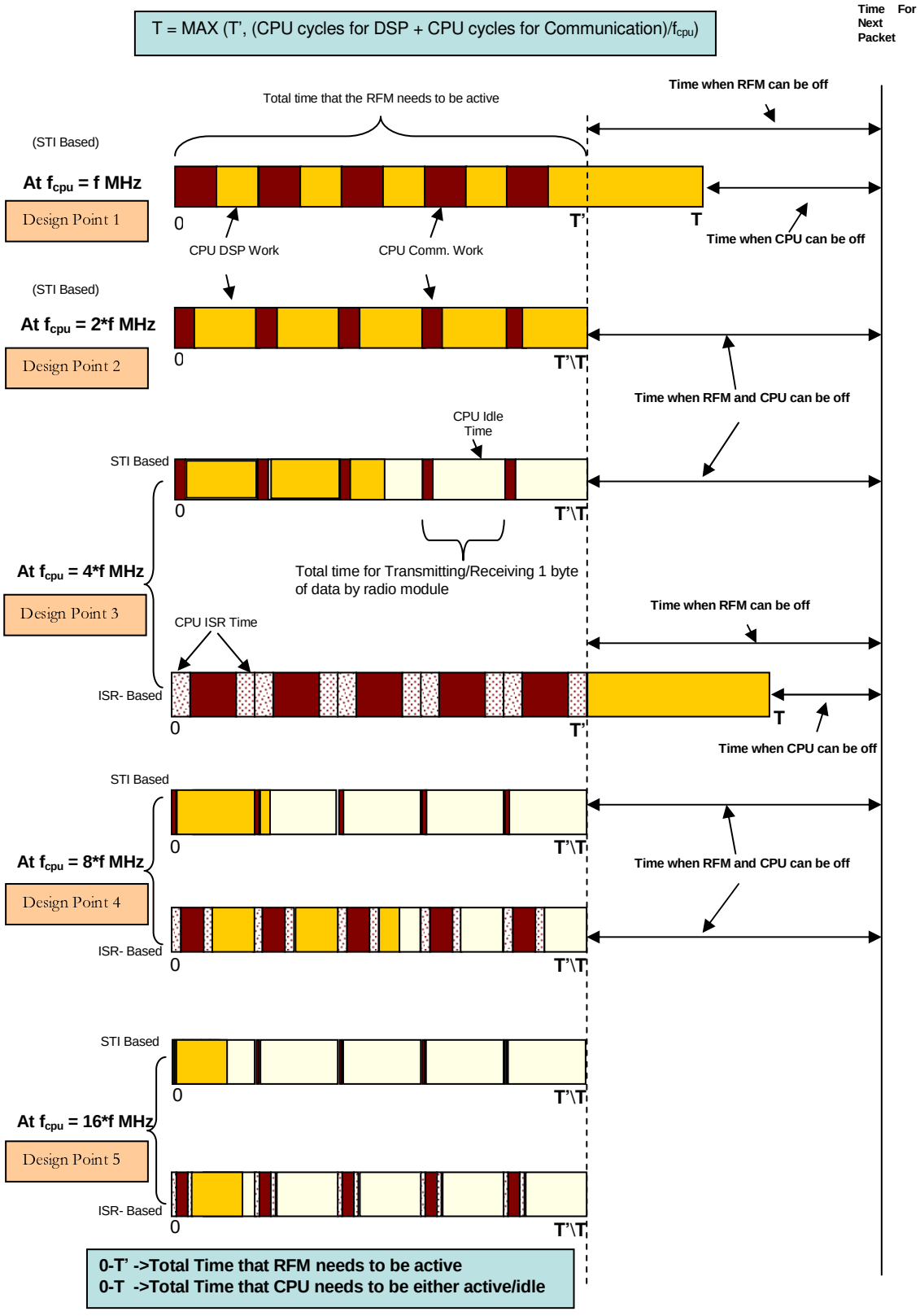

Figure 5-4: Effects of increasing frequency on energy consumption

Figure 5-4 shows the effects of increasing frequency on the energy consumption of a node that is communicating at a fixed bit-rate. As can be observed, due to the high overhead introduced by interrupts, ISR-based systems are not even able to run until the frequency is high enough to allow them to execute. This is due to the large number of cycles (235 cycles for the transmission/reception of a byte for the ISR based routines compared to 32 cycles per byte for the STI-based routines) that the ISR-based communication routines take for execution. This amount of interrupt overhead is similar to that observed in [27]. These communication routines need to finish execution before the deadline for the transmission of the next byte is reached. Increasing the frequency hastens the execution of these routines so that, in due course, the time taken by them to execute falls below that necessary for them to finish before the deadline for the next communication period. However, due to the lack of interrupt overhead in the STI-based schemes, they are able to transmit at the same bit-rate at much lower clock frequencies as there are far fewer cycles to execute. This leads to large savings in energy.

As can be seen in Figure 5-4, the MCU can be doing any one of four things at any point in time:

1. Running the code for the communication routines, or
2. Running the code for the DSP routines (FIR/IIR filters), or
3. In idle mode, or
4. In extended standby mode.

The MCU needs to be in active mode for the periods of time where it is running the code for the communication or DSP routines. It can go into idle mode only when it is not running any code but the MCU has to stay on due to the ongoing SPI communications. The idle mode shuts the CPU off but allows the SPI to continue running and hence this adds to the energy savings. After the completion of the SPI communications and the execution of code by the CPU, it can go into extended standby mode where the energy consumption of the CPU is almost 2 orders of magnitude lower than that of any of the other modes. The longer that the

MCU is in extended standby mode the less energy it consumes. The RF module (RFM) can go off as soon as it has finished with the communication portion to save energy.

As the clock frequency of the MCU increases, each execution cycle takes lesser amount of time. So the processing that needs to be done by the processor finishes sooner and sooner. Thus, the ISR-based routine is eventually able to execute at a certain frequency and the STI-based routines start to finish their processing earlier thus leading to an increase in idle time.

Design Point 1:

Initially, at a low MCU clock frequency (f), the ISR-based thread can not even execute and only the STI based thread is able to execute. Ideally, for better throughput and savings in energy, the DSP code should be executed during the idle time between bytes where the processor has finished execution of the communication code and is now waiting for the communication to complete so that it can send out the next byte in the packet. However, at f and below, though the communication code executes within time for the STI-based thread, the DSP-code is not able to finish its execution by the time the communication is complete so the processor has to stay in active mode for the entire duration of the communication and the duration for which it is executing the remainder of the DSP code. This leads to an increase in energy usage of the node due to the fact that the time that the node can go into extended standby mode is reduced by the time T-T' and due to the absence of any idle time within the communication period. As the bit-rate is constant in Figure 5-4, the time at which the RFM can go off stays invariant at T'. In Figure 5-4, all the savings in energy come from the amount of time that the node is in extended standby and idle modes.

Design Point 2:

At 2*f, the DSP code finishes just within the time that the node finishes communication. Here, the processor is able to go into extended standby mode for the maximum possible time at the chosen bit-rate and packet-rate. The maximum length of this time is limited by the difference between the inter-packet time and the actual transmission/reception time of the packet (which is in turn, limited by the bit-rate). This limitation is due to the fact that the processor has to be in idle or active modes for the duration of the transmission/reception of the packet to allow the communication to continue. At this frequency, the processor still has to stay active for the

entire duration of the communication since the processing takes up all the CPU time and hence it is not possible to go into idle mode. The ISR-based thread is still too long to run at this frequency.

Design Point 3:

At 4*f, the ISR-based thread is just able to run and the STI-based thread continues to run. In the STI-based thread, the DSP code is able to finish much faster thus leaving a large amount of idle time during which the processor can go into idle mode and thus save energy in addition to that which it already saves by going into extended standby mode. In the ISR-based thread, all the communication work barely finishes in time and all the DSP work has to now be done afterwards. This leads to a reduction in the time during which the CPU can go into extended standby mode, which leads to an increased energy consumption in addition to that incurred by not being able to go into idle mode. This leads to higher energy consumption in the ISR-based thread over the STI-based thread.

Design Point 4:

At 8*f, the available idle time increases for the STI-based thread and it is able to go into idle-mode for longer periods of time. The ISR-based thread is now able to finish its processing of the DSP code in advance so that it also has some idle time available to go into idle mode. Here too the ISR-based implementation uses more energy than the STI-based one but, here the difference between the two is reduced due to the fact that the ISR-based thread is now able to go into extended standby mode for all of its available standby time too in addition to going to idle mode for a short while.

Design Point 5:

At 16*f, the available idle time increases for both the STI-based thread and the ISR-based thread and both of them are able to go into idle mode for that duration. Here the difference between the two energy consumptions decreases further.

Hence, the trend that can be seen here is that as frequency increases, the difference between the energy consumption of a node that runs STI-based threads and one that runs ISR-based threads reduces. This is due to the fact that the effect of the increase in execution time

introduced by interrupt overhead becomes lower at higher frequencies as each cycle executes in a shorter amount of time when compared to the inter-byte time of communication. However, higher frequencies lead to higher current consumption for the active, idle and extended standby modes of the processor thus leading to an overall increase in energy consumption.

STI-based schemes allow the processor to run at a lower frequency while transmitting at the required bit-rate and completing the execution of the DSP code by the time the communication ends so that it can go into extended standby mode earlier (subject to the limitations stated earlier) and into idle mode whenever possible. The ISR-based schemes are able to run only at higher frequencies and only much higher frequencies allow them to finish early enough to make use of the standby and idle times. This leads to large savings in power and energy for the STI-based schemes as can be seen in the results that follow.

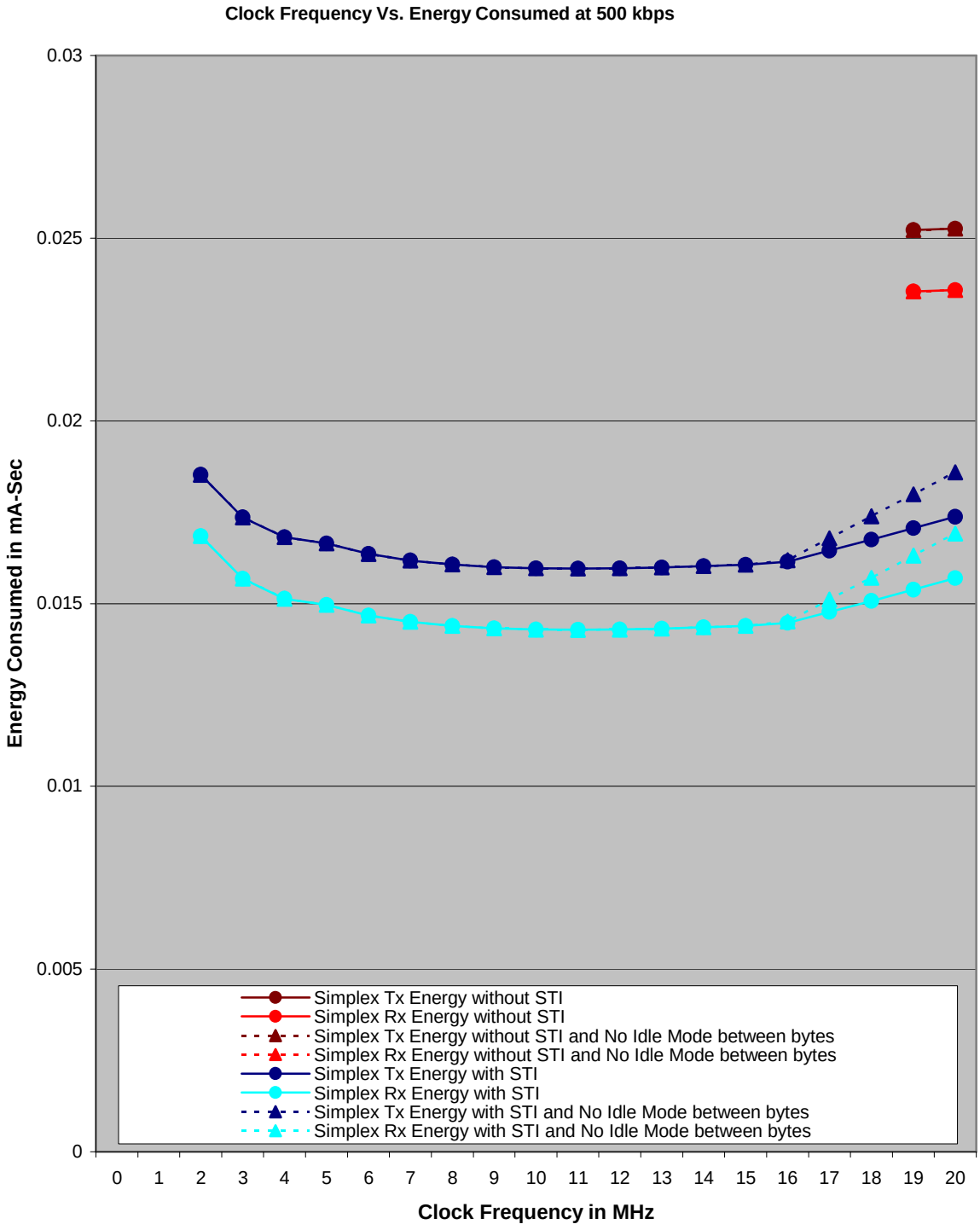

Figure 5-5: Variation in Energy consumption with Clock Frequency at a bit rate of 500 kbps

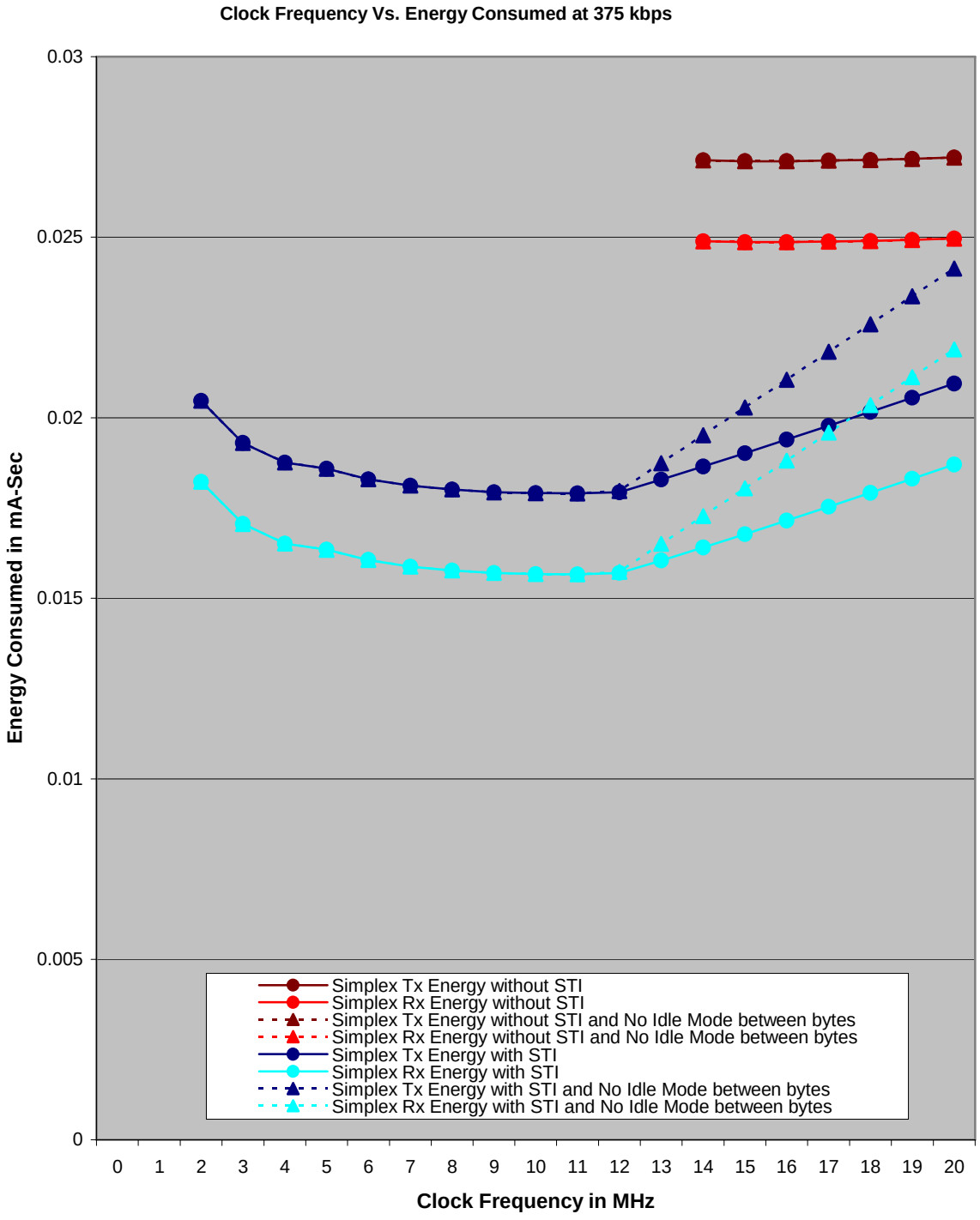

Figure 5-6: Variation in Energy consumption with Clock Frequency at a bit rate of 375 kbps

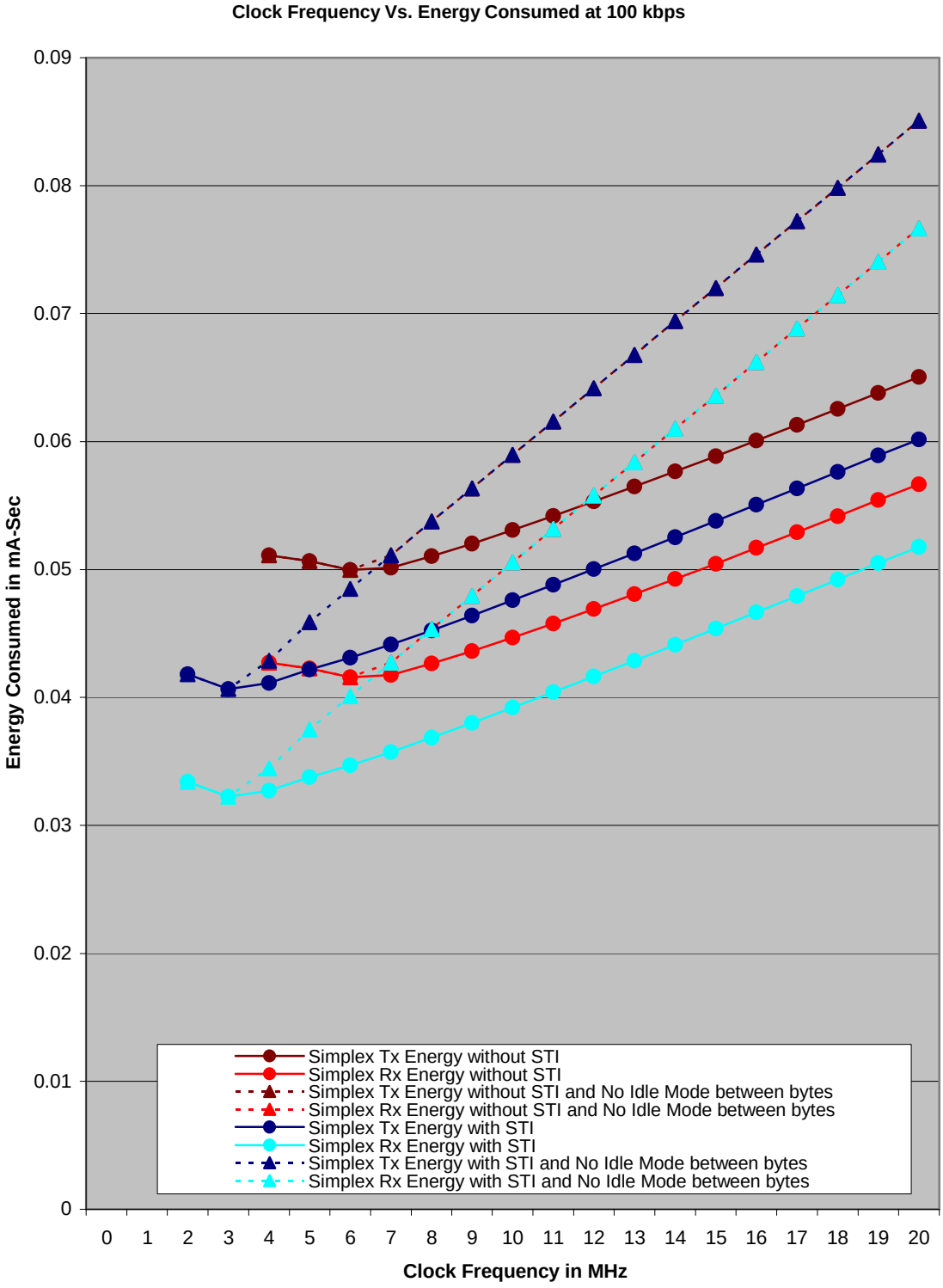

Figure 5-7: Variation in Energy consumption with Clock Frequency at a bit rate of 100 kbps

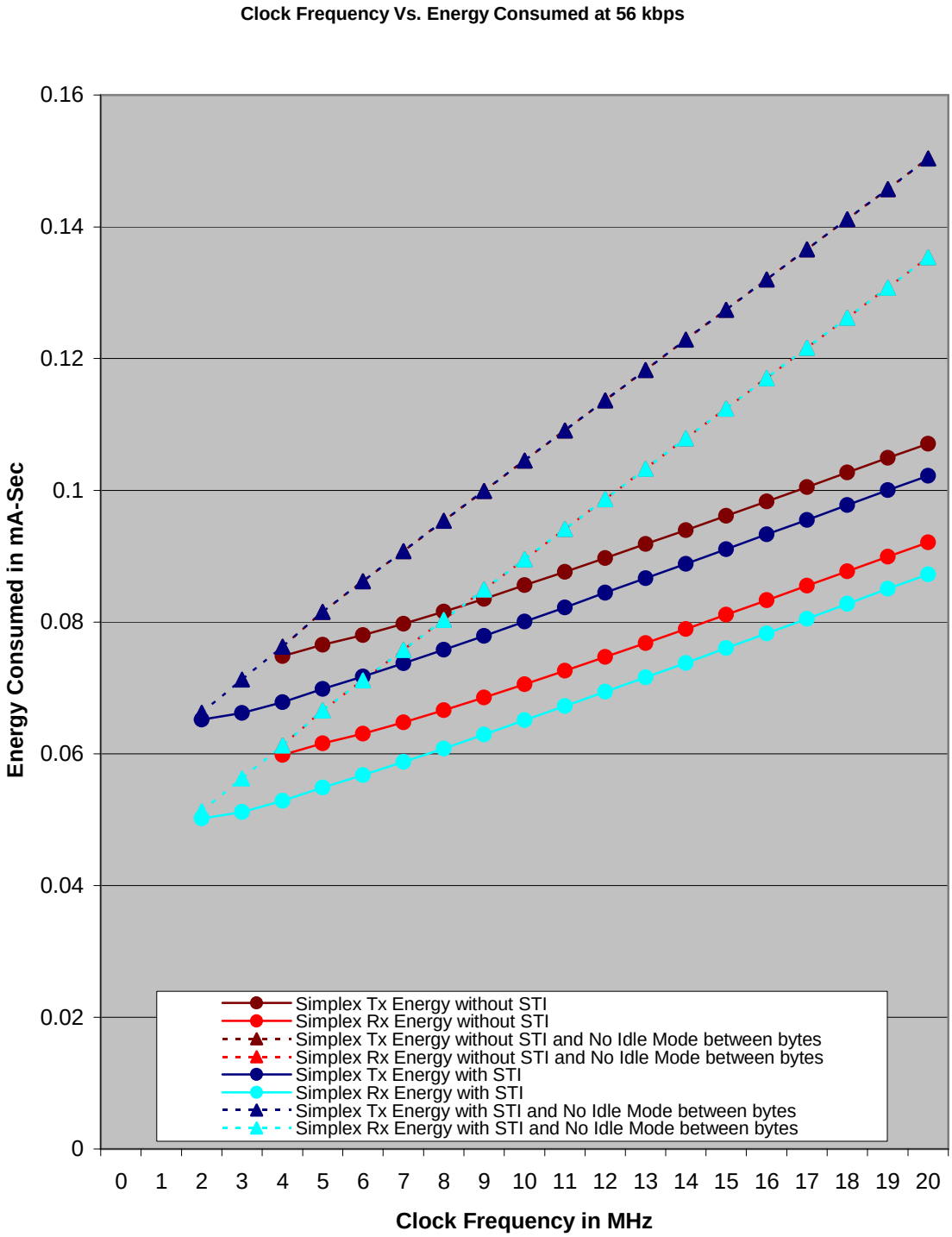

Figure 5-8: Variation in Energy consumption with Clock Frequency at a bit rate of 56 kbps

The graphs Figure 5-5, Figure 5-6, Figure 5-7 and Figure 5-8 show that there is a considerable savings in energy that can be obtained by using STI. All the graphs are for a constant bit-rate, so the power savings that can be achieved by turning the RFM off early is constant across the all the lines in a graph as can be observed in Figure 5-4 and hence all the savings observed in the graphs is due to savings from the CPU. The major savings in energy come from the standby time i.e. the time used by the CPU to go to extended standby mode due to its very low current consumption and low increase in magnitude with clock frequency.

All the graphs show an initial sharp decrease in the energy consumption with the increase in frequency followed by a period of very slow to no decrease in the energy consumption with increase in clock frequency. This phenomenon can be attributed to that fact that increase in frequency leads to a decrease in cycle time and this causes the work that needs to be performed by the CPU to take a shorter amount of time. Thus, more work can be accomplished in the time between the transmission/reception of any two bytes of a packet. This leads to less work that needs be done after the completion of communication and hence more time during which the CPU can stay in extended standby mode and less time that it needs to stay in active mode. This leads to energy savings for the node. The doubling of clock frequency causes the work to be executed in half the amount of time it would take with the original frequency. During the initial phase of sharp decrease, the percentage of frequency increase is also sharp i.e. the first step of increase of 1 unit between 1 MHz and 2 MHz amounts to a doubling (factor of 2 increase) of the frequency and hence halving of the number of cycles that the CPU work takes and hence the energy consumption drops sharply. The next unit increase leads to a 150% increase (factor of 1.5) in the frequency and a corresponding drop in energy consumption. Thus, let the lower frequency be f_1, the higher frequency be f_2, the energy consumption of the node at f_1 be E_1 and that at f_2 be E_2, then the relation between them may be defined as:

$$E_2 \propto (1-f_1/f_2)*E_1$$

Equation 5-1: Proportionality relation between energy consumption and frequency increase

Or, $E_2 = k_1*(1-f_1/f_2)*E_1 + k_2$,

Equation 5-2: Change in Energy consumption with frequency increase

Where, k_1 is the proportionality constant and k_2 is a constant that is dependent on the rate of increase of the active currents with increase in clock frequency.

The above equations are valid only for those values of frequencies during which the CPU work can not be completed during the communication period.

As can be determined from the equation the drop in energy consumption begins to taper off as the value of $(1-f_1/f_2)$ decreases with increase in the values of f_1 and f_2. This is what is observed in the graphs till the point where the CPU work just finishes by the end of the communication period. After this point, the energy consumption begins to increase with the increase in frequency. This can be attributed to the fact that after this point, the CPU work finishes before the end of the communication period. This leaves some time between transmission of bytes that is unused where the CPU can not turn off (extended standby mode) but has to stay in idle/active mode. If the CPU has an idle mode, as is present in the Atmega128, during which the communication process can proceed while the CPU itself is turned off then it can choose to go into idle mode to save energy and switch to active mode when necessary, else it has to continue to stay in active mode. In the case of the Atmega128, the energy consumption in the idle mode is about 50% of that in the active mode as opposed to that in extended standby mode, which is about two orders of magnitude less than that in active mode. After this point in the graphs, the period of time for which the CPU is in extended standby mode stays constant as it is now at its upper limit that is defined by the communication bit-rate:

Maximum Standby time = Time between the commencement of transmission of two packets – Time taken for communication of the packet.

Equation 5-3: Maximum Standby Time

Standby time at any frequency = Time between the commencement of transmission of two packets – Maximum(Time taken for communication of the packet, Time taken by CPU to complete processing).

Equation 5-4: Standby Time at any frequency

From this point on, the decrease in the active period is only possible by the CPU going into idle mode if it has no other task to perform between transmission/reception of bytes. The amount of time that the CPU is in idle/active mode is constant at the value of the difference

between the inter-packet time and the standby time. For processors that do not have the ability to go into idle mode, the total time that the processor has to stay in active mode is constant. Hence, the energy consumption of the node now begins to increase with frequency due to the increase in active current with the increase in frequency. For processors like the Atmega128 that possess the ability to go into idle mode, the energy consumption still continues to increase (albeit at a lower slope as the idle current ~= 50% of active current for the atmega128) since the idle current of the atmega128 also increases with increase in clock frequency. Also, with the increase in frequency, the amount of time that the processor needs to stay in active mode (t_{active}) decreases, which increases the time that the processor needs to stay in idle mode (t_{idle}) since $t_{active}+t_{idle}$= a constant as mentioned earlier. However, since the idle current is a large fraction (>= ½) of the active current, the overall energy consumption still increases, albeit at a lower rate than that in the processors that do not have the ability to go into idle mode.

As is evident from the graphs as well as the equations, the above discussion applies to both the STI- and the ISR- based threads. However, the STI-based threads consume less energy than the ISR-based threads. This is due to the fact that the STI-based threads are able to run at a lower frequency than the ISR-based threads, which leads to energy savings and that there are far fewer cycles of communication work that need to be performed in the STI-based threads due to the lack of interrupt overhead that forms a large part of the cycles of the ISR-based thread. This leads to larger amounts of idle time available to the STI-based implementations. This idle time is used to go into idle mode to conserve energy. Also, the STI-based threads are able to use up all the standby time to go into standby mode and are able to do so at an earlier time by finishing their work well in advance of the time stipulated for the communication of the packet.

A related interesting observation is that increasing the number of bytes per packet will also increase the number of cycles of interrupt overhead per packet, which is used by STI for performing DSP work and sleep. Hence, in this case the STI-based implementation will be able to sleep in idle mode for a longer period of time than the ISR implementation. This increase in the amount of sleep will lead to a larger amount of savings in energy for the STI implementation. The shape of the curves will remain the same as in the above case, just increasing the difference between the STI and ISR versions in terms of energy consumption.

Another observation that can be made by looking at the four figures Figure 5-5, Figure 5-6, Figure 5-7 and Figure 5-8 is that as the bit-rate increases, the percentage of energy saved with STI increases. The ISR and the STI implementations are both running at the same bit-rates, so the savings due to the RFM turning off early are the same in both the threads. However, the increase in bit-rates causes the overall energy consumption to reduce due to the fact that the RFM is able to turn off earlier with higher bit-rates. This also means that the communications end sooner and with the STI implementations it is possible to end the processing work sooner than the ISR implementations can possibly do due to the overhead of interrupts. This causes increased standby time and decreased active time with STI implementations, which leads to a lower energy consumption than the ISR-based implementations are capable of. One thing to bear in mind is that this particular portion of energy savings is limited by the difference between the number of cycles that the ISR-based thread has to execute as compared to that which the STI-based version has to execute. Hence, as can be seen in the graphs, the difference in the least energy consumptions of the STI- and ISR- based threads (difference between the lowest points in the graphs of the two implementations) is almost constant across the various bit-rates. This difference is directly proportional to the difference in the number of cycles between the two implementations. Hence, with the decrease in overall energy consumption, the total energy savings become a larger percentage of the overall energy thus leading to a higher percentage of energy saved with higher bit-rates.

The second experiment is to observe the variation in the energy consumption of the node with changes in the bit-rate of communication.

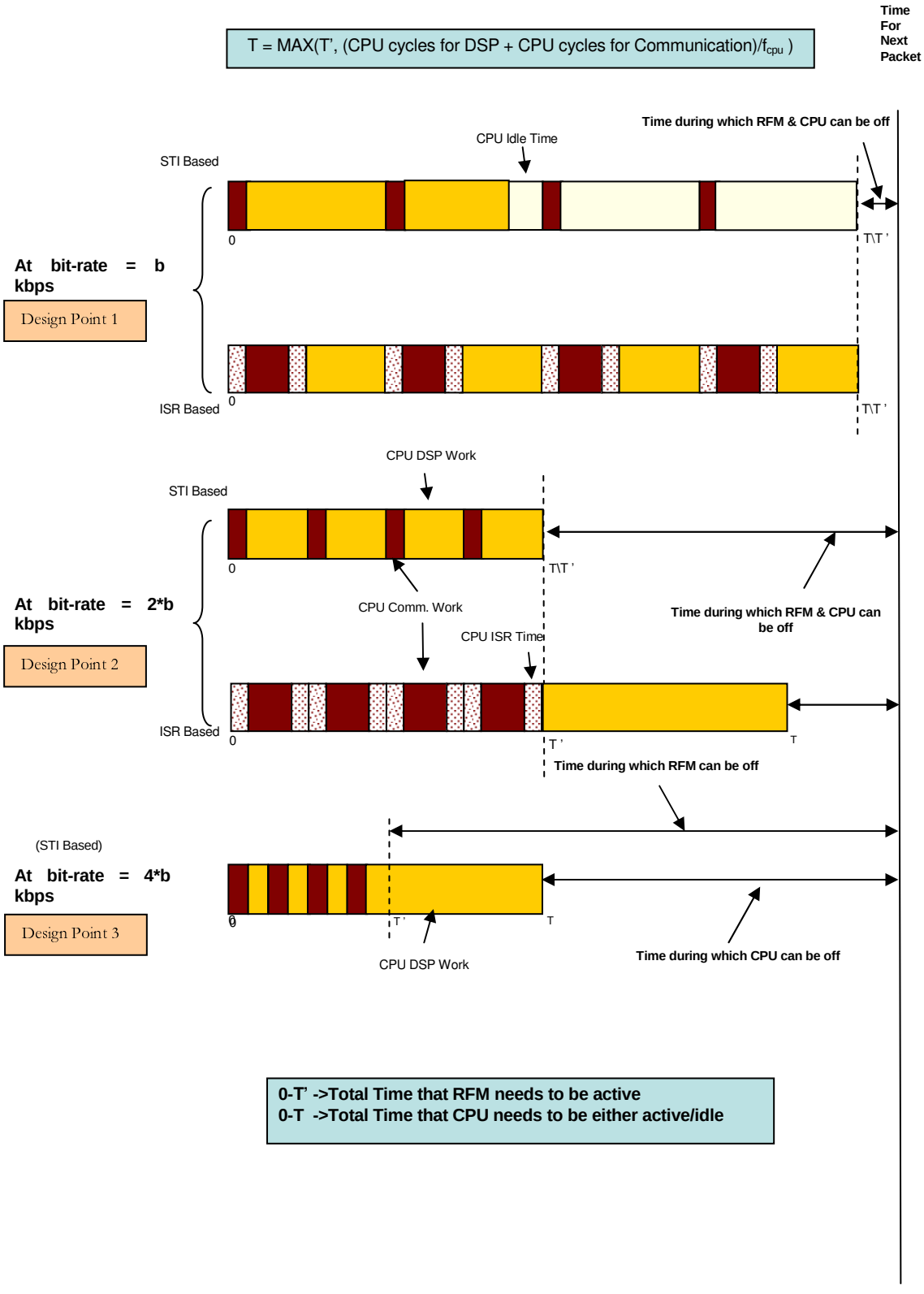

Figure 5-9: Effects of increasing bit-rate on energy consumption

Figure 5-9 shows the change in the energy consumption of the node with the increase in the communication bit-rate of the node. Here, since the frequency of the MCU clock is constant, the total amount of time taken by the CPU to execute instructions for communication and the DSP work (total processing time) is constant across all the bit-rates. Thus, the savings in energy in this case depend primarily on:

1. The RFM - How soon can the RFM be turned off after communication (T'):

   This is governed by the bit-rate of communication. The higher the bit-rate, the sooner the RFM can be turned off and the greater is the amount of energy that can be saved.

2. The CPU - How much lower is the transmission/reception time (T') than the total processing time (T):

   If the communication time is lesser than or equal to the total processing time, the CPU has to stay in active mode for the entire duration of the total processing time since the processing time becomes the bottleneck in this situation. The minimum amount of time that the CPU needs to be in active mode is fixed (since the frequency and the number of cycles are fixed) and thus the power consumed by the CPU will remain constant in this case. The energy consumption is the least in such a situation due to the RFM turning off early and the CPU is using the minimum amount of energy possible since it now does not have to wait for the RFM to complete communication. If the communication time were to be greater than the processing time, the CPU could go into idle mode and save energy but would not be able to go into extended standby mode until the end of communication, so the amount of energy saved would be reduced. The bottleneck in this case would be the communication time and the larger the communication time, the longer the RFM has to stay on, which also leads to increased energy consumption.

Design Point 1:

In Figure 5-9, at the low bit-rate of b kbps, the communication is just beginning to be possible for both the ISR and the STI-based threads as the bit-rate is just above the minimum bit-rate required to maintain the specified packet-rate. At this bit-rate, the CPU standby time is minimal and so is the amount of time that the RFM can go to sleep. This means that the CPU is required to be in active or idle mode for a long time. This leads to higher consumption of energy by the node. Due to the low bit-rate, the overall time taken to transmit a packet (T') is large and so is the time between the execution of the last instruction by the CPU for the communication of a byte and the first instruction for the communication of the next byte in the packet by the node. This time is completely used up in the ISR-based thread for the execution of the DSP work at this bit-rate thereby forcing the CPU to stay in active mode for the entire duration of the communication. The STI-based thread, due to the elimination of interrupt overhead, is able to complete the execution of the DSP work early and uses the remaining idle time to go to idle mode in order to save energy. Due to this, the energy used by the STI-based thread is much lower than that used by the ISR-based thread.

Design Point 2:

At the bit-rate of 2*b kbps, the RFM now completes transmission in half the time that it took for b kbps. This allows the RFM to go to sleep earlier and thus conserve energy. The increase in bit-rate also leads to the decrease in the time (by half) between the execution of the last instruction by the CPU for the communication of a byte and the first instruction for the communication of the next byte in the packet by the node. Hence the instructions for the DSP work that finished within the communication period for the ISR-based thread earlier now may not do so and may have to be executed after the communication has ceased. The ISR-based thread in fact just finishes its communication work by the end of the communication period and is unable to perform any DSP work during that time and hence has to postpone it for after the communication period. This increases the time that the CPU is required to be in active mode. The STI-based thread in this case is just able to finish its work by the time the communication is completed. Here the CPU is able to go to extended standby mode and the RFM is able to go to sleep as soon as the communication is complete which is much earlier than is possible for the ISR-based thread due to its large interrupt overhead. This leads to a

large savings in energy for the STI-based thread as the CPU is able to go to extended standby mode much earlier.

Design Point 3:

At the bit-rate of 4*b kbps, the overhead of interrupts does not permit the ISR-based thread to execute since the execution of the communication instructions would take longer than the required communication period for that bit-rate. Only the STI-based thread is able to run here. Here, the time taken for communication is lower than that for the total work. The high bit-rate ensures that the RFM can go to sleep earlier thus saving energy. The CPU can go to extended standby mode only after the completion of execution of all the DSP work.

Here, the progression of events show that as the bit-rate increases the RFM is able to turn off earlier and save energy. It also shows that, as bit-rate increases, CPU has to wait for lesser and lesser amounts of time for the communication to finish until a point is reached where the communication actually finishes before the computation does. After this point, there are no more savings that can be gained from the CPU by increasing bit-rate, since the CPU does not have to wait for the RFM anymore and can go to extended standby mode as soon as it finishes processing expeditiously. All the savings hereafter come from the RFM turning off sooner due to increasing bit-rate.

Hence, as bit-rate increases, there is considerable energy savings from turning off the RFM earlier. Higher bit-rates are not possible with ISR-based schemes; hence the amount of energy that can be saved by turning the RFM off early is limited. The STI-based schemes are capable of much higher data-rates and hence can turn the RFM off much earlier and save more energy. Both the implementations, however, are subject to a lower limit of CPU energy consumption due to the fact that the time taken by the CPU to complete its work is constant as mentioned earlier.

The total time that the CPU needs to be active/idle is given by:

$T = Maximum\ (T', T_{work})$, *where* T_{work} = *CPU cycles for DSP and communication/ Frequency of the cpu*

Equation 5-5: Total CPU active/idle time

If T= T_{work}, the CPU has to be in active mode for the entire duration, but if T=T', the CPU can move to idle mode for the duration of T- T_{work} in order to save energy. At higher bit-rates, T' becomes so small that T= T_{work} and at lower bit-rates T' can be greater than T_{work} and then T=T'.

The charts below demonstrate the behavior of the energy consumption of the node with increase in bit-rate:

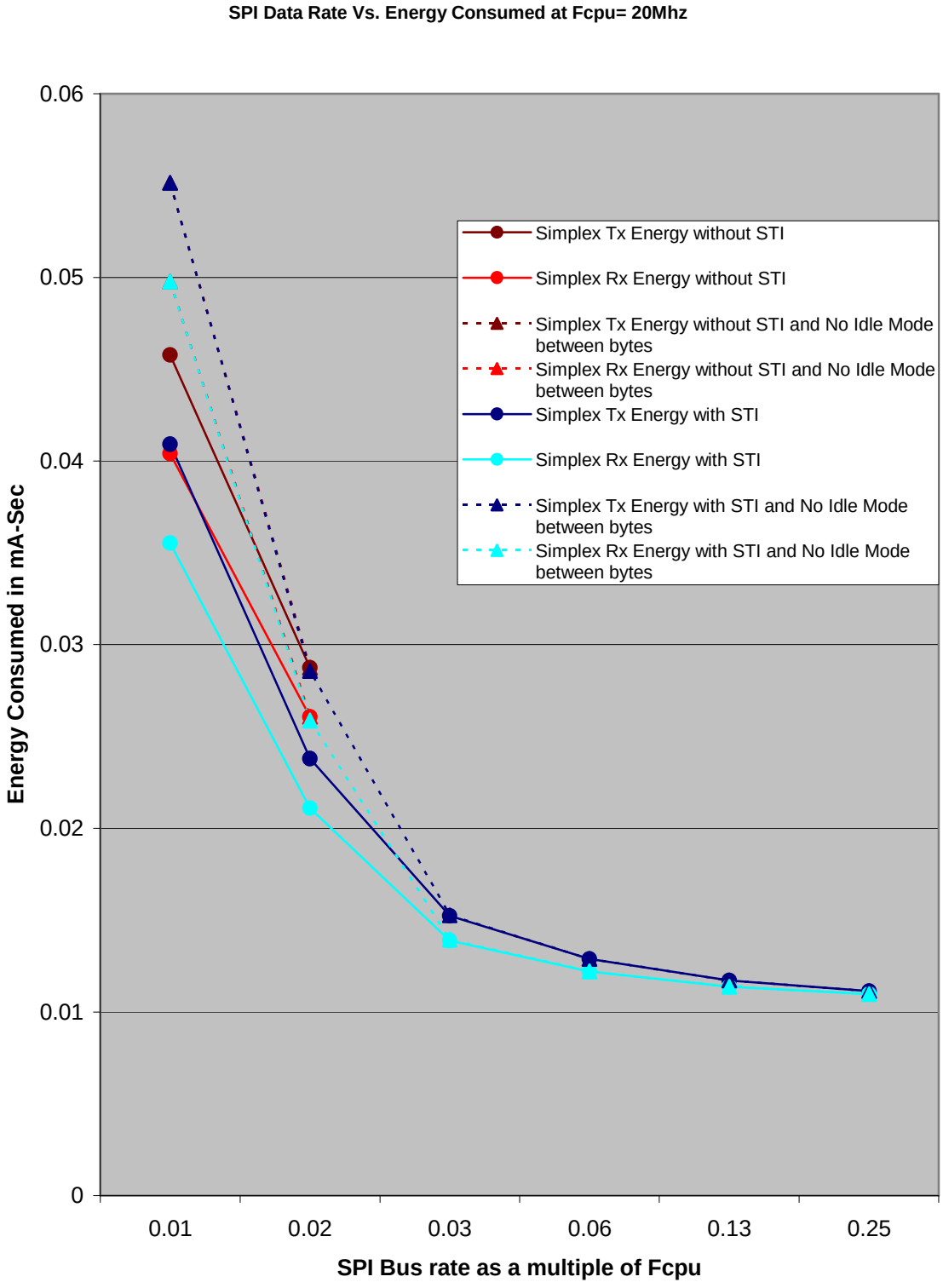

Figure 5-10: Variation in Energy consumption with Data Rate of communication at a clock frequency of 20 MHz

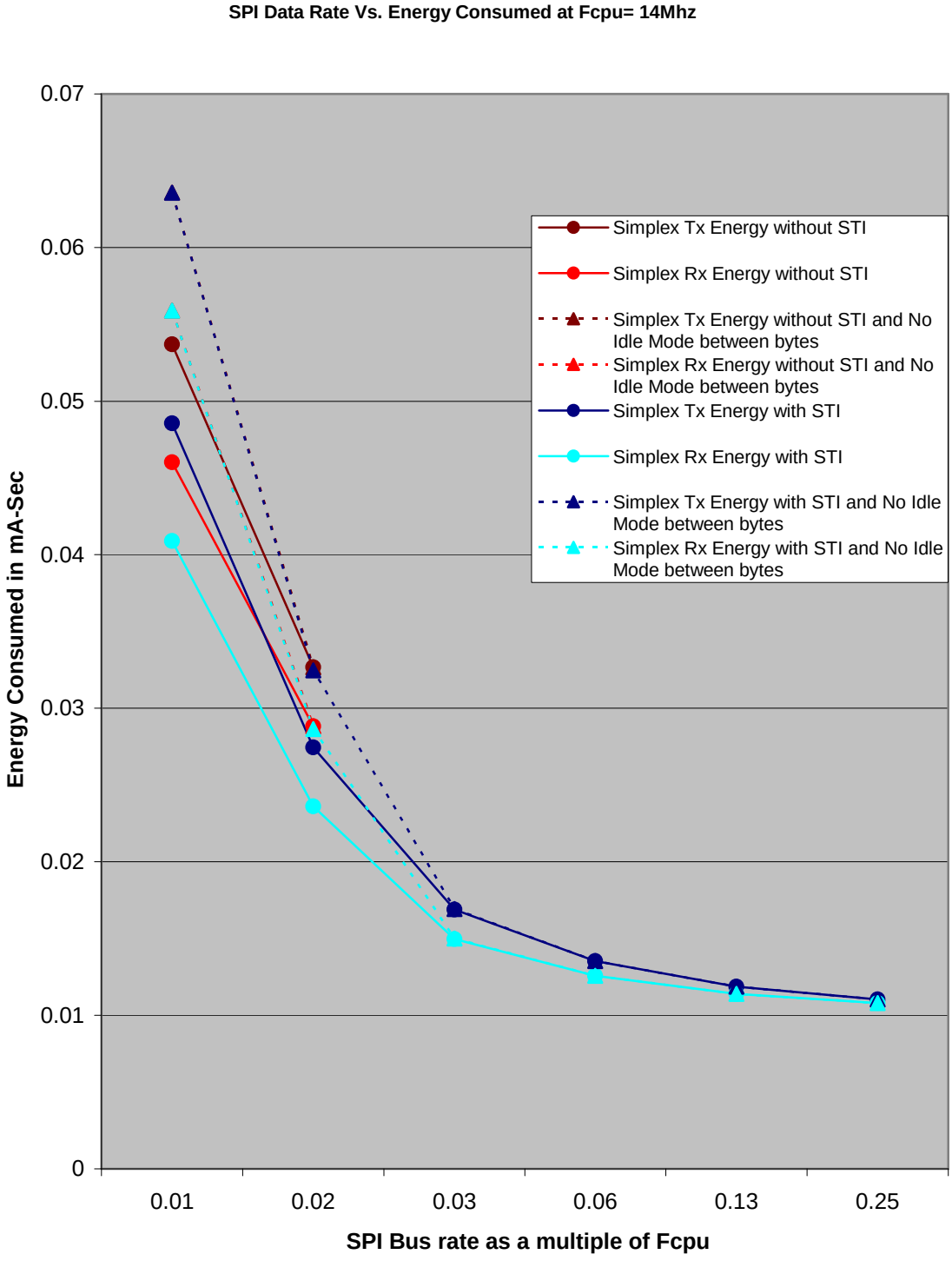

Figure 5-11: Variation in Energy consumption with Data Rate of communication at a clock frequency of 14 MHz

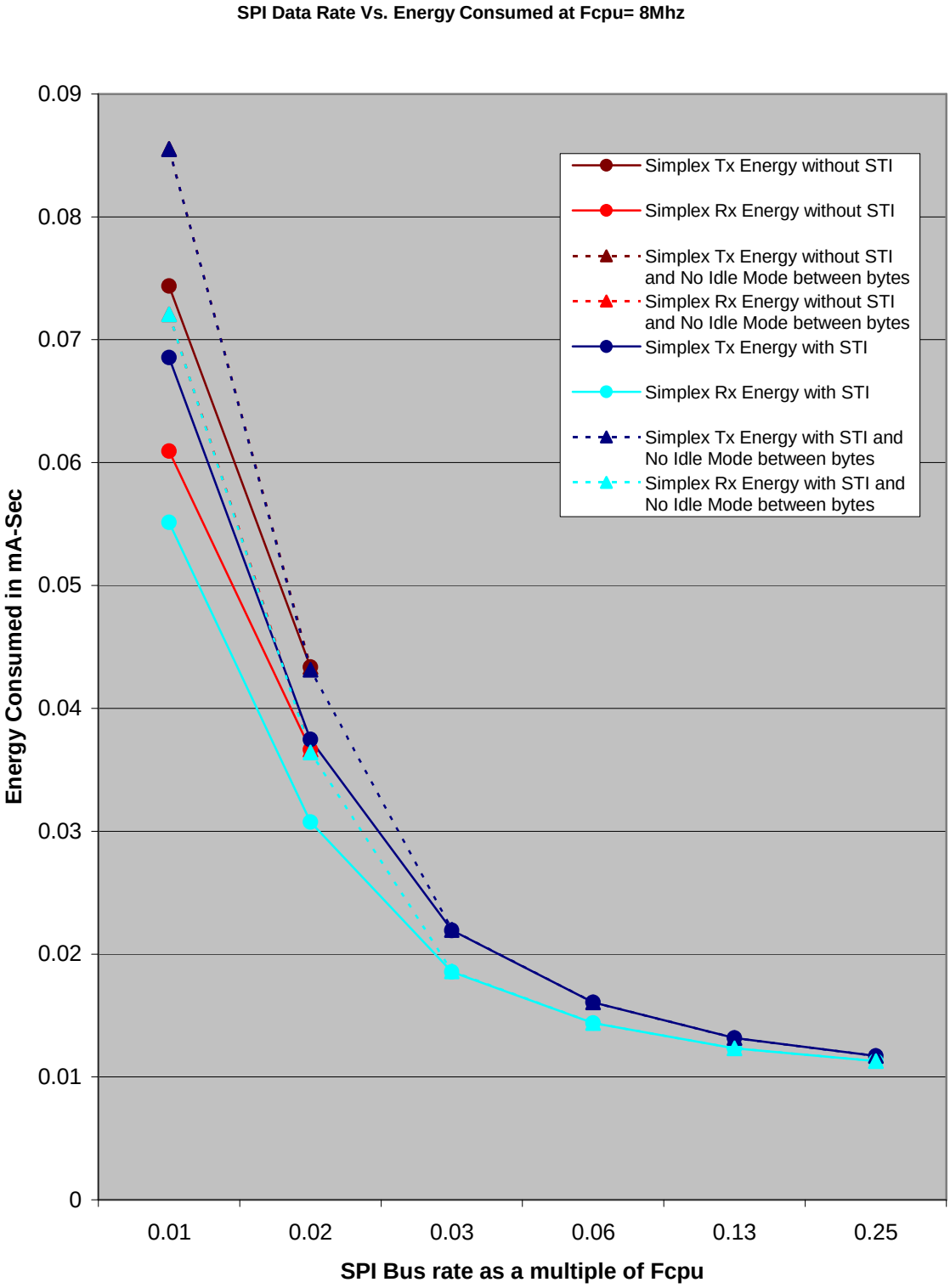

Figure 5-12: Variation in Energy consumption with Data Rate of communication at a clock frequency of 8 MHz

In Figure 5-10, Figure 5-11 and Figure 5-12, it is seen that the energy consumption falls sharply with the increase in bit-rate initially and then begins to level off later. At low bit-rates, the standby time is low and the CPU has to stay in either active or idle mode for a long period of time. If the CPU is one that does not have an idle sleep mode, then it has to stay in the active mode for the entire duration of communication since the communication takes much longer than the computation. Such a CPU will consume more energy than one with idle mode for all the cases where the communication takes longer than the computation and the CPU is forced to wait for the communication to complete so that it can go into extended standby mode. However, in cases where the computation time is greater than the communication time, the CPU is unable to go into idle mode and hence the energy consumption curves of the two kinds of CPUs are identical.

With the doubling of the bit-rate, the RFM is able to turn off at half the time since the communication takes half the previous amount of time. This has the additional effect that CPU needs to wait for much lesser time for the completion of the communication. The duration that the RFM needs to stay on with increase in bit-rate is given by the following equation:

$T_{rfm\_new} \alpha (1/(2^n))*T_{rfm\_MAX}$, *where n=current bit rate/least bit rate and* T_{rfm_MAX} *is the maximum amount of time that the RFM needs to stay on, which occurs at the least bit rate.*

$T_{rfm\_new} = k*(1/(2^n))*T_{rfm\_MAX}$, *where k= proportionality constant.*

Equation 5-6: Relation between RFM active time and bit-rate

The energy consumed is directly proportional to the time that the RFM needs to stay on and the duration that the CPU needs to be in active/idle mode. The duration that the CPU needs to be in active/idle mode is directly dependent on the time that the RFM needs to stay on for the bit-rates at which the computation time is lower than the communication time. Hence, the energy consumption may be shown as follows:

$E_{new} \alpha k1*T_{rfm\_new} + k2* T_{cpu\_new}$, where T_{cpu_new}=time taken by the CPU at the increased bit-rate, k1, k2 are constants that depend on the current consumptions of the RFM and CPU respectively at the chosen frequency and transmission power.

Equation 5-7: Relation between energy and active time for CPU and RFM

Hence, the E_{new} initially decreases sharply in an inverse-exponential manner due to the exponential decrease in T_{rfm_new} and T_{cpu_new}, which is directly proportional to the value of T_{rfm_new} when $T_{rfm\_new} >= T_{cpu\_new}$. When $T_{rfm\_new} < T_{cpu\_new}$, the value of T_{cpu_new} becomes independent of the value of T_{rfm_new} as explained earlier. This leads to a lower limit, beyond which the E_{new} can not decrease. The relation between E_{new} and the bit-rate follows an inverse-exponential curve throughout due to its dependency on the value of T_{rfm_new} as can be observed in the graphs.

Another observation from these graphs is that the maximum energy consumption seems to decrease with increase in frequency, which is different from that which is seen in the previous set of results. This illusion is due to the fact that the bit-rates on the X-axis of the graphs shows the bit-rates as a multiple of the clock frequency and the value of 0.01 for the SPI bus-rate is actually a much higher bit-rate for f_{cpu}=20 MHz than for f_{cpu}=8 MHz. We have seen that the energy consumed drops inverse-exponentially with the increase in bit-rate, which is the same phenomenon that can be seen here.

The amount of energy consumed by the STI-based thread is always lower than that consumed by the ISR-based thread due to the larger number of cycles that the ISR-based threads need to execute to achieve the same result as the STI-based thread.

It is also obvious from these graphs that the STI-based system is capable of a much higher communication rate than the ISR-based thread and this can be used to turn the RFM off much earlier and achieve very substantial savings in energy.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this thesis we have seen that it is possible with STI to cause a significant reduction in energy consumption of a sensor node when compared to that of the current implementations using ISRs. STI allows for communication at high bit-rates and thus low energy consumption due to the RFM turning off early. This when taken along with the elimination of interrupt overhead leads to further savings in energy. STI makes it possible for a node to produce the same results as an ISR-based version while running at a much lower frequency due to the elimination of interrupt overhead. The node consumes much less energy when running at lower frequencies. Since both the idle and active currents of the CPU increase with the increase in clock frequency, as is shown in the graphs (Clock Frequency vs. Energy Consumed), there is a point where the node consumes the least amount of energy. The energy consumption of the node before and after this point is higher than it is at this point. This point manifests itself when the computation time fits exactly within the idle time of the communication; not more and not less. This point is much lower for the STI-based threads than the ISR-based threads and this optimum clock frequency is what the nodes should attempt to run at.

Also observable is that the energy consumption decreases with increase in bit-rate (SPI Bus Rate vs. Energy Consumed) and that this decrease follows an exponential curve where the percentage of decrease in energy consumption reduces as the frequency continues to increase. The knee of this curve where the sharp drop ends is the optimal bit-rate.

Thus, it is possible to choose a combination of the optimum frequency for this optimum bit-rate at which the node consumes the least amount of energy. This point is much lower for the STI-based nodes than the ISR-based nodes. Also, the difference between the energy consumptions of the ISR-based and STI-based nodes increases even further if a higher packet

size is chosen since the amount of ISR overhead increases with the number of bytes in the packet.

6.2 Future Work

The function of a distributed wireless sensor node comprises of many tasks, not the least of which are routing updates, self-reconfiguration related tasks like election of cluster heads, collaborative signal processing, which reduces the signal processing load on a single sensor, periodic sampling of various sensors like acoustic, seismic, infrared, still/motion video-camera, etc., periodic communication of task results and more. Of these many of them are real-time tasks such as sensor sampling and others are non-real time tasks like processing routing updates which can be queued and processed. Such a combination of real-time and non-real-time is conducive to the use of STI. STI may be applied to as many of these tasks as possible and the energy consumption of such a completely STI-based node should be evaluated with that of an ISR-based node and the energy savings be recorded. This evaluation will give the complete picture of the amount of energy that can be saved with STI and the increase in the longevity of the node and the sensor network due to STI. The results and graphs presented in this thesis may be used as a basis to envisage the outcome of and to plan this experiment.

REFERENCES

[1] G. Asada, M. Dong, T. Lin, F. Newberg, G. Pottie, W. Kaiser, and H. Marcy, "Wireless Integrated Network Sensors: Low Power Systems on a Chip". *In Proceedings of the European Solid State Circuits Conference*, 1998.

[2] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next century challenges: Scalable coordination in sensor networks". *In Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 263-270, 1999.

[3] J. Kahn, R. Katz, and K. Pister, "Next century challenges: mobile networking for Smart Dust". *In Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 271-278, 1999.

[4] M. Bhardwaj, T. Garnett, and A.P. Chandrakasan, "Upper Bounds on the Lifetime of Sensor Networks". *In Proceedings of International Conference on Communications*, 2001.

[5] A. Chandrakasan et al, "Design Considerations for Distributed Microsensor Systems". *In Proceedings of IEEE Custom Integrated Circuits Conference*, 1999.

[6] C. Shen, C. Srisathapornphat, and C. Jaikeo, "Sensor information networking architecture and applications". *IEEE Personal Communications*, 8(4):52–59, August 2001.

[7] Y. Yu, R. Govindan, and D. Estrin, "Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks". *University of California at Los Angeles Computer Science Department, Tech. Rep. UCLA/CSD-TR-01-0023*, May 2001.

[8] W. Rabiner Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocols for wireless microsensor networks". *In Proceedings of HICSS*, Maui, Hawaii, January 2000.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, "System Architecture Directions for Networked Sensors". *ASPLOS* 2000.

[10] A. Sinha, and A. Chandrakasan, "Dynamic power management in wireless sensor networks". *IEEE Design and Test of Computers 18, 2.* Mar. 2001.

[11] V. Raghunathan, C. Schurgers, S. Park, and M. B. Srivastava, "Energy-aware wireless microsensor networks". *IEEE Signal Processing Magazine*, 19(2):40–50, 2002.

[12] M. Bhardwaj, R. Min, and A. Chandrakasan, "Power-aware systems". *In Proceedings of 34th Asilomar Conference on Signals, Systems and Computers*, November 2000.

[13] J. Karn, R. Katz, and K. Pister, "Next century challenges: Mobile networking for smart dust". *In Proceedings of ACM MobiCom'99,* Aug. 1999, pp. 271-28.

[14] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next century challenges: Scalable coordination in sensor networks". *In Proceedings of ACM MobiCom'99,* Aug. 1999, pp. 263-270.

[15] R. Min, M. Bhardwaj, S. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan, "Low-Power Wireless Sensor Networks". *In VLSI Design*, 2001.

[16] R. Powers, "Advances and trends in primary and small secondary batteries". *IEEE Aerospace and Electronic Systems Magazine, vol. 9, pp. 32-36*, Apr. 1994.

[17] R. Hahn and H. Reichl, "Batteries and power supplies for wearable and ubiquitous computing". *In Proceedings of the 3rd International Symposium on Wearable Computers*, 1999, pp. 168-169.

[18] A. Wang and A. Chandrakasan, "Energy-Efficient DSPs for Wireless Sensor Networks". *In IEEE Signal Processing Magazine, vol. 43(5), pp. 68–78*, July 2002.

[19] A. Dean, "Software thread integration for Hardware to Software Migration". *PhD. Dissertation, Carnegie Mellon University*, May 2000.

[20] A. Dean, R. Grzybowski, "A High-Temperature Embedded Network Interface Using Software Thread Integration". *Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)* October 1-3, 1999, Washington, D.C.

[21] Atmel, Appl. Note AVR223.

[22] A. Dean, J. P. Shen, "Hardware to Software Migration with Real-Time Thread Integration". *EuroMicro Workshop on Digital System Design, Vasteras, Sweden,* August 25-27, 1998

[23] J. Hill, "A Software Architecture Supporting Networked Sensors". *Masters thesis, University of California at Berkeley*, December 2000.

[24] Atmel, Atmega128 Datasheet, 2003.

[25] Chipcon, CC1000 Datasheet, 2002.

[26] N. Kumar, S. Shivshankar and A. Dean, "Asynchronous Software Thread Integration for Efficient Software Implementations of Embedded Communication Protocol Controllers". *North Carolina State University, Raleigh, NC, Technical Report*, 2003.

[27] P. Ganesan, "Efficiently Adding Secure Communications to Networked Low-End Embedded Systems using Software Thread Integration". *Masters thesis, North Carolina State University,* June 2003.