Assessing Performance of Software Defined Radios on Multicore Hardware

Nick Green (University of Illinois at Chicago; Chicago, Illinois, USA ngreen21@uic.edu)
Ugo Buy (University of Illinois at Chicago; Chicago, Illinois, USA; buy@cs.uic.edu)
Redge Bartholomew (Rockwell Collins; Cedar Rapids, Iowa, USA; rgbartho@rockwellcollins.com)

Abstract—Multicore hardware is now ubiquitous in society. While Moore's Law predicts that the number of transistors on a chip will double every two years, its effect on the speeds of single core processors have levelled off. To work around this limitation, hardware designers are producing chips with an increasing number of cores. This allows vast increases of processing on a single chip without increases in clock speeds.

However, this extra processing power does not come for free. There are significant design issues that need to be taken into account, such as parallelisation and synchronisation issues. In some cases there can be bottlenecks that extra cores will not help overcome. Here we empirically assess the potential performance changes when running SDR applications on multicore platforms. We conclude that the performance of SDR can in fact benefit from multicore hardware; however, various factors may have an adverse effect on potential performance gains.

1 Introduction

Computer systems have seen many transformations since their inception. A relatively recent, albeit significant, evolution has been with respect to the CPU. Traditionally, CPUs have been based on a serial pipeline, where an application is executed sequentially. This has given a somewhat easy approach to coding applications. For a developer, it is straightforward to see that one instruction leads onto another, giving a predictable flow to the program.

Over the years, increases in system performance have come from the ever increasing speeds of the CPU. Hardware manufacturers have increased clock speed's by increasing the number of transistors on the CPU while also decreasing their overall size [1]. Moore's Law [2] has applied whereby performance has roughly doubled every two years. This has worked for decades, however, since the mid-2000s there has been a plateau in the performance gains that have been achieved by using this method due to power and heating constraints. As mentioned elsewhere [1], it is possible to increase performance using this method, but processors would need to double in size while offering marginal gains.

However, this does not necessarily mean that Moore's Law has become invalid. Since it was apparent that the traditional method of performance increases would not last forever, hardware manufacturers have been devising new ways of getting the most out of a CPU. The method that has been most successful over recent years has been the multicore CPU.

1.1 Multicore CPU

The multicore CPU is a change from the traditional single core processor in that rather than making a single pipeline faster, it enables multiple pipelines with slightly lower performance to be utilised at once. The optimal outcome, for instance, is that performance may double when doubling the number of cores on a given processor. While this is generally good news [3], it also means that the free lunch for software developers is clearly over. Developers have had the privilege of creating software with the assumption that hardware would increase in performance over the foreseeable future with no effort on their part. An application that runs slowly on a given hardware platform may run at an adequate speed when the very same application is executed on a faster CPU. This is no longer the state of affairs for developers today.

When extra cores are provided on a CPU, the application has to know how to utilise them. This is the biggest effect when programming single vs. multicore systems. In order for the application to take advantage of these hardware capabilities, it must know what parts of the system can be executed in parallel. This can be achieved with the concept of concurrent programming. A single application can make a choice at a particular time when it can divide processing between multiple units of execution, such as a thread. Thankfully, it is at this point that the hardware level of abstraction hides the handling of the cores. Therefore, software need only know concurrency at a software threading level in order to exploit the full performance of multicore hardware.

Existing software can also take advantage of multicore hardware. If an application makes use of multiple threads, which can be a desirable design decision in single core environments, then each of those threads may utilise the multicore feature of the CPU. An issue for existing software is how far it uses concurrency and asynchronous processing. As stated by Amdahl's Law [4], if an application has a particular portion that cannot be parallelised, the ill-effects of this bottleneck will persist no matter how many processors are allocated to the application.

Various factors may limit the performance gains that can be obtained by parallelising software for deployment on multicore hardware. For instance, some I/O operations are serial, meaning that each operation cannot be effectively speeded up, although it is sometimes possible to carry out multiple operations in parallel with each other (e.g., by reading a file while updating a user display). In addition, data dependencies can prevent effective parallelisation. Data dependencies happen, for instance, when a unit produces data which is later consumed by another. In this case, the units affected cannot be executed in parallel, and so on.

1.2 Software Defined Radio

The software defined radio [5] is the application of choice in our research. SDRs are an incredibly flexible way of programming a radio for use under multiple frequencies, bands and codecs. SDRs take much of the custom hardware out of a radio and instead implement hardware functionality in software on a general purpose host, such as a standard desktop PC. This host receives a transmission from a hardware radio and processes it natively. The host is the flexible part of the equation, as it is possible to vary its software (e.g., the operating system), hardware and CPU architecture, while still allowing a vast number of software radio components to be utilised at low cost.

1.3 Research Questions

Here, we are going to investigate if a multicore host platform can improve performance in an SDR environment. Through this research we should be able to answer the following questions:

1) Can an existing SDR benefit from deployment on multicore hardware?
2) Is it possible to extend SDR functionality to make use of multicore features?
3) Are there cases when multicore adversely affects performance? and ultimately
4) Is it worth using multicore hardware for SDR applications?

Our empirical investigation will evaluate SDR performance under different circumstances. In Section 2, we give an overview of software defined radios and the software used to drive them. Sections 3 and 4 detail the platforms that we used for our experiments with OSSIE and GNU Radio, respectively. In Sections 5 and 6, we present our empirical results with OSSIE and Gnu Radio. Finally, in Section 7 we give our conclusions on these results and attempt to answers the research questions above.

2 Software Defined Radio Environment

There are many different options available to users who wish to use a software defined radio. However, there are two main components; the hardware radio and the software host. The first step for us in our research was to decide what environment we should use to carry out our experimentation. A popular choice for this is the versatile USRP [6] from Ettus Research [7]. The USRP is a modular system that can take a number of daughter-boards, each providing different features such as varying bands, a receiver or transmitter. In our case, we wished to perform receiving within the FM frequency range; thus, we opted for the WBX daughter-board with the USB enabled USRP1, which houses the daughter-board.

There are also a few choices that can be made for the software component of the environment. We evaluated two systems in depth. First, we looked at the Joint Tactical Radio System (JTRS) [8] compliant OSSIE [9], which gives a highly modular SDR software solution based on a component graph architecture. Another software framework that we considered is GNU Radio, which like OSSIE is also a graph based system but not JTRS compliant. It is apparent that the graph based architecture is a popular choice for developing streaming based applications, just like some non-SDR related frameworks such as some DirectX APIs [10]. Even though both OSSIE and GNU Radio are architecturally similar, we chose to focus our investigation to GNU Radio due to the maturity of the platform, also considering that OSSIE uses some components of GNU Radio in order to act as an interface to the USRP. Most of the empirical results that we report later on are based on GNU Radio, however, we will also report some of our preliminary results obtained with OSSIE.

3 OSSIE Experimental Setup

In our research, we seek to study empirically the effects of multicore on software defined radio usage. Our first set of experiments involves the OSSIE framework. Since we were planning to investigate in depth real-world applications with GNU Radio, we ran a smaller set of experiments at a more abstract level with OSSIE.

The OSSIE framework is built upon a graph architecture just like GNU Radio; there are many similartities between the two frameworks. One key difference is that OSSIE is JTRS compliant, which in turn makes it highly modular with a CORBA [11] interface between components. In terms of system resources, a significant difference is that OSSIE will then spawn each graph component in a different OS process, whereas GNU Radio spawns components in different threads (i.e., light-weight processes) within the same process.

We performed some preliminary experiments with OSSIE in order to get a feel for applications within the domain. The simple set of experiments that we performed was a pure simulation of a component graph. Using a dummy source component from the OSSIE toolbox (TX Demo), empty packets would be injected into the system at a predefined interval of 1 ms. The goal of our experiments was to determine with how long it would take for a packet to transition to a null sink component (RX Demo), also provided by OSSIE. By imposing a variable load on a set of simulation components, we could then see the effects of the workload relative to packet throughput. Furthermore, we investigated different graph configurations, in particular, serial and parallel graphs, where our workload is either performed in a sequential or a parallel pipeline. The hypothesis here is that since each component within

an OSSIE graph is executed in their own process, splitting workload among components should yield an improvement on a multicore platform, where an improvement is denoted by a decrease in time for a packet to make its transition from the source node to its destination.

Since this set of experiments was only concerned with the OSSIE graph, we did not use any SDR hardware. For the software host, we selected a dual-core PC (with an Intel Core2 i3 CPU) to perform our experiments with Ubuntu 10.04 (32-bit), which is the latest compatible version of Ubuntu that is supported by OSSIE 0.8.2.

4 GNU Radio Experimental Setup

In our experimentation with GNU Radio, we investigated in detail more real-world systems using the USRP antenna. The primary metric for our study is concerned with the all important concern of signal integrity. We created a test framework using GNU Radio, where we could easily modify various parameters and system configurations in relation to the software host. By methodically varying such parameters, we identified properties that could affect overall system quality. For example, if introducing more threads into the system introduced gaps into audio playback from the radio, we could deduce that the introduction of threads in this situation is not desirable for the application. This is very applicable to our work since there are several parameters that are related to multicore hardware, such as process affinity and the number of software threads within the application. On the one hand, process affinity defines the number of cores that can be assigned to a given application. For instance, by setting an application's process affinity to one, we can force an application to run on a single core. Software threads define units that can be executed in parallel within the context of the same OS process. Threads can be executed simultaneously on different cores, if the affinity of the application containing the cores is greater than one.

We ran our experiments with Gnu Radio on a multicore processor, with the Intel Core i7 CPU, which includes four hardware cores with two hyper-threads per core. As we want to use GNU Radio, a reliable host operating system is a Linux-based OS; in our case we chose the 32-bit variant of Ubuntu 12.10.

4.1 Test Framework

The test framework is the basis of our experimentation. GNU Radio [12] offers us a very modular graph-based architecture. The aim of a GNU Radio graph is to transform an input into an output via a series of transforms. An example of a very simple graph is one that contains two components. Figure 1 shows such a graph, where a source component generates a sine wave, which will then pass the signal to an output component which in turn renders it to the PC's speaker. A more complex graph could be made to receive, say, FM radio from a USRP, as can be seen in Figure 2. Such a graph would consist of an input node which interfaces with the USRP hardware, passes through numerous transform nodes by performing operations such as demodulation, with the output then sent to the speaker.

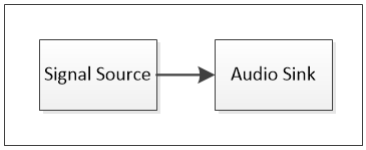

Figure 1. GNU Radio Companion graph that plays back a test signal.

Our experiments used this kind of architecture to create our testing framework. This was done by measuring the amount of data that is output to the speaker. Since SDRs are real-time systems, there has to be a steady flow of data throughout the graph. If there were any blocks in the graph that temporarily stop data flow, samples would be dropped at or near that point. In the case of the previously mentioned graphs, the source components skipped the generation of samples. This could be heard by the user as simple sound jitters. The greater the bottleneck, the more disrupted the output was.

However, to get a more accurate reading we did not use sound quality, but instead counted the samples that made it through the graph. This was done by counting the number of samples per second that passed through a custom component which is placed just before the output component. It is through this sample rate that we determined if degradation was happening, potentially due to a bottleneck in the pipeline. Through our experimentation we varied parameters and observed how these parameters correlated with this sample rate.

We were able to develop two different modes of operation with this framework:

- Manual mode—Here we used GNU Radio Companion [13] to present a user interface (figure 3) to the investigator, which contains various sliders that vary the available parameters. We can specifically vary the following parameters: (1) the total number of software threads executing GNU Radio; (2) The amount of simulated workload imposed on the various cores; and (3) The frequency at which we receive an FM signal. This is an ideal way of quickly gauging the effects of each parameter on system performance. Output can easily be heard via degradation of output.
- Automatic mode—For a deeper analysis, it is also possible to automate parameter modifications via a configuration file. Samples are taken at a set number of intervals (in our setup, 10 samples of 2 second duration) which will indicate the throughput of the system. This throughput gives an insight into the overall system performance.

An important aspect of the framework is our custom experimentation component which not only acts as a hook in getting timing data out of the system, but also as a way of inducing some system-specific logic into the workflow. This is a useful way for a developer in the real-world to see how their code reacts to multicore parameter modifications. A use case of this methodology is to implement a feature within a component which performs a transform, for example, encryption of the signal or multiplexing of two signals. Each case

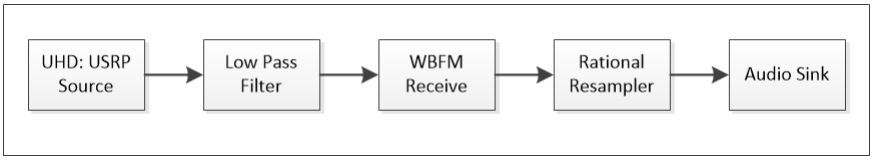

Figure 2. GNU Radio Companion graph for FM receiving and playback.

could introduce their own parameters into the equation which could be exposed via our test framework, allowing a systems engineer to change parameters and see how performance would be affected in their application. In our case, we performed some general processing with variable workload in a so-called Processor Block.

Our component, called a *Processor Block*, performs the simple operation of batch in place floating point division, where the workload is correlated to the number of division performed. However, this operation is designed to utilise multiple cores via threading. When a packet is received from the radio, the workload is shared among a variable number of threads by creating a thread with the job of performing the workload function an assigned number of times (i.e., n floating point divisions).

The hypothesis here is that as we change the number of threads, active cores, and workload, we should see some changes in performance. If performance increases when we introduce more threads and cores into the component, we can conclude that multicore can improve performance in SDR applications. We conducted several experiments varying parameters and component graphs to see the effects that these have on overall system performance.

5 OSSIE Emperical Results

Table 1 shows the results that were collected from the graph in Figure 4. It details the number of iterations of workload that were applied in several cases, along with the amount of time it took for a packet to make its transition from the source filter to the sink on a dual core system. Times for each component shown in Figure 4 are expressed in milliseconds.

When a light workload was applied to the system (Row 1), packets would make their transition from the source to the sink is roughly 1.20 ms. As workload is applied to any custom filter, the transition time increases as expected. Row 2 shows results when the custom filter has a slight workload of 1000 floating point divisions. This gives an expected slow down in the transition speed since the packet will be blocked waiting for the processing to complete. The benefits of multicore come into play when there is a high workload applied to multiple components. Row 4 shows two filters in serial having a workload of 5000 divisions applied to both. The time for a packet to go through the system takes twice as long when running on a single core. This will be due to the parallel processing of the workload in different components occurring in different processes, thus different cores. This hypothesis is backed up by the recurrence of this in other reported cases. We conclude that in our OSSIE simulations, we were able to achieve near-optimal or optimal (linear) speed-ups when using multiple cores.

Serial1 (iterations)	Serial2 (iterations)	Mux (iterations)	Serial4 (iterations)	2 Cores (ms)	1 Core (ms)	Perf. Delta%
1	1	1	1	1.20	1.20	1.00
1	1	1	1000	109	110	0.99
1	1	1	5000	548	548	1.00
1	1	5000	5000	547	1095	0.50
1000	1000	5000	1000	655	876	0.75
1000	1000	1	1	110	220	.050
1000	1000	1	5000	548	781	0.71
5000	5000	5000	5000	1084	2153	0.50

Table 1
Timing results for packet transfer in OSSIE.

6 GNU Radio Empirical Results

In general, our experiments with GNU Radio yielded results that we had expected. However,some empirical results were surprising, which required further investigation. In this section we detail our findings on an experiment by experiment basis, while showing the process that we followed to gain extra information for further clarification.

6.1 How do Multiple Threads Affect Performance

One of the key questions to answer here is how the addition of threads within an application affects overall system performance. To answer this question we looked at the performance difference between using a single thread for all our simulated workload and using up to four threads. Two experiments were executed—one using a test signal graph and another using an FM reception graph.

Figures 5 and 6 show the overall performance of the SDR application for the two different graphs. Performance can be measured based on signal quality, ranging from 0% (no signal) to 100% (full signal). The signal quality was calculated based on the number of dropped packets from the audio stream. If a stream is transmitting 96,000 samples per second, if only 90,000 are processed in one second, we deem the signal quality to be 90,000/96,000 $\simeq$ 84%. When there is a drop in quality we conclude that the workload applied to the graph at that point is greater than the host can handle. From here, all GNU Radio results will plot the trend of workload (x-axis) vs. signal quality (y-axis) based on a set of parameters, such a thread count and active cores.

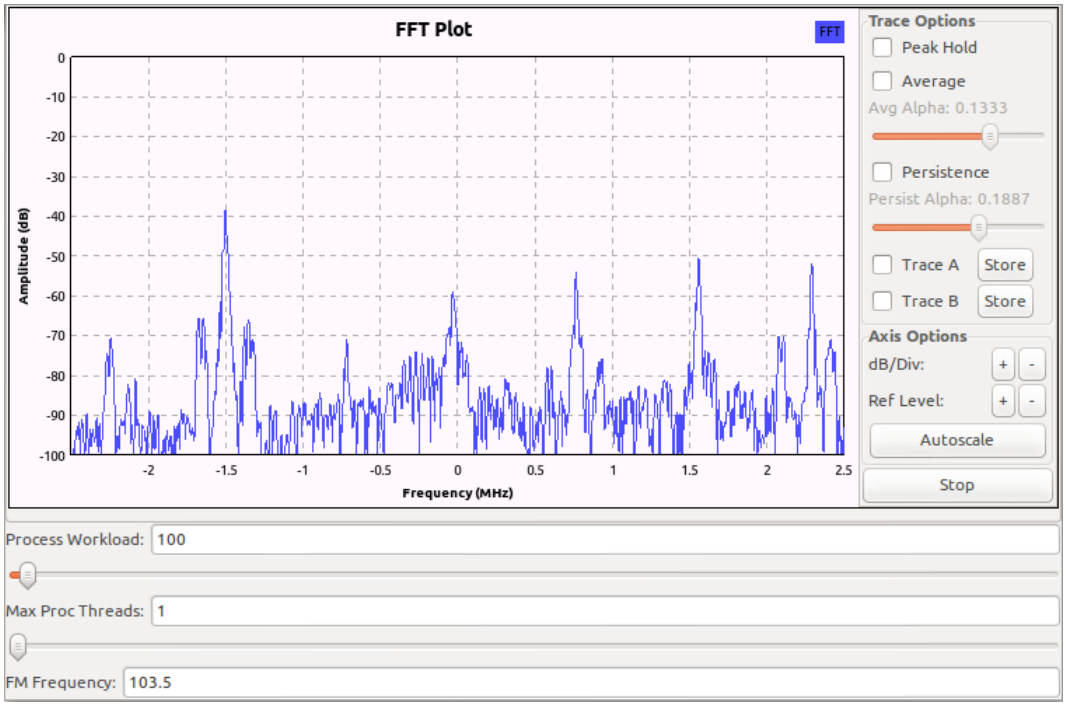

Figure 3. Test Framework GUI in manual mode.

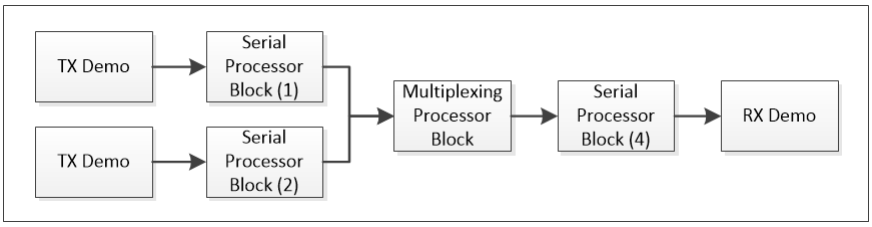

Figure 4. Multiplexing OSSIE simulation graph.

Figure 6 shows how multiple threads affect performance of the host. Maximum throughput is achieved regardless of the parallelism preformed in the Processor Block whilst the workload is low. However, when the workload reaches about 10,000 iterations of the time wasting loop (a sequence of floating point divisions), performance is affected. There is a gradual decline in performance for the single threaded Processor Block; however, the effects of the workload are not seen until later for a multi-threaded block. This clearly shows that the introduction of threads gives an immediate performance improvement over a single threaded solution. This could be attributed to a couple of properties, such as the workload being distributed over different cores. Another possibility is that there could be a blocking call within the Processor Block. We discovered through additional studies that the cause was the former, that is, the way the workload is distributed among the various cores. We discuss this issue further later on when we report about experiments in which we restricted the use of cores.

Another point to mention from this experiment is to what extent performance improved when more threads were introduced. There is a notable improvement when using more threads but the degree of these improvements vary with the workload. For example, at the 40,000 division workload point, doubling the number of threads almost doubles performance (55 vs. 110 vs. 205 units of packets delivered depending on whether 1, 2, or 4 cores are used). This is near the maximum theoretical speed-up that can be achieved by increasing the number of resources. However, looking at another data point (e.g., 60,000 floating point divisions), the performance difference is still substantial, but not quite as dramatic with a performance of 40 vs. 80 vs. 130 units of packets delivered. This could indicate that while introducing threads can improve performance to near optimal levels, there may be other limiting factors.

As for the test signal graph in Figure 5, we see a similar pattern with a different scale (since there is more work in processing an FM signal than generating a sine wave). When the number of utilised threads doubles, the performance also doubles. Even though this is not even using the SDR but a

self contained software experiment, it is useful to see how the parallelisation is not strictly dependent on the source.

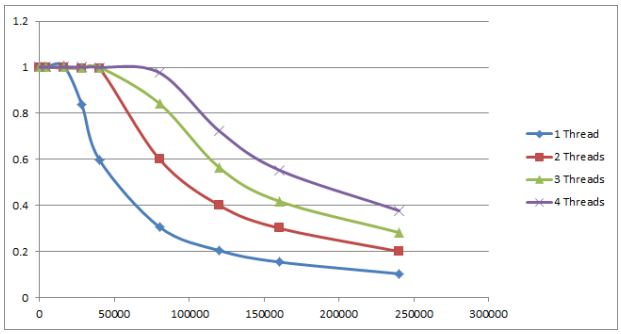

Figure 5. Results showing throughput while playing back a test signal.

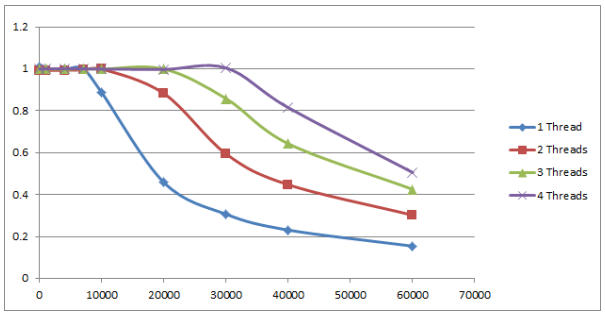

Figure 6. Results showing throughput while playing back FM on multiple cores.

6.2 Threading on a Single Core

We have already shown that introducing threads into the SDR host application may dramatically improve performance. However, it could be the case that software threads are responsible for the improvement and not the introduction of multiple hardware cores (which may be the case if blocking calls are made). Our next set of experiments used the FM receiver on a single core. This was accomplished by setting the process affinity (locking a process to a subset of hardware cores) of GNU Radio Companion to run on only one of the available hardware cores.

Figure 7 shows the performance when only one core is active using the floating point division simulation function. This helps prove our earlier claim that the original performance gains were due to threads being allocated to different cores. Evidently, when there is only a single core running the radio, there are no performance increases when multiple software threads are introduced. This shows that in this system multicore is responsible for the vast performance increases and that software threads simply unlock this potential.

There are two more points of interest here. First, Figure 7 also shows that even if threads are introduced, performance will not be degraded either. This is quite an important point since code that can be written using threads can have instant benefits when run on a multicore system. In such cases, there is the potential for only an insignificant penalty when run on a system with fewer cores.

While software threads can unlock the potential gains that can be accomplished by multicore hardware, the indiscriminate addition of software threads may also hurt performance. Threads do incur an overhead, such as context switching. If a significant number of additional threads were to be created and used, these overheads may accumulate into poor performance.

We also compared the data in Figure 7 to the multicore execution when using a single threaded Processor Block (Figure 6). Even though in both cases only one software thread is used in the Processor Block, there is higher performance when still using multicore. This could be down to multi-threading elsewhere in GNU Radio. GNU Radio, by default, will run each of its nodes in a different thread. So even if the Processor Block is running in a single thread, the USRP Source block will run in a different thread, as well as each other node in the graph. This is an important finding as we inherit some implicit multi-threaded properties from the framework that can be used by the multicore CPU.

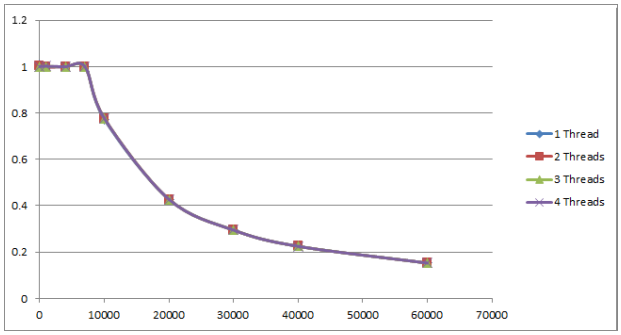

Figure 7. Performance playing back FM on one core.

6.3 The Single Threaded Framework

It was already seen that performance gains can be instantly made in GNU Radio by utilising more cores due to the GNU Radio framework. One interesting option GNU Radio gives to the user is to select different node schedulers in the framework. In particular, a user can choose whether the framework should run using a multi-threaded scheduler (MTS) or single-threaded scheduler (STS), where the multi-threaded mode will execute each component in a different thread, while the single-threaded scheduler runs all node operations in a single thread. It is mentioned that a reason for using the single-threaded mode instead is when there is a need to use hardware, such as graphics cards via Cuda [14], to do some processing in a component.

When each components runs in a single thread, this does not affect the execution of other threads created within that component. Thus, for some insight into this framework property, we ran experiments with a single-threaded Processor Block using either a multi-threaded or single-threaded scheduler. Figure 9

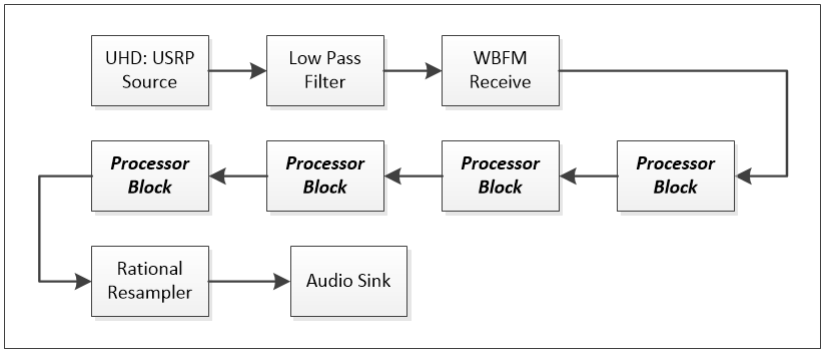

Figure 8. GNU Radio Companion graph with multiple Processor Blocks.

shows the difference between using the two framework modes. This figure shows the performance under both framework modes as well as 1 vs. 4 active cores.

The results shown in the figure are quite interesting. The only case in which there is a significant drop in performance is when the single threaded scheduler is used on multiple cores. Performance actually improves when the scheduler is locked to a particular core. Looking at the Linux system monitor during execution, for the single-threaded scheduler on multiple cores there is quite a bit of variation over what cores the application is run on. This is in contrast to the case when there is only one core for it to use and there is a single flat line of usage on one core. Therefore, even though an application can get instant benefits moving to multi-core, if it is in nature a serial application, like GNU Radio when using the single-threaded scheduler, multicore can actually hurt performance. (See Figure 7.

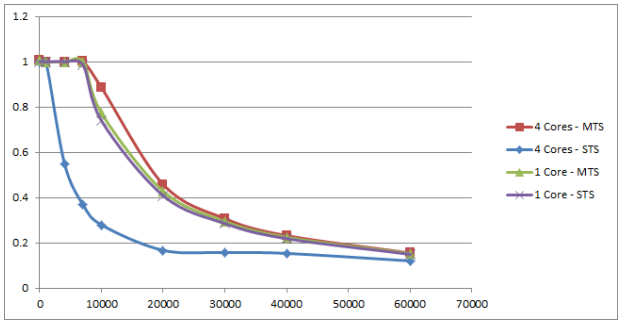

Figure 9. Comparison of scheduler types

6.4 Inter-Component Parallelism

Another variation of the system configuration is to use the inherent multi-threaded framework given in GNU Radio to exploit multicore architectures. Since GNU Radio runs each component in its own thread (in multi-threaded mode) it should be possible to share some workload between components, more specifically, components of the same type. Therefore, instead of running a single Processor Block with four threads, we should achieve similar performance by chaining together four Processor Blocks each using a single thread. Such a GNU Radio graph can be seen in Figure 8. Intuitively, this model has many benefits. The major benefit here is that it mitigates concurrent development from the developer. Since developing multi-threaded code is time consuming, this requires additional expertise on part of the developer, and can introduce concurrency issues. Thus, any mitigation is desirable. This extra encapsulation also reinforces good software engineering principles. Another benefit is reuse, as this component can then be used as many times as necessary via a graph modification instead of editing the code, introducing more threads within a component. One potential downside to this assignment of threads to components is the overhead that may be imposed by the framework, as there will be a cost in performing these extra transitions within the graph.

The results of splitting workload between blocks can be seen in Figure 10. There is the expected improvement when spreading workload over more blocks, similar to intra-component parallelism (i.e., parallelism obtained by creating threads within the Processor Block). However, it can also be seen that there is a notable decrease in performance when moving from the intra-component to inter-component parallelism. As we already mentioned, some performance degradation is expected, however, there are cases when there is a 20% difference in performance. Even though inter-component parallelism has yielded some improvements, there are cases where performance is significantly worse than their intra-component counterparts. We further investigate this issue below.

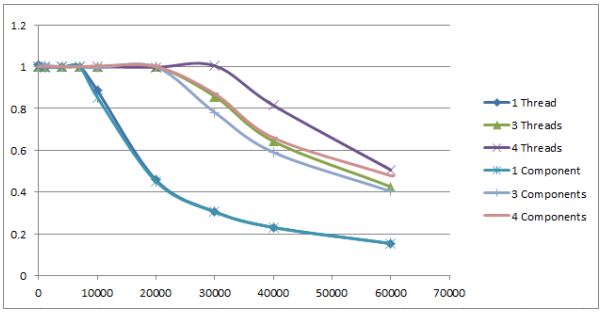

Figure 10. Performance of intra vs. inter-component parallelism.

6.5 Isolating Inter Component Bottleneck

In the previous subsection, we reported finding a test case when inter-component performance differs from intra-component performance. We decided to investigate this discrepancy using performance profilers. Some popular profilers, such as gprof [15], can be fairly intrusive. They can require rebuilding the entire executable which will effectively change the code, hence changing program flow and performance. This may not be desirable, especially when we are trying to identify bottlenecks in a large project.

A less intrusive option is OProfile [16], which has also already been used to successfully profile GNU Radio [17]. Instead of relying on instrumentation, OProfile polls the OS kernel to identify what applications, and functions within those applications, are being executed at any one time. The polling rate is sufficiently large to get an idea of where the program is spending most of its time, whilst having little effect on the overall runtime. The results of profiling our two different experiments, which can be seen in Table 2, were surprising.

Profiling has shown a couple of differences between the two parallelism methods. The first increase is in libpangoft2, which is a text rendering library [18]. This seems like an unlikely culprit in this case and may also be a side-effect of the additional text messages printed by GNU Radio Companion during a period of stress on the graph. The other increase is within the component itself. To be more specific, there was a slight increase in the amount of time the application spent within the simulation function of the Processor Block when aiming for inter-component parallelism. This may suggest that the floating point division is more efficient when executed in different threads within the same component rather than in separate components.

6.6 Differing Simulation Processing

Our finding on Processor Block performance introduced another variable into the pipeline; the type of simulation that is being executed in the Processor Block. So far we have found out that there is more time spent within the simulated processing function when using inter-component threading. We were to discover what processing, if any, promotes this behaviour. Therefore, we created several different test cases that exercise different software properties to see what operations can affect performance. We experimented with various kinds of blocks differing in the kind of "time-wasting" operation that they perform. The kind of operations that we considered include the following:

- In place integer addition and subtraction
- Memory copy with floating point division
- Recursive function calling
- Large integer array reversal (using a temporary variable)
- Large integer array reversal (using XOR swap)
- In place integer addition and subtraction
- An empty nested loop
- Large dynamic memory allocation
- In place floating point addition and subtraction
- In place floating point division

In each of these cases we observed the difference between performance with respect to inter vs. intra-component parallelism. Interestingly, it was found that it is possible to achieve similar performance with either methodology if the Processor Block exhibited certain computational properties.

We consider our previous simulation block, in place floating point division, as our baseline (Figure 10). In this case it is clear there is a performance difference of up to 20% at times. However, when we perform an in place integer addition and subtraction in our experiments (Figure 12), the differences are eradicated with both inter-component and intra-component parallelism yielding similar results. This suggests that there are some inherent properties with the original Processor Block (performing floating-point divisions) that do not work as well when encapsulated in other components. This could be due to contention for shared hardware resources, such as arithmetic units.

We also observed that some operations showed a general degradation between components. Reversing an array of 1 Million integers (Figure **??**) also showed a slight degradation of around 5%; however, the method of performing the swap (using a temporary variable or exclusive or) did not make cause a substantial difference.

There were further issues when experimenting with array reversal methods in intra-component mode. As can be seen in Figure 11, the previous optimal gains were achieved when the number of threads was doubled from 1 to 2. However, the biggest discrepancy seen here from previous intra-component experiments is with the marginal improvement from 3 to 4 threads. The improvement is less than 5%, which is far less than the gains seen in the addition/subtraction experiments. This could hint towards memory or cache related bottlenecks. In such a case, multicore does not offer the gains that we'd hope for due to contention for other hardware resources.

The most significant difference was with respect to dynamic memory allocation. Figure 13 shows a simulation function which allocations a 100MB block of RAM on a system (which has 6GB of RAM). Any kind of concurrent allocation has a significant performance slow-down of over 80%. This indicates that memory allocation in a concurrent environment may cause an exhorbitant performance penalty. We have not looked further into what parameters affect performance; however, using a test framework like the one we developed may be of use to a systems developer. They could add various memory allocation parameters to the framework, such as allocation length, and see how this value affects their system. After all, there may be other limiting factors, such as the memory bus on the motherboard and the speed of RAM.

7 Conclusions and Future Work

Our empirical study leads to several conclusions. First, we have demonstrated a test framework that shows the conditions upon which multicore CPUs and software threading can be effective in an SDR environment. We feel that such a simple interface can be of use to any concurrency project and the ideas here can be domain independent and platform agnostic. The automated approaches here have been invaluable in our

Profile for intra-component parallelism		Profile for inter-component parallelism	
% of Time in Symbol	**Symbol Name**	**% of Time in Symbol**	**Symbol Name**
88.9296	ProcessorBlock::InPlaceFpDivision(int)	89.6622	ProcessorBlock::InPlaceFpDivision(int)
4.3517	fcomplex_dotprod_sse	3.7241	fcomplex_dotprod_sse
1.4393	libpangoft2-1.0.so.0.3000.1	1.2072	no-vmlinux
1.0082	no-vmlinux	1.1898	libpangoft2-1.0.so.0.3000.1
0.8701	python2.7	0.8408	python2.7
0.7006	no-vmlinux	0.6902	no-vmlinux

Table 2
Output of O-Profile, showing system wide CPU utilisation.

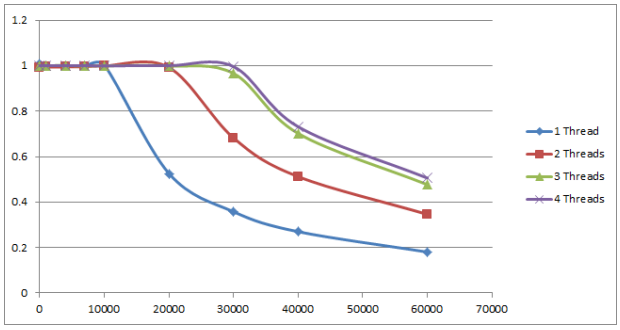

Figure 11. Performance of integer array reversal between threads.

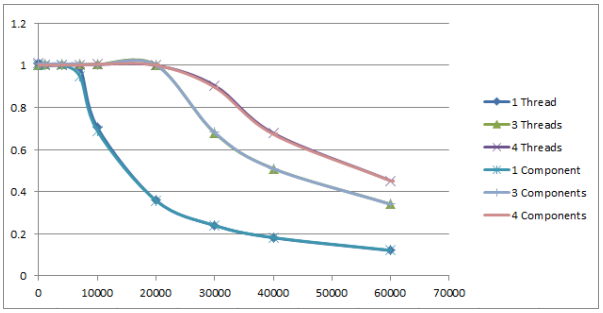

Figure 12. Performance of inter/intra component parallelism with an addition/subtraction simulation function.

experimentation, but use of this approach could reach farther afield, for example, performance testing could be used during the normal automated software build and test cycles. It would be easy to use frameworks similar to ours in test cases; issues could be reported if metrics exceed threshold values. Even for the developer, our framework can be a useful tool in order to tune their use of concurrency. A simple example here is to avoid large concurrent memory allocations.

Secondly, it is clear from our results that going multicore can have instant improvements. In the case of GNU Radio, its use of multiple threads makes it an ideal candidate as switching from one core to four increased performance significantly. In general, parallelising over multiple cores gave good increases in performance. However, if the application is a serial one,

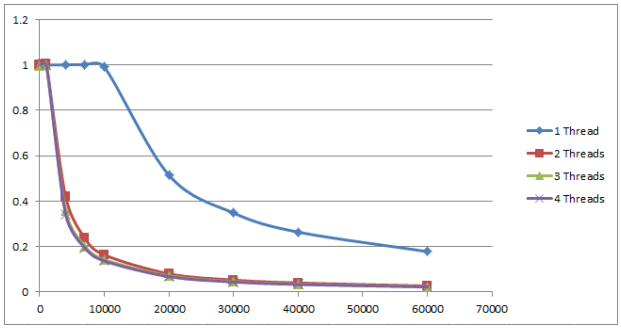

Figure 13. Performance of concurrent dynamic memory allocation.

going to multicore may actually hurt performance. Another setback for multicore is when there is the need to interact with other resources, such as RAM, which can significantly degrade performance.

Finally, even though splitting work between cores can work for a particular case, a systems architecture may come into play. We showed some evidence that splitting work between nodes, while should be promoted as good software design, may have an adverse effect on system performance.

Here, we have shown some results into our investigation on the effects that multicore CPUs have within the domain of software defined radios. Some of the results were conclusive that a vast, near maximal increase in performance can be achieved, but only in certain circumstances. A possible future line of research is to find out what operations can be parallelised, what cannot be parallelised, or maybe to find out what different operations can be performed in parallel in order to get maximum performance out of the system.

8 Acknowledgements

We would like to thank our sponsor, Rockwell Collins, for funding research into this project.

References

[1] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, p. 1113, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1430623

[2] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, p. 5259, 1997. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=591665

[3] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs Journal*, vol. 30, no. 3, p. 202210, 2005. [Online]. Available: http://www.info2.uqam.ca/ tremblay/INF5171/Liens/sutter.pdf

[4] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, p. 483485. [Online]. Available: http://dl.acm.org/citation.cfm?id=1465560

[5] F. K. Jondral, "Software-defined radio: basics and evolution to cognitive radio," *EURASIP Journal on Wireless Communications and Networking*, vol. 2005, no. 3, p. 275283, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1088725

[6] M. Ettus, "Universal software radio peripheral," *Ettus Research, Mountain View, CA*, 2009.

[7] ——, "Ettus research, LLC," *Online information on USRP board). http://www. ettus. com*, 2008.

[8] R. North, N. Browne, and L. Schiavone, "Joint tactical radio system-connecting the GIG to the tactical edge," in *Military Communications Conference, 2006. MILCOM 2006. IEEE*, 2006, p. 16. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4086476

[9] M. Robert, S. Sayed, C. Aguayo, R. Menon, K. Channak, C. Vander Valk, C. Neely, T. Tsou, J. Mandeville, and J. H. Reed, "OSSIE: open source SCA for researchers," in *SDR Forum Technical Conference*, vol. 47, 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.138.7486&rep=rep1&type=pdf

[10] M. D. Pesce, *Programming Microsoft DirectShow for Digital Video, Television, and DVD*. Microsoft Press, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=863598

[11] R. Otte, P. Patrick, and M. Roy, *Understanding CORBA (Common Object Request Broker Architecture)*. Prentice-Hall, Inc., 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=230247

[12] E. Blossom, "GNU radio: tools for exploring the radio frequency spectrum," *Linux journal*, vol. 2004, no. 122, p. 4, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=993251

[13] J. Blum, *GNU Radio Companion*, 2011.

[14] C. Nvidia, *Programming guide*, 2008.

[15] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, p. 120126, 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=806987

[16] J. Levon and P. Elie, *Oprofile: A system profiler for linux*, 2004.

[17] T. W. Rondeau, "Application of artificial intelligence to wireless communications," Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2007. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-10052007-081332/

[18] O. Taylor, *Pango: internationalized text handling*, 2001. [Online]. Available: https://old.lwn.net/2001/features/OLS/pdf/pdf/pango.pdf