Multimedia Streams and the CPU Cache

Mark Hayter[1] and Derek McAuley
University of Cambridge Computer Laboratory

9th March 1994

Abstract

This paper discusses the use of the CPU cache when the processor is being used to manipulate stream data, in particular isochronous media such as as video. The principal idea is to enable data to avoid main memory and flow directly into a portion of the cache. When used with a "cache aware" operating system this method is shown to provide a rich support environment for multimedia manipulation. Experiments to demonstrate this have been performed on a workstation based on the Desk Area Network architecture, and results are presented.

1 Introduction

Multimedia research in the last few years has concentrated on transporting isochronous data across the network and through the workstation while retaining the real-time characteristics to enable smooth display. However, the next range of multimedia applications will be those that do more than just present the data to a human user. They will require some processing to be done on the arriving streams in addition to, or instead of, simply passing the data to the output device.

One common theme that is seen in both multimedia end-system design and the ATM style of networking is a separation of data and control paths. This is frequently done to simplify and optimize the hardware and software involved in the parts of the system where large volumes of data flow. Often in these systems, the video information passes directly from the network interface to the display without intervention by the CPU; it is only relatively rare events with more relaxed time constraints, such as repositioning the window on the screen, that need processing power. This mechanism is highly effective at providing a support mechanism for multimedia applications without either the high bandwidth data streams taking all the CPU time, or execution of other tasks inserting unacceptable jitter in the servicing of the isochronous data.

However, the requirement to keep the CPU out of the data path has led to machines where the processor is never able to access the multimedia data. This approach is unfortunate in that it precludes an interesting range of applications. We believe that a better architecture would keep the data away from the CPU when no processing is required but still enable high speed access by the CPU when the application demands it.

This paper presents the advantages of providing the CPU access to multimedia data by delivery of the stream directly into the CPU cache. This avoids the data passing through main memory and also prevents unnecessary traversal of the machine's interconnect. We refer to a cache used in this way as a "stream cache" (or s-cache).

The desire behind the s-cache is not only to get multimedia data into the processing element fast, but also *get rid of it fast*; that is, the desire it to process data and then flush it as soon as possible, making use of the fact that the data is known to be a stream.

The stream cache mechanism fits well with the Desk Area Network (DAN) architecture [6], which uses an interconnect based on the Asynchronous Transfer Mode (ATM). A machine based on the DAN was used to gain experience with using a stream cache.

[1] Now at Digital Equipment Corporation, Systems Research Center.

2 The Stream Cache

The principle behind cacheing in computer systems is to keep information that is likely to be used in the near future in a store that can be rapidly accessed. This is true on many scales, from retaining information in the sense amplifiers of a DRAM, up to keeping copies of World Wide Web pages in the local server. In this paper we are concerned with the CPU cache, that is the place where the processor keeps data and instructions that it is probably likely to use soon. In many existing systems the cache is divided into sections, possibly of differing sizes, for instructions (i-cache) and data (d-cache). This division is made both because of the observed differences in the access patterns and to prevent interference between the two types. We propose a third portion of the cache, the stream cache (s-cache), for holding stream based data. Both processing of an incoming stream of data and producing output can benefit from this technique; these themes are developed in the next two sections.

A second method for dividing the cache is to force different areas of memory to be separately cached. This can be especially useful in a system where tasks have processing guarantees associated with them – in the same way as in a network switch guaranteed and best-effort traffic may be separated, the operating system may wish to allocate a region of the cache for exclusive use by a process with guarantees, and the rest to the best-effort processes. The operating system will certainly want to be able to dynamically alter this split based on the characteristics of the workload, to enable the most effective use of the cache memory. This desire for this form of dynamic allocation of cache memory applies equally to i-cache, d-cache and and s-cache. Furthermore, once the operating system starts becoming aware of the cache as a machine resource, the next step is to consider if the ratio of the service time of the cache to the processor performance is such as to make it attractive to invoke the processor scheduler when the resource is unavailable (i.e. on a cache miss).

2.1 Caching a Stream

Normally the basis of cacheing is locality – the idea that if some data is used then it is likely that the same or nearby data will also be accessed soon. Therefore when an area of memory is referenced it is retained in the cache and old data may be discarded (or flushed). When processing a stream it is likely that the data that will be accessed is amongst that which has most recently arrived. Hence, the s-cache should hold the most recent data from the stream and new data should cause old to be thrown out. With an s-cache architecture, data arriving from the stream is placed directly into the cache; it does not pass through main memory which avoids unnecessary buffering and a further traversal of the interconnect. A section of the cache effectively becomes a circular buffer holding the latest stream information; update of the cache is asynchronous rather than triggered by a CPU cache miss.

When the CPU tries to access stream data there are three possible outcomes:

- The data has recently arrived in the stream. In this case the item is found in the cache and processing can continue.
- The data has not arrived yet. This is treated as a cache miss and the CPU can be halted until the required item arrives. If the data will not arrive for some (long) period of time the operating system may choose to reschedule the CPU.
- The data is from longer into the past than the buffer can hold. This case should be flagged as an exception to the operating system.

The first two cases have the cache behaving in a fairly standard way, though being updated rather than invalidated by the network traffic. The third is more complicated. The data from the stream has been discarded from the cache and is now lost – there was no main memory behind the cache entry. This is not considered a problem, as in the case of a traditional data stream (for example coming from a fileserver) the process can generally perform some `rewind` operation, while in the case of a real-time stream, the need to access the lost data suggests that processing is getting so far behind the stream that re-evaluation of the processing task and then resynchronisation is required.

To allow an application to use a cached stream the operating system needs to present an abstraction of the low level behaviour. The stream data will need to be addressed in some way. Most streams will have some frame or temporal structure that the application will wish to use; this can be conveniently mapped onto a range of the process' virtual address space[2]. Thus the process may access the stream as an array indexed by frame number.

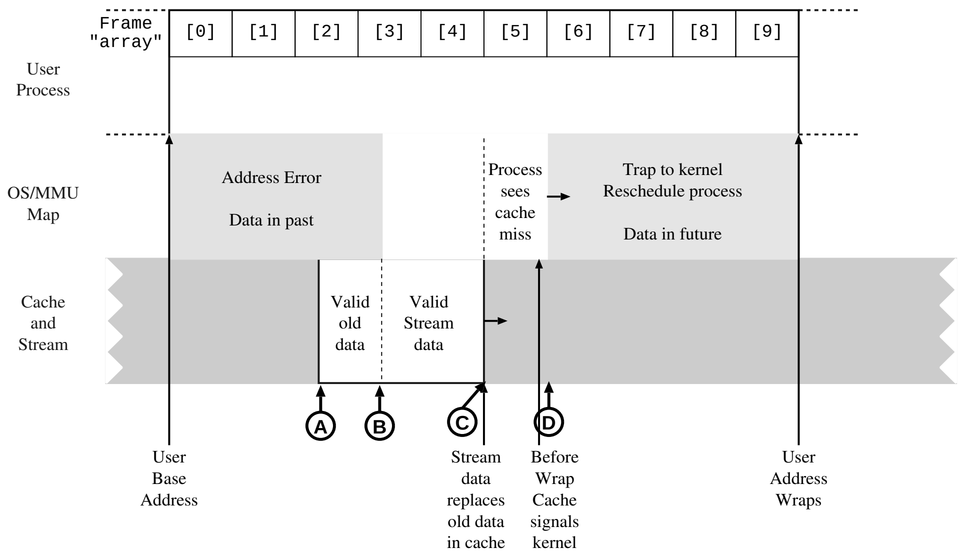

Figure 1: Addressing a Stream

Figure 1 illustrates how the address mapping may be done. User addresses will span some space, possibly having to wrap after a large time has passed. As shown, the operating system will set up the memory management unit (or a more specialised cache address management unit (CAMU)) to trap some of the range and allow other addresses to access the cache. Below point B, accesses are trapped as being too far into the past to be in the cache. This boundary will move in discrete steps (probably some multiple of the page size, though with a CAMU these may be made appropriate to the stream parameters), so that some valid data will be inaccessible (from point A to B) – this is the next data that will be overwritten. Addresses between points B and D are mapped through to the cache. The stream is arriving into this range, so for the lower addresses the processor sees a cache hit and retrieves the data rapidly. Above the stream arrival point (C) the processor is blocked by a cache miss until the data arrives. Past point D the OS considers the data will be long enough arriving that it is worth suspending the process and rescheduling the CPU.

To implement this mechanism the operating system will need to alter the virtual memory mappings to reflect the arrival of data on the stream. Therefore the cache controller needs to inform the OS after some amount of data has arrived. The block size after which the mappings are modified is one of the parameters that can be used to tune the system. If it is too big, a lot of valid data becomes concealed (i.e. the gap between A and B becomes large); if too small, the system will be constantly dealing with cache interrupts and moving the window. A specialised CAMU could automate this process and relieve the OS. Similarly the application will need to know when to

[2]Especially with a 64 bit address space.

wrap round its virtual address range; this may be signalled by an event from the operating system, or be explicitly written into the program.

The second main problem is managing data loss. This is not a problem for streams originating on the same machine, since the communication is reliable; however, the majority of traffic may experience some loss[3]. Loss of data that is *accessed* should be treated as an addressing exception, with the operating system and therefore user process being signalled. Loss of data that is not accessed need not be signalled, though event statistics may be useful. However, the hardware has to be able to recover and ensure addressing consistency – for example, when data loss is noticed the corresponding cache line could be marked as "missing", resulting in a memory error being raised if it is accessed. The details of noticing data loss depend on the transport mechanism used on the stream, in the worst case buffering of a complete block would be required – in the DAN demonstrator built, this would be done by some other node of the machine, such as the network interface.

There are two major advantages to the stream cache system. Firstly, the data from the stream is placed where it is going to be used, namely in the CPU cache. Hence, even if data is not accessed in strictly temporal order, recent information will still be found in the cache. Secondly, the hardware manages the fine grained resynchronisation with the stream, imposing no overhead other than that necessary to wait for the data to arrive. If the window size and advance step parameters are carefully chosen there should be a large performance improvement over traditional stream reception systems. This may be compared with the technique of dallying in an interrupt routine or prior to rescheduling on a semaphore block.

2.2 Output from the Cache

A similar argument to that in the previous section applies in the case of stream output. It is quite likely that output streams will be built up in a random access fashion, and are therefore best formed in the CPU cache. It is commonly believed that network protocol code behaves badly with write-through caches because of the random bit field accesses that tend to be done when building headers.

Output from the cache differs in two important respects from input. In the input case there was no necessity for arriving data to actually be used, but on output the data sent must come from somewhere. This could be achieved by using the data from the previous frame if new values have not been written. Applications which are producing frames by applying inter-frame differences would benefit, but merging of the old and new data could cause unwanted complexity. A second method would be to insist that cache lines for the whole block exist in the cache, written either by the CPU or an input cache stream. In either case the CPU must not be able to alter the cached value of data that has been sent on the stream, or it would be attempting to change the past.

The second choice is how and when the data gets sent. This is in many ways similar to the problem experienced in operating systems with have an integrated VM and IO system such as Multics and Mach. A simple system would be for the CPU to force a "flush" operation on every line, or set of lines, to be sent. This has the disadvantage that it may tie up the CPU for the entire transmit time. A rather more satisfying scheme would be for the data to be transmitted from the cache without processor intervention, in much the same way as incoming stream data arrives. Once a block has been marked to be sent the CPU would be free to continue work, though it should not be permitted to modify the data being sent. Transmission could be done automatically by the hardware to achieve the desired outgoing stream characteristics; or, rather more practically, by the processor issuing an explicit send on the data block.

[3] Although in practice the cell loss rate on an ATM network may even get near the 1 in 10^{12} specified, loss must be accounted for.

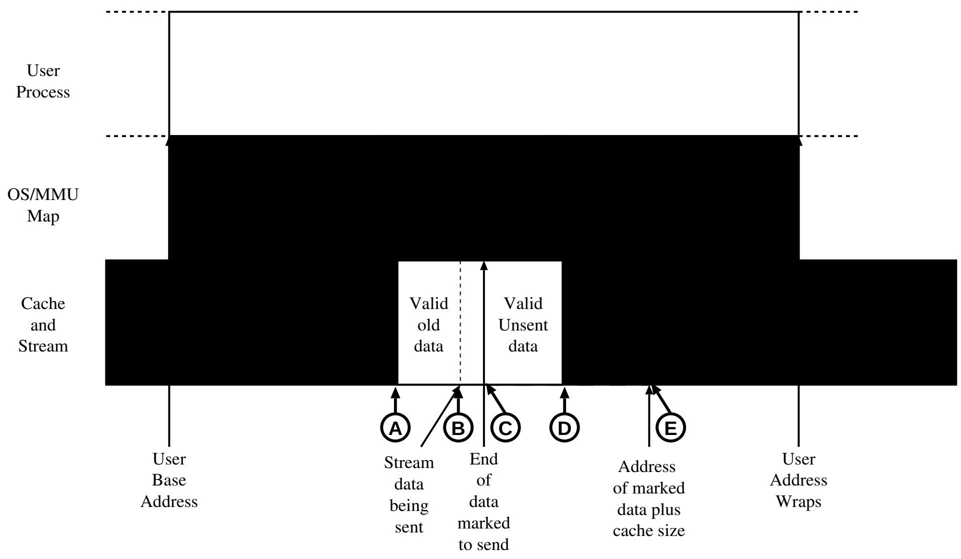

Figure 2: Addressing an Output Stream

Operating system support via the MMU is again required both to ensure that the CPU may read but not alter data that it has sent, and to provide the address range for overlaying the stream. The address map structure will be something like that shown in figure 2. The same sort of sliding virtual memory window could be used as for the incoming case. Again accesses too far into the past (before point A) are trapped. The read-only region (A–C), protects history and contains the data that is currently being sent (point B which moves to meet C). In addition the operating system will need to keep track of the furthest point (D) into the future touched by the user process. As there is only limited cache space, when new stream information is written to the cache, some data from the past must be lost. When a new largest address access is trapped the system must compare it with the current send point plus the cache buffer size (point E). Accesses before this point will cause the window to slide, discarding old data. However, if the window were to move past this point then data that has not been sent would be overwritten corrupting the outgoing stream. These accesses need to be trapped and signalled as an exception. The process will inform the operating system when data is to be sent, so the read-only region can be extended.

3 The Desk Area Network and the CPU node

The Desk Area Network (DAN) is an architecture for a multimedia workstation. Its main feature is the use of an ATM interconnect between the parts of the machine. The architecture was proposed in [6] and has been developed by a series of research projects [2, 5, 8]. The aim is to provide improved system performance by the use of a device and processor interconnect which:

- permits direct device–device as well as CPU–device communications,
- is amenable to implementation as a space division switch fabric,
- provides frequent preemption points,
- allows a simple and efficient high bandwidth connection to an ATM based network.

Similar work is in progress in the ViewStation project [1], and [9], where again ATM is used as the basis for a multimedia machine. The assumption that future high speed networks will be packet based [3] has led to an approach like the DAN but using packet switching internally [4].

Since it is intended for use within a machine the architecture specifies a reliability requirement – when a cell is sent across the DAN, either it reaches its destination or the source is informed that it did not. This allows great simplification of internal protocols. All devices in the system communicate using ATM cells, both for control and data transfers. Devices are designed to support streams flowing into, out of, or through them. The use of ATM within the machine as well as in the local and wide area networks greatly simplifies the network interface. The connection oriented nature of all the data flows is used both to provide routeing information from source (possibly over a LAN/WAN) to destination, and to provide context at the end-point; for example the virtual circuit carrying video stream may have routeing and quality of service information attached to it as it traverses the network, and window co-ordinates in the framestore.

The Desk Area Network project has developed a multimedia workstation as a demonstrator of the architecture. A typical configuration is shown in figure 3. The machine has video and audio capture and presentation nodes and a DSP node in addition to processors and memory. The interconnect of the machine is provided by the ATM switch fabric designed for the Fairisle project [7]. Further details of the demonstrator may be found in [2] and [5].

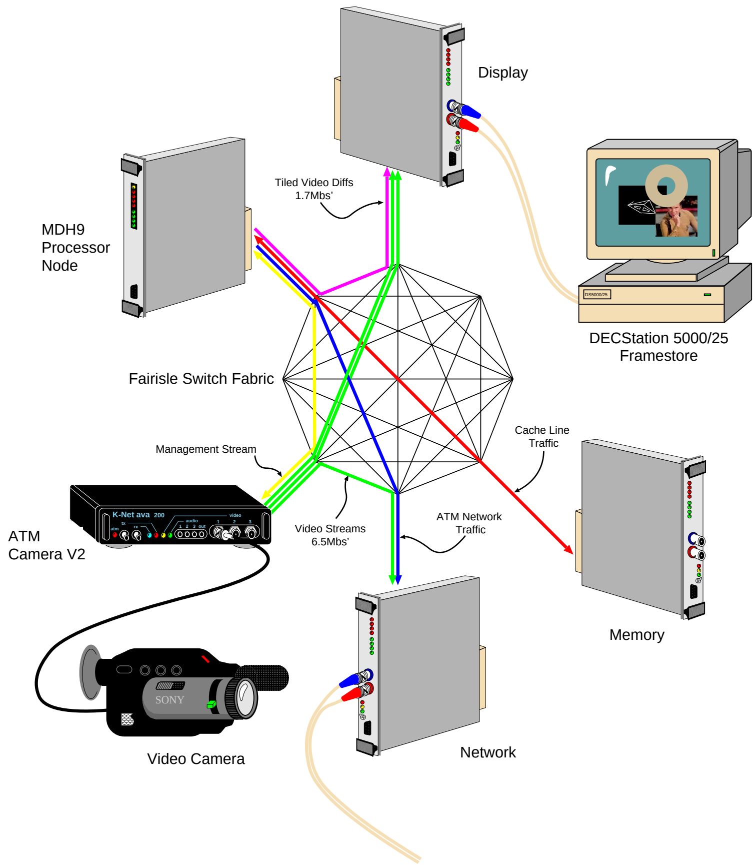

Figure 3: The experimental configuration

A CPU node was developed to allow investigation of the DAN as a processor to memory interconnect. It also permitted the stream cache ideas to be explored. The node consists of an ARM600 RISC microprocessor (which includes an internal MMU and 4K of primary cache) and 256k of secondary cache. The secondary cache controller provides the connection to the ATM fabric and in the event of a cache miss will exchange an ATM cell with one of the memory nodes. The cache is write-back, so the exchange may need to flush a dirty cache line as well as fetch the new one. The request cell therefore contains the requested address and optionally, the data to be written back and write address. The reply consists of the requested data and its address[4]. Using a cache line of 32 bytes allows room for the header and address information to be conveniently fitted into a single ATM cell.

The use of the DAN for both multimedia and cache traffic has been investigated using the demonstrator. The main problems found were in the implementation rather than the architecture, in particular the re-use of the Fairisle switch fabric (which was built as the core for a LAN ATM switch) provided a performance bottleneck. Cache service was shown to be possible in the minimum 2 cell times required for a request–reply exchange between the CPU node and memory. Interference between cache traffic and high bandwidth streams was minimal, and careful interconnect scheduling could further reduce this.

The cache controller on the DAN CPU node currently only supports division of the cache into fixed size regions. In fact both the region of the cache is fixed and the range of remote memory addresses (normally 1Mbyte). Use of the local MMU and a second mapping at the memory node prevents the fixed address range being a problem; however it would be preferable if the size of cache memory for each region were variable. The controller is connected to the DAN interconnect, and is geared to moving ATM cells between the cache and the interconnect. This is exactly what is required for an s-cache, and regions may be marked as an end-point for an incoming data stream. Physical constraints on the current version of the CPU node have prevented regions of the cache also being used as a stream source.

4 Experience with a Stream Cache

Adaptation of the DAN workstation to allow experimentation with cacheing streams and processing multimedia data is still in its initial stages. Both hardware and especially operating systems software enhancements need to be made to support the mechanism. These are being carried out as part of the "Operating System Support for the Desk Area Network" project (usually referred to as DANOS) which has just started. However, some preliminary results showing the use of an input stream cache are reported on in this section.

The existing DAN CPU node is capable of a limited form of input stream caching. The incoming stream must use a specialised ATM Adaptation Layer (AAL), and there is no mechanism for handling data loss. For an initial demonstration and exploration of the ideas, this presents no problem, since the stream can be generated in the correct format (or reformatted) by another node on the same DAN. In addition there is insufficient status information from the hardware to allow the operating system to become fully "cache aware", but there is sufficient for a limited form of the sliding VM window mechanism to be used.

For the stream cache experiments the CPU secondary cache was divided into two parts. The bottom 128Kbytes was used in the conventional manner as a 2 way set associative cache of the 1Mbyte of main memory of the machine. The upper 128Kbytes of the cache held the stream. All the experiments made use of a memory server that was implemented in software on one of the other nodes of the DAN. This takes 8 cell times (33.6 μs) for a cache line fetch, rather than 2 using the hardware server, but it permits information on miss address and time to be recorded.

A version of the Wanda microkernel was extended to provide minimal support for the stream cache. An extra Quality of Service (QoS) type "Cache Stream" was added to the ATM connection establishment protocol stack, to cause the circuit to be setup so that incoming data would flow into the cache hardware rather than being delivered through the kernel I/O buffers. In addition use was made of the MMU to provide the sliding VM window. Page table entries were setup for the region of address space used for the virtual array of incoming frames to map them onto the

[4] The format permits the server to return the cell it received only altering the data field.

stream cache hardware. All the addresses were initially marked invalid. Thus the sliding window starts with zero length. As stream data arrives it expands to 128kbytes to match the cache size before it starts to slide through the address space. A signal should be provided by the stream cache hardware to match movement of the window to data arrival, unfortunately limitations in the hardware prevented this from being used and software emulation was necessary.

A traffic generator was used as the source of the stream. This made simulated video frames of 64×64 pixels each of 32bits, making a whole frame 16kbytes. This frame was packed into cells using the special cache AAL. The rate of delivery of the data (both the inter-cell and inter-frame delay) could be varied.

4.1 Stream Cache Effects on Other Processes

As an example load on the CPU a PAL compiler was used. This consists of three distinct phases: reading and parsing the input equations; optimising them and forming the sum of products form; and writing the output file. To understand the memory behaviour of this application the cache was set to be small (256 lines, direct mapped) and the memory server set to record all addresses fetched. Figure 4 shows the trace for the operating system startup and the execution of the compiler. The three vertical lines indicate the changes between the phases of the example program. Using a dump of the kernel segment map the use of the various areas of memory was determined, and is shown on the right of the figure.

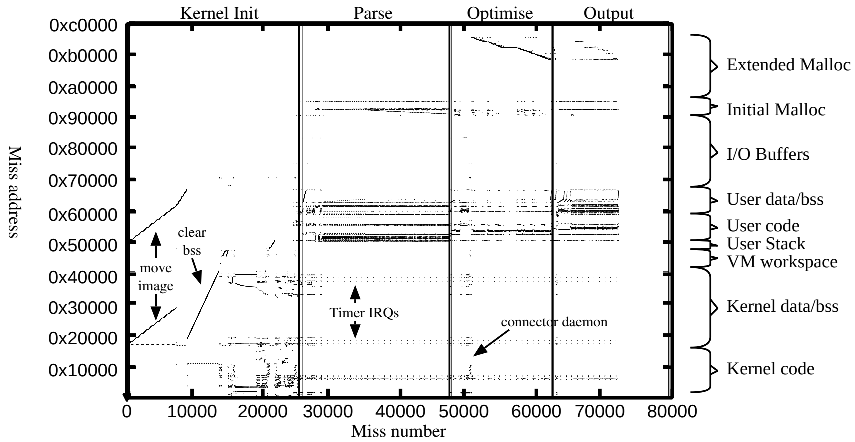

Figure 4: Address of Cache Miss vs Miss Number

The first 10000 misses show the kernel copying the user process to its execution address, and clearing both the user and kernel uninitialised data segments[5](`bss`). Then the kernel performs various initialisations before starting the user process. During the input stage of the compiler the parser code is frequently fetched. In the optimise phase the process has to expand its memory heap (`malloc`) space, and a new segment is seen in use. The kernel activity seen is the timer interrupt code, and the running of the network connection timeout thread at about miss number 50000.

Whenever a cell on a stream VCI arrives at the CPU node the processor has to be stopped while the stream data is inserted into the cache. This time could be reduced or even removed with a fancier design of the cache (dual porting for example). In practice, as shown below, the increase

[5]Wanda defines that this segment will be zeroed before programs are started.

in running time of processes not using the stream cache is relatively small – certainly a lot smaller than the delays caused if the kernel had to process data with similar arrival rates.

The PAL compiler was then run with the cache setup in stream mode as described above. Data streams at various rates was sent into the s-cache region, while the kernel and PAL compiler used the other region as a combined i&d-cache. Table 1 shows the running time of the three main phases of the PAL compiler, executing on the CPU node with various data rate streams arriving. The input and output of the compiler were done to memory buffers so network and disc latencies are not included.

Stream Rate kbps	Parse ms	Process ms	Output ms	Total ms
No stream	261	499	128	888
114	261	499	128	888
900	262	499	128	889
1200	262	500	129	892
4500	264	503	134	901

Table 1: Pal Compiler Execution Time with Cache Stream

4.2 Using the Stream – Spot the Ball

A demonstration was used to investigate the use of the stream cache. "Spot the Ball" receives a stream of frames and follows a "ball" as it moves around between successive frames. The "ball" consists of a 3x3 grid of pixels of a well defined value, other pixels in the frame are randomly coloured. A very simple algorithm is used for the search; it was specifically designed to show up some of the characteristics of the stream cache mechanism.

This application is clearly of no practical use, but it is closely related to tasks that might be performed in a multimedia machine. If a seminar were being relayed, for example, the machine could automatically track the speaker moving around the podium permitting both a centred view of the speaker (which would have to pan smoothly to avoid making the viewer sea-sick) and an unobstructed view of, say, the blackboard being pointed at. In this example the matching algorithm would be more complex but the principles remain the same.

The stream generator assembles the frames and transmits them a line at a time. Each frame is given a different random background, and between frames the ball is moved randomly in both dimensions. Each line is sent through a standard system call, which results in the most likely shape of the stream entering the fabric to be bursts of eight back-to-back cells.

The spotter program, running on the CPU node, defines a frame structure and treats the virtual memory range into which the stream is being received as an array of these frames. This is a very natural way to address the stream – it is simply indexed by frame number. Some care has to be taken to ensure that when the VM range is exhausted the addresses are adjusted. The incoming stream will simply wrap round and begin again at the start of the region, so all the receiving process needs to do is subtract the size of the region from the base address it uses for its "array". A "cache-aware" operating system will have to provide a mechanism for informing the process when to do this.

The algorithm used by the spotting process is to start from the location found for the ball in the previous frame and spiral outwards checking individual pixels until one of the correct colour is found. This is then used as a centre point about which the surrounding pixels are checked. Clearly this is inefficient, but it is very easy to follow the behaviour of the program and hence the system. Provided it is not trapped by the kernel for using an address outside the VM window, the system will stall if the data accessed has not arrived. For the purposes of the experiment, rather than stalling the CPU, it was made to execute a loop in which it spins checking the cache tags for the arrival of the data and incrementing a counter. Since the loop is executing from the CPU internal cache and interrupts are disabled this count gives a measure of the time spent waiting for the data.

The frame size was set to 64×64 with 32 bit pixels, giving 16kbytes per frame. The sliding VM window was thus set to move in 16kbyte blocks, causing a "future access" address fault to occur only the first time a frame was touched. The source was set to generate 10 frames per second, giving a data rate of 1Mbit/s. In one sampled frame the search followed the path shown in figure 5. The start point was (32,32) and the ball, shaded in the figure, had moved to (30,36). In each square the dally count of the first access to that location is shown. The delay waiting for the first location (marked 'X') includes the interframe gap and "future access" trap, so is not comparable, and is not shown.

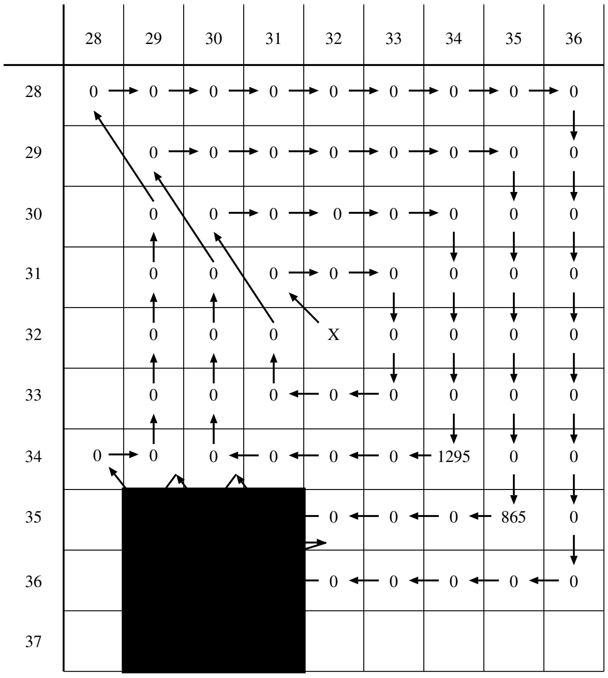

Figure 5: Time (in CPU loops) for Search Access

The results give a clear picture of the operation of the stream cache. All lines prior to the first location (i.e. with a y component less than 32) will have arrived before it so the access delays to pixels in those lines are zero. The time spent searching round the first turn of the spiral is sufficient for the data in the line below the 'X' to have arrived so the CPU does not stop. By the second turn of the spiral the CPU is beginning to get ahead of the stream and it is seen to stall for a while waiting for the lower right pixel (34,34). The same effect is seen on the third turn, but the stall at (35,35) is shorter because the CPU has done more work traversing the longer path. Just after this the ball is encountered for the first time and a defect of the algorithm becomes clear—it searches in locations that the ball cannot possibly occupy. This extra work and the fourth cycle round the spiral take long enough that the CPU is not stalled in the lower right access to (36,36). The ball is then discovered, with a final stall when checking for its lower left corner (29,37).

Once the ball has been located the program has no interest in the rest of the data for that frame and it will progress to the next frame. It is at this point that it is trapped for doing an access far enough into the future to warrant rescheduling the process. The simple handler just adjusts the sliding window and returns control to the user process dally loop. Despite this simplicity, control is given to the kernel and it is clear where an improved scheduler would fit in. The trapping of past accesses was also observed, by making an access more than eight frames into the past.

5 Conclusions

The idea of a "stream cache", where multimedia data is placed directly in the CPU cache, has been proposed. Possible operating system support methods have been suggested to provide a natural abstraction of the data stream to a user application. Finally an example implementation of an input stream cache has been demonstrated.

The use of an s-cache memory certainly adds complexity to the cache control hardware. However, the hardware complexity, as opposed to mental complexity, should be put in perspective; the only unusual component in our system is the cache controller itself (we use standard cache tag memory etc), which is implemented in a pair of Xilinx FPGAs[6] and a couple of standard PALs (10ns 22V10s). The desire for a flexible mapping of i-, d- and s- cache regions would also add some complexity, but again only relatively small parts of the hardware would need changes.

This work continues; having designed the hardware, we believe that only through integration with the operating system and the provision of a straightforward user programming model can we expect applications to make use of it.

[6] A single larger FPGA could have been used, but the design naturally partitioned into address (in an XC3020) and data (in an XC3064) sections. Using two smaller devices significantly reduced the time taken from modifying the hardware description to having a bitstream to load into the FPGA.

6 Bibliography

References

[1] Joel Adam, Henry Houh, and David Tennenhouse. Experience with the vunet: A network architecture for a distributed multimedia system, September 1993. To be presented at the 18th Conference on Local Computer Networks.

[2] P Barham, M Hayter, D McAuley, and I Pratt. Devices on the desk area network, February 1994. Submitted to IEEE JSAC.

[3] Danny Cohen, Gregory Finn, Robert Felderman, and Annette DeSchon. Atomic: A low-cost very high speed lan. Technical Report ISI/RR-92-291, University of Sothern California/Information Sciences Institute, September 1992.

[4] G Finn. An integration of network communication with workstation architecture. *ACM SIGCOMM, Computer Communications Review*, 21(5), October 1991.

[5] MD Hayter. A workstation architecture to support multimedia. Technical Report 319, Cambridge University Computer Laboratory, November 1993. Ph.D. dissertation.

[6] MD Hayter and DR McAuley. The desk area network. *ACM SIGOPS Operating Systems Review*, 25(4), October 1991. Also Cambridge University Computer Lab Tech Report 228, May 1991.

[7] IM Leslie and DR McAuley. Fairisle: An atm network for the local area. *Proc. ACM SIGCOMM*, September 1991.

[8] D McAuley. Operating system support for the desk area network. In *Proceedings of 4th International workshop on OS support for Digital Audio and Video*, November 1993.

[9] T Pfeifer. Micronet machines. new architectural approaches for multimedia end-systems. In *Proceedings of 4th International workshop on OS support for Digital Audio and Video*, November 1993.