Parallelizing Audio Feature Extraction Using an Automatically Partitioned Streaming Dataflow Language.

Eric Battenberg and Mark Murphy
CS 267, Spring 2008

May 19, 2008

Abstract

We implement several versions of an audio feature extraction algorithm in a streaming data-parallel language, StreamIt. We attempt to use the StreamIt infrastructure for a performance study, but are disappointed by portability problems with StreamIt and a lack of optimizations for our target architectures in the StreamIt compiler; however, in certain cases, the compiler does produce noteworthy uniprocessor results.

1 Introduction

There is consensus among both industry and academia that the era of the sequential processor is over. For a variety of reasons, it is infeasible to continue the exponential increase in uniprocessor performance that we enjoyed over the past two decades. If we are to continue to improve the performance of computer systems, we are forced to innovate in radically different ways at the hardware level. So far, the quest to find hardware innovations that do not significantly affect the processor's programming environment has been unsuccessful. It is clear that we need to change the way we program if we are to take advantage of future processing platforms.

The wide array of processor designs available today represents a hybrid of two design goals. First, processor designers still desire to increase the amount of on-chip processing throughput with continuing silicon lithography improvements. In the absence of feasible microarchitectural improvements to processor cores, the most common solution has been to integrate multiple processor cores onto the same die, as well as to add SIMD instruction set extensions.Second, it is necessary to maintain compatibility with existing software infrastructure. Consequently, the interface exposed to the Operating System is usually that of a traditional Cache-Coherent Symmetric Multi-Processor (SMP). Despite several attempts [4, 5], processor designers have yet to find a highly productive general-purpose programming interface to novel processor designs.

In order to improve the performance of an application, software developers now need to produce parallel codes. To take full advantage of the available hardware resources, the end binary must utilize multiple granularities and styles of parallelism. Furthermore, a code optimized for one processor may perform significantly worse on a different processor, even if the processors are binary-compatible. It is clear that the currently available programming interfaces, specifically POSIX Threading with SIMD instruction sets, are insufficient for general software development.

We believe that domain-specific programming environments are a very enticing option. These environments, which may include programming languages, runtime systems, libraries, development tools, or simply programming methodologies, would incorporate optimizations specific to a set of similar applications executing on a particular processor or class of processors. Ideally, these environments would separate the important, but difficult to implement, performance-enhancing program transformations that are meaningful to an application domain. Likely, the resulting code will be less efficient than a hand-optimized code. However, the performance loss is easily justified by eased software development and debugging, since software developers no longer need to reason in terms of threads, locks, SIMD, or cache-coherence.

2 StreamIt

Description of the language, its goals, the claims of the MIT people about what it can do, etc. [3] [8] [9]

StreamIt[8] is a dataflow language that aims to improve the programmability of streaming applications, that is, applications that involve the processing of some sort of constant synchronous flow of informa-

tion. At the same time, the StreamIt compiler attempts to solve the parallel performance problem by automatically optimizing and partitioning computation across multiple processing elements.

The StreamIt language uses a simple C-like syntax to describe the structure of a dataflow program [9]. Computational steps are segmented into function-like constructs called "filters" which can be cascaded in a "pipeline". Task-level parallelism is described using the "split-join" and "feedback loop" constructs. See Figure 1 for illustrations of these basic StreamIt building blocks. Input/output rates are an important part of any streaming application, and the language handles this by using *pop,push,* and *peek* declarations for each filter. *Pop* and *push* handle the input and output rates respectively, while *peek* defines how far into the future the filter can look to access input values. In addition, the built-in *pop()* and *push()* functions are used to access input values and assign output values.

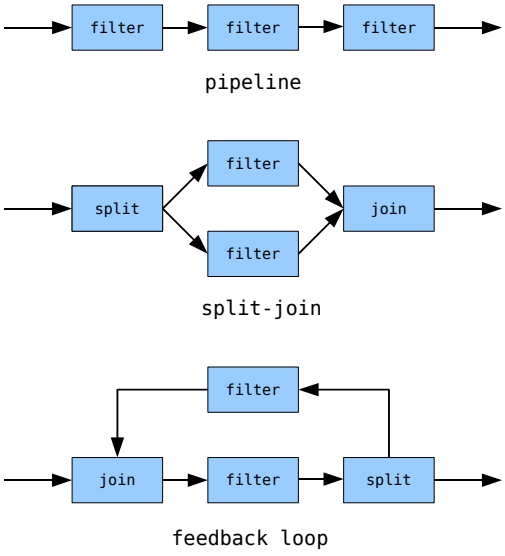

Figure 1: Basic constructs used in a StreamIt program.

Code examples for basic StreamIt constructs are shown in Figure 2.

The StreamIt compiler has been used successfully to optimize streaming applications on MIT's Raw architecture [3] but hasn't seen much attention regarding performance on x86 multicore architectures. The strategy behind the multicore optimizations of the StreamIt compiler is as follows. First, the granularity of the program is coarsened by "fusing" adjacent filters where possible. This reduces the amount of

```
//filter construct
float->float filter Downsample(int N) {
    // downsample by a factor of N
    work pop N push 1 peek 1 {
        int i;
        push(pop()); //output a single input value
        //consume the remaining (N-1) input values
        for(i=0;i<(N-1);i++)
            pop();
    }
}

//splitjoin constructs
float->float splitjoin ExampleSplitJoin() {
    // execute two filters in parallel
    split duplicate;
    add SomeFilterA();
    add SomeFilterB();
    join roundrobin;
}

//pipeline construct
void->void pipeline ExamplePipeline() {
    add Source();
    add Downsample(4);
    add ExampleSplitJoin();
    add Sink();
}
```

Figure 2: Code examples of basic StreamIt constructs.

communication and number of buffer copies required. Second, data parallelism can be exploited by creating separate copies of filters to be used on separate threads. Third, the compiler "pipelines" computation by attempting to optimize the order and distribution of work amongst processing elements in order to minimize idle synchronization time. These optimizations are detailed in [3].

Our aim is to evaluate the productivity of the StreamIt language for implementing music information retrieval feature extraction algorithm as well as the performance obtained using the StreamIt compiler to target the architectures outlined in the following section.

3 Architectures of Interest

The current architectural landscape is chaotic. In the past several years, the techniques used in the 1980's and 1990's for increasing sequential program performance have reached a dead-end. Specifically, we face three fundamental "walls" [1] in further performance improvement for sequential processors. First, we have reached the limits of feasible processor power consumption due both to leakage and to clock frequency scaling. Processor designs have stalled at about 3

GHz since the introduction of the Pentium 4, despite several generations of improvements in silicon lithography. In fact, superscalar designs more recent than the Pentium 4 are far less aggressive in terms of pipeline depth, speculation, and clock frequency, as these performance-enhancing techniques tend to decrease power efficiency. Second, core processor speeds have increased much faster than have DRAM speeds. Cache misses resulting in off-chip memory accesses stall today's superscalar processors for several hundreds of cycles. Since the sequential instruction sets of superscalar processors do not allow the program to effectively express memory-level parallelism, memory latency is the primary performance bottleneck of many codes on modern processors. Finally, despite significant research and development effort, it has been extremely difficult to extract further instruction level parallelism (ILP) without infeasibly complex, large, and power-hungry microarchitectural structures.

Different design teams have responded to these challenges with vastly different design decisions.

3.1 8-Core Clovertown

The most recent architectural iteration available from Intel is the Clovertown, of the Core 2 class of processors. The Clovertown's pipeline is a four-wide out-of-order superscalar implementing the IA32 instruction set with SSE2 SIMD instruction set extensions. The Clovertown machine available to us is a dual socket $\times$ quad-core, running at 2.66 GHz. Each of the eight cores includes 32 KB of L1 data cache, and each pair of cores shares a 4 MB L2 cache. For our chosen benchmark application, the aggregate 16 MB L2 cache is more than enough to contain the entire data set. Off-chip memory bandwidth will only be important for inter-core communication. On cache-coherent architectures such as Clovertown, all inter-core communication must occur via cache-coherency traffic. The two sockets in our Clovertown system share access to a single chipset, which contains the memory controllers for the system. These links have 10.6 GB/s of bandwidth.

3.2 4-Core Opteron

Our second target machine is a dual core $\times$ dual socket AMD Opteron 2214 system. The cores operate at 2.2 GHz and can fetch, decode, and retire three x86 instructions per cycle. The memory hierarchy consists of private 64 KB L1 data caches, and private 1 MB L2 Victim caches. There is no cache-sharing between cores, but all caches are coherent. Inter-socket communication is provided over a single bidirectional 4 GB/s HyperTransport link. Memory controllers are integrated onto the same die as the cores. Although there is less bandwidth available for inter-socket communication, the lack of an intermediary chipset and the utilization of the HyperTransport protocol may provide a lower latency communication interface.

3.3 8-core (128 thread) Niagara 2

The Niagara 2, or UltraSparc T2 processor, represents a significant divergence in design philosophy from the previous two architectures. Rather than focusing on single-thread sequential performance, the T2 was designed as a System-on-chip solution for high-throughput server computing. The T2 chip integrates not only eight multithreaded processor cores and 4 Megabytes of shared L2 cache, but also two 10 Gigabit Ethernet controllers, a PCI-E controller, and four FB-DIMM Memory controllers. The cores themselves are dual-pipeline, in-order, and implement the Sparc V9 instruction set. Operating at 1.4 GHz, a T2 core supports 8 fine-grained hardware threads. These threads are divided into thread groups of 4 threads, and each thread group can issue to one of the pipelines each cycle. Each individual thread executes slowly relative to a thread on a superscalar processor, but the aggregate instruction throughput of all 16 pipelines (8 cores) can potentially be much higher.

Note that on none of the multi-core processors does StreamIt give us an interface to specify processor affinities. Nor is the StreamIt compiler aware of the cache-coherence topology of the system. Thus we expect that the organization of communication over cache coherence traffic will be far from optimal on the systems with non-uniform inter-core communication latencies and bandwidths. On the single-socket Niagara system, however, we should not expect such issues to arise, since there is uniform latency from each L1 cache to each other L1 cache, and from each core to the shared L2 cores to

4 Music Information Retrieval

Music information retrieval (MIR) is a field that combines signal processing, machine learning, and psychoacoustics to provide music listeners with content-based information on collections of digital audio. Typical applications include automatic playlist generation, content recommendation, music notation transcriptions, and content-based search indexing and organization. The front end of any MIR task usually

involves some method of feature extraction that attempts to collect meaningful low-level data about a passage of music to be used in subsequent processing steps. Usually this entails the extraction of short-time spectral features which are computed on a frequency and amplitude scale more meaningful to the human auditory system.

An example of this type of feature, well-known in the speech recognition community, are Mel-Frequency Cepstral Coefficients (MFCCs) [10][6]. The extraction of these features begins by summing short-time spectral energy (computed using an FFT) into frequency bands distributed on a meaningful perceptual scale. These band energies are then amplitude-compressed using a decibel log scale and then dimensionality-reduced using the discrete cosine transform. The problem with this method is that in order to achieve the necessary spectral resolution at lower frequencies, the duration of the short-time analysis frame needs to be around 23ms worth of audio samples long, which doesn't give fine enough time resolution for certain types of musical rhythm analysis.

One way to combat this time/frequency resolution trade-off is to perform a frequency warping on the audio signal prior to applying the FFT. An appropriately-designed warping will place each band of interest within the bandwidth of a single FFT coefficient, thereby allowing the time domain analysis window to be reduced in size to twice the number of desired bands. This results in a reduction of required time window from 23ms to as low as about 1ms.

The following section explains our implementation of this type of warped-frequency spectral feature extraction.

5 Algorithms Implemented

The process of warping the frequency axis is accomplished using a tapped and downsampled cascade of all-pass filters [2]. The intuition behind this scheme is illustrated in Figure 3, in which the unit delay elements of an FFT filterbank are replaced with frequency-selective delay elements in the form of all-pass filters, $A(z)$. It has been shown that a perceptually meaningful warping can be approximated using simple first order all-pass filters of the form:

$$A(z) = \frac{z^{-1} - \lambda}{1 - \lambda z^{-1}} \quad (1)$$

At 44kHz sampling frequency, the resulting warping optimally fits the bark perceptual frequency scale when the parameter $\lambda \approx 0.7565$ [7].

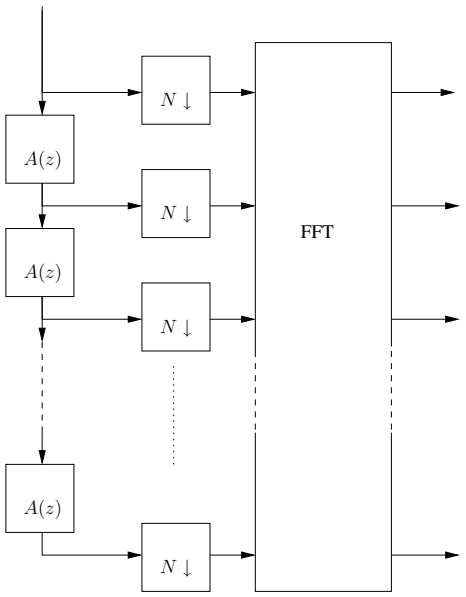

Figure 3: FFT filterbank using delay elements replaced with all-pass frequency-selective delays ($A(z)$). The boxes containing "$N \downarrow$" are Nth-order downsamplers.

The primary concern when computing this warped spectral transform is the complexity involved in the all-pass chain. Each all-pass filter is an IIR filter that must retain state information; therefore, the savings of fast FFT filtering cannot be employed here. The main bottleneck becomes the all-pass chain, and due to its streaming cascade structure, it is a prime candidate for description, optimization, and parallelization using StreamIt.

We implemented two versions of the spectral feature extraction process in StreamIt. First, motivated by way in which StreamIt seems to handle the partitioning of coarse-grained program structure, we produced a coarse-grained all-pass chain structure, which can be viewed in Figure 4. The all-pass structure involves only a single pipeline that contains all-pass filter elements which act on some of the data while passing other data untouched. Large-scale downsampling is carried out at the end of the all-pass chain. Computing the all-pass chain in this manner adds a considerable amount of integer operations involved in the passing of the unused data at each step; however, it was our hope that structuring computation in this way would lead to an easy partitioning for the StreamIt compiler.

For the second implementation, we used a fine-grained structuring that is almost exactly translates the block diagram in Figure 3. The structure, which can be viewed in Figure 5, breaks up the filtering and downsampling into separate StreamIt "filters",

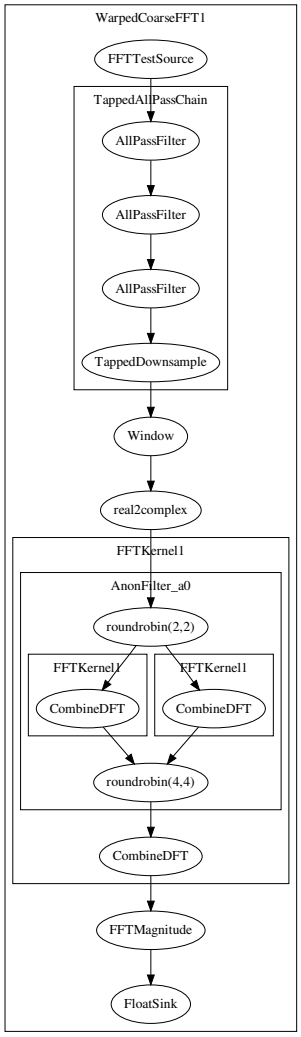

Figure 4: Coarse-grained all-pass chain for a frame size of 4

so that only the minimum amount of operations are performed for each part of the chain. We anticipated this implementation would perform much better on a uniprocessor, but felt that because of its recursive structure, it would not parallelize well.

Figure 5 shows the stream graph of the complete fine-grained program, including the FFT, frame windowing, and complex-valued operations. A uselessly small FFT size (4) is used for simplicity. Typical applications may employ anywhere from 16 to 64 bands, which would require FFT sizes between 32 and 128. For frame sizes such as these, the stream graph would be significantly larger.

6 Performance Data Obtained

We implemented all of the aforementioned codes in the StreamIt language, in order to evaluate their performance on our platforms. Performance data for the Coarse-grained All-pass filter bank were obtained on our Clovertown and Opteron Systems. We directed the StreamIt compiler to generate five different versions of the code, for 1, 2, 4, 6, and 8 threads, at optimization level -O1. Higher optimization levels did not finish compilation, as the StreamIt compiler exhausted all of the machine's 2 GB of DRAM. We then compiled the code with the GNU g++ compiler at optimization level -O3. Table 1 and table 2 display the execution times of the code on the two x86 target machines. All times in these tables are are in seconds. The first column shows the number of threads used. The second column shows the wall-clock execution time of the code. To demonstrate the effect of thread-creation overheads, the third column shows the amount of time spent inside system calls. To demonstrate the parallelization efficiency, the fourth column shows the sum of the execution times of all threads. The fifth column shows the speedup over the single-threaded StreamIt code.

The StreamIt compiler generates a "work estimate", containing an estimate of the amount of work each thread must do in each iteration of a steady-state streaming computation. The last column of tables 1 and 2 shows the expected speedup from parallelization, based on the length of the critical path (i.e. the thread with the longest work estimate), over the single-threaded StreamIt code.

Clearly, the realized performance falls short of the expected performance. Since, as we will shortly see, the single-threaded performance of the code is very acceptable relative to our other implementations, we expect that the lackluster performance is due primarily to the communication-topology agnosticism

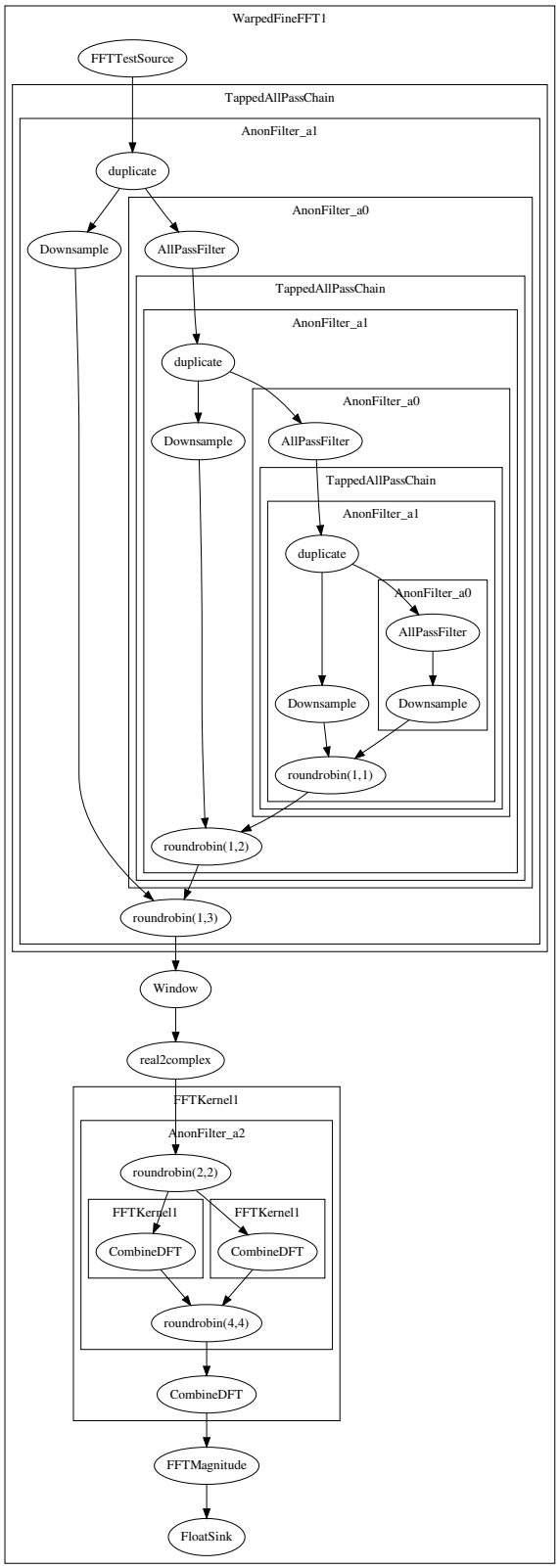

Figure 5: Fine-grained all-pass chain for a frame size of 4

of the StreamIt compiler. The fact that most positive speedups reported from StreamIt are on the MIT Raw architecture (with a very regular, high-performance, and compiler target-able communication fabric) supports this hypothesis.

Note that we were unable to execute the code on the most interesting of our multicore targets, the Niagara processor. Section 7 details our difficulties in running those tests.

6.1 Core 2 Duo T9300 (Penryn)

Because of the difficulty encountered in porting the StreamIt compiler to our target architectures, we decided to test our code on a dual-core machine to which we had root access. This greatly simplified the compiler build due to its need for a particular antiquated version of java and other non-standard packages. Naturally, as graduate students, we were limited to one of our own personal laptops to meet this requirement.

The Intel Core 2 Duo T9300 (codename Penryn) has two cores each clocked at 2.5GHz which share a 6MB level 2 cache. The level 1 data and instruction caches are each 32kB in size.

Figure 6 shows the timing results obtained for the coarse-structured implementation run on one and two cores and compiled with StreamIt optimizations on and off (-O1 and -O0). Speedup is observed for the three largest frame sizes using the -O1 optimizations; however, with optimizations off any speedup is negligible and overall it actually performs better.

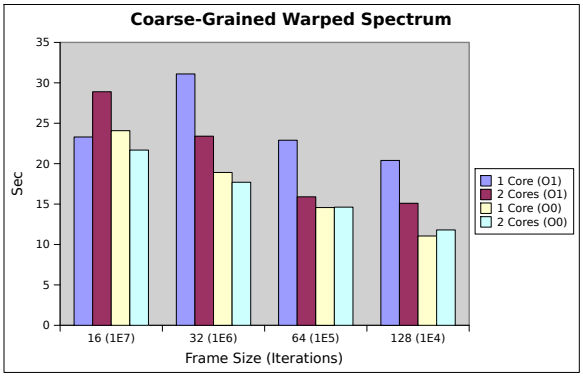

Figure 6: Coarse-Grained implementation performance on 1 and 2 cores at two optimization levels.

This dismal parallelization and optimization can most likely be attributed to two elements. First, the StreamIt compiler does not seem to be optimized at all for the basic consumer-grade x86 dual core architecture. Second, one of the three primary StreamIt compiler optimizations is to fuse or coarsen adjacent

filters when they do not contain state information. Since the all-pass filters in this implementation are IIR filters, they are "stateful" and may be hurt by this optimization or not allowed to use it.

The results for the fine-structure implementation are even more disheartening for those hoping to get free parallel performance out of the StreamIt compiler. For the two frame sizes that compiled (see Figure 7), a significant slowdown occurred when moving to two cores. However, this implementation benefited greatly from use of the -O1 optimization flag.

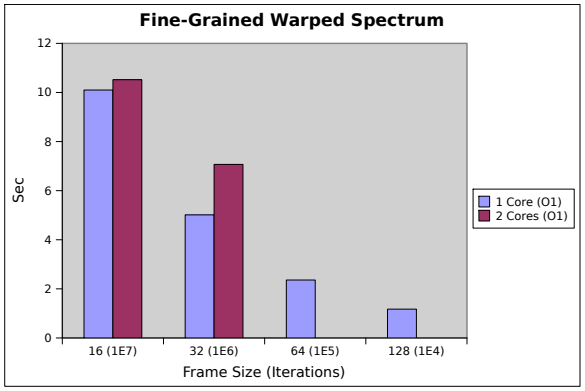

Figure 7: Fine-Grained implementation performance on 1 and 2 cores at two optimization levels.

Comparing the best runs of each of the two methods with a thoughtfully optimized Matlab implementation (Figure 8), we see that the fine-structured StreamIt implementation (compiled with the -O1 flag) performed significantly better using only one core than the other two did using two cores. This result is encouraging for those attracted by the simple StreamIt style of programming applications who would like to reap performance gains on a uniprocessor architecture without having to worry about all the details of low-level C programming.

7 Performance Data Missing

To our despair, the process of porting the StreamIt infrastructure to our set of target architectures was more difficult than anticipated. We had hoped that the similarities of the UNIX environment and cache-coherent architectures would be sufficient to port the runtime and StreamIt compiler generated C++ code. Here, we document our experience to the contrary.

Our strategy for compiling StreamIt code for multiple platforms attempts to leverage the fact that the StreamIt compiler's output is C++ code. That is, the input StreamIt program is parsed and optimized based on its pipe-and-filter streaming struc-

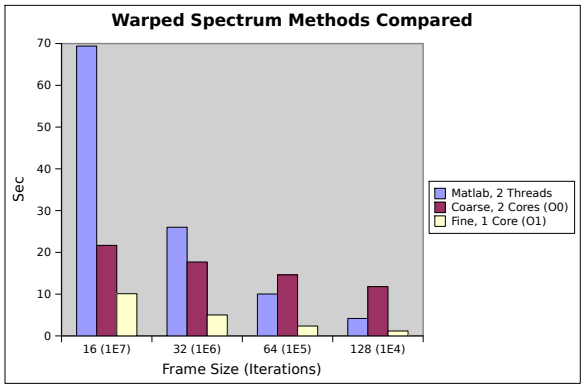

Figure 8: Performance using Matlab, coarse-grained on two cores (-O0), and fine-grained on one core (-O1)

ture. The output of the StreamIt compiler is a C++ program utilizing a hand-written C++ runtime library. Threads are created via the PThreads POSIX threading library, and assigned statically by the StreamIt compiler with a subset of the computation of the program. Communication takes the form of inter-thread shared circular queues.

Ideally, we would be able to compile the C++ runtime and generated code with a platform-specific compiler and run the binaries. However, even after successfully generating binaries in this manner, the code did not execute correctly. The runtime library is over 5,000 lines of complex multi-threaded C++ code, and the generated C++ programs are usually over 26,000 lines long. The StreamIt system does not provide any high-level debugger, and tracing high-level events through the morass of automatically generated C++ code is nearly impossible.

We were able to produce binaries for the Fine-grained version of the Warped Delay filter bank for all targets. Likewise, we were able to produce binaries of all codes for the Niagara processor. We repeatedly encountered two bugs. First, code run on Niagara would always terminate after approximately 1.07 seconds, regardless of how long the code should have taken. Since the execution of the C++ code quickly disappears into virtual functions and multiple threads, even this relatively simple-sounding bug was impossible to trace. Second, some codes exit almost immediately, providing a rather cryptic error message from the runtime relating somehow to a broken communication link between threads. The runtime code that generates this message is uncommented, and we were unable to track down its source. Our best theory is that the runtime code contains inter-thread synchronization bugs; these frequently are only correctable by re-writing significant portions of the code.

In short, the StreamIt code is "research-grade" at best. Most of the performance results provided in the StreamIt literature are from runs on the MIT Raw architecture, which bears little similarity whatsoever to our target platforms. Moreover, very few, if any, substantial codes have been written for any target by programmers outside of the StreamIt research group. Consequently the code base released to the public is in a rather immature state. For example, one early bug we found was caused by the runtime simply not checking whether the return value of a C Standard IO `fopen()` call was valid. During our work, we encountered several such "simple" problems that inspire little confidence in the quality of the StreamIt code base.

8 Conclusion

Despite our inability to obtain a full set of results, we can still draw some insight into the applicability of StreamIt for MIR applications. First, we believe that a high-level programming interface is necessary for productivity. Second, it is crucial for performance that as much of the overhead associated with a high-level language be pushed to compile-time, rather than run-time. Finally, we believe that StreamIt's compiler is missing some crucial optimizations for multi-core performance.

It is difficult to dispute the claim that high-level programming interfaces increase productivity. If a programming environment is well-suited to the target application, then it is unlikely that the programmer effort necessary to adopt the new interface will be prohibitive. In the case of StreamIt for MIR applications, the StreamIt language allows an efficient way of expressing the parallelism inherent in our applications. By providing a runtime environment that implements the pipe-and-filter streaming communication semantics, StreamIt allowed us to focus entirely on the implementation of the algorithms.

StreamIt prevents most of the overheads associated with its high-level language from being run-time bottlenecks. Compared with our equivalent Matlab implementation, our uniprocessor StreamIt implementation is much more efficient. We believe this is due to StreamIt's substantial and thorough lowering into C++ code. The StreamIt language is not entirely dissimilar from C++, this transformation is not prohibitively difficult. Matlab, on the other hand, is a dynamic and interpreted language whose efficiency comes primarily from highly-optimized implementations of intrinsic functions.

However, StreamIt was still unable to provide satisfactory scaling behavior on our multicore platforms. We believe that this is primarily due to the StreamIt compiler's agnosticism of the cache-coherence topology of our target processors. Since previous StreamIt literature has been able to demonstrate satisfactory performance on other multi-core processors, we do not believe that StreamIt's approach is fundamentally flawed. Rather, for StreamIt to be useful on the cache-coherent multi-core processors that are widely available, the compiler needs to incorporate the relative communication costs between different cores into its optimizations. The generated code should then utilize processor affinity interfaces provided by most modern operating systems to ensure that its performance estimates are valid.

References

[1] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.

[2] C. Braccini and A. Oppenheim. Unequal bandwidth spectral analysis using digital frequency warping. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 22(4):236–244, 1974.

[3] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, 2006.

[4] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006.

[5] C. NVIDIA. Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*, 2007.

[6] L. Rabiner and B. Juang. *Fundamentals of speech recognition*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993.

[7] J. Smith III and J. Abel. Bark and ERB bilinear transforms. *Speech and Audio Processing, IEEE Transactions on*, 7(6):697–708, 1999.

[8] Streamit language specification version 2.1. `http://www.cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf`, September 2006.

[9] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *International Conference on Compiler Construction*, 4, 2002.

[10] F. Zheng, G. Zhang, and Z. Song. Comparison of Different Implementations of MFCC. *Journal of Computer Science & Technology*, 16(6):582–589, 2001.

Threads	Wall Time	System Time	Sum time	Speedup	Expected
1	19.32	0.00	18.30	1.00	1.00
2	14.08	0.27	19.37	1.37	1.93
4	11.80	0.36	21.07	1.63	3.95
6	13.19	0.72	24.34	1.46	5.53
8	12.39	1.01	26.23	1.56	7.307

Table 1: Coarse-grained filter bank scaling on Clovertown. All times shown are in seconds.

Threads	Wall Time	System Time	Sum time	Speedup	Expected
1	33.82	0.01	31.59	1.00	1.00
2	22.59	0.26	27.33	1.50	1.93
4	21.19	0.83	27.23	1.60	3.95

Table 2: Coarse-grained All-pass filter bank on Opteron