

# Thresholding for Optimal Data Processing in a Software Defined Radio Kernel

Michael L Dickens, Brian P Dunn, and J. Nicholas Laneman  
Department of Electrical Engineering  
275 Fitzpatrick Hall  
Notre Dame, IN, USA  
{mdickens, bdunn2, jnl}@nd.edu

**Abstract**—We discuss a developing software-defined radio framework and kernel, *Surfer*, which, among other changes compared with known current frameworks, moves control of when to process a block and how much data to process into the blocks themselves via buffer thresholds. Varying threshold values shows a trade-off between data processing latency and throughput, with respect to overhead time. Processing and overhead timing statistics are collected during runtime and can be retrieved by a user’s constraint function in order to allow for user-based optimal data processing. A simple waveform example is included to demonstrate system functionality as well as provide insights into the latency-throughput tradeoff.

**Index Terms**—Software Defined Radio, Framework, Signal Processing Time, Overhead Time, Optimal Signal Processing, Optimal Latency, Optimal Throughput, Latency / Throughput Trade-off

## I. INTRODUCTION

IN the kernel of any software-defined radio (SDR) implementation, no matter if processing data streams or packets, time is spent both in performing overhead tasks and in processing data streams or packets. Assuming the traditional directed acyclic graph data-flow signal-processing application in which nodes and edges represent processing blocks and dependencies, respectively, the primary responsibilities of the SDR kernel include verifying that there are no cycles in the graph, setting up and otherwise handling data buffers for each block, and controlling the flow of data through the graph. The controller must determine a number of factors including when a block is ready to be processed, how much data to process, and where the data should be processed. These factors can be determined a priori or during runtime, and this paper investigates the runtime determination of the first two factors.

The need for computing these factors motivates consideration of how a centralized single-threaded controller becomes a processing bottleneck, and how

this bottleneck can be mitigated by distributing these computations into other areas of the SDR kernel. The resulting SDR implementation becomes multi-threaded and distributable, able to leverage emerging multi-core and many-core processor architectures. We provide background for this discussion in Section II. Our technique for spreading the controller overhead load throughout the SDR kernel revolves about (1) assigning a threshold value to each input and output buffer; (2) separating blocks into paired kernel-space and user-space classes, i.e., overhead and signal processing tasks; (3) separating blocks from block processing threads – which we call runners; and (4) using a pool of runners, each possibly assigned to a specific CPU or core. As this paper investigates an SDR kernel implementation, we generally use the term *block* to refer to the kernel-space class.

As data is read from or written into each buffer, the blocks using that buffer are independently responsible for keeping track of whether the buffer has met its threshold, i.e., if the amount of data in the buffer is above or below the threshold value, for input and output buffers respectively. By assigning a threshold value per buffer, the computations to determine when a block has met all of its thresholds and is ready to be processed, and how much data to process, becomes an a priori decision. Further, because each block determines whether or not it is ready to process independently of every other block, the load for this overhead factor is spread across all runners. By separating blocks from runners, and using a pool of runners, multiple blocks can be processed as an interval on a given runner, providing graph-theoretic optimal data processing [1]. Further, each runner can execute on any processor known to the kernel, allowing for the possibility of heterogenous signal processing.

In order to demonstrate our approach, we have created a new SDR framework and kernel, *Surfer*, a high level overview of which is further described in

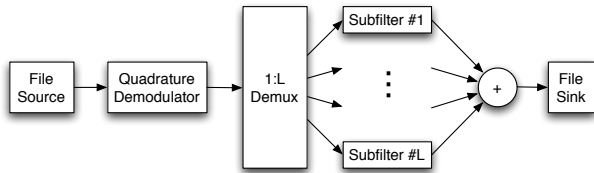


Fig. 1. Data-flow graph of a downsampling narrowband FM demodulator

Section III. The use of thresholds, whether assigned statically a-priori or controlled dynamically during runtime, allows for the possibility of signal processing optimized for some specific constraint, e.g., minimum latency or maximum throughput, while still maintaining a minimum quality of service, e.g., a minimum sample rate. Throughout this paper, we provide some example results of *Surfer* runtime statistics collection, and discuss how those statistics can be used for setting thresholds for optimal signal processing. We summarize our work to date in Section IV, and discuss the anticipated evolution of *Surfer* in Section V.

## II. BACKGROUND

Modern SDR frameworks, e.g., GNU Radio [2] and JTRS [3], implement a user's application by connecting a set of computation blocks together into a directed acyclic graph. Each block performs a simple computation, such as adding together element by element two vectors of 32-bit integers. Chaining simple blocks to form this graph allows for the creation of simple or complex applications, e.g., an FM transmitter, a repeater, or an HDTV receiver. Some frameworks, e.g., JTRS', generate each application indirectly, via a markup language that defines the block, connections, and other variables, while other frameworks, e.g., GNU Radio, generate each application directly using a standard application programming interface (API) via a scripting or compiled language, e.g., Python or C++. No matter how the application is defined or programmed, a graph abstraction can be created from it. Figure 1 provides an example graph for the narrowband FM (NBFM) demodulator that will be used as an example throughout this paper to demonstrate system functionality as well as provide insights into optimal data processing.

Given the graph, signal processing occurs as data flows through the graph from source(s) to sink(s). Each source retrieves data, e.g., from audio hardware, a file, a TCP/IP socket, and places it into an output buffer that is used solely by this source and any

blocks connected to it. The data generated by this source is then read from this same buffer by each block connected as an output to this source; the input data is processed and the resulting generated data written to this block's output buffer(s). Once all of these blocks have read the data, then the space the data occupies is freed allowing more data to be written by that source. In this manner, data flows through the graph from block to block and eventually into the sink(s) where data processing terminates, e.g., writing the data to audio hardware, a file, a TCP/IP socket. The graph can be viewed as a pipeline, with each stage representing a particular block and the size of the stage being the amount of data handled; the total pipeline length is the end-to-end data-processing latency. Hence, an important property of this data-flow is how much data is processed by any specific block: smaller processing amounts across all blocks result in a shorter pipeline and lower latency while larger amounts result in higher end-to-end latency.

Executing a graph using an SDR kernel results in both overhead and data processing time. Overhead time is spent on tasks including block creation and deletion, verifying no graph cycles with the addition or removal of blocks, updating buffer variables, e.g., offsets, pointers, checking for overflow or underflow, when new data is written into a buffer or current data is read from a buffer, and checking to see when a given block is ready to be processed, i.e., when enough input data and free output buffer space are simultaneously available. For the purposes of feedback-control of processing conditions, overhead time would ideally be independent of the time spent performing signal processing and input and output buffer sizes so-as to be able to remove it from control computations. That said, control can still be performed if all timing statistics are known.

In some SDR kernels, e.g., the GNU Radio single-threaded scheduler, most of the runtime overhead tasks are handled by a centralized controller executing in a single thread. The controller represents a necessary component of any SDR kernel, its execution is purely overhead time, and it often presents a processing bottleneck: as the number of blocks grows, the load on this single controller also grows, eventually saturating the controller with tasks and introducing a delay in handling any given task [4]. This type of controller provides scalable performance as the number of blocks grows up to the saturation point. In order to provide scalability beyond this point, the controller functionality must be distributed across mul-

multiple threads; GNU Radio provides this functionality via its thread per block scheduler, by giving each thread / block with its own single-threaded scheduler. Assigning a single thread for all blocks and a single thread per block represent extremes on the continuum of block and thread allocation, with the former requiring the least system resources and the latter the most. A resource-use compromise made on some modern multi-threaded systems is to create a pool of threads, and assign tasks for processing as needed and as threads are available in the pool [5]; *Surfer* uses this methodology with a pool of runners and blocks.

When a block is ready for processing, overhead time must be spent determining how much input data will be provided to the block and output data generated by it. Assuming that overhead time is roughly constant no matter how much data is processed, then less data will generally result in lower individual block input-to-output latency (seconds) but also lower throughput (items / second), while more data will generally result in higher latency but also higher throughput. In real hardware resources are finite, hence increasing the amount of processed data will eventually reach a saturation point, e.g., when the processor reaches maximum utilization or the amount of data no longer fits into local processor caches, at which point throughput will decrease from its peak and latency will increase significantly.

The overhead time spent determining the block processing factors will vary with the total number of block buffers as well as each block type. Thus there will always be *some* trade-off between latency and throughput with respect to the overhead time and amount of data processed, and, given the correct runtime information about these trade-offs, data processing can be optimized in terms of processing time versus overhead time. In order to provide the end-user access to optimizing this trade-off, as well as to investigate many other SDR implementation concepts, we have developed a SDR kernel, *Surfer*, trying to combine the best of current SDR kernels, as discussed in the next section.

### III. RELEVANT *Surfer* FUNCTIONALITY AND MAIN RESULTS

*Surfer* is a nascent SDR kernel, with enough current functionality to show off certain benefits including the potential for optimal data processing. An overview of specific parts of *Surfer* and some of its application programming interfaces (APIs) is shown in Figure 2, and suggesting some of the extensibil-

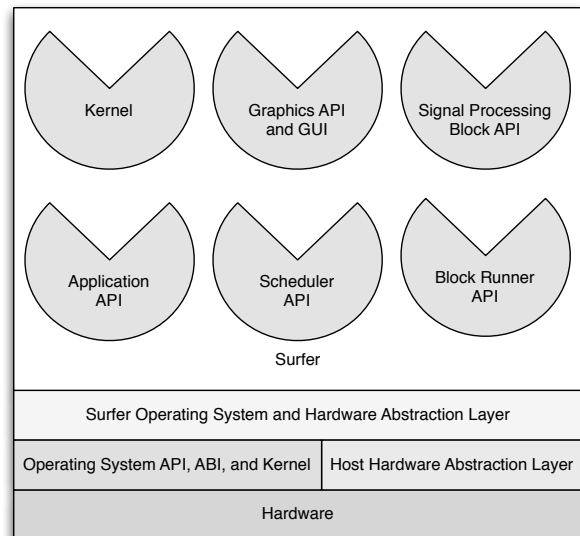


Fig. 2. Overview of *Surfer* and some of its APIs

ity provided to the user: any API can be easily overloaded by the user via C++ inheritance. Hence the user can experiment with a new scheduler, write different signal processing blocks, or even create new runners for processing blocks. *Surfer* was designed with certain goals in mind; those that are most relevant to this paper are now discussed.

#### A. Minimal Background Requirements

*Surfer* is written in C++ and uses STDC++ for various functionality. The GNU Autotools are required to configure and build *Surfer*, and the host operating system must provide a POSIX interface. Certain OS-based functionality, such as user-space context switching and thread cpu affinity, are used when available but otherwise ignored. For the purpose of minimizing compile-time complexity, we separate *Surfer* functionality into two primary components: an OS compatibility layer and the *Surfer* kernel itself that solely depends on the OS compatibility layer. This separation of kernel from OS allows for much easier porting to other OSs and that *Surfer*-based applications can be made OS-independent, thus providing significant application portability.

#### B. Runtime Statistics Collection

As part of the overhead in *Surfer*, each block has built into it methods to calculate timing statistics for both overhead tasks and data processing computations. The timing statistics are available to the user's application, and could be used to dynamically change

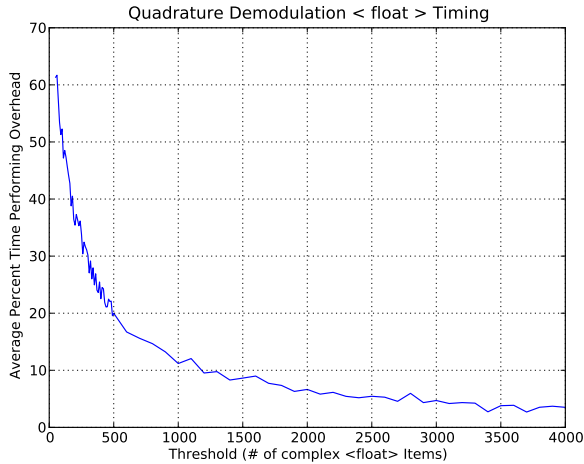


Fig. 3. Timing of unoptimized quadrature demodulator, as percent of the execution time spent as overhead, in *Surfer* executing on an Apple MacBook Pro 2.6 GHz running Mac OS X 10.5 and only standard background applications

threshold values to optimize data processing according to various criteria. Currently, statistics collection happens during runtime only, and only recent data samples are used to compute the statistics. Knowing the threshold value, these statistics can be used to compute average throughput per block or for the graph as a whole.

Using *Surfer* to execute the NBFM decoder application illustrated in Figure 1, we collected runtime timing statistics using various threshold values from 50 to 4000 items (of input type complex <float>). The timing statistics for the quadrature demodulator block alone are plotted in Figure 3 as the average percent of overhead time relative to the average total time for processing at the given threshold; other blocks provide similar statistics. Examination of the data shows that at low threshold values overhead time is actually greater than processing time, while at high threshold values overhead time decreases to less than 5% of the total time. Thus, purely from the computational efficiency perspective, higher thresholds result in higher efficiency. Looking at the timing plot in Figure 4, the overhead time remains essentially constant for all threshold values. Combining this result with the prior plot, the average processing time grows almost linearly with threshold value.

### C. Spreading of Overhead via Thresholds

In order to provide a compromise between system resource use and performance, *Surfer* moves away from the single-threaded centralized controller, pushing almost all of the decision functionality into blocks

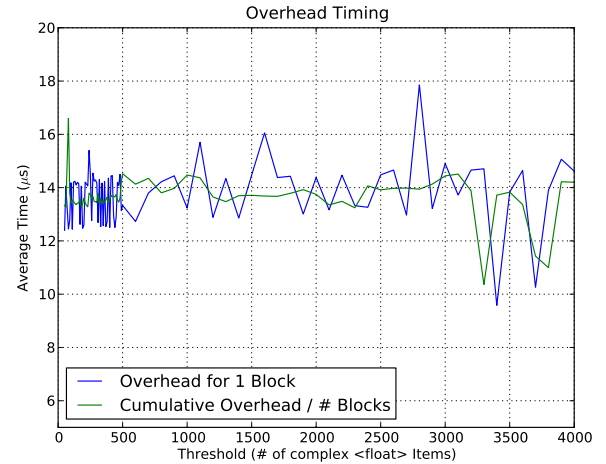


Fig. 4. Illustration of per-block overhead timing in *Surfer* executing on an Apple MacBook Pro 2.6 GHz running Mac OS X 10.5 and only standard background applications

via the use of thresholds. As a result, overhead time is roughly constant for any given block and is proportional to the number of blocks, as demonstrated in Figure 4. There is no bottleneck in the decision of processing factors, because these calculations are spread across the whole pool of runners, effectively creating a distributed controller and providing load-balancing.

Each buffer is paired with a processing threshold that determines when there is enough data in the buffer for the block to which it is attached to be processed. For input (output) buffers, once the amount of data is above (below) the threshold, the paired block is set as ready to process. Once all buffers attached to a given block are ready to process, then the block delivers itself to a subsystem for processing a threshold's worth of data. As depicted in Figure 5 for the quadrature demodulator in our example, low threshold values result in a low average latency but also low throughput, while high threshold values result in high latency. For this particular experiment, the throughput saturates at only moderate threshold values. Currently, threshold values are set a-priori by the user's application.

Figure 5 presents a trade-off that will be found in all SDRs: for a given throughput, assuming it can be handled by the CPU handling processing, there is a minimum latency, and hence a minimum threshold value as used in *Surfer*, at which that throughput can be achieved. For the case of our example, the throughput and timing statistics for both individual blocks as well as the whole graph are well represented by the

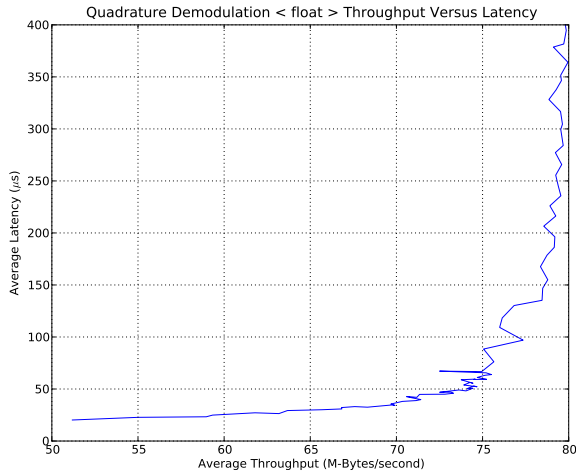


Fig. 5. Latency versus throughput of an unoptimized quadrature demodulator executing on an Apple MacBook Pro 2.6 GHz running Mac OS X 10.5 and only standard background applications

figures in this paper, and thus can be used in a relatively simple fashion to provide optimal data processing. That said, if the latency is not small enough for the required throughput, or vice versa, then the given processor cannot be used with the given blocks in real-time for this application. In this case, options are to upgrade to a faster processor or devise new block implementations to provide higher throughput while not significantly changing latency.

#### IV. CONCLUSIONS

We have developed initial functionality of *Surfer*, an SDR framework and kernel that uses buffers threshold values to control when a block is processed and how much data to process, and decouples blocks from processing threads. By moving this control into the blocks themselves and processing blocks using a pool of threads, overhead time is spread among the thread pool, resulting in a more balanced overhead processing load and eliminating a potential bottleneck. By varying threshold values, we have shown a trade-off between data processing latency and throughput with respect to overhead time. *Surfer* integrates processing and overhead timing runtime statistics collection, and makes these values available to the user to allow for optimal data processing. We demonstrate the potential for user-controlled threshold values via a simple waveform, which demonstrates system functionality as well as provide insights into the latency-throughput tradeoff.

#### V. FUTURE WORK

Although *Surfer* is still in early stages of development, there are plenty of directions in which we will be expanding it in the future. With each added feature, it is our goal to be enhancing the kernel's functionality while decreasing complexity for both end users and developers. Some areas for expansion include:

- controlling thresholds via a monitor thread dynamically;
- providing both runtime scheduling and a-priori elastic scheduling;
- compiling a user's application into a static executable;
- saving and loading of statistics, to allow for optimal a-priori scheduling;
- using thread cpu affinity and memory affinity;
- augmenting statistics collection to include data transport latency;
- maximize use of a-priori known information (connection data type and rate; graph connections) in order to provide 'optimal' SDR performance according to a user's constraint;
- constraining where a given block will be processed via user-defined functions; and
- creating runners for a hybrid cpu / gpu SDR.

#### REFERENCES

- [1] O. Sinnen, *Task Scheduling for Parallel Systems*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2007, Wiley Series on Parallel and Distributed Computing.
- [2] GNU Radio Website, accessed January 2010. [Online]. Available: [www.gnuradio.org](http://www.gnuradio.org)
- [3] Software Communications Architecture revision 2.2.2, accessed January 2010. [Online]. Available: [sca.jpeojtrs.mil](http://sca.jpeojtrs.mil)
- [4] D. P. Bertsekas and R. G. Gallager, *Data Networks*. Prentice Hall, January 1992, 2nd Edition.
- [5] Apple Mac OS X 10.6: Grand Central Dispatch, accessed January 2010. [Online]. Available: [www.apple.com/macosx/technology/#grandcentral](http://www.apple.com/macosx/technology/#grandcentral)

#### VI. BIOGRAPHIES

**Michael L. Dickens** is a Ph.D. candidate in Electrical Engineering at the University of Notre Dame. He received a B.S. from MIT in 1991, and a M.S. degree from the University of Notre Dame in 2001, both in Electrical Engineering. He has more than 10 years of industry experience, having worked at the Oak Ridge National Labs (Oak Ridge, TN), Bolt Beranek and Newman ("BBN", Cambridge, MA) including in the Internetworking Division, and most recently the MITRE Corporation (Bedford, MA). His research interests span all aspects of programming for software

radios – from operating system boot codes to kernels, signal-processing implementations to user interfaces.

**Brian P. Dunn** is a Ph.D. candidate in Electrical Engineering at the University of Notre Dame. He received a B.S. degree from Purdue University in 2003 and a M.S. degree from the University of Notre Dame in 2005, both in Electrical Engineering. His research interests lie within wireless communications, spanning from information theory to software radio. He has worked for Crown Audio and BAE Systems, and recently won the 2008 Notre Dame McCloskey Business Plan Competition for a plan to commercialize software radios.

**J. Nicholas Laneman** is an Associate Professor in Electrical Engineering at Notre Dame and has served as a regular consultant to industry (including two startup companies), government, and intellectual property firms for the past decade. He has a Ph.D. in signal processing and communications from MIT (2002) and his research focuses on software radio, wireless communications, and information theory. He is author or co-author on over 50 publications and co-inventor on 5 US patents, with a number of provisional patents pending. Dr. Laneman received a 2006 Presidential Early-Career Award for Scientists and Engineers (PECASE), a 2006 National Science Foundation (NSF) CAREER Award, a 2003 Oak Ridge Associated Universities (ORAU) Ralph E. Powe Junior Faculty Enhancement Award, and the 2001 MIT EECS Harold L. Hazen Graduate Teaching Award. He is a member of IEEE, ASEE, and Sigma Xi.