A buffer-management system to simplify the design of streaming applications

Philip Garcia
University of Wisconsin-Madison
pcgarcia@wisc.edu

Johannes Helander
Microsoft Research
johannes.helander@microsoft.com

December 20, 2007

Abstract

In recent years there has been increased interest in the design of parallel systems. Multicore processors have become commonplace in general purpose computing, and have become increasingly important in the design of new embedded devices. These systems require a new approach to programming, as standard programming models are based upon a single thread of execution that cannot fully utilize the processing resources of the processor.

In this work we propose a method for parallelizing applications through the usage of stream buffers. While stream processing has been around for many years, our system is designed to fully decouple application control flow from the programming model, and creates a simple environment through which programmers can combine various components to create a full application.

1 Introduction

Parallel programming is a difficult task that is not adequately solved by current tools. Designing such applications requires the developer to explicitly order the execution of threads, and ensure correctness for the near infinite overlays of the multiple threads [8]. Such applications are difficult because contemporary programming models are designed to work with sequential Von Neuman computers that assume serial execution of a program. When programming Von Neuman systems, the control and data flow are explicitly given in the programming language.

When attempting to design parallel systems within the constraints of contemporary programming languages (C, C++, Java, etc.), this explicit description of control and dataflow is no longer inherent. While the application can easily show the progression of control and data within a single thread of execution, they cannot adequately express the interactions of the various threads within the application. Without constraints governing their execution. it is almost impossible to design an application that behaves predictably and correctly[3, 8]. While structures such as semaphores, barriers etc. have eased the design of parallel programs, their usage is generally left to experts who have studied parallel algorithms in great depths [8]. Additionally using these structures is error prone, and hard to visualize, as the control flow of the application is no longer contiguous. By this I mean that the control flow can only be observed by examining multiple parts of an application that specify the application's behavior under exceptional conditions.

Much research has been done on parallel program, however until recently much of this work was limited to the high performance computing (HPC) community, where a relatively few number of experts developed scientific applications. Many techniques developed by the HPC community have concentrated on the parallel execution of loops, using the single program multiple data (SPMD) paradigm[4, 7]. This work can greatly accelerate many scientific codes, however, but itself it cannot accelerate many current applictions. More recently, this paradigm has been used by compilers to automatically extract parallelism from loops or small subgraphs of computation[10, 2]. However extracting parallelism from a large application is made much more difficult by the interdependencies between these subgraphs.

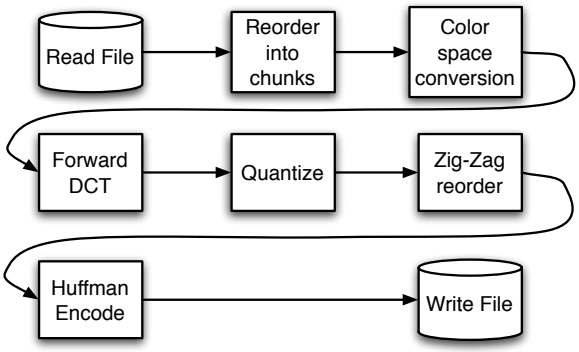

Figure 1: An example pipeline expressing the flow of data in a JPEG encoder.

2 Execution Environment

In order to simplify the design of parallel programs, we believe that an application should be designed with concurrency in mind. This helps to separate the application flow from the control and data flow that exist within a single component. In order to accomplish this goal, we propose breaking down an application using a form of the pipe and filter design pattern[9]. By doing this we split the application into a data flow graph that expresses the program as a series of filters or "components" that the data flows between. Figure 1 shows an example of this paradigm when applied to a JPEG encoder. Under the pipe and filter paradigm, data is passed between the components, and all of the components in the system are capable of running in parallel (assuming all of them have input data available to them, and a buffer in which they can write output data)

It is important to differentiate our components from more general functions and subcomponents used in programming. In our design, a component can consist of many different functions or subcomponents, and the execution within a single component is purely serial. A component may not call another component (although it's output can be passed to another one using a buffer), and must accept one or more buffers of data as input and/or generate one or more buffers of data as output.

We propose doing this by using a standardized buffer structure combined with the pipe and filter design paradigm to create applications. Applications for our system are expressed using a simple XML syntax that should facilitate the design of a graphical tool for expressing parallelism. This approach allows for hierarchical formations and patterns to be used to help simplify the system's descriptions. The methods of extracting/specifying parallelism are described in detail in Section 4

Our system is designed to support applications that run on anything from small embedded systems to large distributed multiprocessor systems. This means that our applications can be "compiled" for parallel or serial execution. Invoking the application to run as a "single-threaded" process can be useful for development purposes. It is much simpler to ensure that the functionality of the program is correct and debug the application when it is executing serially. Once proper serial execution is ensured, the developer can then start adding parallelism to the picture. As all, or most of the locking code is handled by the system, there should be few concurrency bugs, however such bugs can still occur depending on the component. It is our belief that developers who follow our design methodology will find it much simpler to develop "bug free" multithreaded applications.

Our current implementation reads in an XML description of the application at execution time, and dynamically creates and initializes the buffers and windows that will be used to push data through the system. All filter chains should include a "data pump" as well as a "drain" to fill the pipeline with data or to finish executing data. A pump or drain could consist of a component that simply reads or writes data to/from a file or sa network. The overhead of dynamically creating these filter chains should not be much more significant then calling a function linked from a library. This flexibility allows an application to dynamically load filter chains during execution, or even create a new filter chain from a current chain.

While there has been much work in using streams to express parallelism[1, 11], many of them have not allowed the application designer to adequately express their applications, or required the application designer to express the details of the buffers used to pass data between pipeline stages. We will therefore focus our design on the buffer system used to pass data between the various stages in the pipeline, as well as the window interface that is exposed to the programmer.

3 Buffer Structure

The buffers, and their accessors are designed to be both efficient and easy to use. This task is difficult as not all applications can specify the size of data that is held within their buffers. The various constraints placed on the buffer means that no single buffer structure is sufficient for all situations. Because of these constraints, we have created a layered approach to the buffers. Figure 2 shows a mapping of how the buffers are layered on top of the system's virtual memory.

When using our tool, a buffer pool is created, based on the applications needs, and information specified within it. While the pool should be relatively static in size, it is designed to be flexible to adapt to changing application conditions. At the physical layer there are two pools that we choose buffers from: the buffer header pool, as well as the buffer pool itself. The headers can easily be expressed as an array, or chain of arrays, while the actual buffers are expressed as a list of buffers of varying sizes.

The physical buffers are contained within a logical buffer chain. A logical buffer chain consists of one or more buffers "chained" together by the header structures. Each link in the buffer is responsible for keeping track of its ownership, size, sequence number, length, and whether it is the last buffer in a stream.

The size of the logical/physical buffers is obviously dependent on the application. A component that passes lists of integer numbers will obviously have different constraints placed on the buffer size than a component that is passing multiple KB data structures (although for this case it might make more sense to pass pointers to structures contained on the heap). The buffer structure should therefore conform so it can not only allow multiple data sizes, but also allow for multiple data elements to be passed at a time. In Section 6 we will examine the impact of varying the size and number of buffers on our test application.

3.1 Virtual Buffers

Many applications that use our buffers need to be very efficient in their handling of the buffer data. Additionally, some applications (such as the BSD TCP stack) regularly rearrange buffers. These operations are not well supported with most buffer structures, as operations such as insertion or deletion often require copying the data before the insertion point to a new buffer, inserting the new data, and then copying the data after the insertion point into the new buffer. These operations can be highly expensive, and are not optimal for many applications.

To support this rearranging of data, we introduce the concept of virtual buffers. In Figure 2 we see an example of virtual buffers mapped onto physical buffers. In this example portions of the logical buffers have been "carved" out of the virtual buffers to produce a virtual mapping of the data in the buffers.

To support these virtual buffers, we have created functions that can insert one or more data objects into a virtual buffer (or more general within a window inside the buffer). Additionally, functions exist to delete one or more objects in the virtual buffer, or to copy data between buffers.

```
FFT_Component()
1  while ( InWin.Advance(1))
2        do
3            OutWin.Advance(1)
4            InBuff ← InWin.Get(In)
5            OutBuff ← OutWin.Get(Out)
6            in ← in + 1
7            out ← out + 1
8            DoFFT(InBuff, OutBuff)
```

Figure 3: An example of how a window buffer might be used to perform a DCT upon the input buffer, and write the results to an output buffer.

3.2 Buffer Windows

To better manage the buffers, we have created a system of "windows" that look into a buffer. All actions that a component performs on a buffer occur through this buffer window. A buffer window allows array like access to a buffer, but still limits the component's access to the buffer. Limiting access to the buffer allows the system to share buffers between components, increasing concurrency. Figure 3 shows an example of how a buffer might be used in a program.

By using windows to look into the buffer, we solve the issues that may occur in components where item i-n may be needed concurrently with item i. This removes the need to have to either

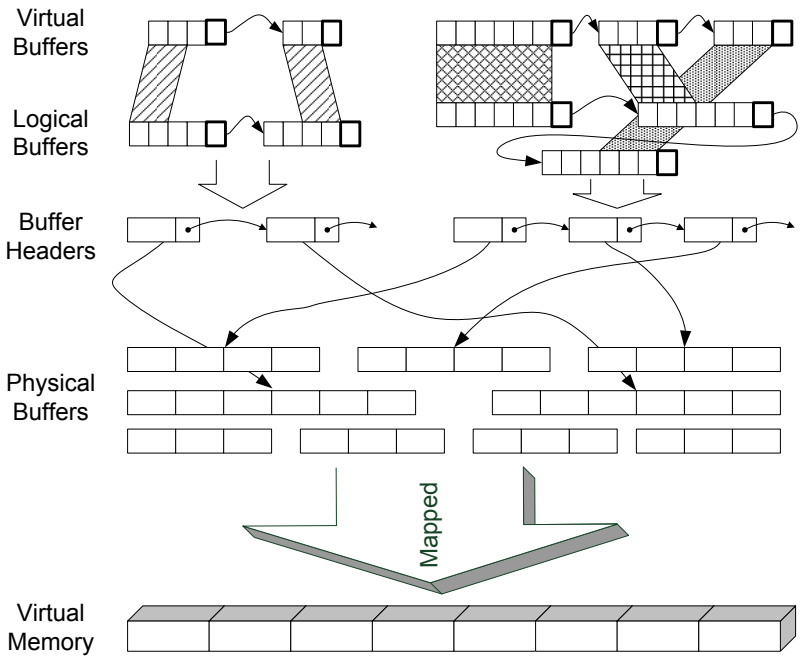

Figure 2: Depiction of how the different layers of our buffer manager interact, showing the mapping of virtual buffers down to the virtual memory subsystem.

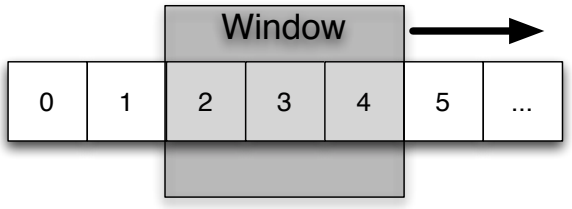

Figure 4: Depiction of how the data window moves along data contained within a virtual buffer.

"rewind" a buffer to access "older" data items or first copy the data items into local memory. By using a window we can can explicitly define which items in the buffer are currently in use, and can request to access more or less data based on outside conditions. Figure 4 shows how a window moves along its input (or output) data buffer.

A window will generally be set to operate in either input or output mode. In input mode, the buffer is used to read data into a component, and in output mode the component writes data to the buffer. However these two modes of operation can not efficiently work with all types of data. For instance if a component within the application inputs a list of integers and outputs a list of the same integers plus one, it would be more efficient to overwrite the input data with the generated output data. This can reduce memory requirements, and reduce cache contention. We solve this problem by creating special "In-Out" buffers.

Another aspect that the buffer manager must be aware of is whether the data being passed contain pointers. When multiple components can access data not directly contained within the buffer structures, it is left up to the application developer to ensure the correctness of the application. This is best done by either locking these external data structures, or preferably by not letting a component access the data pointed to by the pointer once the pointer is no longer in the buffer's window. In this situation, when component R_i advances its window so that it can no longer see a pointer it read/wrote, R_i should remove any local references to the pointer that it might have. Additionally the system must be aware if pointers are passed, as any component R_{i+j} $(j > 0)$ that uses R_i's output buffer must run on a processing element that shares the same address space as component R_i.

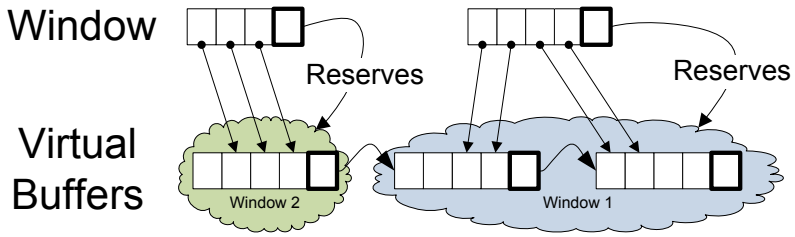

Figure 5: Depiction of how a window operates on a virtual buffer.

3.3 Window modes

Because different applications have different ways of accessing data, and different performance requirements, no single window interface is acceptable. We have therefore decided on three modes of operation that should make programming with these buffers simpler. Under the first two modes of operation, we assume that the data stored in the buffer is of a fixed size, while under the third mode, the data can be of any variable size.

Under the first mode, we allow the component to access any member of the window individually. For instance, if the window size was large enough to hold 5 objects of *object_size*, the programmer could specify to access items W_0, W_1, W_2, W_3, or W_4. However, because the window can cross multiple virtual buffers, the programmer can not directly perform address calculations. I.e. the address of W_1 is not $W_0 + object\_size$. Because of this, this buffer mode is not recommended for components where the amount of work performed on each data item is small. For example, text processing would be extremely inefficient, as the address of every character would have to be looked up.

Under the second mode of operation, the window manager ensures that every window presented to the component is contiguous in memory. This is done by examining the window boundaries whenever the window is moved, and, in the event that the window crosses a physical buffer chain boundary (it can cross a logical one provided that the physical buffers are contiguous in memory), use a temporary buffer that has a copy of the appropriate data located within it. This method is appropriate for many text processing algorithms, but it is important that the window is relatively small compared with the size of the buffer to minimize the chance that a window straddles a buffer boundary, as well as the minimize the amount of data that must be copied to temporary buffers.

The third method of dealing with windows is to directly expose the programmer to the underlying buffer structures. While this model is not an ideal way of working with data, such a model is required for components where the data sizes are unknown. This is a requirement for applications performing bit-level data manipulation such as Huffman encoding. This is because the size of each element can not be known in advance, and it is entirely possible that independent data elements will straddle buffer boundaries. While this method makes dealing with the stream data more difficult, most components that manipulate data in this way require careful coding to begin with, and a couple additional constraints should not greatly complicate these algorithm. This method can also be used for text processing algorithms where the performance hit of the second method is too great.

To help facilitate the usage of these different window accessors, we plan on creating a set of components that can perform common operations. These methods will be accessible under all of the window modes, and will support copies, insertion, deletion etc. Additionally these components will be capable of performing complex virtual buffer manipulations that can greatly increase the efficiency of such operations.

We currently have implemented the first (single access) window mode, however we have left the development of the other access modes for future work.

4 Extracting Parallelism

Extracting parallelism from an application requires the designer to fully specify the flow of data amongst the various components in the program. To express this parallelism we created a tool that

```
<AppStream xmlns="http://tempuri.org/X-Buffer/0.01">
  <Stream>
    <Component type="COB\CFromPPM.cob"
                  name="Source" init="true"/>
    <Component type="COB\CColorConv.cob"
                  name="Conv" init="false"/>
    <Component type="COB\CFDCT.cob"
                  name="DCT" init="false"/>
    <Component type="COB\CQuant.cob"
                  name="Quant" init="false"/>
    <Component type="COB\CHuff.cob"
                  name="drain" init="false"/>
  </Stream>
  <Chain>
    <link><name>Source</name><Win>0</Win></link>
    <link><name>Conv</name><Win>1</Win></link>
  </Chain>
   <Chain>
    <link><name>Conv</name><Win>0</Win></link>
    <link><name>DCT</name><Win>1</Win></link>
  </Chain>
  <Chain>
    <link><name>DCT</name><Win>0</Win></link>
    <link><name>Quant</name><Win>1</Win></link>
  </Chain>
  <Chain>
    <link><name>Quant</name><Win>0</Win></link>
    <link><name>drain</name><Win>0</Win></link>
  </Chain>
  <Initializer>SestupJPEG</Initializer>
</AppStream>
```

Figure 6: An example XML file used to describe a JPEG encoder

reads in an XML description of the program's components, and automatically creates the buffer structures and control code necessary to run the program on the target platform. It is believed that the layout of the XML could be done within a module designed for Visio. This will allow a graphical layout that generates the application description. The design of this module is left for future work, and this work will concentrate on the tools necessary to generate the applications.

In order to take advantage of streaming parallelism we must know some information about the flow of data in the system. For instance, many applications first initialize many data structures before other components can execute. These components must run to completion, and may provide global data that the other stages read from (if the other components modify these structures, care must be given to ensure correctness of the program).

Each component in a filter chain can optionally have an initializer function. The initializer function is passed a parameter containing data it needs to operate. To coordinate the multiple initializer function, the XML file describing the program contains an initializer field. This allows the developer to specify a single function that initializes all of the components within it. This makes it clear where initialization should occur resulting in more readable and maintainable code. Synchronization of all the components is handled internally, so that any component that requires initialization can not execute until all threads have finished their execution. This simplifies the model, and helps to avoid race conditions.

To simplify the descriptions of the program, subsections of the program should be expressed as modules that can contain multiple components. This allows the system designer to express hierarchy in the design, and makes it easier to express multiple paths that can exist within an application. The expression of multiple execution paths will be discussed in more detail in Section 4.1.

Figure 6 shows an example XML stream that we used for our JPEG encoder. In this example we first define each component, and then describe the chains that bind the various components together.

4.1 Multiple data paths

As our tools are designed for both contemporary and future computer architectures, and target multiple platforms, it is important that we allow the application designer to allow for multiple implementations of a module. This can be useful for applications that can be accelerated through ISA extensions (MMX, SSE2, etc) or special purpose hardware. While this work concentrates on designs for software, or designs that can be implemented in reconfigurable hardware, this approach applies equally well to systems with special-purpose coprocessors such as graphics cards.

When a developer wishes to choose between multiple implementations they can include a base module of the implementation. They will then define submodules that can be used to implement the base module's functionality. This is similar to using virtual functions in C++ classes, and defining subclasses that implement these functions differently (such as having a regular and vector-optimized implementation). The submodules can be defined as a single component, or as another module consisting of multiple components. Allowing multiple components to replace a single one can be useful in instances where a software component must first

rearrange the data so it can be more efficiently processed by a hardware module. Additionally for architectures where the hardware can not access memory, it may be necessary for additional pipeline stages to be added that store and load data from the actual hardware.

Splitting the implementation from the description of the program is important when considering the applications future maintainability. This provides a straightforward approach to refactoring an application to take advantage of new special purpose hardware or software components. Additionally, on a reconfigurable system, it allows the system designer to easily specify multiple hardware kernels with varying area/power/performance/accuracy trade offs.

4.2 Scheduling

Scheduling the various threads contained within an application is a non-trivial task. For a small embedded system it might be beneficial to define a static schedule that specifies where and how threads execute. However more complex systems will demand dynamic scheduling. For instance, a system containing components that can be implemented in either software or reconfigurable hardware should take into account, the size of the hardware kernel, the speed of the software version, the speed of the hardware version, how often the hardware is used, amongst other criteria [5]. As this work is focusing on the design of the system as a whole, we are leaving the individual scheduling components for future work.

5 Example Application: JPEG

To test our platform we developed a JPEG benchmarked based of the reference IJG code. We used the basic design show in Figure 1, and used the XML description shown in Figure 6 to implement our benchmark.

This benchmark combines some of the stages shown in 1, and instead has a component that reads in the PPM source image, one that converts the image from RGB to YCbCr (and downsamples the Cr and Cv components), an FDCT component, a quantizer, as well as a component that performs zigzag reordering and Huffman encoding. Each stage in this pipeline was capable of running independently.

We used the Microsoft Invisible Computing platform (also known as MMLite) to implement our benchmark [6]. The invisible computing platform is designed to facilitate embedded systems programming, and allows compilation to a number of different operating environments, including windows.

We chose to examine JPEG encoding because it is a common operation (particularly in embedded devices), and is an example of an application made up of many different components that each require extensive execution time. In Table 1 we see the results of profiling the reference JPEG code under linux. While our implementation was done in Windows, we hose to profile it under Linux as we lacked the tools to perform such an analysis under windows.

Operation	Percent of Execution
Colorspace Conversion	30%
Huffman Encoding	26%
Forward DCT	15%
Quantization	25 %
Downsampling	3%

Table 1: Profile of the reference IJG JPEG encoder when running on a 2.4GHz Core 2 processor.

This table shows that the JPEG pipeline is fairly well balanced between the different stages. Because of this, a streaming implementation of JPEG should be able to take full advantage of the inherent parallelism found between the independent stages.

6 Results

We tested our stream-buffere implementation on a quad-core 2.4GHz Core 2. When running the JPEG reference code we saw that it took approximately 4 seconds to encode a 256MB image under linux (using gcc and -O3), and approximately 7 seconds to run when compiled using Visual Studio under windows.

In Figure 7 we compare the performance of our streaming JPEG encoder with the IJG reference implementation. This graph shows that for our test, the reference code significantly outperformed our implementation. However it is believed that with future modifications we can tweak our encoder to beat out the reference implementation. We will

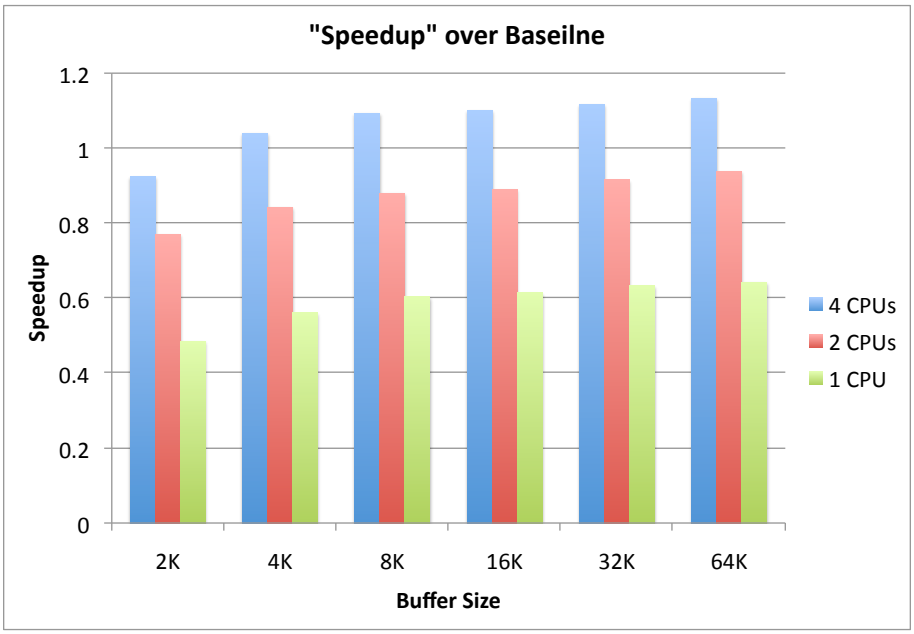

Figure 7: The "speedup" we obtained when running our JPEG encoder when compared to the reference baseline.

examine ideas for improving performance in Section 6.1.

While our baseline performance was not as good as anticipated, our results follow the expected trend. As Figure 8 shows, as we increased the number of CPUs, our speedup increased, obtaining a speedup of 40% with two threads (speedup here is measured over our multithreaded baseline when constrained to run on a single CPU), and a speedup of about 80% when we use 4 cores. While this speedup is not as good as the profiling results suggested we could get, we do not have an accurate profile of JPEG's execution when running under Windows, and are unsure how expensive some of the synchronization primitive we are using are.

It is also interesting to note in Figure 9 that the absolute performance of the system increased as we increased the size of our buffers. While this was expected for small buffer sizes (where having larger buffers can significantly decrease synchronization overheads), it was unexpected when we used larger buffer sizes. While not shown, when we increased the buffer size to 512KB, we noticed little change over the 64KB values. This suggests that the communication overhead was overshadowed by the synchronization overhead. This might be due to the fact that even when the buffers could not fit within the processor's L2 cache, the streaming bandwidth needed was less than that of main memory, and on chip prefetchers were able to adequately preload the data for us.

6.1 Future Work

As our performance was so poor, we plan on modifying our source to try and minimize the overhead of our buffer management system. While we are currently unsure of the exact reasons why our code performed worse than the baseline singlethreaded code, we have some ideas as to why it performed so poorly.

The development environment used was targeting embedded applications, and was not designed around running windows threads. The windows implementation was designed to work, and be useful for debugging your embedded applications, however performance was not its primary concern. This constraint is likely helping to limit our performance.

Additionally, the synchronization methods used were not optimized to run the fastest. We believe that by using better profiling tools we can measure what is causing our slowdown, and attack the problem at the source. This should result in code

Figure 8: The "speedup" we obtained when running our JPEG encoder when compared to the reference baseline.

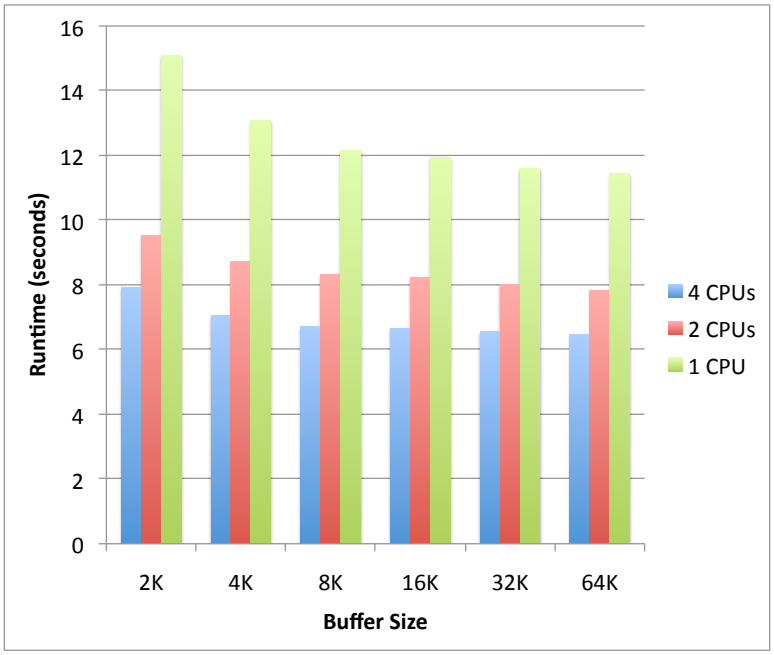

Figure 9: The absolute runtime (in seconds) of our JPEG encoder as we varied the size of our stream buffers.

that not only performs better but likely code that scales better as we increase the number of threads (with a likely maximum of 4).

We are also thinking of implementing other versions of JPEG that implement some form of SPMD parallelism, and that exploit processor features such as SSE2 to better optimize algorithms within the JPEG runtime. Both of these methods should increase overall performance, although using a more optimized algorithm should not effect the multiprocessor scaling of the algorithm.

References

[1] Paulo Barthelmess and Clarence A. Ellis. The threadmill architecture for stream-oriented human communication analysis applications. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 61–68, New York, NY, USA, 2004. ACM Press.

[2] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David I. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[3] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. pages 527–543, 2002.

[4] Leonardo Dagum. Openmp: A proposed industry standard api for shared memory programming. October 1997.

[5] Wenyin Fu and Katherine Compton. An execution environment for reconfigurable computing. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 149–158, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Johannes Helander and Alessandro Forin. Mmlite: a highly componentized system architecture. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 96–103, New York, NY, USA, 1998. ACM.

[7] Intel. Intel thread building blocks. 2007.

[8] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[9] Dennis M. Ritchie. The evolution of the unix time-sharing system. In *Language Design and Programming Methodology*, 1979.

[10] Shane Ryoo, Sain-Zee Ueng, Christopher I. Rodrigues, Robert E. Kidd, Matthew I. Frank, and Wen mei W. Hwu. Automatic discovery of coarse-grained parallelism in media applications. In *Transactions on HiPEAC*, pages 194–213. Springer-Verlag Berlin Heidelberg, 2007.

[11] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.