# Stream Caching:
# Optimizing Data Flow within Commodity Visualization Clusters

Nathaniel Duca
Johns Hopkins University*
duca@jhu.edu

Peter D. Kirchner
James T. Klosowski
IBM T. J. Watson Research Center
{pdk,jklosow}@us.ibm.com

## Abstract

Our work with Chromium on the IBM Deep View system focuses on visualizaiton systems that run commodity software on distributed systems built with commodity graphics hardware. This paper describes our work to optimize the bandwidth requirements of such a graphics pipeline. We introduce a novel framework called Stream Caching for the caching of frequently-repeating streams, and use it to optimize the output of *unmodified* OpenGL applications in real-time.

Our modifications to Chromium are useful for a number of commodity visualization problems. The Stream Caching framework can be used for remote-visualization, improvement of graphics pipeline primitive throughput, or even to allow unmodified OpenGL applications to support computationally costly features. We suspect that this technology can be used to greatly improve the value of commodity visualization.

## 1 Introduction

The Chromium project [6] and supporting work [5, 4] provide a powerful streaming framework for different types of commodity visualization. Chromium allows the graphics pipeline to be constructed as a multi-host DAG of Stream Processing Units (SPUs). This allows modification of the output of standard, unmodified OpenGL applications. Our team has shown the utility of this framework for commodity parallel visualization with the Deep View system [7].

We have observed that OpenGL applications tend to be wasteful with their driver bandwidth, specifying the same geometry across many frames without using display lists. This is frustrating within Chromium because its pipeline bandwidth is heavily limited by both software and network throughputs. The ideal solution to this bandwidth problem is to produce an "application optimization" plug-in for Chromium that locates and caches frequently used command sequences, thus preventing redundant data flow throughout the pipeline.

We have not seen this approach described in the literature. This is not to say that the idea is entirely new. A number of different fields have designed application specific parallels to this: text compression, multimedia, and other fields uses similar application-specific approaches. While a graphics-specific solution is equally possible, we have discovered that a General Purpose Stream Caching framework can be created that is applicable to areas beyond visualization. We devote the first part of our paper to discussion of this novel technology.

A Chromium implementation containing General Purpose Stream Caches makes the code for optimization SPUs trivial, as shown in Figure 1. This code is so effective that we use it to produce the results in this paper. To this end, section 3 describes the integration of General Purpose Stream Caches into Chromium, while section 4 shows some of our preliminary performance results for the work. We close the paper in section 5 with a discussion of commodity uses of a Stream Cached Chromium.

```
void autocache_Begin(GLenum type) {
   child.CacheBegin(ANONYMOUS);
   child.Begin(type);
}

void autocache_End() {
   child.End();
   child.CacheEnd();
}
```

Figure 1: The code for a simple application-accelerator SPU within a Stream Caching framework.

## 2 General Purpose Stream Caches

Stream caching is based on producer-consumer locality. If the stream producer repeats the same record across its communication channel to the consumer, why waste bandwidth in retransmission and re-computation? This motivates the idea of a stream cache, which amounts to caching based on locality-of-generation: on the producer, locate frequently repeating substreams and place them in a cache on the consumer. Then, collapse subsequent uses of the cached substream into small meta-refrences. Because stream caching is done within the Stream Processing paradigm, multiple *Stream Caches* can be chained together to produce various types of bandwidth optimizations, parallel or otherwise.

A *stream cache* is a state machine that acts as a switch between the cache and an SPU chain. The key design point of a stream cache is that it is controlled by operations embedded in the same stream that is being cached. Three operations are used to control the cache: `CacheBegin`, `CacheEnd`, and `CacheExec`. The CacheBegin command diverts the stream into a comparison routine that tries to generate a cache hit or miss for subsequent incoming bytes. When an uncached substream is received, the entire substream is forwarded and added to the cache. When a cached substream is received, it is replaced with a single CacheExec command. Caches can be configured to respond to CacheExec commands differently, depending on their location in the pipeline. One setting simply forwards the Exec commands through the SPU chain; however, the more useful setting is an expand mode,

*While on Internship at IBM Research

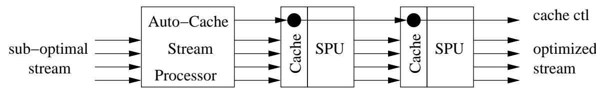


Figure 2: Cache control commands are embedded into a stream using a standard SPU. These commands can then control an arbitrary chain or DAG of SPUs.

where a CacheExec command is replaced with the corresponding cached substream. A stream cache is a relatively simple structure; the key point is that it is a byte-level mechanism that caches substreams based on special control commands embedded in that stream.

Use of a Stream Cache is quite simple. A stream producer needs only to embed the proper Begin-End cache commands into the stream; the byte cache will handle the more involved processes of repeat identification, compression, and re-expansion. Applications can do this directly, or, more interestingly, a Stream Processing Unit can perform the process automatically, as shown in figure 2.

What makes General Purpose Stream Caching useful is its flexibility. It allows a producer uncertain about when and how repetitions in a stream occur to still perform caching operations. One does not have this ability using previous techniques. For example, display lists, where the producer must be certain of when and where a repetition will occur and be able to assign a unique identifier to that repetition. While knowing this as an application programmer is reasonably easy, it is nearly impossible at the streaming level. That stream caching can do this distinguishes it from previous techniques.

# 3 Implementation

Use of General Purpose Stream Caching in Chromium involves some significant modifications to Chromium code. However, they are non-disruptive: Chromium runs in exactly the same fashion with Stream Caching as without. The changes required are twofold. First, we produce an AutoCacheSPU, which embeds Cache Control commands into an OpenGL command stream and serves as the actual "application acceleration SPU". Second, we modify the Chromium framework to contain General Purpose Stream Caches.

## 3.1 AutoCache SPU

The job of an AutoCacheSPU (figure 1) is to locate areas of likely repeat in an application command stream. By the construction of the Stream Caching framework, the AutoCacheSPU does not have to always guess correctly: misses are naturally handled by the system. OpenGL is extremely helpful in identifying areas of potential repeat with an an application stream. The `glBegin`/`glEnd` block are perfect cues that can be used to place CacheBegin-End markers. While we have settled on this approach for the time being, some important dynamic optimization work must be done to prevent small glBegin-End blocks (e.g. single triangle blocks) from becoming caching candidates. At the same time, some very interesting work can be done to implement display lists with caching calls: doing so would greatly simplify the process of handling display lists within Chromium.

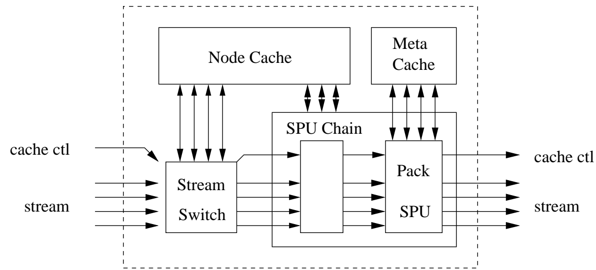


Figure 3: Detail of the modifications made to the Chromium server. Every Chromium server contains a full cache, plus a modified PackSPU that allows caches to work across hosts.

## 3.2 Stream Caching Chromium

We use mostly standard caching techniques to implement a stream cache within Chromium. The stream cache is a map that is keyed by an automatically generated unique identifier and an optional application specified name. Every stream is indexed into the cache using a checksum (CRC-32) on its contents. Then, an optional custom ID is provided with each CacheBegin command that is stored within a secondary map. This allows operation analogous to OpenGL display lists: construction with a specific ID and subsequent execution with that stored ID.

An important consideration in stream cache design is memory consumption. A stream cache is intended to store large substreams, so to a point, a certain amount of memory is necessary. However, in many places along the pipeline, all that is required for cache processing is knowledge of downstream cache contents, not the actual cached substreams. We support this idea of a meta-cache, which allows lightweight yet fully-functional caches to exist through the entire pipeline. To prevent unbounded memory growth in regular caches, we also implement a LRU substream replacement scheme. This is clearly not the most efficient choice; for example, a least frequently used scheme can be implemented in the same framework to give better results on certain programs.

Our modifications to Chromium place these Stream Caches at various places within the Stream Processing graph. Our basic rule is to place a cache at every point in the pipeline where buffering already takes place. As shown in figure 3, we place a full cache at the entry to every non-application SPU chain (CRServer) and a meta-cache at the end of every Non-Rendering SPU chain (PackSPU, TilesortSPU, etc.). Our approach tries to minimize caching overhead by amortizing it across the operations required for networked rendering.

We have implemented two autonomic behaviors into the caching system that handle the two extremes of caching. For a scenario with a low hit ratio, the entire caching system can be switched off by bypassing the AutoCacheSPU dispatch table. Conversely, applications with high hit ratios will spend a great deal of time pointlessly specifying geometry that has already been hit in the cache. For this scenario, we have implemented a Driver-Bypass option, that diverts the application stream into a No-Op SPU until it reaches the CacheEnd function, allowing the application to quickly move on to more pertinent code.

The net effects of these modifications are threefold: first, entire chunks of OpenGL commands can be converted at any point in the stream into meta-execute commands. These commands also propagate through the SPU chain, which allows SPUs to operate directly on Cache records (enabling

the advanced features of Section 5). Toward the end of the SPU chain, meta-executed substreams are re-expanded to the cached substream and passed to the renderer. The true benefit of this architecture is that, when a substream is cached, it still passes through the SPU chain with streaming semantics, and thus can be operated on in the standard fashion.

# 4 Results

We have had encouraging results with the first implementation of a Chromium caching architecture. For large streams, the overhead of stream caching is easily justified. At small stream sizes, miscellaneous overheads severely hurt performance. The decision to use Stream Caching comes down to an analysis of application style (therefore, how well AutoCacheSPU works), incurred caching overheads shown in figure 4, and the bandwidth/computation savings that result. This section explores our findings in these areas.

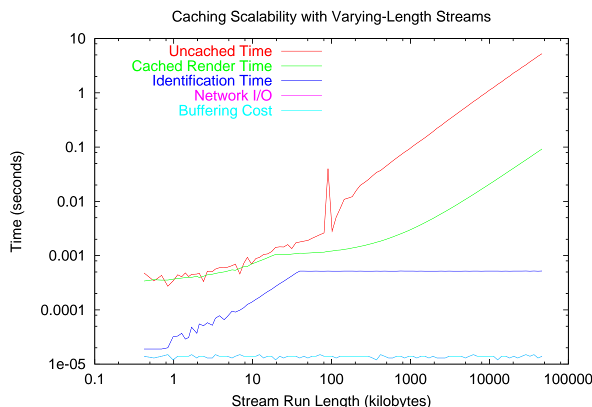


Figure 4: The overhead of stream caching is directly related to stream run length because of buffering costs, rather than identification costs. Note the streaming performance (topmost) against cached performance (second) to see the time gains achieved using caching.

With respect to overheads, our present implementation prevents identification costs from growing too rapidly by checksumming only a small portion of the entire buffered stream. However, at low stream sizes, identification can still be quite costly, as figure 4 shows. More complex but significantly more efficient solutions exist to this problem that we have not yet pursued (e.g. a hierarchy of checksums).

Another bottleneck in our system is in stream buffering. We have taken care to implement it as efficiently as possible: as mentioned earlier, we can achieve cache hits before the entire stream has been buffered, which allows us to discard the remainder of the stream without buffering (by remapping the head SPU pointers to No-Op equivalents, thus avoiding conditionals). Unfortunately, the big cost in this scenario is driver entry. Future optimizations may still be possible.

At the moment, caching runs on single-context applications, which severely limits our test population. However, we show here a few simple to understand their respective stream-caching characteristics. Figures 5-7 show the bandwidth requirements in cached and uncached scenarios for Newave, a wave propagation simulation, Atlantis, an example of a frame-similar animation, and Quake 3, a heavily optimized application. These bandwidth measurements can be used to infer cache hit and miss ratios.

Several broad observations are possible from these data. Applications with mixed dynamic simulation and static viewing modes, modeled by Newave (figure 5), work extremely well with caching, achieving massive compression ratios of up to 53:1. This is well matched to typical Sci-Viz workloads. Logically, however, the more dynamic an application becomes, the poorer caching performance it achieves: there are points where this may be appropriate, as shown in figure 6 by Atlantis, where hits across frames are still possible. However, aggressively optimized and face-culled applications such as Quake 3 (figure 7), do not gain much from caching. As the next section discusses, performance on applications is partly an incurable function of the application's style, but more importantly, a function of AutoCacheSPU design.

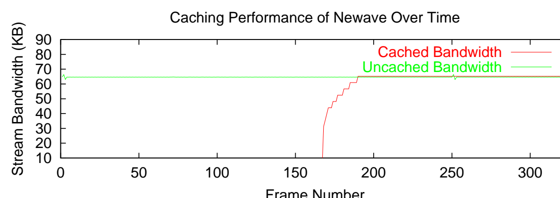


Figure 5: CAD and scientific visualization applications benefit the most from caching. Here, the GLUT wave-propagation simulation (Newave) toggles from view to simulate mode to demonstrate cached vs. uncached bandwidth requirements.

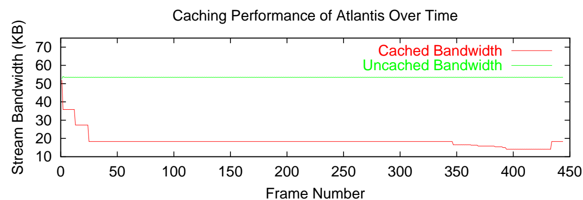


Figure 6: Applications showing progressive update, such as Atlantis, can still be optimized by caching (3:1 in this case), but not to the extent of the previous example.

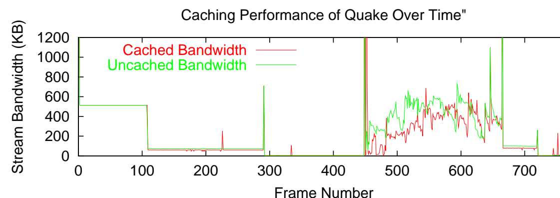


Figure 7: There are some cases where caching simply cannot succeed: Quake 3 is such an example. The best thing do to in this scenario is shut off caching.

As mentioned in the charts, Stream Caching is able to get compression improvements that depend on the character of the application. In principle, any repeated substream can be compressed to 12 bytes, giving a variable compression ratio of $n/12$ for every repeat of an $n$ byte cached substream. Compare this to results on X11-graphics compression, which were

able to achieve a 1.6:1 compression ratio [1]. While network latency remains the principal barrier to remote visualization, our implementation should, out of the box, be extremely useful for the purpose of remote visualization.

# 5 Future Work

The use of General Purpose Stream Caching within Chromium is a work in progress. There are a number of different potential areas of pursuit. Framework improvements must be made, and more clever optimization SPUs can be built. Finally, advanced pipeline behaviors can be constructed by SPU-level inspection and modification of cached substreams.

## 5.1 Framework

The earlier discussion about caching overhead speaks volumes about necessary future work on the caching framework. What we have now is a proof-of-concept with unacceptably large identification and buffering costs. The primary area of work in the framework area will be to implement efficient techniques for performing stream caching. We believe that there is plenty of room for improvement in this area.

Simultaneously, however, much work can be done to improve the "AutoCacheSPU" described earlier, which should lead to leaps in cached application performance. This unit might be best built with dynamic optimization algorithms: perhaps some techniques from the real field can be applied to this problem. In the short term, the performance of AutoCaching can be greatly improved with a component that prevents single-triangle cache blocks. The AutoCache approach talked about here produces large batches of such blocks, which hurts performance because the caching system is optimized for much larger substream sizes. Solutions to this problem must be pursued to realize proper performance gains. At the same time, multi-context support needs to be fixed within the autocaching framework. This change is simply a matter of time and effort, rather than feasibility.

## 5.2 Advanced Stream Processing

One limitation of stream caching is its restriction to low-level operations. Stream caching hybridizes the stream processing metaphor by automatically constructing a lightweight scene graph. Our implementation exposes this data structure to stream processing units, allowing them to implement complex processing operations that can be used to produce more complex output than originally intended by the application.

We can think of a number of immediate applications of this notion. Stylistic rendering [10] depends heavily on bulk-understanding of a frame, which is provided with caching. Remote visualization [11] tasks are further improved in the area of geometry compression [8, 2], which is otherwise considered an off-line process. A similar potential use for this is for driver-supported level-of-detail, either in the form of vertex clustering [9] or continuous LoDs [3]. In general, the presence of cached blocks allows the benefit of time-consuming computations on a substream to be spread-out across the lifespan of the substream (in contrast to the lifespan of a vertex in standard Chromium). In another area, Stream Caching can be applied toward sort-first architectures, where we can use the persistence of substreams to perform tilesort operations directly among different tile servers. By taking advantage of the bisection bandwidth of the cluster interconnect, we should be able to greatly improve the triangle throughput and scalability of the tilesort system. In summary, the presence of persistent graphics chunks within the graphics stream opens up a number of very interesting opportunities that are traditionally restricted to scene-graph architectures.

# 6 Conclusion

While caching was originally intended as an optimization to curb excessive data motion through the Chromium pipeline on commodity graphics equipment, the data that Stream Caching provides is applicable to a wide variety of commodity visualization problems. In general, Stream Caching allows "intelligent streaming data flow" throughout a pipeline. In this paper, we have discussed two creations toward this end: first, General Purpose Stream Caching, and second, the techniques required to use Stream Caching for OpenGL application acceleration. A stream-caching Chromium is even more flexible than standard Chromium: it can be bandwidth and computationally efficient, and support a variety of powerful programming models traditionally unsupported by OpenGL. In general, we believe that the long viability of caching will come from its use for creating powerful, versatile additions to commodity visualization pipelines, of which optimization is only a part.

# References


[1] J. Danskin and P. Hanrahan. Compression performance of the xremote protocol. In *1994 Data Compression Conference*, 1994.

[2] M. Deering. Geometry compression. In *Proceedings of SIGGRAPH 1995*, pages 13–20, 1995.

[3] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH 1996*, pages 99–108, 1996.

[4] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed rendering for scalable displays. In *Proceedings of SuperComputing 2000*, page 30, 2000.

[5] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: a scalable graphics system for clusters. In *Proceedings of SIGGRAPH 2001*, pages 129–140, 2001.

[6] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A streamprocessing framework for interactive rendering on clusters. In *Proceedings of SIGGRAPH 2002*, pages 129–140, 2002.

[7] J. T. Klosowski, P. Kirchner, J. Valuyeva, G. Abram, C. Morris, R. Wolfe, and T. Jackman. Deep view: High-resolution reality. *IEEE Computer Graphics and Applications*, 22(3):12–15, May 2002.

[8] J. E. Lengyel. Compression of time-dependent geometry. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 89–95, 1999.

[9] K.-L. Low and T.-S. Tan. Model simplification using vertex-clustering. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, 1997.

[10] A. Mohr and M. Gleicher. HijackGL: Reconstructing from streams for stylized rendering. In *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, 2002.

[11] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 85–92, 2001.