Experiences from Porting the Contiki Operating System to a Popular Hardware Platform

George Oikonomou, Iain Phillips
Computer Science, Loughborough University, Loughborough, LE11 3TU, UK
Email: {G.Oikonomou, I.W.Phillips}@lboro.ac.uk

***Abstract*—In contrast to original belief, recent work has demonstrated the viability of IPv6-based Wireless Sensor Networks (WSNs). This has led to significant research and standardization efforts with outcomes such as the "IPv6 over Low-Power Wireless Personal Area Networks" (6LoWPAN) specification. The Contiki embedded operating system is an important open source, multi-platform effort to implement 6LoWPAN functionality for constrained devices. Alongside its RFC-compliant TCP/IP stack (uIP), it provides support for 6LoWPAN and many related standards. As part of our work, we have made considerable fixes and enhancements to one of Contiki's ports. In the process, we made significant optimizations and a thorough evaluation of Contiki's memory and code footprint characteristics, focusing on network-related functionality. In this paper we present our experiences from the porting process, we disclose our optimizations and demonstrate their significance. Lastly, we discuss a method of using Contiki to deploy an embedded Internet-to-6LoWPAN router. Our porting work has been made available to the community under the terms of the Contiki license.**

***Index Terms*—6LoWPAN, RPL, IPv6 for Embedded Devices**

I. Introduction

Over the last decade the wireless sensor network (WSN) research community members have invested considerable effort towards the design of novel network protocols and architectures. Originally, it was believed that internet protocols would not be applicable [1] and that WSN design would benefit by steering away from the abstraction of layered architectures [2]. Thus, significant portion of the development and evaluation effort focused on the link layer, while protocol optimizations have mainly been guided by application-specific requirements.

However, the release of uIP demonstrated the viability of RFC-compliant TCP/IP stacks for embedded devices [3]. Subsequently, the IETF 6LoWPAN working group published RFC 4944, discussing a method of carrying IPv6 datagrams within 802.15.4 low power radio frames [4]. Among other topics, this and subsequent documents specify methods for the fragmentation of datagrams and the compression of their headers. Further work demonstrated that a purely IPv6-based network architecture is not only viable but can also outperform application-centric designs [5]. For those networks, the emerging standard for routing is the "IPv6 Routing Protocol for Low power and Lossy Networks" (RPL) [6]. Lastly, IETF members have been working on optimizing IPv6 neighbor discovery (ND) for 6LoWPAN and similar networks [7].

Meanwhile, the uIP stack was integrated into the Contiki embedded operating system and was enhanced to include IPv6 support (uIPv6) [8]. Additionally, the Contiki OS includes an implementation of the RPL routing protocol [9] as well as mature 6LoWPAN support (IPv6 datagram fragmentation and header compression). Contiki has been designed with portability in mind and has been used successfully on multiple diverse hardware platforms. One of these porting efforts has been for the Sensinode devices, which are based on the Chipcon/Texas Instruments CC2430 and CC2431 System-on-Chip (SoC) solutions. However, the effort was discontinued during the early months of 2010, leaving the port in a defunct state.

As part of our work, we resumed this porting effort; we fixed existing problems, we enhanced it by implementing multiple additional features and we made it available to the community[1]. In this paper we discuss our experiences: We contribute an in-depth analysis of optimizations required to overcome hardware limitations and we analyze the port's code and memory footprint characteristics.

In order to better understand this work, the reader needs to be familiar with the CC2430 architecture. We thus introduce some important concepts in section II. Our fixes and improvements to the Contiki/Sensinode port are presented and evaluated in section III. Motivated by the *Wireless Embedded Internet* vision of interconnected IPv6-based WSNs, in section IV we evaluate an embedded 6LoWPAN-to-IPv6 router implemented with Contiki. This paper concludes with a discussion of future work and open issues in section V.

II. Hardware

Our hardware has been manufactured by Sensinode Ltd and is based on the Chipcon/Texas Instruments CC2430 and CC2431 System-on-Chip (SoC) solutions. These SoCs combine an Intel 8051 micro controller (MCU), 8 KBytes of volatile RAM, up to 128 KBytes of non-volatile flash, a 2.4GHz low power RF transceiver and various sensing elements. Compared to the industry standard 8051, the MCU on the CC2430 offers performance enhancements while using the same instruction set. The most noteworthy improvements are faster instruction execution and the addition of a second data pointer (dptr) [10]. The CC2431 is equipped with a location estimation hardware module but is otherwise identical to the CC2430 [11].

Sensinode devices combine a CC2430-based SoC with additional sensing elements (e.g. 3-axis accelerometer, light sensor), a USB connector and for some models an RS232 connector. They can be powered from battery or over USB.

[1]http://nets-www.lboro.ac.uk/george/contiki-sensinode/

A. Memory Spaces

Memory on the CC2430 is logically separated into four distinct but partially overlapping memory spaces: i) DATA ii) XDATA iii) SFR and iv) CODE. DATA and XDATA are read/write memory spaces for program data and are physically located in the 8 KByte volatile ram during execution. The former can be accessed with a single MCU instruction whereas the latter can only be addressed indirectly, taking 4-5 MCU instructions. The SFR space provides access to the hardware's special function registers (for instance the data, instruction and stack pointers and peripheral-controlling registers). Lastly, the read-only CODE space is used for the application code and constants and is physically located on flash.

B. Code Banking

The CODE memory space on the CC2430 is used to address flash memory, but it can only address up to 64 Kbytes at a time. Expanding CODE beyond this limit relies on a technique called *code banking*, which is necessary for embedded software images with a code footprint greater than 64 Kbytes.

Flash memory on the CC2430 is broken down into up to four 32 KByte blocks. With *code banking*, the lower 32 KBytes of the CODE memory space always address the first block (also called *common area*) while the higher 32 KBytes can switch, during execution, from addressing one block to addressing another. The side effect of this technique is that it is impossible to directly invoke code in one switched area from another. Function calls (and returns) across switched banks are implemented by invoking intermediate code (often called a *trampoline*) which resides in the common area. This mechanism adds an overhead of additional MCU cycles per call, which is why functions used very frequently should be positioned in the common area when possible. An additional side effect of code banking is that interrupt service routines must always reside in the common area. The exact details of how to develop bankable software are toolchain-dependent, usually rely on compiler hints and language extensions and require additional effort from the code author. This is discussed further in sec. III-A.

III. The Contiki Sensinode Port

The contiki code repository contains an existing port for older CC2430-based Sensinode devices, supporting basic functionality and designed to build with the Small Device C Compiler (SDCC), discussed in the next section. However, since before the release of contiki version 2.4 (Feb. 2010), the sensinode port has been in a defunct state: it fails to compile due to API changes, toolchain incompatibilities and non-standard code. Code banking is not fully configured, leading to bugs which only appear at runtime. Additionally, enabling IPv6 results in frequent stack overflows during execution, further discussed in sec. III-D.

As part of our work, we have solved all compilation and linking errors in the original code and we have added more comprehensive banking support, permitting us to successfully execute images with code footprint of more than 100 KBytes (sec. III-B). Additionally, we have extended the port by implementing drivers for additional features, including Analogue to Digital Conversions (ADC), all sensing elements (accelerometer, light sensor, voltage and temperature sensors), hardware watchdog timer, general purpose buttons and the CC2431 location engine [11], [12]. Lastly, we have implemented optimizations in order to reduce stack usage and avoid overflows (sec. III-D).

A. The Small Device C Compiler - Memory Models

Due to the 8051 MCU's unique characteristics there are only a handful of toolchains available, most of them being commercial products. The most noteworthy open source solution is the Small Device C Compiler (SDCC), a "*... retargettable, optimizing ANSI - C compiler that targets the Intel 8051, Maxim 80DS390, Zilog Z80 and the Motorola 68HC08 based MCUs.*" [13]. The 8051 target is actively maintained and developed, providing very good support for the MCU. The toolchain also provides support for code banking, as discussed in section II-B. However, it is unaware of the CC2430's architectural enhancements and as a result can not exploit features such as the presence of a second data pointer.

When building a project with SDCC, the developer has a choice of four different *memory models*: small, medium, large, huge. The choice of memory model influences the compiler's variable allocation algorithm as well as code banking. Due to the size of the contiki operating system, the small and medium model are not suitable. With the *large* memory model, functions residing in switched banks must be marked as such by the developer with a keyword. When encountering a call to one of those functions, the compiler automatically replaces the call with a *trampoline* invocation. With the *huge* model, SDCC treats all functions as if they resided in a switched bank and replaces all function calls with a banked call. This simplifies the development process considerably, since the code author no longer has to use keywords to flag banked functions. However, this method also increases code size considerably and adds some additional burden on the stack. In the following sections, we thoroughly evaluate the impact of code banking with both methods on code size as well as stack utilization.

B. Port Configuration and Footprint

Our contiki port has been configured to use the CSMA module for the MAC layer and the sicslowmac module for the radio duty cycling (RDC) layer (which effectively keeps the radio always on). In the 6LoWPAN adaptation layer we use the most recent header compression scheme available in contiki. The network stack is configured with UDP and ICMPv6 enabled, while routing is handled by RPL.

In Table I lists the code and memory footprint for various components of the contiki operating system and for both SDCC memory models. One observation of interest is that when RPL is in use, code footprint increases considerably. Note that this only includes message generation and handling but does not include routing table processing nor packet forwarding (those

TABLE I
FLASH AND RAM USAGE IN BYTES WITH CODE BANKING (LARGE / HUGE)

	In Flash		In RAM
	CODE	CONST	XRAM
System			
Core	7750 / 8153	0 / 0	348 / 349
Standard Libraries	2649 / 2661	0 / 0	0 / 0
Timers	5596 / 6026	0 / 0	33 / 35
Architecture Specifics (Port)			
RF driver	2358 / 2595	20 / 30	3 / 3
Sensor drivers [a]	1669 / 1992	49 / 58	27 / 27
Other drivers [b]	1565 / 1708	0 / 0	13 / 14
Network			
Buffers	2917 / 3243	0 / 0	2087 / 2087
Data Structures [c]	8687 / 9148	0 / 0	778 / 778
UDP	2474 / 2726	0 / 0	25 / 27
uIPv6	4422 / 4563	0 / 0	550 / 550
ICMPv6	1320 / 1374	0 / 0	16 / 16
Neigh. Disc.	7460 / 7869	0 / 0	275 / 277
6LoWPAN	7565 / 7997	36 / 38	25 / 25
MAC	5445 / 5848	20 / 26	118 / 123
Radio Cycling	1105 / 1370	26 / 32	55 / 55
Routing (RPL)	20891 / 21537	0 / 0	721 / 730
Border Router	903 / 1020	10 / 12	68 / 69
SLIP	1962 / 2214	25 / 27	291 / 293
Totals [d]	88071 / 93645	279 / 314	5513 / 5538

[a] ADC and button drivers
[b] UART, clock, timers, watchdog
[c] Network interface and address table, routing table, ND table
[d] Some modules are included in the totals but not explicitly listed (e.g. debugging functions)

are listed in separate categories). A second important observation is that while building contiki with the huge memory model makes code banking trivial, it also has an impact on code size. We further analyze this in the next section. Lastly, for simplicity reasons, we have currently configured the port with TCP disabled. Enabling it adds approximately 11 KBytes of code and increases ram usage by about 600 bytes. This suggests that it would be feasible to fit an image with TCP on the device flash. This would require banking configuration and potentially some stack optimizations similar to the ones discussed in section III-D.

C. Impact of Banking on Code Footprint

In order to evaluate the impact of banking on code footprint, we built sample firmware images with three methods: i) large model without banking (only possible when image size is $<$ 64KB), ii) large model with banking, iii) huge model (thus, implicitly with banking). The total code footprints for each image and method are summarized in Table II, with column one containing the image name. *Blind Node* is the image that we used for the evaluation of the CC2431 Location Engine (which is not discussed here due to space limitations). *Border Router* is the software discussed in sec. IV and *UDP Server* was used for the stack depth evaluation in sec. III-D. Notice that two of the aforementioned images have too large a footprint and may not be built without banking support (*N/A* cells in the table). The *% Increase* column lists the percentile increase in code size caused by the huge memory model

TABLE II
IMAGE CODE FOOTPRINT (IN BYTES) WITH AND WITHOUT BANKING

	Large No Banking	*Large Banked*	*Huge Banked*	*% Increase*
Blind Node	40658	41091	45006	9.53%
Border Router	N/A	85015	90348	6.27%
UDP Server	N/A	88716	94127	6.10%

TABLE III
SUMMARY OF PER-FILE BANKING OVERHEAD ON CODE FOOTPRINT

	Large No Banking	*Large Banked*	*Huge Banked*
Sample	40 object files		
Average Code Footprint	1919	1964	2110
Average % Increase	N/A	2.35%	9.98%
Minimum % Increase	N/A	0.40%	2.13%
Maximum % Increase	N/A	26.09%	51.78%

compared to the large model with banking. We observe that the huge memory model induces a code footprint overhead of between approximately 6% and 10%.

To better understand the exact huge model overhead, we further analyzed it on a per-file basis, using a sample of 40 object files used by our builds. Table III summarizes the results. We found out that, compared to non-bankable code, the average increase in code size was 2.35% when building with the large model and banking support. With the huge memory model, the average increase across the same files is 9.98%. For the same model and sample, the values had high variance ranging from as low as 2.13%, up to a 51.78% increase. The explanation for this diversity is that the code increase for each file depends on the number of function definitions and the number of outgoing function calls. Therefore, a very large file but containing only a single function without many external calls will increase less in size than a smaller file with many function definitions and many external calls. Furthermore, a file with many calls to functions among which only 1 is bankable will increase in size a lot with the huge model (since all calls will be treated as if they were bankable) but considerably less so with the large banked model (since calls to non banked functions will result in zero overhead).

D. Stack Depth Optimizations

As discussed in section II-A, the CC2430 memory layout defines a fast access, 256 byte memory space (DATA). This is shared among variables allocated to internal RAM, MCU register banks and all bit-addressable variables. The remainder is occupied by the stack, leading to a theoretic maximum stack size of 256 bytes. Our builds allocate 223 bytes for the stack, which is very limited and poses a very significant challenge. After having written all the necessary hardware drivers, we found out that enabling 6LoWPAN and RPL leads to very deep function call hierarchies during execution, resulting in frequent stack overflows and node crashes.

In order to circumvent this problem, we implemented a series of optimizations. The first set, which we will call *Stage 1*,

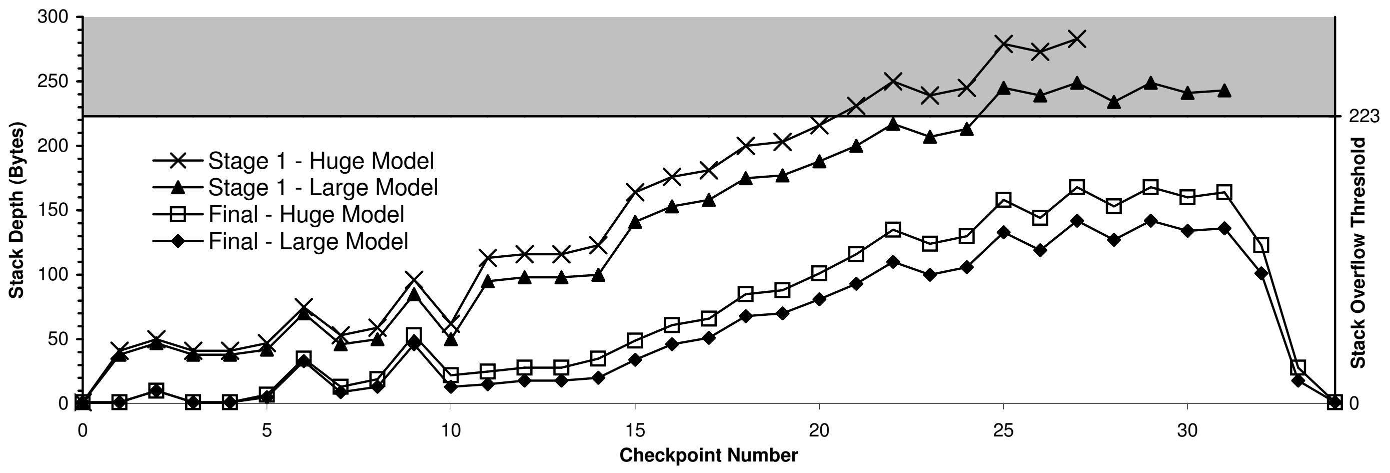

Fig. 1. MCU stack depth evolution over time, starting with the reception of a data packet. The *white area* represents normal operation whereas the *grey area* represents overflows.

involved changing the declaration of large variables in multiple positions of the RPL code. Examples of large variables are those used to hold IPv6 addresses (16 bytes) or the information stored in an RPL DIO message (80 bytes). When those variables are declared dynamically with local scope, they get allocated on stack at runtime. With a maximum stack size of 223 bytes, this allocation policy results in frequent overflows. In our port we have moved many of those variables to the external RAM memory space, significantly reducing stack burden at a trade off of slightly larger memory footprint. For instance, compared to the original code, *Stage 1* optimizations reduce maximum stack depth by 97 bytes during the processing of incoming RPL DIO messages, thus preventing a stack overflow.

The next change involved re-writing the radio driver for the CC2430. Instead of implementing it as a contiki process receiving asynchronous events when a network packet arrives, we directly poll it from the `main()` function periodically. The third change revolves around contiki's implementation of synchronous events, which results in significant stack overhead. For some code execution branches, it is possible to predict which process is meant to receive what event. It is thus possible to replace the event post with a simple call to the receiving function.

To better illustrate the stack overflow issue and the significance of our optimizations, we placed a series of checkpoints in the code, one at each layer of the network stack and in various points in the RPL code. In Contiki, we implemented a very simple UDP echo server and issued request messages from a remote client. Upon reception of a request, we observed and recorded stack depth at each of the aforementioned checkpoints. We repeated this experiment with *Stage 1* optimizations only as well as with the final version of the code (all optimizations in place), using both SDCC's memory models. Please note that it was impossible to reproduce this measurement with the original code since, for the reasons outlined earlier, a stack overflow would occur upon the reception of the first RPL DIO message (long before reception of our UDP client's request).

Fig. 1 illustrates the results. Reception of a UDP datagram from the client ($X = 0$) triggers a series of events, displayed from left to right on the X axis. The packet is forwarded up the network stack until it gets delivered to the server process ($X = 15$). The server generates a response which moves down the network stack until it gets transmitted by the radio driver ($X = 22$). This is followed by a series of callback notifications before the execution branch returns. The *horizontal line* at 223 bytes represents the stack overflow threshold. The lines with *crosses* and *triangles* depict stack behaviour after *Stage 1* optimizations, while *squares* and *diamonds* illustrate the final optimized code. Notice how without all optimizations, stack overflows occur regardless of memory model (lines above the overflow threshold have been inferred from the corresponding non overflowing ones).

In the figure, one can observe three points where the lines diverge abruptly. The first one at $X = 1$ is due to our change in the implementation of the radio driver, as discussed above. This optimization results in a 37 byte difference in stack depth when using the large memory model (40 bytes with huge). The second and third points at $X = 11$ and $X = 15$ are due to our synchronous events optimization, reducing stack depth with the large memory model by 43 and 27 bytes respectively (48 and 29 when building with the huge model). Including some other minor changes (e.g. bypassing redundant function calls), for this specific execution branch we have achieved maximum stack depth reduction by 120 and 129 bytes respectively for the two memory models. This is a quite significant quantity, considering that the total size is only 223 bytes.

IV. Connecting the 6LoWPAN to the Internet

For data collection, management and monitoring purposes, it is often desirable to connect a 6LOWPAN to the internet. To achieve this, a gateway device is necessary in order to forward packets to and from the WSN. On the wireless side, the gateway can run RPL or any other routing protocol for 6LOWPANs, while on the internet side it will run some other

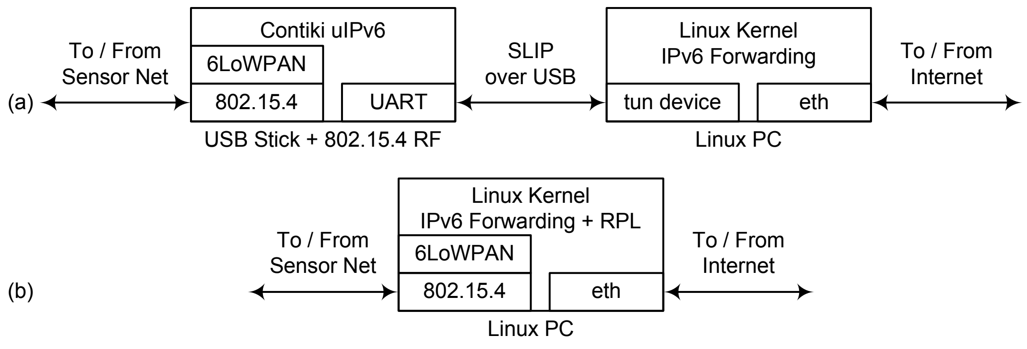

Fig. 2. Two methods of connecting the 6LOWPAN network with the internet: MCU-hosted gateway (a) or kernel-hosted gateway (b). Both approaches make the wireless sensor network seamlessly accessible from the internet over IPv6.

standard routing mechanism.

It is possible to achieve this functionality with a PC, by attaching an 802.15.4 device to it (e.g. over USB). The PC's kernel will handle IPV6 forwarding between the internet and the sensor network. It will also run the routing protocol on the internet side. For the 6LOWPAN side, there are two approaches as to which device will handle the routing and IPV6 logic i) hosted by the PC ii) hosted by the MCU.

In the former case, the entire logic is handled by the computer's kernel, while the wireless device only handles packet transmission and reception [Fig. 2 (b)]. In order to achieve this setup, one needs a driver for the device and a software implementation of 6LOWPAN functionality for the computer's operating system (routing, IPV6 header compression, 6LOWPAN ND). This approach may sound straightforward but there are caveats. Crucially, the only mature 6LOWPAN and RPL implementations have been written for embedded operating systems such as Contiki. For example, while some efforts have been made to add 6LOWPAN functionality to the linux kernel, so far they have not been widely adopted. The same applies for RPL implementations while there are only a few drivers which would permit directly attaching 802.15.4 devices to a PC.

The alternative is to host all the 6LOWPAN logic on the embedded device [Fig. 2 (a)]. In this case, RPL-generated routing tables and IPV6 neighbor tables are stored on the wireless device's RAM and all processing is performed by the micro controller. This includes IPV6 header compression and all software functionality required for the correct operation of the radio transceiver. In order for this to work, it is necessary to configure a virtual network device (tunnel) which will be used for the exchange of packets between the USB device and the computer's kernel. Lastly, the USB device will need to forward all traffic from the USB to the 6LOWPAN and vice-versa. While slightly more complex, this approach has the advantage that it capitalizes on existing and relatively mature 6LOWPAN and RPL implementations.

Table IV summarizes the advantages and disadvantages of each approach. The *MCU hosted* approach suffers from the restrictions imposed by embedded devices. Memory limitations restrict the size of routing and neighbor tables while MCU speeds have a negative impact on the performance of the routing algorithm, packet processing and forwarding.

TABLE IV
TWO APPROACHES TO IMPLEMENT RPL BORDER ROUTERS IN LINUX

	MCU Hosted	*Kernel Hosted*
Mature 6LOWPAN	Yes	No
Mature RPL	Yes	No
Device Drivers	Many Ports	No
Performance	Low	High
Hardware Restrictions	Severe	-
Complexity	Moderate	Low
Device-to-PC Interface	PPP/SLIP	Directly attached (e.g. USB)

A. Evaluation of a Contiki Embedded Router

We configured our port discussed in sec III and built an MCU-hosted IPV6-to-6LOWPAN router. The image is for the Sensinode N601 NanoRouters, equipped with a CC2430 system on chip and a USB interface. On the wireless side, our router uses RPL while on the internet side, the linux host can implement any standard routing mechanism. Communication between the embedded device and the hosting Linux PC is achieved over the Serial Line Internet Protocol (SLIP) (over USB). Subsequently, we investigated this solution's scalability with increasing network size.

The border router's total code footprint is approximately 85 KBytes, with SLIP forwarding functionality occupying around 1 KByte (line *Border Router* in Table I). Memory footprint is more interesting, since it directly influences the scalability of this approach. As the number of nodes in the 6LOWPAN increases, so do the size requirements for the routing and neighbor tables on the border router. In Contiki, the maximum size of each of those tables is configurable and space gets pre-allocated *statically* in RAM (XDATA memory space). Static allocation has the advantage that the size is constant and predictable: new entries will not cause the table to expand, they will merely occupy some space in the pre-allocated area. The drawback is that special attention must be paid during configuration to prevent under (or over) allocation. On the CC2430, total external RAM space is 7936 bytes. The router firmware image has 5059 bytes allocated for storage of all

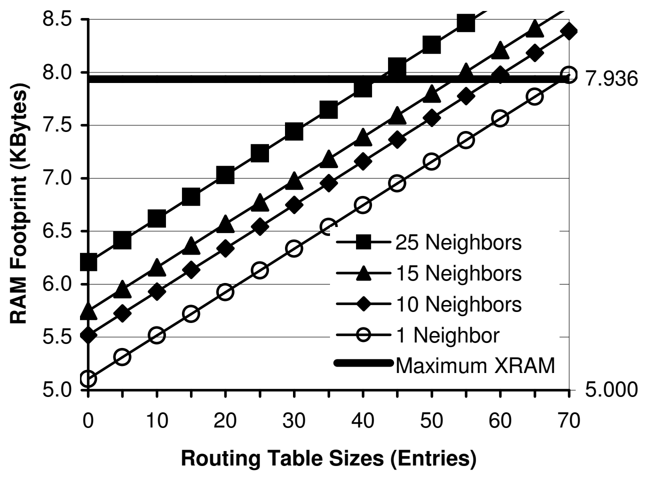

Fig. 3. RAM usage for different sizes of the neighbor and routing table.

necessary variables, leaving a total of 2877 bytes available for the two tables. Choosing correct configuration values can be a balancing act. The normal approach would dictate for more routes than neighbors. However, in a dense network topology (e.g. extended mesh) it may prove necessary to decrease the number of routes in favor of more neighbors.

In Fig. 3, we plot the total external RAM footprint increase for different configurations. The *thick horizontal* line at 7936 bytes represents the maximum possible XRAM allocation (configurations resulting a larger footprint will cause a linker error). Fig. 4 presents a different view of the same information. Each additional route occupies 41 bytes while each neighbor 46. Thus, a configuration with r routes and n neighbors is valid if $41r + 46n \leq 2877$. In the figure, this restriction is represented by the *white area* while the *negative slope line* represents the area's limits. Any combination within the *shaded area* would result in a linker error.

V. Conclusions and Future Work

While IPv6-based protocols and algorithms have been proposed and evaluated, most of the work has so far taken place in simulated environments. On the other hand, the most significant large scale WSN deployments have been using application-specific, link layer-based design. Thus, the community currently lacks understanding of how well the emerging standards for 6LoWPANs will work in a real field environment. Our future planning includes field experiments and evaluation of 6LoWPAN and RPL in terms of scalability, performance and energy requirements with increasing network size and traffic. We are also in the process of implementing our own version of the Constrained Application Protocol (CoAP) for Contiki. Evaluating this protocol in the field and enhancing it or suggesting alternatives is also part of our future plans. The work discussed in this paper will form the basis of our future efforts.

The current Contiki implementations of the embedded IPv6 stack and the 6LoWPAN adaptation layer have reached a stage of maturity. The RPL implementation adheres to recent versions of the internet draft to great extent but it would benefit from code size and memory usage optimizations. Since the RPL specification has been moving forward with leaps over the past months, it would make sense to delay any such optimizations until the specification becomes more mature. Lastly, one feature that is currently missing from Contiki, is support for 6LoWPAN ND.

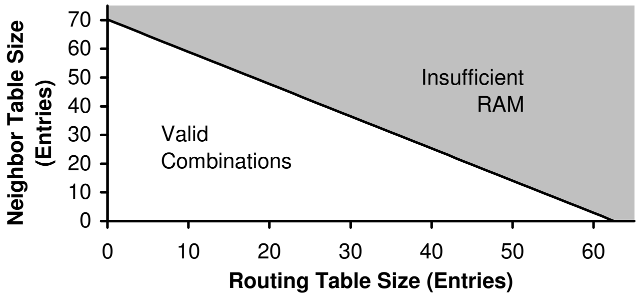

Fig. 4. Valid combinations for neighbor and routing table sizes fall within the boundaries of the *white surface*.

References

[1] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next century challenges: Scalable coordination in sensor networks," in *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*. Seattle, Washington, USA: ACM, Aug. 1999, pp. 263–270.

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Not.*, vol. 35, pp. 93–104, November 2000.

[3] A. Dunkels, "Full TCP/IP for 8-bit architectures," in *Proc. 1st international conference on Mobile systems, applications and services - MobiSys '03*, May 2003, pp. 85–98.

[4] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 packets over IEEE 802.15.4 networks," RFC 4944, Sep. 2007.

[5] J. W. Hui and D. E. Culler, "IP is dead, long live IP for wireless sensor networks," in *Proc. 6th ACM conference on Embedded network sensor systems (SenSys '08)*. New York, NY, USA: ACM, 2008, pp. 15–28.

[6] T. Winter, Ed., P. Thubert, Ed., A. Brandt, T. Clausen, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, and J. P. Vasseur, "RPL: IPv6 Routing Protocol for Low power and Lossy Networks," IETF Internet Draft, Oct. 2010.

[7] E. Shelby, Zach, S. Chakrabarti, and E. Nordmark, "Neighbor discovery optimization for low-power and lossy networks," IETF Internet Draft, Dec. 2010.

[8] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels, "Making sensor networks IPv6 ready," in *Proc. 6th ACM conference on Embedded network sensor systems (SenSys '08)*, Raleigh, North Carolina, USA, Nov. 2008.

[9] N. Tsiftes, J. Eriksson, and A. Dunkels, "Poster abstract: Low-power wireless IPv6 routing with ContikiRPL," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010)*, Stockholm, Sweden, Apr. 2010.

[10] "A True System on Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee®," CC2430 Data Sheet (rev. 2.1), May 2007.

[11] "System on Chip for 2.4 GHz ZigBee® / IEEE 802.15.4 with Location Engine," CC2431 Data Sheet (rev. 2.0.1), May 2007.

[12] K. Aamodt, "CC2431 Location Engine," Texas Instruments Application Note AN042 (Rev. 1.0), Jul. 2006.

[13] "Sdcc - Small Device C Compiler," retrieved 2011. [Online]. Available: http://sdcc.sourceforge.net/