Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

Chaining Communications Algorithms with Process Networks

Oliver FAUST, Bernhard SPUTH and David ENDLER

DSP Centre, Ngee Ann Polytechnic, Block 8 #06-09,
Clementi Road 535, 599489 Singapore, Singapore
faust_o@web.de, bernhard@erg.abdn.ac.uk, david.endler@t-online.de

Alastair R. ALLEN

Departments of Engineering and Bio-Medical Physics,
University of Aberdeen, Aberdeen, UK

Abstract. *Software Defined Radio* (SDR) requires a reliable, fast and flexible method to chain parametrisable algorithms. *Communicating Sequential Processes* (CSP) is a design methodology, which offers exactly these properties. This paper explores the idea of using a Java implementation of CSP (JCSP) to model a flexible algorithm chain for Software Defined Radio. JCSP offers the opportunity to distribute algorithms on different processors in a multiprocessor environment, which gives a speed up and keeps the system flexible. If more processing power is required another processor can be added. In order to cope with the high data rate requirement of SDR, optimized data transfer schemes were developed. The goal was to increase the overall system efficiency by reducing the synchronisation overhead of a data transfer between two algorithms. To justify the use of CSP in SDR, a system incorporating CSP was compared with a conventional system, in single and multiprocessor environments.

1 Introduction

A Software Defined Radio (SDR) is a single device which is capable of performing different wireless communications functions at different times [1, 2]. Such devices avoid communication breakdowns by adjusting to new environments. Such a breakdown happened during the first gulf war (1991): the U.S. military observed that their operations were hindered by incompatible radio equipment. This was the reason why the SPEAKeasy project [3] was launched by different branches of the U.S. military as one of the first attempts to create a Software Defined Radio.

Algorithms are used to model the functionality of an SDR device at a specific time. It is not efficient to state a specific functionality in the form of a single sequential algorithm, because this does not allow for reusing parts of the algorithm to model other functionality. Therefore, it is desirable to have general parametrisable algorithms, generic enough to be employed in different models. This requires the ability to execute multiple algorithms and to move data freely between them. The execution of the individual algorithm should only depend on the data, such that only data processing, and not waiting for data, requires processing resources. This reduces the processing time, because one processor, or a processor network, can be shared among multiple algorithms. The design of such systems is one of the goals of Communicating Sequential Processes (CSP) [4]. The sharing of processing resources is achieved by executing multiple processes concurrently. A process incorporates a rule defining the relationship between data input and output. For Software Defined Radio the rules are stated as algorithms. Data can be exchanged between processes by means of channels

connecting the individual processes. Part of the power of CSP is the way the data exchange is synchronised between individual processes. The formal correctness of a given CSP system can be proved mathematically. This allows the separation of the functionality from the synchronisation, leading to data driven systems. The SDR concept is an extension of a data driven system, such that not only the data exchange between algorithms is data dependent, but even the algorithms themselves depend on the data, i.e. the algorithms change depending on the data to be processed.

The price for having parallelism in an SDR system is the increased synchronisation overhead, which decreases the data throughput. But data throughput is one of the factors which limits the capability of a particular SDR system. The practical part of this paper (Section 4) details buffer reuse as one of the methods to increase the data throughput. For a particular channel, the total time spent on synchronisation depends on the synchronisation overhead and on how often the channel is used. If a fixed data-rate is assumed for a particular channel, then the data frame size will determine how frequently a channel is used. The effect of different frame sizes on the data throughput is shown in the measurements detailed in Section 6.

2 Discussion: Sequential versus Concurrent Data Processing

Data processing is a repeating three-step process: fetching a data block, processing it and storing the result. Based on this definition it is possible to state the optimal condition for data processing: A data processing system is called optimal if it can meet the processing requirements while utilising as little resources as possible. Under these constraints a data processing design which utilises a single processor to 100% is optimal, due to the fact that the processor is the limiting resource. In a sequential design the three steps for data processing are performed in a single loop. Under practical considerations a sequential design has several problems.

In a real-time environment, the data fetching, processing, and result storage steps run with a constant data rate. This means that data fetching and storing steps require time. The processor does not perform these steps, these are done by external entities. The processor waits for these entities to signal completion. If the data processing is performed sequentially, the wait time cannot be utilised for data processing. To achieve the desired output rate, the processing has to be done in the remaining time, which requires a faster processor. Therefore, a sequential design is not an optimal solution for real-time signal processing. The situation worsens when executing a sequential design in a multiprocessor environment, because only one processor is utilized. The only way to speed up a given sequential design is to use a faster processor! There is a linear relationship between the execution speed of a sequential design and the processor speed.

To utilise wait times as well as multiple processors, the fetching, processing and storing steps should be performed concurrently. Each of the steps forms a small-scale sequential algorithm. These concurrently executing steps need to exchange the results of their labour. These exchanges must be safe, i.e. the concurrent operating steps must perform synchronised data exchange. The synchronisation of the interconnects requires processing power, but this is a small price to pay for a nearly full utilisation of multiple processors.

3 Some SDR Algorithms

An algorithm produces specific output from specific input, and the input-output relationship is normally expressed in mathematical terms. The discussion here is restricted to the algorithms used in the system implementation (see Section 4). These algorithms come from the

area of digital communications, which is a field where Software Defined Radios are widely employed.

Most of the transmission signals used for digital communications are defined in the baseband domain. This allows the description of the signal independently from the actual transmission frequency. In a transmitter a Digital Up Converter (DUC) is used to shift the base band signal to a transmission band. A receiver incorporates a Digital Down Converter (DDC) for the inverse operation. To state a distributed model for both the DUC and DDC functionality, the following five different algorithms are required:

- Up-Sampler, **L** ↑;
- Finite Impulse Response Filter, **FIR**;
- I/Q Combiner, **IQ/IF**;
- I/Q Splitter, **IF/IQ**;
- Down-Sampler, **L** ↓;

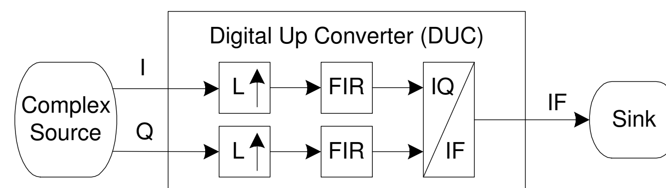

Figure 1: Digital Up Converter algorithm chain

A detailed description of these algorithms can be found in various Digital Signal Processing text books [5, 6]. Figure 1 shows how these algorithms are connected in order to represent the DUC functionality. The output of the source is an arbitrary complex baseband signal, the real part of this signal is represented by an I (Inphase) signal and the imaginary part is represented by a Q (Quadrature) signal. Representing the base band signal via I and Q has the advantage that the subsequent algorithms process real instead of complex numbers: this simplifies the algorithm implementation. The upsampling and filtering combination is used to increase the sample frequency of the baseband signal. This operation allows the I/Q Combiner to shift the baseband signal into an intermediate frequency (IF). The IF signal is a real valued transmission signal. The value of the IF (intermediate frequency) must be stated as a rational number (numerator and denominator) to ensure that the signal can be created with a digital system. In other words, it must be possible to represent one or multiple periods with a finite (integer) number of samples.

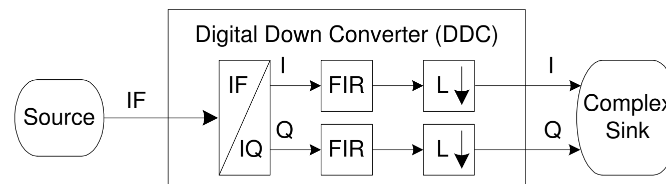

Figure 2: Digital Down Converter algorithm chain

Figure 2 details the block diagram for the DDC. The I/Q splitter shifts the received IF signal down into the baseband domain. This baseband signal has an unnecessarily high sample frequency, i.e. the highest possible signal frequency is much lower than half of the sample frequency [7]. The filtering and down sampler combination reduces the sample frequency. This operation is necessary because the sample frequency determines the processing speed for the subsequent baseband processing algorithms. For simplicity these algorithms are represented by the complex sink block in Figure 2.

If the FIR algorithm is parametrisable, the same implementation can be used for DUC and DDC. This reduces the implementation time and improves the quality of the result, because well known standard implementations can be used. Due to the fact that a receiver merely performs the inverse operations of a transmitter, such synergy effects are quite frequent in digital communication systems. For SDR, the synergy effects are not limited to transmitter and receiver symmetries, they extend to similarities between different standards [8]. The DUC and DDC functionalities are perfect examples of synergies across different standards. If the individual algorithms can be parametrised, there is only one implementation required to accommodate a multitude of different standards.

4 Implementation Aspects

SDR systems can be used in various types of environments, such as client PCs, broadcasting or embedded systems. As the algorithms stay the same, they should be executable in the different environments without any changes. In the Java environment, the compiler translates the source code into Java Byte code, which is executable by a Java Virtual Machine (JVM). JVMs are available for nearly all types of environments, making the Java environment ideal for implementing SDR systems.

The system under discussion here consists of the algorithm chains shown in Figures 1 and 2. To be able to verify the correct functioning of the algorithm chain, it was decided to process files compatible with the FhG Software Radio [9]. This is software that decodes signals according to the Digital Radio Mondiale (DRM) standard [10]. The FhG Software Radio is able to decode files containing DRM signals in either IF or IQ format. The files contain samples of 16 bit resolution, which corresponds to the short data type in Java. As all the algorithm implementations operate with integer values, a data conversion has to be performed.

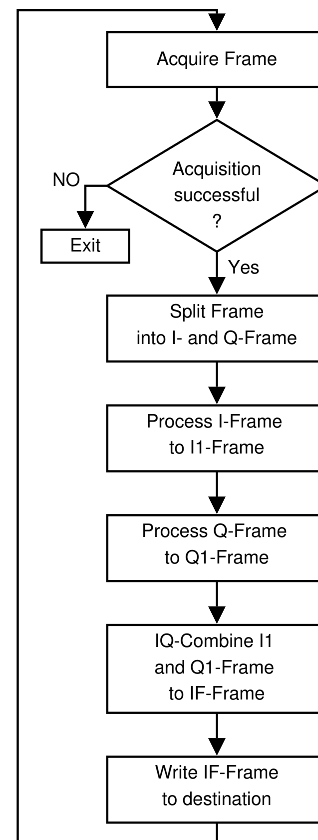

Figure 3: Finite State Machine of the DUC chain.

For the Digital Up Converter (DUC) of Figure 1 a complex source, providing the I and Q components, is required. As the file format stores these two signal components in a multiplexed form, a de-multiplexing step is required. This de-multiplexing step will also perform the data type conversion from short to integer. For the Digital Down Converter (DDC) of Figure 2, an IQ-Multiplexer is required in order to write files compatible with the Fraunhofer Software.

Processing single samples introduces a high number of function calls, which in turn decreases the performance of the system. To avoid this, groups of samples, so-called frames, are constructed and processed together. Increasing the frame-size results in a greater delay, as the system has to wait longer for the frames to become available. The frame-size depends on the processing system and the particular application. In SDR the applications are not fixed, some applications allow more delay than others. For example, a broadcast standard allows more delay than a communication standard, because a human is able to detect communication delays that are larger than 20ms. This is the reason why in the SDR system under discussion the frame size must be flexible. This is relatively unusual for CSP systems; normally all parameters are optimised for fastest processing without latency constraints.

4.1 Sequential Implementation

For implementing the sequential approach, each algorithm of the DUC and DDC chains is implemented as a Java class. The DUC and DDC chains require the signal data to be processed by the algorithms in a fixed way. Therefore, each chain is represented by a function, which instantiates the algorithms and calls each of them according to the schedule. This is equivalent to a finite state machine, as shown in Figure 3 for the DUC chain.

4.1.1 Optimisation of the Sequential Implementation

In the sequential implementation each algorithm allocates a new output buffer and disposes of the input buffer. Both allocation and disposing of buffers takes time, which could be spared if the output buffer could be recycled. Since only one algorithm is executing at a time, and the frame-size does not change during runtime, recycling of the buffer is possible.

4.2 Concurrent Implementation using CSP

CSP is not included in the standard Java distribution, so a Java implementation of CSP had to be found. *Communicating Threads for Java* (CTJ) [11, 12, 13, 14] and *Communicating Sequential Processes for Java* (JCSP) [15, 16, 13, 14] seemed to be what we were looking for. JCSP and CTJ have a similar goal and therefore provide a similar functionality, and to choose between them was not easy. In the end, the exhaustive documentation, with lots of examples plus lots of other support material, made JCSP the implementation of choice.

In CSP, each process executes independently from other processes: this makes it possible to execute multiple processes concurrently. Data exchange between processes is only possible by using unidirectional channels which act as a synchronisation mechanism. Every process has channel-inputs and channel-outputs, depending on its communication requirements. The combination of processes and their interconnection channels is called a process network.

To convert the sequential implementation into a concurrent implementation, we have to transfer the sequential algorithm chain into a process network. To create this process network it is necessary to identify the independent components of the sequential algorithm chain, and convert each into a process. Taking a look at block diagrams, Figures 1 and 2, identifies that each block performs its operation independently from other blocks. Therefore, each block is converted into a process. This is done by implementing a wrapper class. This wrapper class has an instance of the algorithm class as member, whilst providing a CSP conforming interface. To create the corresponding process network is now only a matter of interconnecting the processes according to the block diagrams.

As the concurrent approach can only utilise as many processors as it has processes, a large number of processes is desirable. For the developer, on the other hand, a large number of processes becomes difficult to handle. It is difficult for the programmer to maintain an overview of the complete method/function: this leads to insecurity. Fortunately, CSP allows the creation of components and their use in a hierarchical fashion [17]. This is enabled by the fact that a process network in CSP is nothing else than a process. Therefore, it is possible to use predefined process networks, as components, to build a larger process network. This process network's interface will simply consist of externally connected channel inputs and outputs. This technique of component building was used to create the DUC and DDC processes of Figures 1 and 2. The process-networks for the IQ-Combiner and IQ-Splitter are shown in Figure 4.

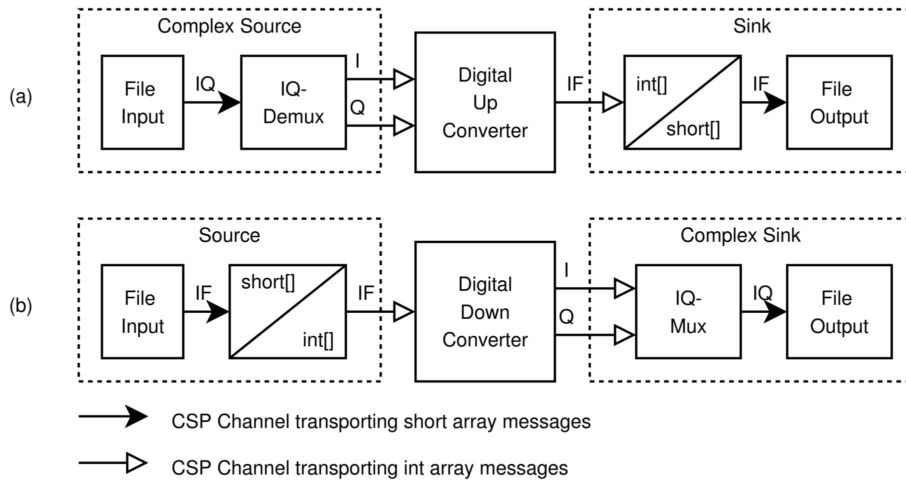

Figure 4: Process Networks for IQ-Combiner and IQ-Splitter.

4.2.1 Improving the Data Throughput

As previously stated, it is desirable to avoid unnecessary buffer allocation and disposal. In the sequential implementation this was achieved by allocating a single output buffer, which was then used as input by the following algorithm. Using the same scheme in the concurrent implementation, where the algorithms are executed in parallel, could result in one algorithm writing to the buffer while another reads from the same buffer. The result would be wrong, rendering this implementation useless.

If we assume that each algorithm / process only operates on a single buffer per channel at one time, it is sufficient to use two buffers which are exchanged between two processes. This approach has two drawbacks. First, the size of the output buffers is fixed over the complete runtime of the network. Second, each passing of output frames requires two channel communications: one to send the new output buffer and one to receive the now empty output buffer. The two communications increase the synchronisation overheads, increasing the runtime.

In order to avoid the second channel operation, the sending process needs to keep a reference to both output buffers. One buffer is marked as *output-buffer*, to store processing results, while the other buffer is marked as *away-buffer*. The *away-buffer* is used by the next process as its input. Once the process has completed processing its input, it sends the *output-buffer* to the next process and swaps *output-buffer* and *away-buffer*. The switching of the buffers is handled by a so-called BufferKeeper object. Figure 5 illustrates the working of the scheme.

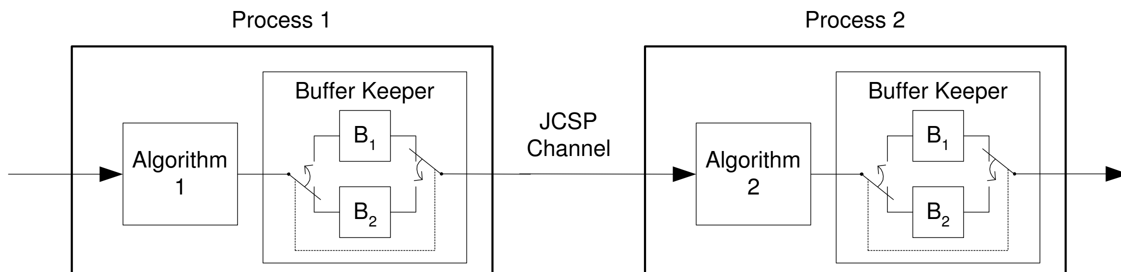

Figure 5: Buffer Reuse Scheme in the Algorithm Chain

For this scheme to work it is important to know when the receiver has processed the previous buffer. This is solved by constraining the receiver to request a new buffer only after the previous buffer is processed. With this method it is safe to perform the output buffer switch after the sending operation has succeeded.

5 Comparison between Sequential and Concurrent Implementation

The concurrent implementation has multiple advantages over the sequential implementation.

5.1 Easy to Apply, Easy to Debug

It is easy to convert sequential algorithms into CSP processes. This is done by creating a process which performs the channel operations, and uses the algorithm implementation to create the desired output from the input. This approach provides a clean division between functionality and synchronisation. This scheme allows the development and verification of functionality and synchronisation separately. This results in simpler test setups and therefore faster development.

5.2 CSP Enables Compact Designs

In CSP everything is a process, resulting in process networks also being processes. This enables the abstraction of recurring process networks in the form of processes. For the user of the resulting process it does not matter if there is a single process or a process network inside. These processes allow us to build components [17], which can be tested individually (unit testing) [18]. This component based approach accelerates and simplifies the development of algorithm chains, due to the use of previously developed and tested components. The ability to layer CSP based systems produces compact designs: these can be easier to understand and are therefore desirable [19]. The DUC chain, for instance, has two layers:

1. Top Layer: Complex Source, Digital Up Converter, Sink.
2. Bottom Layer:
    - Complex Source: File Input, IQ-Demux
    - Digital Up Converter: I-UpSampler, I-FIR, Q-UpSampler, Q-FIR, IQ-Combiner
    - Sink: Int2Short, File Output.

5.3 Taking Advantage of Multiple CPUs

A concurrent implementation can provide a speedup S on multiple CPUs. The speedup is defined as [20]:

$$S = \frac{T_S}{T_N} \tag{1}$$

with

- $T_S =$ optimal sequential processing time; the best time that can be achieved on a single processor using the best sequential algorithm
- $T_N =$ concurrent processing time; the actual time achieved on an N-processor system with the concurrent algorithm and a specific scheduling method being considered.

In our case, the time of the sequential implementation is considered as T_S. The scheduling is fixed by the fact that the sequence of the algorithms is fixed by the task itself. The runtime of the sequential implementation is nearly independent of the frame-size. For the concurrent approach, larger frame-sizes result in fewer channel operations and therefore in less synchronisation overhead. That means its runtime decreases with increasing frame-sizes. A graph

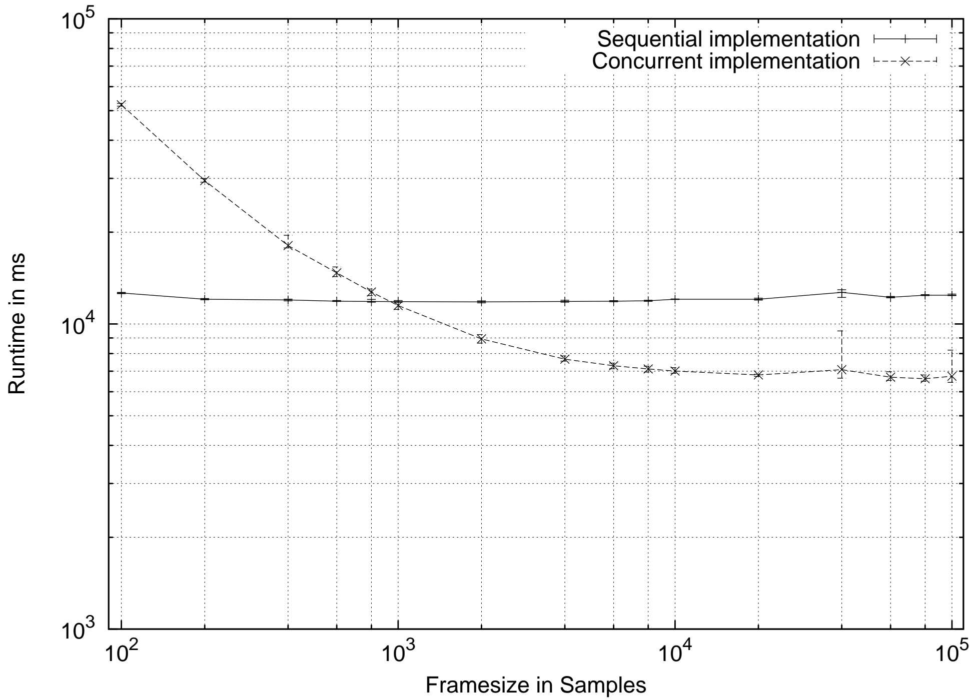

Figure 6: Comparing the runtime of concurrent and sequential implementation, on a dual CPU machine.

showing the dependency of the runtime of sequential and concurrent implementations on the frame-size is given in Figure 6.

More interesting than the raw speedup, which depends on the number of CPUs, is whether the efficiency of the concurrent implementation scales on multiple CPUs. The efficiency E, of a concurrent implementation is defined as:

$$E = \frac{S}{N} = \frac{T_S}{T_N \cdot N} \tag{2}$$

with N as the number of CPUs in the system. A plot of the efficiency versus the frame-size is given in Figure 7.

5.4 Wait Cycles Introduced by the IO Subsystem can be used for Signal Processing

The runtime of the sequential approach is defined as the summation of all steps of processing:

$$T_S = \sum_{i=1}^{m} t_i \tag{3}$$

with:

- $t_i =$ execution time for one step of a sequential implementation
- $m =$ number of steps in a sequential implementations.

Assuming that the sequential algorithm chain is connected to a slow source or sink, the time spent waiting for the IO subsystem is reflected in the processing time. The concurrent implementation can utilise this wait time to perform processing. The processing time is determined solely by the IO subsystem if the processing is faster than the acquiring or storing of the data. Due to this fact a concurrent implementation, on a single CPU system, can be faster than a sequential implementation.

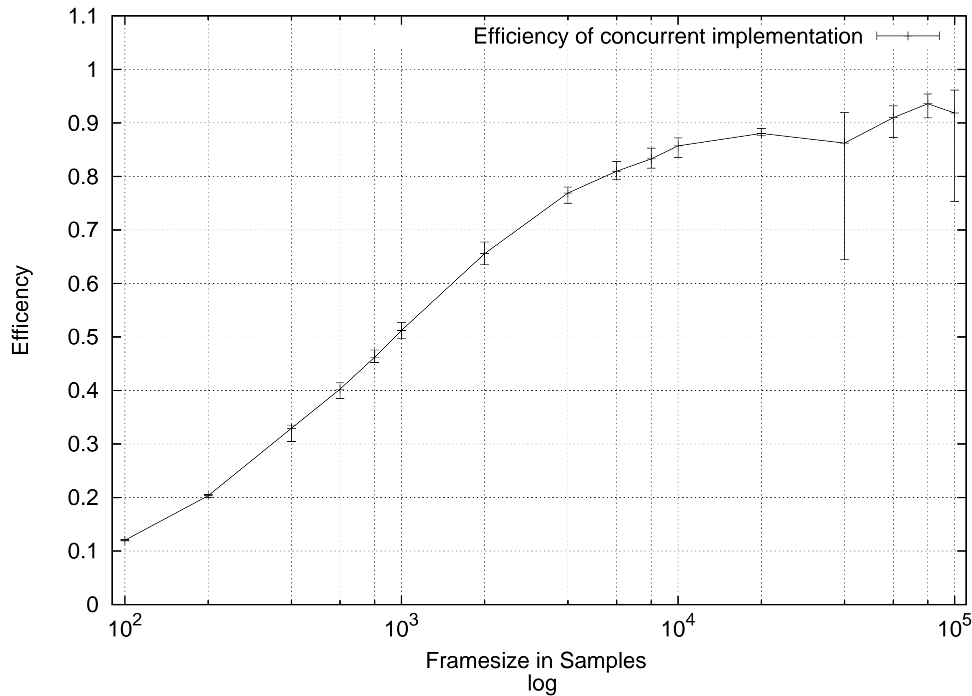

Figure 7: Efficiency of the JCSP implementation, on a dual CPU machine over the frame size.

6 Measurements

All measurements were performed using an HP workstation x4000, equipped with two Intel P4 Xeon 1.5GHz CPUs with 256kB cache and 512MB Rambus RAM. J2SDK build 1.4.2_01-b06 for Linux, available from Sun, was used as the Java environment. The Java environment was installed on Slackware Linux 9.1 running Linux Kernel 2.6.3, from `kernel.org`.

For all measurements, the Digital Up and Down Converter chains were parametrised:

DUC chain:

- Up-sampling factor: 4
- Numerator: 1
- Denominator: 4

DDC chain:

- Down-sampling factor: 4
- Numerator: 1
- Denominator: 4

All measurements represent the combined runtime of the DUC and the DDC chain.

6.1 Efficiency

Measurement Setup: In order to avoid undue influence by the IO sub-system, a source and sink which operate with virtually no latency were developed. The source supplies a specific number of samples before signalling end of data. To show that the runtime depends on the frame-size, the source is able to supply frames of any size. The implementation of the sink is an empty function, which gets called, but does not perform any processing. This setup allows us to measure the runtime of the algorithm chains without disturbance from the IO sub-system.

The runtime of the implementations were measured assuming a round trip, first digital up converting the data, then digital down converting it. The DUC chain is sourced with $6 \cdot 10^6$ IQ-samples producing $24 \cdot 10^6$ IF-samples – due to the performed up-sampling. The DUC chain is sourced with these $24 \cdot 10^6$ samples.

For measuring the optimal sequential processing time (T_S), the machine was loaded with a single CPU kernel. For each frame-size, ten runtime measurements were taken and the lowest result was taken as T_S for this frame-size.

To determine the concurrent processing time (T_N), also ten measurements per frame-size were performed.

Measurement Results: The efficiency graph in Figure 7, shows the dependency of the concurrent implementation on the frame-size. Generally speaking, the larger the frames get, the less synchronisation overhead is introduced. In our case the efficiency is over 80% with frame-sizes of 6000 samples upwards.

6.2 Slow IO Sub-system

The aim of this measurement was to show that a concurrent implementation has advantages even on a single CPU system. This is for instance the case when a single CPU system acquires the data to process from a slow IO sub-system without caching.

Measurement Setup: In order to create reproducible measurement results, a data rate controlled source was developed. The data rate was specified as samples per second sps. The source holds a variable containing the number of available samples, the *as-counter*. The as-counter gets incremented, with a samples per tick value spt, by a periodic timer thread. In Java (on Linux) a periodic timer thread can have a minimum period length of 1ms (1000 ticks per second tps). The particular kernel version used, provided the necessary timer resolution. This results in $spt = \frac{sps}{tps}$. Once the periodic timer thread has performed the increment, it wakes up a possibly waiting receiver. If the as-counter value is larger or equal to the frame-size, the receiver decrements the frame-size from the counter and processes the data. This scheme provides a stable source for data rates that are integer multiples of 1000. This source represents a data rate controlled source with cache, since the periodic timer thread always increments the as-counter. The flowchart of the source is shown in Figure 8. For reasons of simplicity, the synchronisation is not included.

To remove the caching feature from this source, it is necessary to stop incrementing the as-counter while there is no reader waiting. This was done by introducing a reader waiting flag, which is only set when a reader is waiting. The periodic timer thread checks this flag and only when it is set increments the as-counter. The only problem is, that in the case that the processing of a frame takes less than 1ms, the runtime is only determined by the source. This is due to the fact that for the timer thread the receiver-waiting flag is constantly set. This results in a measurement where the sequential approach would have always requested the next frame, before it processed the current frame. In short, the source would behave as if it would perform caching. Therefore, the measurements were done only with large frame-sizes, which in any case took longer to process than 1ms. The source was parametrised to provide a total of $72 \cdot 10^6$ samples with a sample rate of $1 \cdot 10^6$ samples/s. This results in the source requiring $72s$ to provide the data.

For this test run a frame-size of $100,000$ samples was chosen. For each test case, 10 runs were performed and the mean value used to produce Figure 9.

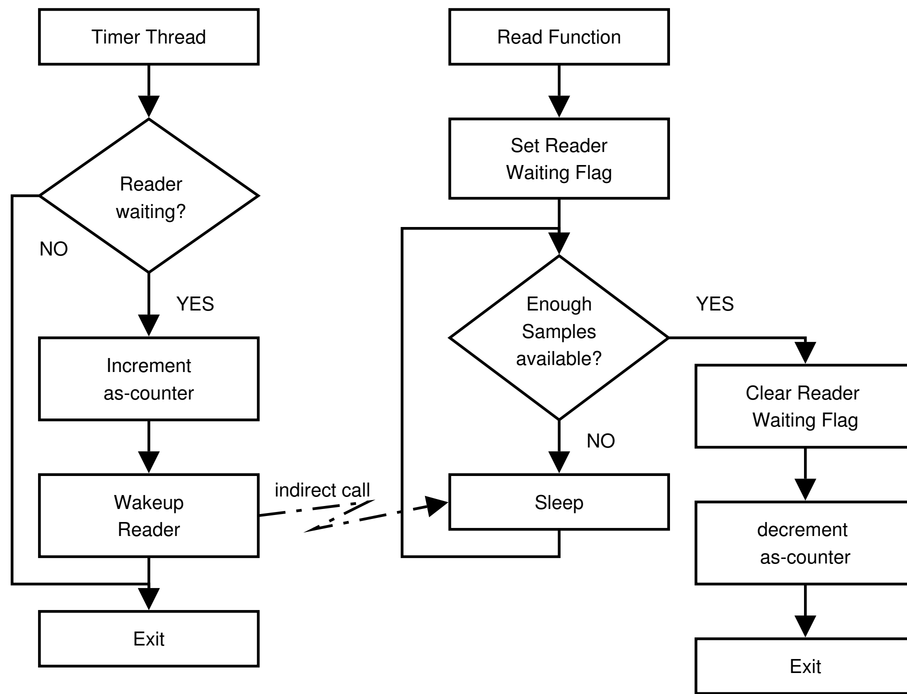

Figure 8: Flowchart of the rate controlled cache less source

Measurement Results: The bar graph in Figure 9 compares the runtime of sequential and concurrent implementations, using different sources. The sequential implementation requires $96s$ to process data from a slow source. The slow source alone requires $72s$ to provide the data, leaving $96s - 72s = 24s$, which is the processing time of the sequential implementation. The sequential implementation sourced by the low latency (fast) source requires $24.6s$ to process the same amount of data. This supports the validity of the statement made in Equation 3 that the runtime of the sequential approach is the summation of all steps of processing.

The concurrent implementation, on the other hand, requires $72.1s$, which is very close to the $72s$ the source requires to provide the data. This shows that a concurrent implementation can utilise wait times introduced by external entities to perform processing. A concurrent implementation can have runtime advantages even on a single processor system.

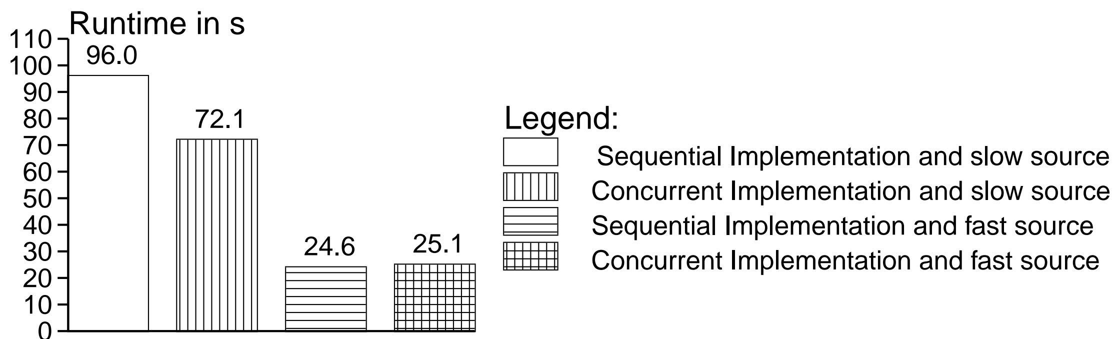

Figure 9: Impact of a slow source on the runtime of the different implementations on a single CPU machine

7 Conclusions

The use of CSP technology in SDR systems is beneficial, because of increased flexibility and reliability. Compared with other, monolithic, implementations the CSP approach offers a speedup in multiprocessor systems, because the algorithms can be executed in parallel. Multiprocessor systems require some sort of synchronization in order to keep track of which algorithm is used to process what data. For Software Defined Radio systems these synchronization methods are not the core problem, therefore it is advisable to use a standard method. CSP represents such a standard method, which offers the opportunity to prove the absence of deadlocks. Other implementations which cater for parallel processing require sophisticated synchronization methods. In the best case these other synchronization methods are reinventions, but most of the time they fall short in reliability and flexibility compared with the CSP approach.

A flexible way of distributing the processor load is required in order to make full use of a multiprocessor environment. CSP offers the opportunity to utilize additional processing resources. The question is: how good are the algorithms when distributed over the available processors. SDR requires mainly an algorithm chain where the processing speed is determined by the time requirement of the slowest process in the chain. The goal for a load balancer is to find an optimal processing load distribution in a multiprocessor system. This task is left for a future undertaking.

References

[1] Joseph Mitola. *The software radio architecture*. IEEE Commun. Mag., no.5, pp.26-38, May 1995.

[2] Joseph Mitola. *Software radio architecture: A mathematical perspective*. IEEE J. Sel. Areas Commun., vol.17, no.4, pp.514-538, April 1999.

[3] R. J. Lackey and D. W. Upmal. *Speakeasy: The Military Software Radio*. IEEE Communications Magazine, vol. 33, no. 5, pp. 56-61, May 1995.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 2003.

[5] Sanjit K. Mitra. *Digital Signal Processing – A Computer-Based Approach*. Mc Graw Hill, Avenue of the Americans, New York, NY 10020 United States of America, second edition, 2001.

[6] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing Principles, Algorithms, And Applications*. Prentice Hall, Upper Saddle River, New Jersey 07485 United States of America, 1996.

[7] C.E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, July/October 1948.

[8] Jeffrey H. Reed. *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall, Upper Saddle River, New Jersey 07485 United States of America, 2002.

[9] Fraunhofer. *Homepage FhG Software Radio*. `http://www.iis.fraunhofer.de/dab/products/drmreceiver/`.

[10] *DRM standard IEC 62272-1 Ed. 1: Digital Radio Mondiale (DRM) - Part 1: System Specification*. International Electrotechnical Commission, IEC Central Office 3, rue de Varemb? P.O. Box 131 CH - 1211 GENEVA 20 Switzerland, 2003-03.

[11] André W. P. Bakkers, Jan F. Broenink, Gerald H. Hilderink, and Wiek Vervoort. Communicating Java Threads. In André W. P. Bakkers, editor, *Proceedings of WoTUG-20: Parallel Programming and Java*, pages 48–76. IOS Press, the Netherlands, 1997.

[12] Gerald H. Hilderink. CTJ (Communicating Threads for Java) Home Page. Available at: `http://www.ce.utwente.nl/javapp/` Retrieved July, 2004.

[13] Peter H. Welch, André W. P. Bakkers, G. S. Stiles, and Gerald H. Hilderink. CSP for Java: Multithreading for All. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 277–278. IOS Press, the Netherlands, 1999.

[14] Peter H. Welch, Gerald H. Hilderink, and Nan C. Schaller. Using Java for Parallel Computing - JCSP versus CTJ. In Peter H. Welch and Andr W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 205–226. IOS Press, the Netherlands, 2000.

[15] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[16] Peter H. Welch. Java Communicating Sequential Processes Home Page. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/` Retrieved July, 2004.

[17] Peter H. Welch. Communicating Processes, Components and Scalable Systems. Homepage, May 2001. `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/components.pdf`.

[18] Steven C. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, One Microsoft Way, Redmond, Washington, 1993.

[19] Eric Steven Raymond. *The Art of Unix Programming*. Pearson Education, Inc., Pearson Education Inc, Rights and Contracts Department, 75 Arlington Street, Suite 300, Boston, MA 02116, first edition, 2004.

[20] Randy Chow and Theodore Johnson. *Distributed Operating Systems & Algorithms*. Addison Wesley, 2725 Sand Hill Road, Menlo Park, CA 94025, 1997.