SpotCore: A Power-Efficient Embedded Processor for Intelligent Sensor Networks

Mbou Eyole-Monono
University of Cambridge
Computer Laboratory
15 JJ Thomson Avenue
Cambridge CB3 0FD, UK
me265@cam.ac.uk

Robert Harle
University of Cambridge
Computer Laboratory
15 JJ Thomson Avenue
Cambridge CB3 0FD, UK
rkh23@cam.ac.uk

Andrew Rose
ARM Ltd.
Processor Division
110 Fulbourn Road
Cambridge CB1 9NJ, UK
Andrew.Rose@arm.com

ABSTRACT

Sensor platforms designed with mobility in mind, such as body networks, have inherent scalability problems arising from the conflicting demand for high processing capabilities (to collect, compress, and filter data) and the need for low-power, resource-constrained hardware. This paper presents a CPU design which seeks to optimize processing for a sensor network by improving performance in a power-efficient and scalable manner. We demonstrate the crucial design decisions and trade-offs required in developing such a processing platform and demonstrate that a minimalist design saves power without adverse impact on performance. In addition, we address the problem of scalability in a multi-threaded environment through the development of a novel scheduling algorithm implemented directly in hardware.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*pipeline processors, RISC architectures*

General Terms

Measurement, Performance, Design, Experimentation, Languages

Keywords

Low-power, Sensor networks, Microprocessors, Embedded systems, Threads, Schedulers

1. INTRODUCTION

In recent years, advances in wireless communications have fostered the notion of many independent sensors distributed throughout an environment or across an object, continually sensing and reacting to the current state. Beyond reliable communication, the scenario of body sensor nets offers many challenges: powering the sensor nodes, processing information efficiently within the network, and bringing costs down. Sensing inside or around a human body, for example, requires tiny nodes (aesthetics) connected invisibly (so as not to restrict motion) with low power draw (so we can use smaller, lighter batteries that need changing less frequently or some form of energy-scavenging). In fact, power is often the major barrier: it limits communications range, operating lifetime, and processing capabilities.

This paper presents a novel approach to enhancing the computational capabilities of sensor networks without the penalty of high power consumption. Currently, any high-performance back-end processing required in sensor networks is often offloaded to desktop or server clusters as discussed in [21]. However, this scenario suffers from at least five drawbacks:

1. Compute-intensive applications cannot run within the sensor network in order to improve the quality of the data by adapting the sensor parameters in real-time.
2. High-end computing systems may be remote from the sensor network and the bandwidth required to transport data to and from the cluster will be very large.
3. A connection to a high-performance computer cluster would not always be available for certain deployments of sensor networks.
4. Server clusters are currently not power-efficient and the latency may be unpredictable.
5. As sensor systems scale in size and complexity the energy efficiency of a distributed computation will be far from optimal if individual processing elements are not power-efficient.

Significant advances in body nets can be achieved by processing biomedical data locally within the body sensor network using processing units designed with such extremely power-constrained networks in mind. The processor described herein, named *SpotCore*, is a small, fast, and open-source CPU core designed with emphasis on power efficiency, flexibility and scalability, which it is hoped will stimulate research into high-performance processing within sensor networks. It is designed in the Verilog hardware description language and is purely synthesisable.

The rest of this paper is organised as follows. Section 2 summarises some important research into the reduction of

power consumption. Section 3 covers the important architectural points which directed the development of SpotCore and explains the crucial combination of features which make it unique among the plethora of RISC processors available. Section 4 examines the instruction set in detail. Some integrated circuit synthesis and experimental results are presented in Section 5. Section 6 is concerned with optimising scheduling within a sensor-driven computational platform, aiming to make this as lightweight as possible without sacrificing robustness. We conclude in Section 7.

2. RELATED WORK

Nazhandali [16] notes that designing energy-efficient sensor processors is a fairly recent undertaking and describes how an ultra-low energy processor may be designed by combining optimisations at the microarchitectural and instruction set levels, with subthreshold voltage circuits [17]. These circuits often involve significantly lower clock frequencies[22] and are thus successful in reducing power for applications that do not require a high throughput. However, this limitation is too great in the general case.

Many sensor network platforms have used off-the-shelf components which were not primarily designed for the strict ultra-low power environments they are then embedded within. Virantha et al. [6] present a novel architecture based on an asynchronous 16-bit RISC core. By using asynchronous design techniques this processor, known as SNAP/LE, can reduce its power consumption because not all parts of the circuit are actively changing state, and there is no power-hungry clock-tree. To simplify verification (often a problem in asynchronous circuits) they adopt a quasi delay-insensitive design approach. Their reported worst-case energy consumption figure is 300pJ/instruction and they note that this stands out favourably when compared to approximately 1500pJ/instruction for an off-the-shelf Atmel microcontroller. The design principles of SNAP/LE differ from those of SpotCore since the latter retains a synchronous design methodology (the most viable route for integrated circuit synthesis) and focusses instead on optimising the instruction set and processor architecture. The researchers in [13] use a low-power compilation methodology to save energy within a wireless sensor network by making optimisations at the microprocessor instruction execution level.

The SNAP/LE project also shows how the execution time of a given task can be reduced relative to an Atmel microcontroller running TinyOS on a Berkeley MICA mote, by using a scheduler implemented in hardware and tightly-coupled to the processor. This technique yields significant power improvements, and is adopted in SpotCore. However, we present a more scalable hardware-based scheduler which is capable of supporting not only event-driven execution but true multi-threading, and which achieves a better degree of fairness than the simple non-pre-emptive FIFO-based scheduler used in SNAP/LE. Mota et al. [15] also take a hardware-oriented approach and show that they can improve the information-processing capability of sensor network nodes by re-implementing tasks as hardware modules.

Warneke et al. [23] produced a design which improves power-efficiency by having separate hardware subsystems which can be shutdown independently, elaborate clock-gating, and guarded ALU inputs. However, the design uses no datapath pipelining in a bid to avoid the associated hardware overhead. This in turn limits the maximum clock frequency. However, the designers note that the platform known as "Smart Dust" will be used in low data rate scenarios where high clock frequencies are not normally needed. At 500kHz and 1V, the design utilises 12pJ/instruction. With the possibility of collaborative processing between sensor platforms and the high level of interest in in-network processing, much higher levels of performance may be required and hardware limitations on the design speed are inadvisable. The instantaneous power might be reduced at lower frequencies (and voltage) but the overall energy consumption might be worse if the execution time is not also reduced through careful instruction set design.

Ciaran et al. [14] present a survey of different processor architectures for wireless sensor networks and observe that current microprocessors have limited capabilities for handling complex data-processing tasks. The Texas Instruments MSP430 [9] emerged as the best architecture in the survey, with the smallest power consumption figures compared to the Atmel ATMega128L and the Microchip PIC18.

The i-Bean [19] uses dual processors clocked at different speeds to improve power efficiency. The recently announced Imote2 [3] from Crossbow technology uses a high-performance, low-power 32-bit PXA271 XScale processor and is capable of dynamic voltage and frequency scaling from 13MHz to 416MHz. The platform also incorporates a DSP coprocessor to accelerate multimedia operations by extending the XScale instruction set. Preliminary data suggests that, with the radio circuitry off, the rest of the chip comprising the CPU and memory consume about 2mW/MHz. It is an interesting fact that on this platform, which is touted as the most power-efficient sensor platform, the processing elements consume as much as 40% of the overall power consumption when the radio circuitry is on; indicating that research into more power-efficient cores is at least as important as research into low-power communication interfaces in the quest to reduce the overall power consumption of sensor platforms.

3. THE SPOTCORE ARCHITECTURE

The design of SpotCore is primarily motivated by the desire to integrate as much essential functionality as possible into a single core whilst taking great care in the instruction set and processor design to avoid the introduction of redundant hardware. There are many optimisations which can be applied to the basic RISC pipeline (see [7]) but it is important to identify a set of reliable optimisations which would still yield a reasonable performance from a highly minimalist design philosophy. While more pipeline stages will enable the design to be clocked at higher frequencies, by putting less work or logic in each stage, this adds complexity, increases hardware size and worsens the branch or exception penalty. We observe that processing in sensor networks is of a highly concurrent nature as there might be multiple data streams requiring analysis. We envisage that as these networks scale, this parallelism is going to increase dramatically. This leads to an increased probability of many context-switches so it is desirable to keep as little state internal to the processor as possible (but relevant to any given thread). In addition, SpotCore is being designed for low-

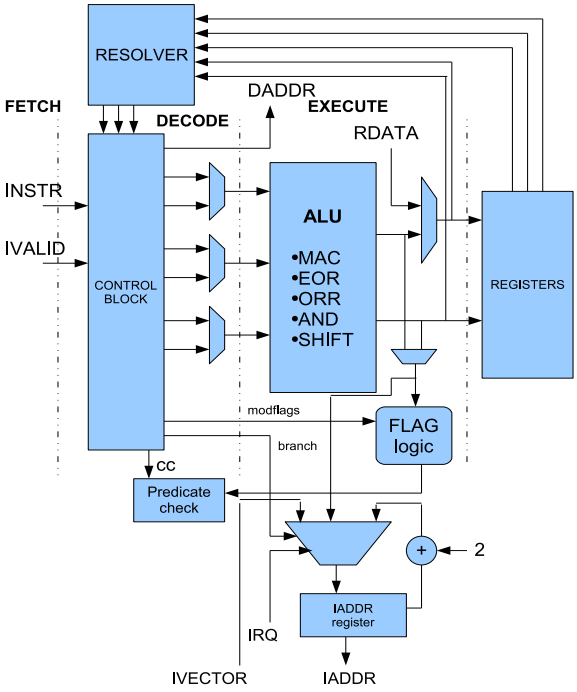

Figure 1: The internal structure of SpotCore

power environments where extremely high clock frequencies in the gigahertz range are not feasible due to the substantial increase in power requirements. The critical path length constraint can be relaxed as a result. Deeply-pipelined processors will typically need a great deal of logic to transfer information contained in instructions still in the pipeline to preceding instructions.

Using an instruction width of just 16 bits instead of 32 bits or higher reaps power savings by reducing the bandwidth requirement of instruction memory and may additionally lead to high code density. The datapath of SpotCore (comprising registers, internal buses and functional units) is 32 bits wide. The important parts of the internal structure and datapath are shown in Figure 1.

The instructions are split into different classes depending on whether they are dyadic, monadic or require no operands. This enables us to attain a highly orthogonal instruction set design which makes maximum use of the available encoding space. This encoding scheme is illustrated in detail later. Most of the data processing instructions (Add, Subtract, Multiply, AND etc) on SpotCore are dyadic but due to the restrictions on instruction length only register contents may be used as operands. This is in contrast to the plurality of addressing modes used on an ARM processor [2] and it greatly simplifies the addressing scheme leading to more compact decode logic. In addition, register lookup uses indices specified in fixed parts of an instruction in order to further simplify decode.

All SpotCore instructions are conditional — the predicated instruction scheme has been found to be successful in ARM processors [2] and Intel's IA-64 architecture [11] by reducing the number of branches. However, in order to save encoding space we mandate that a conditional instruction cannot also be "flag-modifying". This means we can use just one field to specify whether an instruction sets or clears the flags or is itself conditional on some flags set previously within the processor. This field is restricted to 3 bits. The trade-off arose from observing many instruction traces of code compiled for an ARM processor and not finding many instances of conditional instructions which modified flags. The combination of this 3-bit field and a 4-bit primary opcode field leaves only 9 bits for encoding the registers being accessed. This in turn restricts the number of visible and directly addressable registers to eight.

Another power-saving measure seeks to reduce the number of register ports — two read ports are adequate if we use only simple dyadic instructions. However, we eventually incorporated three read ports and two write ports in order to be able to support certain very useful instructions:

STR r0,[r1],r2 => Store the value in r0 at the address pointed to by the value in r1 and update the register r1 with the sum of the values in r1 and r2.

MLA r0,r1,r2 => Place X + Y*Z in register r0, where X,Y, and Z are the values in r0,r1 and r2 respectively.

LDR r0,[r2],r3 => Load r0 with the value at the address pointed to by the value in r2 and update the register r2 with the sum of the values in r2 and r3.

SORT r0,r1 => Swaps data values so that the register with the higher index contains the higher value.

These instructions and a few others necessitate either three simultaneous reads from or two simultaneous writes into the register file. However, the presence of two write ports creates an opportunity to improve the execute stage by arranging the logic so its more critical pathways feed into a less heavily-loaded write port and thereby creating more balanced timing in that pipeline stage.

SpotCore has a smaller register file than most embedded RISC processors and no banked registers. In order to save time on exception-entry, register-file stacking is managed by hardware. Register 7 is the program counter and the stack pointer is internal. SpotCore also maintains an internal link register which is saved automatically when nested subroutines are detected. This increases the number of general purpose registers available and also saves time since it can be stacked in parallel with branching. A separate instruction is provided to recover the link register value if necessary. We are currently researching more ways of reducing the performance impact of this small register file through advanced compilation techniques. SpotCore simply operates in one of two modes — *trusted* or *untrusted*, and avoids any expensive exception-handling scheme.

SpotCore has a branch penalty of 2 cycles. Due to the fairly mild impact of branching and exceptions on its short

pipeline, it was decided that the performance boost afforded by branch prediction in this case would not justify the extra hardware needed. However, in order to mitigate the impact of branching in the common scenario involving fixed branches at the end of iterative blocks of code such as in FOR loops, a LOOP instruction was added to the instruction set. The concept is similar to that used in the Intel x86 architecture [10] but our mechanism is different and the occurrence of nested loops is detected and handled automatically by the SpotCore hardware. The purpose of this instruction is to ensure that the process of checking for the last iteration at the end of the loop body can happen while the loop body itself is being executed so the pipeline can be filled with the correct set of instructions and the effect of the branch is hidden. As a result, an explicit branch at the end of the loop, and the penalty associated with it, are avoided. To illustrate this point the following ARM code sequence A can be rewritten as code sequence B on SpotCore.

```
;Code sequence A
             MOV r0,#10 ;set up loop counter
label        LDR r1,[r2],r3 ;loop starts here
             ;rest of loop body
             LDR r4,[r5],r3
             SUBS r0,r0,#1
             BNE label
             ;other instructions


;Code sequence B
             LOOP r1,r0
             MOV r0,#10 ;set up loop counter
             ;set up loop end address
             MOV r1,#end_address
             LDR r1,[r2],r3 ;loop starts here
             ;rest of loop body
end_address  LDR r4,[r5],r3
             ;other instructions
```

Notice that the registers used to set up the loop in the LOOP instruction can be re-used within the loop. In the case of a nested loop, the information held in the loop state machine in the CPU is written out to memory automatically as the new "loop state" is created in the two instructions following the a LOOP instruction. A special internal register holding a pointer to the base of the memory structure holding loop state information at various levels must be set up during initialisation.

4. INSTRUCTION SET DESIGN

The SpotCore instruction set is largely influenced by the many common RISC instruction sets. SpotCore attempts to include many of the most common instructions without violating our size and power constraints (Figure 2), while also including some special instructions for thread management. Many of our instructions would typically be found in digital signal processors.

Apart from the simple register addressing mode shown in the table, the 32-bit ARM has other addressing modes where the second operand could be an immediate value or even specified as the result of a shift operation. Since we did not want

SpotCore	ARM	TI MSP430	MIPS32
ADD rX,rY,rZ	ADD rX,rY,rZ	ADD src,dst	ADD rd, rs, rt
SUB rX,rY,rZ	SUB rX,rY,rZ	SUB src,dst	SUB rd, rs, rt
ORR rX,rY,rZ	ORR rX,rY,rZ	BIS src,dst	OR rd, rs, rt
AND rX,rY,rZ	AND rX,rY,rZ	AND src,dst	AND rd, rs, rt
EOR rX,rY,rZ	EOR rX,rY,rZ	XOR src,dst	XOR rd,rs,rt
BIC rX,rY,rZ	BIC rX,rY,rZ	BIC src,dst	----------
MUL rX,rY,rZ	MUL rX,rY,rZ	----------	MUL rd,rs,rt
MLA rX,rY,rZ	MLA rX,rY,rZ	----------	MADD rd,rs,rt
LDR rX,[rY]	LDR rX,[rY]	----------	LW rt,offest(base)
STR rX,[rY]	STR rX,[rY]	----------	SW rt,offest(base)
LDR rX,[rY],rZ	LDR rX,[rY],rZ	----------	LWXC1
STR rX,[rY],rZ	STR rX,[rY],rZ	----------	SWXC1
MOV rX,#N	MOV rX,#N	----------	----------
B or BL label	B or BL label	JMP label	B offset
BL rX	BLX rX	----------	JALR rX
LSL or LSR rX,#Sh	MOV r0,r1,LSR #Sh	----------	SLL / SLR
PUSH {r0-r5}	STM sp!,{r0-r12}	PUSH src	----------
POP {r0-r5}	LDM sp!,{r0-r12}	POP dst	----------
RETURN {r0-r5}	LDM sp!,{r0-r12,pc}	RET	----------
LSL rX,rY	MOV r0,r1,LSR rX	(rotation)	SLLV
LSR rX,rY	MOV r0,r1,LSR rX	(rotation)	SLRV
INV rX,rY	MVN rX,rY	INV dst	----------
INCR rX,rY	ADD rX,rX,#1	INC dst	ADDI rd, rs, IMM
DECR rX,rY	SUB rX,rX,#1	DEC dst	----------
ABS rX,rY	----------	----------	ABS fd,fs
SORT rX,rY	----------	----------	----------
NEG rX,rY	RSB rX,rY,#0	----------	NEG fd,fs
CLZ rX,rY	CLZ rX.rY	----------	CLZ rd,rs
BITREV rX,rY	----------	----------	----------
LOOP rX,rY	----------	----------	----------
Thread instructions	----------	----------	----------

Figure 2: Comparison of SpotCore, ARM, TI MSP430, and MIPS32 instruction sets

the shifter to be in the critical path and were limited in our available encoding space, we created separate instructions for getting immediate values into the processor and for shifting. Although this might represent a performance problem if these types of instructions are used frequently, we can however assert that the code size relative to a 32-bit instruction set is unaffected as the two operations simply become two 16-bit instructions.

The MSP430 is a 16-bit RISC CPU which lies at the heart of many sensor boards. It has 16 registers, 4 of which are treated specially — program counter, status register and constant generator which is particularly important because it provides six frequently used immediate values thereby reducing code size. However, unlike SpotCore and ARM not all MSP430 instructions are conditional. Only a few MIPS32 instructions are predicated despite having a 32-bit instruction set.

Following the implementation of an "ARM-like" instruction set, we were able to add some useful but uncommon single-cycle instructions without any drastic effect on operational parameters. These extra instructions are ABS (get absolute value), SORT (arrange values in registers based on their numeric size), BITREV (bit-reversal useful in Fast-Fourier Transform algorithm), the LOOP instruction described previously, and some thread management instructions described later. We also added the facility to load a relatively large literal from the instruction stream as data. Our move instruction (MOV) has a special bit which if set

will treat the next instruction as data, and append the last 5 bits of the move instruction to that. This means that besides the usual data memory access instructions we can either load a 5-bit value in one instruction or a 21-bit value in two instructions. In contrast, one cannot load an arbitrary 21-bit value within a single (32-bit) ARM instruction but have to encode the immediate operand as an 8-bit constant and a 4-bit (even-number) rotate which is applied to it.

The four main SpotCore instruction formats are shown in Figures 3, 4, 5 and 6, while Figures 7, 8, 9 and 10 show the exceptions to these standard formats. What we are trying to portray in these figures, is the fact that the width (number of bits), meaning, and placement of many sections of the instruction are kept as consistent as possible between instructions in a bid to simplify the decoding logic. Similarly, by splitting the instruction set into different classes depending on their requirements, with regards to the number of registers required for a particular operation, we achieve a compact layout which favours an efficient design. POP and PUSH instructions read from and write to the stack respectively. The RETURN instruction is similar to the POP instruction with the only difference being the fact that it also loads the PC with the preserved link register value. In these instructions, the bit field [5:0] is used to encode the set of registers which must be stacked to allow flexibility, akin to the ARM stack-manipulation instructions.

5. SYNTHESIS AND TEST CODE RESULTS

Since the energy usage of sensor applications is a product of power and time, it is important to reduce both the power consumption and the execution time of a set of instructions. We synthesised our Verilog design using Synopsys Design Compiler with a UMC 130nm technology library. The worst-case power estimate obtained was 0.03mW/MHz which looks auspicious compared to the TI MSP430 (0.4mW/MHz, CPU only). Admittedly the TI MSP430 is a complete System-On-Chip comprising memory and other peripherals (watchdog,timer,UART etc), but this power reduction is very significant as it is more than an order of magnitude smaller than the power consumed by the TI CPU when it is operating alone with all the peripherals powered down. We note that the more lightweight embedded ARM processors — ARM7TDMI and the ARM Cortex M3 have power figures of 0.06mW/MHz and 0.09mW/MHz (130nm technology and speed-optimised) respectively [1]. The Cortex-M3 [20] implements a new 16-bit variant of the ARM instruction set known as Thumb-2 which is capable of improving code density while maintaining a high level of performance. Figure 11 shows the code size and execution times of different processors running the same digital filter algorithm on 4000 input samples. The performance advantage of SpotCore in this IIR filter experiment is largely due to its ability to know precisely where branches within loops occur; and this improvement is significant for a large number of programs as loops are very common programming constructs. In addition, judging from Figure 2, about 70% of our instruction set is ARM compatible which is significant as there exists a wealth of reliable benchmarks for that instruction set. While many of our instructions are also similar to those in the TI MSP430 instruction set, we believe we gained a definite performance advantage because our core supports

[15:12]	[11:9]	[8:6]	[5:3]	[2:0]
opcode1	Predicate	rX	rY	rZ

Figure 3: Class 1 instruction

[15:12]	[11:9]	[8:6]	[5:3]	[2:0]
opcode1	Predicate	opcode2	rY	rZ

Figure 4: Class 2 instruction

[15:12]	[11:9]	[8:6]	[5:3]	[2:0]
opcode1	Predicate	opcode2	opcode3	rZ

Figure 5: Class 3 instruction

[15:12]	[11:9]	[8:6]	[5:3]	[2:0]
opcode1	Predicate	opcode2	opcode3	opcode4

Figure 6: Class 4 instruction

[15:12]	[11:9]	[8:6]	[5]	[4:0]
opcode1	Predicate	rX	direction	shiftAmount

Figure 7: Shift by Immediate

[15:12]	[11:9]	[8]	[7:0]
opcode1	Predicate	Link?	<signed offset>

Figure 8: Branch instruction

[15:12]	[11:9]	[8:6]	[5:0]
opcode1	Predicate	opcode2	r0—r5

Figure 9: PUSH, POP, RETURN

[15:12]	[11:9]	[8:6]	[5]	[4:0]
opcode1	Predicate	rX	Next?	Value

Figure 10: MOVE instruction

	Execution time (ms)	Code size (bytes)
SpotCore	20.2	50
ARM7TDMI	23.5	92
TI MSP430	38.5	95

Figure 11: IIR Filter Code results

dyadic instead of monadic data-processing instructions. In addition, the TI MSP430 does not support direct multiplication within the processor datapath but relies instead on a system peripheral which limits performance because a data access is required.

6. ROBUST ZERO-OVERHEAD SCHEDULING

In addition to building low-energy processor cores, optimising the manner in which threads are loaded and removed, and the associated scheduling scheme, maximises processor utilisation and improves power efficiency. An effective thread management strategy should also scale with the number of threads and processing elements. Due to the small physical area of SpotCore it is envisaged that it will be used not only in a multithreaded environment but alongside other cores in a multiprocessor; and in this case it is desirable to have transparent thread migration between cores. The central innovation in this section is the development of a thread management policy which runs directly in hardware without requiring any CPU time unlike conventional operating systems. This hardware module which we call *TopDog* shares a connection with the processor memory and interrupt interfaces, and can dispatch threads to the processor based on its internal scheduling algorithm. This module elevates the level of performance possible as the processor does not have to keep switching to some kind of supervisory mode in order to check the status of other threads. It also improves scalability in a system comprising multiple cores (Figure 12) by providing a common, fast arbitration mechanism. This is applicable in situations where a shared bus is feasible such as symmetric multiprocessors with up to about 16 cores.

In summary, the TopDog carries out the following tasks which are deemed to be crucial to efficient operation when many threads are present:

- Fast and clean creation, reloading, and switching of threads
- Implements a scheduling algorithm with fairness considerations from the ground up
- Synchronisation of threads
- Stores thread control blocks (TCB) for different threads and can modify each via simple instructions issued from the processor
- Holds interrupt vectors and priorities, and uses a common CPU access mechanism for interrupts and other threads

Rather than leaving the scheduling decisions entirely up to the operating system, the programmer can specify certain

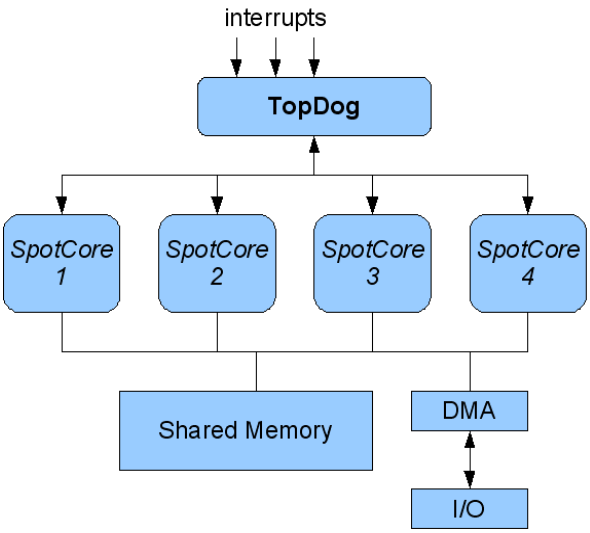

Figure 12: Processing Subsystem with TopDog Scheduler

parameters to the TopDog to enable it to achieve the right level of Quality-of-Service (QoS). This development was inspired by Nemesis [18] which is an OS designed to provide applications such as multimedia applications which are very time-sensitive with some form of QoS guarantees with respect to CPU and I/O resources. The scheduling decisions in TopDog are based on the following three parameters which achieve the appropriate balance between ease-of-use and robustness — PRIORITY, ON_TIME, and OFF_TIME.

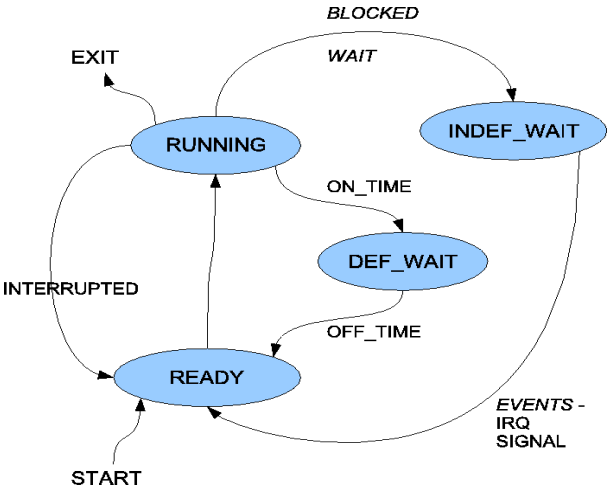

Figure 13: The TopDog scheduler state machine

These parameters relate to the thread state diagram in Figure 13 and their utility is explained as follows. At the most elementary level, if threads are of the same priority, they gain access to the processor core based on a First-Come-First-Served (FCFS or simply FIFO) scheme. There are 16 priority levels and the TopDog will remove lower priority threads from the processor so a higher priority one can run. Unfortunately, this might very easily lead to starvation of some threads if there are many high priority threads. As

a result we designed a system which allows the programmer to specify a maximum "ON_TIME" and a minimum "OFF_TIME" for each thread. Together with the priority value, they can be used to fine-tune performance because the priority value controls how quickly the thread gets to execute when it is in the READY state, the ON_TIME controls how long it is allowed to spend on a processor, and the OFF_TIME determines the delay between getting preempted at the end of its execution time, and being able returning to the READY state again. The scheme has enough flexibility to support a very broad range of CPU access schemes without resorting to a high-level thread library which will impact performance.

While the hardware does not guarantee that thread starvation will be avoided, it does provide the sensor system programmer with more scope for controlling thread scheduling than is currently available. The effect of the ON_TIME and OFF_TIME parameters is to ensure that the execution probability of any thread does not have a strong dependence on the number of threads of a higher or the same priority. One good policy for fairness would be to ensure that low priority threads have more t_{ON} and less t_{OFF}, while high priority threads have less t_{ON} and more t_{OFF}.

We shall now compare our scheduler with those commonly found in resource-constrained environments. MicroC/OS-II is a pre-emptive real-time kernel written in C, which runs on embedded processors such as the Motorola 68k, ARM7, and Altera Nios II. It supports dynamic priorities but no two tasks (threads) may have the same priority. The highest priority thread always runs but may be superseded by an interrupt service routine. Its main drawback is that a low priority thread can wait for an arbitrarily long period of time since the highest priority thread must run to completion or get blocked before it can run. It is believed that this approach does not scale well because fairness is not integral to the operating system and it is harder to give the lower priority thread any QoS as the execution times of many higher priority threads are indeterminate. The TopDog scheduler avoids this undesirable scenario by providing the ability to control the dominance of higher priority threads in a direct way. The uC/OS-II kernel code occupies about 2K bytes and can consume about 5% of CPU time. It can support up to 255 tasks. While the TopDog scheduler has fewer priority levels, it can support multiple threads of the same priority and can manage up to 512 threads.

TinyOS [8] is an open-source embedded operating system developed at University of California Berkeley and is very popular among developers of applications for Wireless Sensor Networks (e.g. [12]). It operates on many different platforms, speeds development, and is useful for testing research ideas. It is written in nesC which is a C-like structured component-based language. In addition to the standard functions of task scheduling and interrupt handling, it also performs encryption and power management. The major system components include drivers for the radio interface, UART, memory and timer. Other components include drivers for the LED interface, an I^2C protocol implementation and a CRC packet filter. It has an event-driven architecture which is sufficiently abstract to enable the creation of cross-platform applications while remaining lightweight. The main structural elements are configurations which connect components, modules which define how components behave by implementing commands and event-handlers, and interfaces which define the interaction between any two components. Unfortunately, TinyOS provides only an elementary concurrency model with limited operating system support for a large number of threads or a platform with more than one processor. It has no inherent ability to specify multiple priority levels. The two main system threads comprise tasks and hardware event-handlers respectively. Tasks must run to completion and cannot preempt other tasks while they may be preempted by hardware interrupts.

The SpotCore approach differs from this by allowing threads in our system to pre-empt other threads regularly and by allowing the programmer to specify QoS constraints in an explicit manner. Contiki [4] builds on the event-driven model by utilising "protothreads" [5] — lightweight threads which can facilitate multithreading. Since each protothread does not need its own stack, protothreads have been promoted as ideal for memory-constrained systems. The conditional blocking wait abstraction provided by protothreads avoids the complexity involved in dealing with explicit state machines which is common when developing software for event-driven systems. The TopDog gives threads the ability to block until a condition variable becomes true or an external event of interest (interrupt) occurs. We argue that our approach is more scalable as there is no overhead in terms of CPU time or code size and we can also reap the benefits of hardware acceleration of the scheduling algorithm.

Rather than communicating with the TopDog module as a memory-mapped peripheral on the system bus, we created special instructions to speed-up access and promote flexibility with different memory architectures as no pointers have to be calculated.

The SpotCore instructions which are used to communicate with the TopDog are:

- SET_VECTOR — this sets the address from which the new thread will start executing
- SET_PRIORITY — this specifies a 4-bit priority value for the thread being created
- SET_ONTIME — sets the parameter t_{ON} (10-bit value)
- SET_OFFTIME — sets the parameter t_{OFF} (10-bit value)
- FORK — activates the thread state machine
- EXIT — removes all active references to the exiting thread
- SIGNAL — decrements the specified variable held in one of the TopDog memory banks (this behaves in a manner similar to a conventional semaphore in that each update is atomic and an event is generated when the count value reaches zero)
- WAIT — blocks or suspends a thread until a specified event such as a semaphore value reaching zero or an external interrupt occurs.

- CHECK — similar to wait but non-blocking
- SET_SIGNAL — initialise a given semaphore

The state diagram shown in Figure 13 is implemented by multiple memory blocks and a few associated logic controllers. The total memory required to manage 512 threads with 16 priority levels, and 64 signals is about 720 bytes. The controllers were simple enough so that the hardware footprint of the TopDog module is barely over 2000 gates. Assuming a clock frequency of 10MHz, the worst-case latency between one thread sending a signal to the TopDog, and another one becoming dispatched after the TopDog has updated its READY queue due to the signal and determined the thread which now has the highest priority, is 2 μs. The key to this fast inter-thread communication and synchronisation mechanism was designing the logic which maps events to thread IDs and that which performs priority analysis on the READY queue, to operate as concurrently as possible.

7. CONCLUSIONS

In this paper, we have applied a selection of low-power CPU design strategies to develop a highly-optimised processor design which can meet performance goals in a power-efficient manner. We demonstrated a 48% improvement in the execution time of an IIR filter routine over the TI MSP430 which is widely used on sensor platforms, and a 14% improvement over an ARM7 processor. Our synthesis results prove the extremely lightweight nature of the design; and coupled with reduced execution times, it can enable significant energy-savings to be made in the realm of sentient computing. We also noted improvements in code density, and discussed the possibility of very fast thread management using our hardware-based scheduler module. We hope these results will stimulate more research into suitable primitives for expressing computation in the creation of very power-conscious CPU designs within the sensor network research community. Our future work will involve leveraging the small size and performance of the SpotCore CPU to build multiprocessing hubs which will take advantage of the high degree of data-parallelism inherent in sensor networks. It is our belief that such lightweight processing elements will form the cornerstone of scalable sentient computing.

Acknowledgements

Many thanks to Andy Hopper, Alastair Tse, Andrew Rice, and Ripduman Sohan for their inspiration. The generous financial support from ARM Ltd. is gratefully acknowledged.

8. REFERENCES

[1] http://www.arm.com/products/cpus/.

[2] ARM Ltd. *ARM Architecture Reference Manual.*

[3] Crossbow Technology, Inc. *Imote2 High-perfromance Wireless Sensor Network Node.*

[4] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004.

[5] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-driven Programming of Memory-Constrained Embedded Systems. In *Sensys' 06*, November 2006.

[6] V. Ekanayake, I. Clinton Kelly, and R. Manohar. An Ultra Low-Power Processor for Sensor Networks. In *Architectural Support for Programming Languages and Operating Systems*, 2004.

[7] J. L. Hennessey and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2003.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, and K. P. David Culler. System architecture directions for networked sensors. In *ASPLOS-IX*, 2000.

[9] T. Instruments. *MSP430 Ultra-Low-Power Microcontrollers.*

[10] Intel. *Intel Architecture Software Developer's Manual: Instruction Set Reference.*

[11] Intel. *Intel Itanium Architecture Software Developer's Manual*, 2006.

[12] R. M. Kling. Intel mote: An Enhanced Sensor Network Node.

[13] N. Lane and A. Campbell. The influence of Microprocessor Instructions on the energy consumption of wireless sensor networks. In *Third Workshop on Embedded Networked Sensors (EmNets 2006)*, 2006.

[14] C. Lynch and F. O'Reilly. Processor Choice For Wireless Sensor Networks. In *Workshop on Real-World Wireless Sensor Networks*, 2005.

[15] A. Mota, L. B. Oliveira, F. F. Rocha, R. Riserio, A. A. F. Loureiro, C. J. C. Jr., H. C. Wong, and E. Nakamura. WISENEP: A Network Processor for Wireless Sensor Networks. *ISCC*, 0:8–14, 2006.

[16] L. Nazhandali. *Architectural Optimisation for Performance- and Energy-Constrained Sensor Processors.* PhD thesis, University of Michigan, 2006.

[17] L. Nazhandali, M. Minuth, B. Zhai, J. Olson, T. Austin, and D. Blaauw. A Second-Generation Sensor Network Processor with Application-Driven Memory Optimizations and Out-of-Order Execution. In *ACM/IEEE International Conference on Compilers, Architecture, and Sythesis for Embedded Systems*, September 2005.

[18] D. Reed and R. Fairbairns. Nemesis Kernel Overview, May 1997.

[19] S. Rhee, D. Seetharam, S. Liu, N. Wang, and J. Xiao. i-Bean Network: An Ultra-Low Power Wireless Sensor Network. In *Ubicomp*, 2003.

[20] S. Sadasivan. An Introduction to the ARM Cortex-M3 Processor. Technical report, ARM Ltd., 2006.

[21] C.-K. Tham. *Sensor Network and Configuration: Fundamentals, Techniques, Platforms and Experiments.* Springer-Verlag, June/July 2006.

[22] A. Wang and A. P. Chandrakasan. A 180mV FFT processor using subthreshold circuit techniques. In *Digest of technical papers, 2004 IEEE International Solid State Circuits Conference*, volume 1, pages 310–319, 2004.

[23] B. Warneke and K. Pister. An Ultra-Low Energy Microcontroller for Smart Dust Wireless Sensor Networks. In *International Solid-State Circuits Conference*, 2004.