

Triangulation in Ad-Hoc Networks: An Energy Efficient Solution

Chris Savarese, Yashesh Shroff, Greg Lawrence

CS252, EECS Department
University of California at Berkeley

Abstract

The realization of a triangulation algorithm in an ad-hoc network of heterogeneous nodes requires a low power solution. This paper presents a study of three commercial architectures, several versions of a triangulation algorithm, and discussions on possible optimizations to both hardware and software for greater energy savings. Energy consumption levels for most variants are estimated through simulation or speculation. The target energy consumption level was $10\mu\text{J}$. Predicted results after the implementation of all proposed optimizations suggest an overall cost of $27\mu\text{J}$ or less. Proposals for further optimizations are provided.

1. Introduction

The deployment of a densely populated, ad-hoc network of heterogeneous nodes has many attractive applications; climate control in large buildings, interactive display exhibits, personal monitoring and navigation systems, and smart home environments are a few immediately obvious possibilities. This market of applications naturally dictates that nodes must have a long lifetime, implying that whatever power source each individual node employs must last indefinitely to avoid the need for frequent servicing. Another consequence of these applications is the desire for each node to be able to triangulate its position with respect to its neighboring nodes. For example, the triangulation feature would allow climate control to be localized to different parts of a building based on the readings taken from nodes moving through different control zones. Smart homes and interactive exhibits with triangulation abilities could use the positions of people wearing nodes relative to nodes fixed on interactive objects to further customize an environment. Finally, personal monitoring and navigation systems present obvious motivations for triangulation.

If done carelessly, the realization of a triangulation algorithm for an ad-hoc network could easily become so computationally intensive as to consume unacceptable amounts of power. This paper presents the research and findings concerning the effects of implementing several different optimizations to a triangulation algorithm and to

the hardware on which it is performed. The general strategy used in guiding this research was to start from a standard realization of the algorithm on a commercial processor, the StrongARM 1100, and measure the power consumption on that platform. We then estimated improvements gained from various optimizations and from other architectures based on the performance of the StrongARM and the differences between it and the implementation in question. In some situations we were able to run simulations to further characterize the performance of a particular alteration. For each of the optimization techniques studied, we will discuss the details of the technique itself, the motivation behind considering the technique, the results or expected results derived from doing so, and possible improvements to the technique.

2. Algorithm Development

It is important to emphasize that the nodes in the type of network being considered are heterogeneous, meaning that any behavior described for one node can be expected of any and all other nodes as well. For the sake of clarity, we will often talk of the triangulation algorithms from the perspective of a particular node, thus seemingly inferring that only that node is working through the algorithm. In reality though, every node will be behaving in approximately the same manner as the one being considered.

The computations are based on the distances between each of the nodes, as measured by received signal strength indicator (RSSI). The ranges between each of the nodes are shared with neighboring nodes, such that each node is aware of the range data between all possible pairs of nodes within its neighborhood. These range values are prone to as much as $\pm 50\%$ error.

In the following discussions, a *known* node is one for which the coordinates of its position are known, and an *unknown* node is one for which the coordinates of its position are not known.

2.1 The Standard Algorithm

We began our research with an algorithm based on a least squares solution incorporating the coordinates and ranges of all the nodes surrounding an unknown node. The immediately obvious flaw to this approach is the assumption that all of the surrounding nodes are known. Nevertheless, we foresaw that the standard and the more robust generalized algorithms would incorporate many similar features, especially regarding power consumption, and we decided to start with the standard algorithm in order to attain preliminary results. The adaptations necessary for the generalized algorithm and their consequences will be dealt with in subsequent sections.

The coordinates of each neighboring node and their respective distances from the unknown node are represented linearly:

$$\begin{bmatrix} (x_1 - x_n) & (y_1 - y_n) & (z_1 - z_n) \\ (x_2 - x_n) & (y_2 - y_n) & (z_2 - z_n) \\ \vdots & \vdots & \vdots \\ (x_{n-1} - x_n) & (y_{n-1} - y_n) & (z_{n-1} - z_n) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

Where (x_i, y_i, z_i) are the coordinates of known node i , (x, y, z) are the unknown coordinates, and b_i is a scalar term based on the distance from node i to the unknown node and the squared coordinates of nodes i and n [1]. This overdetermined least squares problem is solved using a modified Gram-Schmidt QR factorization to find the coordinates (x, y, z) of the unknown node:

Q-R factorization of matrix $A_{m \times n}$:

```

for i = 1:n
    q_i = a_i
    for j = 1:(i-1)
        r_ji = q_j^T q_i
        q_i = q_i - r_ji q_j
    end
    r_ii = norm(q_i)
    q_i = q_i / r_ii
end

```

The solution to the least squares problem is now straightforward:

$$x = (A^T A)^{-1} A^T b = (R^T Q^T Q R)^{-1} R^T Q^T b$$

$$x = (R^T R)^{-1} R^T Q^T b = R^{-1} R^{-T} R^T Q^T b$$

$$x = R^{-1} Q^T b$$

LU decomposition of R and backsubstitution of the columns of Q^T allow for the elimination of one matrix multiplication, but the remaining operations still result in a substantial amount of computation.

In all, we have two very computationally intensive tasks in determining the n -dimensional position of x based on

the locations of $m+1$ nodes: the QR decomposition, and the calculation of x from Q and R . The total cost of these operations is given by:

$$O(2n^2 m - \frac{2}{3} n^3)$$

Beutel [1] shows that the inaccuracy introduced by the errors in the range measurements is substantially reduced for $n=7$ or more nodes.

2.2 The Generalized Algorithm

In an ad-hoc network such as the one being considered, there is no guarantee that surrounding nodes will know their own coordinates, and so a more robust algorithm is called for.

We developed an algorithm that evolves over time based on the range values between all the nodes in a neighborhood. This algorithm determines the locations of unknown nodes one at a time in the order that they establish communication, making assumptions where necessary, and compensating for errors through corrections and redundant calculations as more information becomes available. These assumptions are needed at first in order to deal with the underdetermined set of equations presented by the first few nodes. This description of the general algorithm assumes the perspective of node n_0 .

The algorithm begins with the assumption that n_0 is located at $(0,0,0)$. The first node to establish communication, n_1 , is assumed to be located at $(r_{01}, 0, 0)$, where r_{01} is the RSSI-determined distance between n_0 and n_1 . The location of the next node, n_2 , can then be explicitly solved for, given two assumptions: first the square root involved in finding y_2 is assumed to yield a positive result, and second, z_2 is assumed to be 0.

$$x_2 = \frac{r_{01}^2 + r_{02}^2 - r_{12}^2}{2r_{01}} \quad y_2 = \sqrt{r_{02}^2 - x_2^2}$$

The next node, n_3 , is handled much like n_2 , except that only one assumption is made: that the square root involved in finding z_3 is positive.

$$x_3 = \frac{r_{01}^2 + r_{03}^2 - r_{13}^2}{2r_{01}} \quad y_3 = \frac{r_{03}^2 - r_{23}^2 + x_2^2 + y_2^2 - 2x_2 x_3}{2y_2}$$

$$z_3 = \sqrt{r_{03}^2 - x_3^2 - y_3^2}$$

From this point forth, the system of equations used to solve for further nodes is no longer underdetermined, and so the standard algorithm can be employed for each new node.

There are two major problems with this algorithm. First, although all of the assumptions are designed to establish correct locations for the first 4 nodes relative to each other, they do not in any way guarantee a correct orientation of the nodes' locations with respect to a fixed refer-

ence point. This is, of course, unavoidable, since until a known node enters the network, no fixed reference point is available. When the first known node, n_k , enters the network, n_0 determines n_k 's location relative to its own concept of (0,0,0) using the standard algorithm. It then compares this relative location of n_k to the known location of n_k , determining the shift needed to translate one onto the other. Every location in n_0 's memory, including n_0 itself, is then translated accordingly, and future computations adopt these new locations. A second known node would then provide enough information to determine a rotation to be applied to all of the locations in n_0 's memory, and a third known node may or may not result in a flip. All nodes should now be oriented correctly, and all false assumptions should be fixed.

The second major problem is that of error due to inaccurate range measurements. The standard algorithm deals with this problem by using overdetermined systems of equations to average out the errors, but this algorithm defeats this approach by considering new nodes one at a time. One fix to this problem is to recalculate the positions of the nodes using the standard algorithm several times after a considerable number of nodes have been located. Again, based loosely on Beutel's analysis [1], we predict that 5 nodes recalculated 2 or 3 times will yield relatively accurate results. As we were unable to verify this approach, these numbers are highly speculative.

The generalized algorithm is separated into two classifications: startup and update. The startup mode of the algorithm refers to the initial assumptions, computations, and redundant iterations necessary to achieve convergence detailed above. Once convergence is achieved, the algorithm shifts to update mode, in which nodes' locations can be added, removed, or changed with only one iteration of the standard algorithm.

2.3 Fixed Point Arithmetic

As expected, we found floating point operations to be extremely costly, so we created a Q16 fixed point version of the standard algorithm. Our fixed point implementation was inefficient in two important ways. First, it utilized shifts before and after each multiplication operation for normalization, and second, each fixed point operation required a function call, resulting in large amounts of overhead. Despite these problems, we still realized considerable energy savings over the floating point version of the standard algorithm. These results and further commentary on more efficient fixed point solutions are included in later sections of this paper.

2.4 The CORDIC Algorithm

CORDIC (COordinate Rotation DIgital Computer) is an iterative, dual mode algorithm that rotates two-element

vectors using only additions and shifts [2]. The vector mode determines the CORDIC controls that define an angle of rotation, and the rotate mode uses these controls to perform rotations. Successive calls to the CORDIC algorithm can be used to perform a QR factorization on a matrix with large performance gains derived from not using any multiplications or divisions. Furthermore, the algorithm is very easily parallelized, and a proper hardware implementation can further increase the performance of CORDIC-based QR factorization [3].

The CORDIC algorithm naturally fits into specialized hardware that is extremely modular in design. The most basic unit is the CORDIC cell, which consists of two logic blocks for determining sign, two buffers, two barrel shifters, four multiplexers, and two ALUs [2]. In general, three CORDIC cells are grouped together with small memory units to form a CORDIC supercell capable of handling complex numbers [3]. As our application deals with purely real data, our supercell would contain only one CORDIC cell and memory. Next, n CORDIC supercells are cascaded, where n is the number of minirotations per CORDIC rotation. In general, n should be equal to the number of bits of precision desired. This cascaded configuration creates a natural pipeline: a new rotation problem can be started at the completion of every CORDIC stage [3]. Finally, an application-specific arrangement of supercell strings exploits parallelism. For QR factorization, the strings are staggered by one supercell, and the number of strings is dependent on the size of the matrix [3].

3. Experiments with Commercial Architectures

We simulated the standard algorithm on 3 commercial architectures: the StrongARM 1100, the Tensilica Xtensa, and the Texas Instruments TMS320C62 DSP. Parameters for each experiment included floating vs. fixed point arithmetic and the number of nodes in the network. This section of the paper introduces each of the architectures, and the next section reviews the results of these trials.

3.1 The StrongARM 1100

The StrongARM is a 32-bit general-purpose, 160MHz, single-issue, low power, high performance microprocessor designed specifically for embedded applications. It contains 16/8 kB, 32-way set-associative instruction and data caches, respectively. The split instruction and data caches are features common to all three architectures that we explored. It simplifies control and allows for full pipelining of load and store instructions. It is important to note that the StrongARM does not have a floating-point unit; instead, it relies on software implementation of floating point operations. The StrongARM 1100 has 31 32-bit registers, 16 of which are available at a time.

The StrongARM was chosen to be our control case throughout our research. We used this platform to gain an early assessment of the types of performance and power consumption we should expect from other architectures and optimizations.

3.2 The Tensilica Xtensa

Xtensa is an instruction-set extensible processor from Tensilica. The core processor can be configured to match the application needs. Beyond that, it is also possible to add TIE instructions to get additional performance and code-density improvement. We customized the major features of the core processor based on the StrongARM 1100 processor to provide for a better basis of comparison between the two:

- Clock speed: 170MHz
- I/D cache: 16kB, direct
- 32 Registers (32-bits)

3.3 The Texas Instruments ‘C62 DSP

After some work with the first two processors, we quickly arrived at a set of conclusions that led us to believe that a DSP core was the natural choice for a triangulation algorithm that relies so heavily on matrix manipulations. To maintain fair comparison, we chose the DSP processor with features most similar to the StrongARM processor. Simulation results presented here are based on Texas Instrument’s TMS320C6211 (‘C62) DSP processor. The ‘C62 is a 32-bit signal processor with multiple functional units to speed up real-time computations. Like the StrongARM, the ‘C62 also has a MAC and a barrel shifter at the end of the MAC/ALU data-path to allow bit shifting of the final product. The ‘C62 is a 150MHz, 400mW processor that operates at a core supply voltage of 1.5V.

4. Time and Energy Profiling

Since we know the nominal power consumption and cycle count for the ARM, Xtensa, and ‘C62 processors, we can determine the energy consumed for a single loop of the standard algorithm using the following equation:

$$Energy = (\text{nominal core power}) (\text{cycles}) (\text{clock period})$$

Nominal core power is 200mW for the StrongARM, 200mW for the Xtensa, and 400mW for the ‘C62 at low activity. For each architecture, the cycle count is measured by the simulators, and clock frequency is known. It is important to note that the cycle count for the ARM processor was calculated from simulations using zero cache misses, since the ARM emulator is not capable of simulating cache activity.

We profiled the ARM processor in two other ways to confirm our faith in using cycle counts as a sound way of getting energy estimates.

- *Itemized instruction profiling*: Current consumption for each ARM instruction has been characterized previously [6]. Using instruction counts with current consumption and cycles itemized for each instruction executed, we can get an accurate estimate of energy per standard loop.
- *ARM testboard*: A third method for determining energy consumption was carried out by executing the code on an actual StrongARM processor. A cycle count from an on-board 3.6864MHz clock was measured and used to approximate energy consumption.

Energy estimates obtained from itemized instruction counts and profiling on a physical ARM board were in consonance with our predictions based on cycle counts.

4.1 Fixed Point vs. Floating Point

We profiled floating and fixed point versions of the standard algorithm with 7 nodes on all 3 architectures in order to find their respective energy costs:

<i>Processor</i>	<i>Floating</i>	<i>Fixed</i>	<i>Savings</i>
StrongARM	68 μ J	43 μ J	37%
Tensilica	144 μ J	69 μ J	52%
TI ‘C62	115 μ J	73 μ J	37%

Table 1: Energy consumption (Fixed vs. Floating point)

To our surprise, we found that the StrongARM processor out-performed both of the other two processors. To be fair, the Xtensa was never intended to be built without TIE instructions, and so by not taking advantage of this feature, we failed to use the Xtensa to the fullest of its capabilities.

Although the cycle count for the standard algorithm with 7 nodes was about 25% lower on the ‘C62 than on the StrongARM, the core power for the ‘C62 is twice that of the StrongARM. The discrepancy in cycle counts suggests that the DSP core was indeed helpful, but only if such a core can be found in a processor with a lower core power rating.

5. Further Optimizations and Energy Forecasts

In this section of the paper we discuss some of the optimizations we researched but never had the chance to implement, and we attempt to predict the energy savings achieved from each alteration to the standard algorithm running on the StrongARM processor with fixed-point arithmetic.

5.1 The Cost of the Generalized Algorithm

The first four nodes considered in startup mode are handled with assumptions and explicit calculations. After that, the algorithm employs the standard algorithm with a steadily increasing number of nodes for each iteration. Using simulated cycle counts from the fixed-point version of the standard algorithm for different numbers of nodes on the StrongARM, we were able to estimate the expected power consumption for startup mode in a network with 7 nodes, and with two waves of recalculations for increased accuracy. Our conclusion for these parameters was 884 μ J. Fortunately, this happens only once when the network is first brought on line.

The update mode, which is the part of the algorithm that occurs regularly, is essentially one pass of the standard algorithm. We found that the cost of the standard algorithm varies linearly with the number of nodes in the network. For 7 nodes on the StrongARM with fixed-point math, this cost is about 43 μ J. It should be noted that this number is based on the addition of a node, and that a change in a node's position is assumed to be handled as a removal and subsequent addition of a node. A more efficient approach would be to update the matrix involved in the least squares computation based on the movement of a node, instead of performing a full pass through the standard algorithm. This possibility is discussed in the next subsection.

5.2 Employing the CORDIC Algorithm

CORDIC performs more operations than the Gram-Schmidt version of QR factorization, but each CORDIC operation is much less costly and highly parallelized. Overall, the parallelization results in greater performance, and the less expensive operations sum up to considerable power savings. Through fixed-point StrongARM simulations, we determined that the Gram-Schmidt QR factorization used in the standard algorithm for 7 nodes consumes about 17.3 μ J. Analysis of the CORDIC QR algorithm leads us to believe that it will only consume 8.2 μ J, less than half that of the Gram-Schmidt version. This translates directly to a savings of 9.1 μ J for the entire update algorithm, bringing it down to about 34 μ J.

The implementation of a single string of n CORDIC supercells would greatly reduce the overhead incurred when CORDIC is performed on a traditional general purpose architecture. Furthermore, the hardware parallelization of several strings of CORDIC supercells would probably lead to substantial performance gains, but it is not clear that it would have a large effect on the number of operations performed, and thus the power consumed. It could be argued that better performance could allow for a lower clock speed, which would certainly result in energy savings. It is assumed, however, that the triangulation algorithm will not be the most performance-critical code run-

ning on a node in this type of network, and so a performance increase in the triangulation algorithm alone would not necessarily allow for a slower clock speed. Thus, it was concluded that a hardware implementation of a single supercell string could be used to achieve a small reduction in energy consumption, but that several parallel strings would not further reduce energy consumption by any noticeable amount. We were unable to derive any more accurate forecasts for these alterations.

A more efficient means of performing the update mode of the generalized algorithm may be possible using CORDIC. Rader [3] discusses means of updating triangular matrices based on new column inputs. Specifically, he proposes a means of solving a least squares problem by discovering a unitary matrix Q that zeroes out the last column of the constructed matrix $[A|b]$ (from $Ax=b$). The Q matrix is implied by the CORDIC controls used to zero out the b column, and then x is found almost directly from the last column of Q . If the construction of Q from the CORDIC controls is as effortless as Rader implies, this method could be used to replace the entire QR factorization and backsubstitution least squares portion of our standard algorithm. We would then expect to see anywhere from a 25% to a 50% reduction in power consumption. Further research needs to be done on this point though before this claim can be backed up with any evidence.

5.3 Optimized Fixed Point Arithmetic

In our current version of fixed point arithmetic, every multiplication involves two normalization shifts to protect against overflow. Most DSP processors have large accumulator registers to avoid this overhead, allowing for a single normalization shift at the end of a sequence of multiplications. All three of the processors that we have studied contain such an accumulator register.

Another area that can be improved is function call overhead. We originally compiled our code with optimizations turned off to get a fair comparison between the three processors. Optimizations, especially function inlining, should reduce the instruction count significantly since a large percentage of time is spent on function call overhead in very short functions.

Examination of the fixed-point multiply function implementation shows that there are 15 instructions involving function overhead, 13 instructions dealing with absolute value calculations, and 2 shift instructions. These instructions can all be eliminated by making use of the accumulator registers and by inlining functions. A 7 node network on the StrongARM requires about 100 multiplications for standard triangulation. Therefore, we can expect to save approximately 6.8 μ J (100 multiplications \times 30 instructions \times 2.257 nJ per instruction). Similar savings can be expected on the other two processors as well.

Finally, choosing the right fixed-point format can also reduce the number of normalization operations. Using fewer fractional digits reduces precision, but allows for less frequent normalization. There are commercial tools available that will profile fixed-point C code and give run-time statistics, including an overflow count and the needed precision to keep the cumulative error below some given tolerance level [4]. Making use of these tools should help to reduce energy consumption even further.

5.4 Hardware Specific Code

Traditionally DSP code has been written in assembly language because of poor compiler efficiency. Most DSP processor compilers would take high-level language code written in C and create assembly language that would be only about 20-30% as efficient as code written in assembly language directly [5]. We did not attempt to write a QR factorization algorithm in assembly, but instead we inspected the algorithms, confirming our suspicions that the most computationally detrimental code sequence is the multiply/add operation:

```
for (j=0; j<qNODES; j++) {  
    col[i] += b[j]*y[i][j]; }  
}
```

One of the most compelling reasons for running our code on both the StrongARM and 'C62 platforms was ability to use the built-in MAC functionality available on both processors, and yet neither compiler recognized this opportunity in our C code. Considering that we are looping through this code a number of times, it is very important to optimize this line, and so an implementation that is part C and part inline assembly language should be considered for future work.

5.5 Combined Savings Due to Optimizations

The fixed point standard algorithm consumed 43 μ J on the StrongARM processor for a network of 7 nodes. A better fixed point library is expected to save about 7 μ J. If a successful application of CORDIC updates can be achieved, we modestly predict a further 25% energy reduction. Combining these improvements yields an estimated energy consumption of $(43-7) \times 0.75 = 27\mu\text{J}$. As mentioned earlier, customizing the code for the DSP platform would probably help the algorithm's performance, but such a strategy is not expected to reduce energy consumption to a level competitive with the StrongARM until a DSP with a lower core power is employed. On the contrary, customizing code for the StrongARM, and especially making use of the MAC functionality, could result in less energy consumption. The effects of such an effort have not been estimated here.

6. Conclusions

Of the platforms simulated, the StrongARM processor was found to execute the fixed point standard algorithm for 7 nodes with the lowest energy consumption: 43 μ J. It was estimated that the cost of the update mode of the general algorithm could be reduced to 27 μ J or less through optimization of fixed point operations and use of the CORDIC algorithm. Further energy savings could be achieved by customizing the code for the StrongARM platform or using a DSP with a lower core power than that of the 'C62.

7. Acknowledgements

The authors acknowledge the contributions of Jan Raebaey, Bart Kienhuis, Fred Burghardt, Marlene Wan, Vandana Prabhu, and Kostas Sarrigeorgidis to this research.

8. References

- [1] J. Beutel, *Geolocation in a PicoRadio Environment*, MS Dissertation, UC Berkeley, 1999.
- [2] Y. H. Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing," IEEE Signal Processing Magazine, July 1992.
- [3] C. M. Rader, "VLSI Systolic Arrays for Adaptive Nulling," IEEE Signal Processing Magazine, July 1996.
- [4] *Frontier Design's A/RT Library*, <http://www.frontierd.com/artlibrary.htm>.
- [5] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, p. 711, John Wiley & Sons, Inc. 1999.
- [6] *Average Current Per Instruction for SA110*, <http://infopad.eecs.berkeley.edu/~laued/StrongARM/Imeasurements.html>.