Integer-based Optimisations for Resource-constrained Sensor Platforms

Michael Zoumboulakis and George Roussos

School of Computer Science and Information Systems
Birkbeck College, University of London
{mz,gr}@dcs.bbk.ac.uk

Abstract. In this paper we argue that the fundamental constrains of WSNs impose the need to re-discover programming optimisation techniques that were widely used a few decades ago but are less common today, at least in the conventional computing arena. Integer techniques, code tuning and profiling are absolutely essential in the world of the very small devices. We present three alternative methods of integer programming: scaling, fixed-point and rational arithmetic. These techniques are complemented by a brief review of bitwise and general optimisation techniques. As artifact of the usefulness of these techniques, we discuss the implementation details of a data mining algorithm that gained over a factor of 10 improvement in performance as a result of integer programming. We conclude by presenting a widely accepted time model adapted for a WSN platform.

Key words: Wireless Sensor Networks, Optimisation, Integer Techniques, Fixed-Point Arithmetic, Rational Arithmetic, Data Mining

1 Introduction

There was a time when programmers used to go to great lengths in order to ensure that their programs were efficient. Computers were slow and expensive pushing program performance optimisation very high on the list of software engineering priorities. As the years passed, advances in circuit design and memory density translated into faster machine speeds. Today we live in an era of powerful, transistor-dense, multi-core machines with abundant memory and storage. Inevitably, this fact had an impact in modern software engineering priorities: programmers chose to attach more gravity to other important goals such as stability, portability and maintainability whilst performance and efficiency gradually dropped towards the bottom of the priorities list.

However, the resource-constrained end of the Wireless Sensor Network (WSN) spectrum is comprised of devices whose datasheets are very similar to those of computers from a couple of decades ago. Furthermore, we know that the rate of growth of WSN nodes' capabilities is much slower than that of their consumer electronics counterparts. For instance, memory in low-cost, ultra-low power devices does not track Moore's law — a micro-controller RAM costs three orders

of magnitude more than PC SRAM and five orders more than PC DRAM [12]. Even more importantly, energy density does not seem to track Moore's law either — over a decade energy density of commercially available batteries has changed only modestly [4].

These fundamental constraints suggest that programming optimisation techniques widely used a few decades ago need to be re-discovered. Consequently, we address the importance of programming efficiency in WSNs and we discuss methods that contribute to the attainment of this goal.

In the remainder of the paper, we draw from our own experience of implementing an established data mining algorithm in a resource-constrained platform and we present the direct performance gains of applying a few straightforward coding techniques. Moreover, we focus on integer programming complemented by bitwise techniques and general code tuning — combined, they form a toolbox of techniques that can greatly improve program efficiency. Lastly, we adapt the conventional time model of [2] for a sensor platform with a view to identify costly operations and provide rough estimates of the time needed for typical tasks.

2 Case Study: TinySAX — Efficient Implementation of a Data Mining Algorithm

Symbolic Aggregate Approximation (SAX) [7] is a widely-accepted and mature algorithm used for data mining. Essentially SAX performs *Symbolic Conversion*: it maps an ordered sequence of sensor readings to a sequence of letters from a finite alphabet Σ according to a set of well-defined rules. Formally, the input is a sequence of readings $\bar{r} = \langle r_1, r_2, \ldots, r_n \rangle, r_i \in \mathbb{Z}$ and the output is a sequence of letters $\bar{s} = \langle s_1, s_2, \ldots, s_m \rangle, s_i \in \Sigma$ with $0 < m \leq n$. In this respect SAX is a discretisation technique.

Applying SAX to a time-series stream has many attractive properties such as dimensionality reduction due to the Piecewise Aggregate Approximation (PAA) that is central to the algorithm. A PAA representation has been found to rival more sophisticated dimensionality reduction techniques such as Fourier transforms and wavelets [6]. The core idea of obtaining a PAA from a sequence of readings is based on dividing the sequence into m equal-sized frames and taking the mean of the values that fall within each frame. The PAA representation comprises the vector of the mean values. The final step involves transforming the PAA into a sequence of equiprobable symbols. For this, SAX uses a sorted list of breakpoints (found in statistical tables) $\bar{\beta} = \langle \beta_1, \beta_2, \ldots, \beta_{\alpha-1} \rangle$, with α =size of the alphabet, such that the area under a N(0,1) Gaussian curve from β_i to β_{i+1} is $1/a$ with β_0 and β_α set to $-\infty$ and ∞ respectively [14]. The final string is obtained according to the range of the PAA coefficients i.e. if a coefficient is less than β_1 it is mapped to 1, if it is lies between β_1 and β_2 is is mapped to 2 and so forth. The precise details of the PAA transformation and the further symbolic transformation are well-described in the SAX literature [7, 6] (and references therein).

Table 1. Performance Times (in ms) for a Symbolic Conversion using naive implementation with floating-point operations.

Operation	*Size* *40*		*80*		*120*	
S2. Standardise						
a. Mean	12ms	(10.62%)	24ms	(10.62%)	37ms	(11.01%)
b. Std Dev	42ms	(37.17%)	81ms	(35.84%)	120ms	(35.71%)
c. Subtract & Divide	25ms	(22.12%)	53ms	(23.45%)	78ms	(23.21%)
S3. Get PAA	24ms	(21.24%)	48ms	(21.24%)	73ms	(21.73%)
S4. Get Symbols	10ms	(8.85%)	20ms	(8.85%)	28ms	(8.33%)
Total Time	113ms		226ms		336ms	
RAM Image Size (Bytes)	766		846		926	

One of the common uses of SAX is for *Anomaly Detection* in a time-series. We use SAX in the WSN setting for a similar purpose, namely *Complex Event Detection*. We use the following definition of Complex Events:

Complex Event. An interesting or unusual pattern in the data gathered and processed by WSN nodes that can be very difficult or even impossible to detect using threshold-based techniques.

This method allows us to detect many complex patterns simultaneously without significant overhead to the resources of the WSN nodes. We use SAX in an online fashion where sensor nodes continuously convert sliding windows of readings to symbols and then compare them against patterns of symbols submitted by users as event interests.

SAX relies on various operations involving floating-point numbers and a first implementation attempt was running prohibitively slow on the target platform (TMote Sky [16]). A single symbolic conversion was taking approximately 113ms for a sliding window of 40 readings. Profiling the code, by timing the entry and exit points of every function, provided the information shown on Table 1.

2.1 Refactoring SAX into TinySAX

This slow implementation had to be reviewed and re-written in a manner suitable for the constraints of the WSN. The outcome was TinySAX; an efficient integer-only implementation that reduced the active CPU time by more than a factor of 10. This was achieved by a combination of the following actions, listed in order of importance:

– **Integer Programming**. Firstly, all floating point variables were replaced by integers. Functions such as standardisation — that relies on the costly formula $\bar{z_i} = \frac{r_i - \mu}{\sigma}$ where μ is the mean and σ is the Standard Deviation of the sliding window $\bar{r}$ — were re-written with elimination of the division operation. Instead, the breakpoint vector $\bar{\beta}$ was scaled by σ. The size of the breakpoint

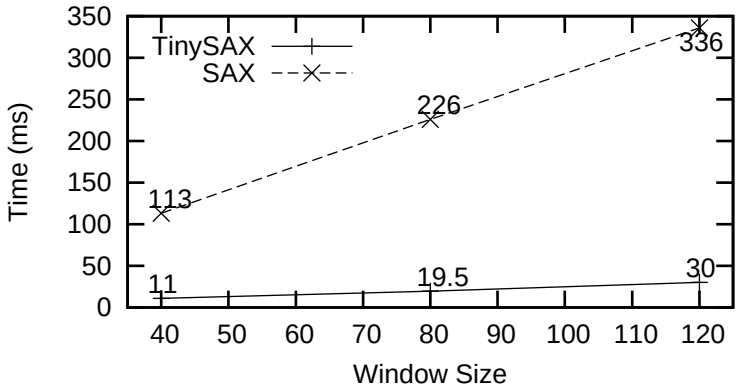

Fig. 1. Comparison of running times for two alternative implementations of the same data mining algorithm on a TMote Sky. SAX refers to an implementation that relies heavily on a number of floating point operations. TinySAX refers to an optimised implementation that has been re-written using integer programming.

vector is equal to the size of the alphabet. Since the latter is typically much smaller than the size of the sliding window, scaling the former involves significantly less operations than dividing each r_i by σ. Furthermore, as we will see in section 8, division is much costlier than multiplication especially for double words (i.e. 32-bit integers). Secondly, the breakpoints vector was *binary scaled* (i.e. multiplied by a power of 2) offline to map the floating point numbers to integers. To counteract this, the PAA coefficients were also scaled by the same scaling factor.

– **Bitwise Techniques**. Functions such as the square root were replaced by fast integer implementations coded in a manner that utilised bit-level operations. This gave an additional performance boost to the code.
– **General Optimisations**. Unrolling and consolidating loops and the choice of appropriate variables (i.e. unsigned *int*s for indices) trimmed off the final excess milliseconds from the implementation.

The results for TinySAX, in terms of time, are shown in Figure 1 — the difference in current consumption for a typical sliding window of length 40 is 73.7mA for the integer implementation compared to 251.74mA for the naive implementation (assuming a 1Hz sampling frequency).

In the sections to follow we are going to describe from a more general perspective some of the programming techniques used to transform SAX into TinySAX — an efficient data mining algorithm for constrained devices.

3 Optimisations: Strategies, Tools and Techniques

There are a few different alternatives when deciding to optimise for performance. One of the first things that should be decided is whether there is a need for op-

timisation: a fundamental questions is *"is the program good enough already"* [2]. The answer depends on the context in which the program must execute. If it has been stress-tested under different realistic workloads and it is found to perform well, then clearly there is little need to optimise. Aggressively optimising a program still in the early design stage can adversely impact the quality of the code and compromise its correctness. In the words of Donald Knuth [8]: *"we should forget about small inefficiencies, say about 97% of the time: premature optimisation is the root of all evil"*. A program should be subjected to an optimisation process once it has been profiled and its runtime behaviour is well-understood.

If optimisation passes the above suitability test, it should then be decided whether to opt for node-level optimisation, network-wide optimisation, or both. Node-level optimisation targets the specific program image that executes on a single node and attempts to re-design it so it is highly efficient. Network-wide optimisation, as the name suggests, targets the entire network: local decisions of individual nodes may be sub-optimal for the benefit of the whole network e.g. an example of this is distribution of processing load among nodes or avoiding bottlenecks such as the funnelling effect [1] that penalises nodes near a base station. Network-wide optimisation largely depends on application-specific factors and network characteristics such as density and therefore is beyond the scope of this paper. The techniques that follow are primarily aimed at node-level optimisation.

4 Integer Techniques

Integer techniques is a broad term for a family of methods relating to computer programs using integral data types. Although the term has been used to refer to techniques that sometimes use a mixture of floating-point and integer numbers, in this context we use it to refer to programs that use exclusively integers.

The motivation for re-introducing integer techniques into WSN programs stems from the following:

1. **Floating Point on Software**. The majority of low-end microcontrollers (MCUs) lack Floating Point Units (FPUs). Floating point numbers are represented in software and operations involving these data types are inherently slow and expensive.
2. **Lack of Standardisation**. Programs that use floats in WSNs are less portable since there exist variations in the floating point representation across different compilers and microcontrollers. Programs in heterogeneous networks that need to communicate floating point values over the radio, suffer from the added complexity that the lack of a common standard introduces.
3. **Application needs**. Many real-world applications have a requirement for operations involving real numbers — typical examples are object tracking and target estimation applications. In general, a large number of trigonometric and statistic functions are prime candidates for integer transformation.

In the following, we will discuss some of the specific techniques that are available to a programmer.

4.1 Scaling

We have briefly discussed scaling in section 2.1 where it was used in the implementation of TinySAX. Scaling is a well-understood technique that involves multiplying a floating point number by an integer, usually a power of two (binary scaling) to represent it as an integer. Arithmetic operations can then be performed using the scaled numbers and the result can be divided by the same scaling factor. As an example consider the floating point numbers -0.52 and 1.28. Scaling both by 2^{11} and rounding toward zero gives -1064 and 2621 respectively. Any arithmetic operation can be performed using the integers: i.e. multiplication would yield -2788744. If we divide the result by the same scaling factor we obtain -1361. To retrieve the floating point result we divide again and obtain 0.6645 (the exact floating point result is 0.6656).

Care must be taken to prevent arithmetic overflow. This is usually done by performing some analysis of the range of values that a program variable will take. The maximum of these values is taken and scaled. To accommodate the maximum value x_{max}, M bits will be needed with $\log_2 x_{max} \leq M$. Numerical precision requirements vary across applications, so N bits must be reserved to satisfy the accuracy requirements Δx with $2^{-N} \leq \Delta x$. The total number of bits that will be needed to represent the floating point number in an integer type within the accuracy requirements is therefore $M+N$. There is a tradeoff between the maximum range of a variable and the accuracy required [17], but this is very much application dependent.

Embedded operating systems such as TinyOS [13] employ scaling techniques in the design of certain components — an example of this is the implementation of Timers: a second equals 1,024 milliseconds. So if we need to represent the time value of 1.875 seconds we scale it by 2^{10} and use 1,920 to perform any arithmetic operations. This level of accuracy is good enough for many applications.

Generally speaking, there will always be a tradeoff between speed and accuracy: if an application requires very high numerical fidelity in its calculations then performance may need to be sacrificed, for instance by using arbitrary precision data types.

4.2 Fixed-point arithmetic

Fixed-point arithmetic is very closely related to the scaling technique but it involves slightly higher degree of programming effort. It is estimated that approximately 30 % of the software development for the first NASA Apollo space missions was spent on fixed-point arithmetic and scaling [3]. Nowadays, it is still widely used by DSP programmers, graphics developers, computer typesetting and other applications where there is a combined need for real-valued numbers and high performance.

In floating-point numbers the radix point is generally allowed to be determined dynamically or to "float" thus being capable of representing a wide range of values. Conversely, in fixed-point arithmetic the radix point is fixed. There is a fixed number of digits after the radix point reserved for the fractional part.

Let N represent the total number of bits needed to accommodate a fixed-point variable. We can then split a type of length N into Integer bits and Fractional bits. The integer bits are to the left of the hypothetical or implied binary radix point and the fractional bits are the remaining bits to the right. We use the notation (I, F) [1] to represent the Integer part and the Fractional part respectively. The number of bits needed for an unsigned type is $I + F$ and for a signed type $I + F + 1$. The range of unsigned fixed-point numbers is from 0 to $2^i - 2^{-f}$ while the range for singed fixed-point numbers is from -2^{i-1} to $2^{i-1} - 2^{-f}$, where i and f are the number of bits needed to represent the integer and fractional parts respectively. To obtain the value of unsigned and signed numbers in (I, F) (fixed-point) representation we use equations 1 and 2 respectively [19]:

$$x = 2^{-f} \sum_{n=0}^{N-1} 2^n x_n \tag{1}$$

$$x = 2^{-f} \left(-2^{N-1} x_{N-1} + \sum_{n=0}^{N-2} 2^n x_n \right) \tag{2}$$

Where x_n denotes bit n of x.

Consider the following example: an unsigned fixed-point data type of the form $(3, 5)$. This type can accommodate a range from 0 to 7.96875 and has the following representation:

← *Integer Part* →			⟵ *Fractional Part* ⟶				
I	**I**	**I**	**F**	**F**	**F**	**F**	**F**

The number $(00101011)_2$ is then $2^{-5}(1 + 2 + 2^3 + 2^5) = 1.34375$. Similarly, if we were using a signed type $(3, 5)$ requiring $(I + F) + 1 = 9$ bits the range would become $-4 \leq x \leq 3.96875$, with the MSB reserved for sign in two's complement notation. The code (in C) needed for the transformation is shown below:

```
#define FB 5 /* (3,5) Representation */
float f=1.34375; uint8_t fp;
fp = (uint8_t)(f * (1<<FB) + (f>=0 ? 0.5 : -0.5));
f = (float) fp / (1<<FB); /* Fixed-point back to float */
```

All arithmetic operations are defined using the fixed-point representation but care needs to be taken to ensure that the types are in the same sign representation

[1] Other common notations are (M, N) and Q.

and that the result does not overflow. Operations can still be performed on numbers with different bit (I, F) representations but somewhat more effort is required for bit alignment.

Although measurable performance improvements can be gained by using fixed-point arithmetic, it suffers from certain disadvantages: it can hinder code portability. Moreover, it can make the programming task tedious resulting in error-prone code.

Recently, there have been efforts to automate the floating-point to fixed-point conversion process. Notably, [11] and [15] suggest program translators that firstly monitor and collect statistics about the range of floating point variables and then automatically generate a fixed-point isomorphism.

4.3 Rational Arithmetic

An often overlooked alternative that lies between fixed point and floating point arithmetic in complexity, cost and capability is rational (or "no point") arithmetic [5]. In rational arithmetic two variables are used to store each number; A real number is therefore represented as a ratio (fraction) of two numbers. If arithmetic is done on fractions instead of approximations many computations can be performed entirely without any rounding errors or precision loss [9]. However, taking into account that it is advisable to avoid arbitrary precision arithmetic, some loss will occur if rational operations are bounded to a specific type length as it is the typical case.

Rational numbers are represented programmatically as pairs of integers $(u/u\prime)$ with u and $u\prime$ relatively prime and with zero represented as $(0/1)$. Arithmetic operations are defined on pairs of integers and they largely rely on the *greatest common divisor* algorithm. Euclid's algorithm which is by far the simplest for *gcd* calculation has $O(n^2)$ complexity, but a binary *gcd* implementation is somewhat faster.

An example of multiplication between $(u/u\prime)$ and $(v/v\prime)$ and result $(w/w\prime)$ involves the calculation of uv and $u\prime v\prime$. The resulting $(w/w\prime)$ may not be relatively prime but if we let $d = gcd(uv, u\prime v\prime)$ the answer is $w = uv/d$ and $w\prime = u\prime v\prime/d$ [9].

Rational arithmetic using bounded types offers the same range as fixed-point arithmetic and approximately the same accuracy. In terms of simplicity, the arithmetic operations of addition and subtraction are easier to implement [5] in fixed-point whilst division and multiplication have simpler implementations in rational arithmetic. An interface providing basic operations such as addition, multiplication, comparison, and so forth can be developed that offers the ability to use them in the same manner as calling library functions.

The advantage of rational arithmetic over fixed point is that it is more portable and less machine dependent. Of course this assumes that all helper functions (such as *gcd*) have machine-independent implementations.

5 Bitwise Techniques

Bitwise techniques involve the use of the bit-level operators: AND, OR, Exclusive OR (XOR) and left (`<<`) and right shifts (`>>`). The use of bit fiddling allows to operate at a much lower level and it can often assist in making programs more time and space efficient. Bit packing for example is a good technique for packing many values into one data type.

Consider an example from our implementation of TinySAX: the typical alphabet size used is 10 characters with a maximum size of 15 characters. The application has a requirement to send over the radio a string of characters. The smallest data type that a language provides is one byte, however 4 bits are sufficient to represent a character from an alphabet of size 15 (with range 0-15). The listing below shows a code example used to pack 4-bit nibbles into a 64-bit integer.

```
uint64_t y=0; /* 64-bit var to be packed with 16 nibbles */
for (i=0; i < length-1; i++)
 { y = (y+data[i]) << 0x4; }
y += data[length-1];
```

With this code we can encode twice the amount of information in a single variable. If there is a requirement to send such a variable once a second over the radio, in the course of a whole day we save 675 kilobytes (or little over 2 megabytes if we include the packet headers and footers) at the expense of a little computation. The packed value can be unpacked using the code below:

```
for (i=shift=0; i < length; i++, shift+=0x4)
 unpacked_data[(length-1)-i] = (y>>shift)%0x10;
```

Note that the modulo operation is inexpensive since it does not require division i.e. $x \bmod 2^n = x\&(2^n - 1)$ [10].

Bit fields such as the above are particularly useful in WSNs when there is a need to keep track of a neighbour table. For instance if neighbours with identification numbers 1,5,8 are awake, this can be encoded in a bit field in the following manner: $\{1, 0, 0, 0, 1, 0, 0, 1\}$. This a very compact representation and individual bits from the field can be toggled, as neighbours appear or disappear, at a very small computational cost.

Furthermore, bitwise operations are useful for the reduction or elimination of *branching* — a part of the program where control flow is decided on evaluating some condition — in programs. For example the simple line `x - ((x - y) & -(x < y))` can be used to determine the maximum of the two values (x, y) or `(x ^ (x >> (16 - 1))) - (x >> (16 - 1))` to determine the absolute (`abs`) value of x (assuming 16-bit integers).

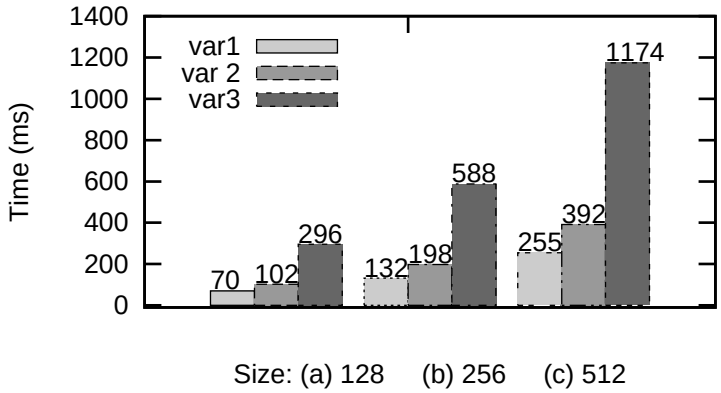

Fig. 2. Running time of 3 variance algorithms on a TMote Sky — Sets A, B, C represent vectors of 128, 256 and 512 random numbers respectively.

The usefulness of bit-level techniques in embedded devices is undisputed — in the words of the author of one of the leading works in bit fiddling [18] *"It is amazing what can done with binary addition and subtraction and maybe some bitwise operations"*. The main criticism for bit fiddling is that it can make programs less portable. The majority of bit-level operations depend on endianness (i.e. the byte ordering) and therefore can make programs platform-specific.

6 Profiling

Profiling or benchmarking is a useful form of empirical analysis that aims to identify which parts of a computer program consume the most time or other resources such as memory, registers or disk. With respect to time, a straightforward way of profiling a program is the use of timers at entry and exit points of functions. Timers are criticised of inaccuracy in conventional operating systems because a multi-tasking computer system is rarely idle and processes contend for CPU time. However this is not entirely true for the majority of low-end sensor nodes where a single-threaded application [2] is the *only* process in the system. Therefore, the use of timers is a fairly accurate profiling measure within the platform timer accuracy and precision envelope.

Theoretical complexity analysis of algorithms can be complemented by empirically obtaining execution times for real-world input. This process can aid in deciding which implementation to include in the final version of a program. As an example consider the results of the empirical analysis of three alternative variance implementations shown in figure 2.

The first algorithm (`var1`) is fast but slightly inaccurate: it has a relative error of .0011%, .037% and .092% on sets of random numbers A, B and C of

[2] For the sake of simplicity, we ignore recent multi-threading abstractions such as TinyThreads and ContikiOS.

Table 2. Results from unrolling a basic loop that executes the one-line statement `c[i]=a[i]+b[i]` in its body.

Degree of Unrolling	Time (ms)
Normal Loop	220ms
2-way Unrolling	173ms
Loop Elimination	112ms

sizes 128, 256 and 512 respectively. Algorithm 2 (`var2`) is precise and fairly fast while algorithm 3 (`var3`) [9] is a highly accurate and numerically stable algorithm.

The use of such analysis can annotate the accuracy-efficiency tradeoff with some numbers and can make the decision-making process easier.

7 Loop Unrolling

If a program performs an operation multiple times it is worth considering *unrolling* the loop — that is explicitly writing each iteration of the loop in sequence [2]. This can only be achieved when the terminating condition of the loop is known in advance.

An example of 2-way unrolling, provided `SIZE` is known in advance and it is an even number, is shown below:

```
for (i=0; i < SIZE; i+=2)
 { c[i] = a[i]+b[i];
   c[i+1] = a[i+1]+b[i+1]; }
```

The loop can be unrolled further as long as (`SIZE` *mod* increment) `==`0 and it can even be eliminated altogether by writing out explicitly every instruction. The runtime results from unrolling the simple loop described above are shown on Table 2.

Another optimisation technique, involves removing any conditional expressions that are inside the body of a loop. This will reduce branching and the code can be re-factored by enclosing a copy of the loop inside the bodies of the `if` and `else` clauses respectively. Similarly, any expensive operations inside the loop body should be examined: for instance if a loop involves an expression that contains division, then this expression should be re-arranged to use multiplication e.g. $(u/u\prime) \leq (v/v\prime)$ is equivalent to $(uv\prime) \leq (vu\prime)$ — the latter is significantly cheaper, as we will see in the next section.

Generally speaking, loops are usually the first part of the program to target. Eliminating the loop completely will almost always yield a gain in performance. Critics of this technique assert that there is a side-effect of higher register usage to keep track of intermediate results. Empirical analysis suggests that this is

Table 3. Performance Times (in ms) for various operations executed by the TMote Sky (timed using TinyOS 2.x and msp430-gcc 3.2.3).

	Arithmetic (Integer)		
Operation	*16-bit*	*32-bit*	*64-bit*
Increment	6,734	12,824	67,271
Addition	7,955	15,319	88,301
Subtraction	7,969	15,310	87,985
Multiplication	145,020	159,060	316,755
Division	226,265	709,720	3,519,055
Remainder	225,375	706,875	3,540,080
	Bitwise		
AND	7,965	15,285	88,095
OR	7,985	15,325	88,360
XOR	7,955	15,315	88,310
SHIFT	56,735	62,880	573,665
	Floating Point Arithmetic		
Assignment & Cast	2,687,535		
Addition	3,438,530		
Subtraction	3,499,920		
Multiplication	4,841,490		
Division	4,041,785		
	Array Comparisons & Swaps		
Straight Comp	104,095		
Comp C Fcn	83,315		
Comp TOS Fcn	83,335		
Comp TOS task	159,055		
Swap C Macro	93,085		
Swap C Function	83,320		
	Max Function		
Straight Max	137,565		
Max C Macro	137,075		
Max C Function	79,110		
	Built-in Sensors		
ReadLight	327,510		
ReadTemperature	154,125		
ReadHumidity	154,175		
ReadIVoltage	321,635		
	External Flash Chip IO		
Reads (4-bytes)	158,750		
Writes (4-bytes)	191,280		

highly compiler-dependent; run-time results can always assist in measuring the benefit for loop elimination.

8 A Time Model for WSNs

In this section, we apply the methodologies introduced by Bentley [2] for the construction of a time model that can be used to identify the relative cost of typical operations. The model is adapted so it tests various platform-specific components such as the time it takes to sample the on-board sensors. The results are shown in Table 3. Reported times are in milliseconds per 5,000 trials. As expected the arithmetic operations that are larger than the MCU word size (16-bit) are taking much longer. For example, 64-bit integer division is almost as slow as floating point division, while for other operations using a 64-bit type is cheaper than floating-point. 64-bit types can be useful for temporary storage i.e. holding results, when scaling is employed.

The lack of FPU slows down floating point operations — by a factor of 432 in the case of addition — making floating point operations unacceptably slow. Surprisingly, bitwise operations are slightly slower than addition and subtraction but shifts are still considerably faster than division and multiplication.

Subscripted array operations are generally fast, and according to [2] this is due to favourable memory access predictability. A TinyOS function — which is platform-independent and hence portable — is infinitesimally slower (by 20ms) than a C function performing the same operation. However, a TinyOS task, which is placed on a separate FIFO queue, is somewhat slower — almost by a factor of 2 — than the C or TinyOS function. The debate of whether a function or a preprocessor macro is faster, appears to have settled on the fact that on a mote the C function is faster than its macro equivalent.

Sampling Light and Internal Voltage sensors is more expensive than sampling Temperature and Humidity — the internal temperature and humidity sensors are on the same chip, hence the similarity in terms of performance. Flash Input-Output is generally slow[3] and as expected writes and slower than reads. The reads are not cached, in contrast with traditional disks and flash chips.

Lastly, we expect this table to provide a suitable guide into the tradeoffs of operations and data types that partially determine the running time of programs.

9 Summary

There is a wide variety of real-world WSN applications that have a need for representation of real numbers in programs. However, many MCUs lack Floating Point Units (FPUs) and as a result computations are performed in software. This has the undesirable impact of slow performance and increased energy cost; both factors are burdensome in any embedded environment. Furthermore the lack of a common floating point representation across different node platforms and compilers can introduce runtime errors in heterogeneous networks that need to communicate such values over the air. In this paper, we focus on three main

[3] The results reported are using the `ConfigStorage` TinyOS component.

integer programming techniques: Scaling, Fixed-point and Rational-point Arithmetic. Each one of these varies in complexity, cost and implementation and a good understanding of their relative strengths and weaknesses is essential for designers who aim to shave off valuable time units from their implementations. Moreover, bitwise techniques and other optimisations such as loop unrolling can assist in compounding significant performance gains. We have highlighted the applicability of these methods, combined with integer techniques, by presenting a case study of an implementation of a data mining algorithm for a WSN platform. By re-writing the program with efficiency as the ultimate goal performance improved by a factor of more than 10. Finally, we assert that operational awareness of a program's performance together with application of an appropriate optimisation strategy are essential factors in software engineering for WSNs with direct implications to network and device lifetime.

References

1. Gahng-Seop Ahn, Se Gi Hong, Emiliano Miluzzo, Andrew T. Campbell, and Francesca Cuomo. Funneling-MAC: a localized, sink-oriented MAC for boosting fidelity in sensor networks. In *SenSys*, pages 293–306. ACM, (2006).
2. Jon Bentley. *Programming Pearls (2nd Edition)*. Addison-Wesley Pub Co, (1999).
3. H. Kreide and D.W. Lambert. Computation: Aerospace Computers in Aircraft, Missiles and Spacecraft. *Space/Aeronaut*, 78, (1964).
4. Joseph M. Hellerstein, Wei Hong, and Samuel Madden. The sensor spectrum: technology, trends, and requirements. *SIGMOD Record*, 32(4):22–27, (2003).
5. Berthold K. P. Horn. Rational arithmetic for minicomputers. *Softw., Pract. Exper.*, 8(2):171–176, (1978).
6. Eamonn Keogh, Jessica Lin, and Ada Fu. HOT SAX: Efficiently Finding the Most Unusual Time Series Subsequence. *IEEE International Conference on Data Mining*, pages 226–233, (2005).
7. Eamonn Keogh, Stefano Lonardi, and Chotirat Ann Ratanamahatana. Towards parameter-free data mining. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 206–215, New York, NY, USA, (2004).
8. Donald E. Knuth. Structured Programming with go to Statements. *ACM Comput. Surv.*, 6(4):261–301, (1974).
9. Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, (1981).
10. Donald E. Knuth. *The Art of Computer Programming: Volume 4, F.1, Binary Tricks and Techniques*. Addison-Wesley, (2009).
11. Ki-Il Kum, Jiyang Kang, and Sung Wonyong. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. In *IEEE Transactions on Circuits and Syst—Part II*, volume 47, pages 840–848, (2000).
12. Philip Levis, Eric Brewer, David Culler, David Gay, Samuel Madden, Neil Patel, Joe Polastre, Scott Shenker, Robert Szewczyk, and Alec Woo. The Emergence of a Networking Primitive in Wireless Sensor Networks. *Communications of the ACM*, 51(7):99–106, (2008).

13. Philip Alexander Levis. TinyOS: An Open Operating System for Wireless Sensor Networks (Invited Seminar). In *MDM '06: Proceedings of the 7th International Conference on Mobile Data Management*, page 63, Washington, DC, USA, (2006).
14. Stefano Lonardi, Jessica Lin, Eamonn J. Keogh, and Bill Chiu. Efficient discovery of unusual patterns in time series. *New Generation Comput.*, 25(1):61–93, (2006).
15. Daniel Menard, Daniel Chillet, and Olivier Sentieys. Floating-to-fixed-point conversion for digital signal processors. *EURASIP J. Appl. Signal Process.*, 2006:77–77, (2006).
16. MoteIV (later renamed to Sentilla). TMote Sky Datasheets and Downloads, `http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf`.
17. Jimfron Tan and Nicholas Kyriakopoulos. Implementation of a Tracking Kalman Filter on a Digital Signal Processor. In *IEEE Transactions on Industrial Electronics*, volume 35, pages 126–135, (1988).
18. Henry S. Warren. *Hacker's Delight.* Addison-Wesley, (2002).
19. Randy Yates. Fixed-point arithmetic: An introduction — digital signal labs, technical reference, `http://www.digitalsignallabs.com/fp.pdf`, (2007).