

# Caching Techniques for Multi-Processor Streaming Architectures

Martijn J. Rutten<sup>1</sup>, Jos T.J. van Eijndhoven<sup>1</sup>, Evert-Jan D. Pol<sup>2</sup>

<sup>1</sup>Philips Research Laboratories

<sup>2</sup>Philips Semiconductors

Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

{martijn.rutten; jos.van.eijndhoven; evert-jan.pol}@philips.com

## ABSTRACT

In the world of complex SoCs for consumer applications, multiprocessor architectures usually deploy caching techniques to alleviate the cost of data communication between processing elements. In this application domain, the characteristics of streaming applications play a dominant role in the design of the multiprocessor architectures. These characteristics not only influence the design at SoC level, but also permeate the design of lower level blocks, such as caches.

This paper proposes three novel techniques for data caching in multiprocessor streaming architectures. These techniques exploit the combination of mechanisms from the domains of stream caching and cache coherency, optimizing the effectiveness of data caches in a streaming multiprocessor context.

Our first implementation targets cache systems in a multiprocessor architecture for high-definition MPEG decoding, where six function-specific processors communicate through shared embedded SRAM. Simulation results demonstrate the effectiveness of the presented techniques, even though the caches are of very small dimensions, i.e., 0.12 mm<sup>2</sup> in 0.12μ technology.

## 1. INTRODUCTION

The convergence of consumer applications in the domestic, automotive, and mobile domains leads to ever more complex products, which need to perform an increasing number of different applications. A common denominator among all of these applications is that they process signal streams, such as audio, video, voice, and combinations thereof.

In order to provide cost-effective solutions for the processing requirements of these applications, consumer electronics vendors are deploying complex SoCs. In the consumer domain, these complex SoCs generally are based on multiprocessor architectures to balance hardware cost, power consumption, and delivered performance. Oftentimes, these multiprocessor SoCs rely on shared memory for inter-processor communication.

With progressing IC technology, the cost of transport of data has become a dominant factor in the SoC design [1]. To compound this difficulty, the bandwidth and latency requirements on this shared-memory increase with the rising application demand for higher resolution and multiple channels. Therefore, architects have devised mechanisms that aid in alleviating the mentioned transport difficulties.

One such technique that is widely applied in SoC designs is caching, e.g., virtually all CPUs incorporate caches to mitigate latency and bandwidth requirements to memory for data and instructions. Caches have been deployed for decades in a wide

range of systems [2]. Many different forms of cache designs have been devised to address the diverse characteristics of different application domains.

Consumer electronics products are centered on media applications, such as digital audio processing, speech processing, digital video coding, image enhancement, etc. A general characteristic of media-processing applications is stream-based processing. This characteristic can be exploited in the design of caching systems [3][4].

Despite the maturity of the field of caching technology, this paper presents three new techniques to enhance the effectiveness of caches in a stream-based context. Section 3 outlines the generic processor ‘shell’ that is key in reducing processor complexity and decoupling the processor from the transport network. Relying on the explicit synchronization mechanism supported by the processor shell, Section 4 introduces our concepts for caching streaming data in a multiprocessor environment. Section 5 outlines a concrete implementation of the described techniques in tiny data caches within the processor shell. This is followed by a simulation setup in Section 6 that demonstrates the effectiveness of the presented techniques.

## 2. RELATED WORK

With the advent of media processing, caching techniques optimized for streaming data have received an increased attention in the form of stream buffers [5], stride-prediction tables [6][7], and stream caches [3]. Oftentimes, these techniques are applied to traditional (multi-way) associative caches. The canonical form for selecting victims in a fully loaded associative cache is the least-recently used (LRU) mechanism [8]. Such victimize strategies are oblivious to the stream associated with cached data, causing cache contention when the processor accesses multiple data streams through its cache. To some extent, techniques have been devised to avoid contention by extracting stream information from the processor’s access pattern and separate cache blocks accordingly [9]. Section 4.1 outlines a more cost-effective cache organization that exploits the processor’s knowledge of the application structure. Such a cache organization fits well to media-processing architectures that frequently adopt a dataflow-oriented model of computation [10].

One of the notoriously difficult problems in the field of cache technology for inter-processor communication is cache coherency [11]. This paper combines the—traditionally separate—domains of stream caching and multiprocessor cache coherency. In Section 4.2, we apply an explicit dataflow synchronization scheme [10][12] to control cache coherency and prefetching, fully transparent to the application tasks. This results in a simpler and more

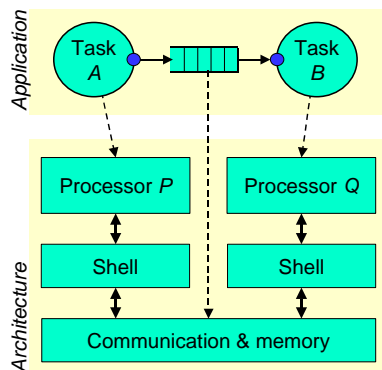
efficient implementation than generic coherency mechanisms such as bus snooping [11]. In addition, it avoids the communication overhead of for instance a write-through architecture [5].

Processors generally apply caches to reduce the latency to access data in memory. For streaming applications, prefetch methods are deployed to predict upcoming I/O operations and further reduce memory access latency [3]. State of the art prefetch techniques address two problems. First, access to a stream must be recognized by matching the addresses of a series of I/O operations and extrapolate this to an expected future I/O access. This recognition of the stream access pattern is troubled by interleaved I/O operations that do not belong to the stream. Second, once a prediction is found that is not available in the cache, the corresponding block of data needs to be fetched from memory. When the prefetch data enters the cache, it may replace data that is still valuable. Therefore, prefetching may be the cause of further cache misses.

Literature on prefetch methods focuses on solving the first problem of predicting future I/O operations [6][7]. Little attention has been given to the second problem of explicitly selecting cached data to be replaced. Section 4.3 proposes a technique that reverses the traditional prefetching approach by predicting cache locations for which the cached data is not expected to be further used. The cache subsequently initiates prefetch actions to fill precisely these cache locations—without overwriting valuable cached data.

### 3. PROCESSOR SHELL

Figure 1 shows the mapping of a streaming application onto parallel processors. The application tasks communicate through streams of data, mapped onto cyclic buffers in shared memory. Communication involves both *data transport* and *synchronization*, which latter is concerned with maintaining and checking filling conditions of the stream buffers. Processors communicate through their *processor shell*: a hardware module that offers generic services to the processor. These services encompass among others data transport, data caching, and inter-task stream synchronization. The shell absorbs a complex part of the system architecture. Their explicit specification allows reuse for each (function-specific) processor, and simplifies processor design.



**Figure 1. Application tasks mapped onto parallel processors. Shells handle inter-processor communication via stream buffers in shared memory.**

Application tasks mapped onto one or more processors explicitly execute a dataflow synchronization mechanism [10][12]. With this mechanism, producing and consuming application tasks syn-

chronize inter-task data transport via shared memory. The synchronization consists of *inquiry* and *commit* actions that provide information on the amount of valid (produced) data and the amount of empty room (consumed data) in memory that is shared between the application tasks. The application tasks initiate these actions, independently from data I/O.

Each shell locally administers the buffer filling of streams associated with tasks executing on the shell’s processor. Shells of different processors exchange messages to keep their buffer administration up to date in such way that inquiry and commit actions of the processor can be served promptly and locally. The following section shows how data caching techniques can exploit such a distributed synchronization scheme.

### 4. STREAMING DATA CACHE

Processors transport all media data to and from their shells through read and write operations. The shells internally compute the actual address into a cyclic stream buffer in shared memory and access the data. The shells provide the read and write interface to hide aspects such as the width of system data paths; data alignment in memory and cyclic buffer addressing, and data stream caching—including coherency and prefetching control. Shells incorporate separate read and write caches that play an important role in decoupling the processor from the global communication network. The following subsections detail the three most important and novel concepts applied in these caches.

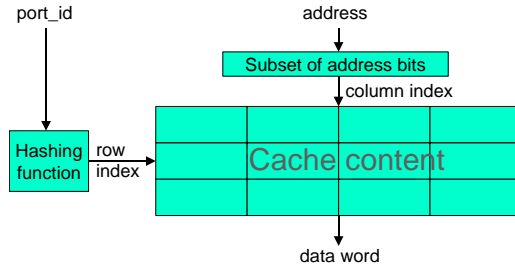
Impartial to the described caching techniques, we chose to separate the read and write data path to more easily support parallel read and write requests, for instance from a pipelined processor. The remainder of this paper focuses on read accesses to the read cache, as these are the most challenging with respect to cache coherency and prefetching. However, the proposed techniques apply equally well to handling write requests.

#### 4.1 Cache indexing through stream IDs

Stream buffers in shared memory compete for shared resources, such as cache lines and a limited number of banks to store address tags. The processor tasks are I/O intensive, requiring efficient cache behavior. Thus, contention on the cache resources leads to large and unpredictable task execution delays. For each read or write access, the processor tasks pass a task and port ID. Note that the port identifier has local scope for each task. The shell combines the task and port IDs to form a stream identifier. To limit cache contention, the shell indexes its read or write caches through this stream ID, effectively decoupling the caching of stream content of different streams.

The stream ID can be used to select a row of cache blocks. However, we chose to share cache rows over different tasks to limit the cost of cache memory in the shell. Thus, the shell only uses the port ID to select a cache row, and cache rows are shared over equivalent port IDs of different tasks. Moreover, instead of directly addressing of the cache row by the port ID, the shell applies a hashing function by which it translates the port ID into a potentially smaller number of cache rows. Figure 2 depicts this cache organization. We chose the hashing function to be a simple modulo operation over the number of rows. This way, a single task may share a single cache row over multiple task ports. This is cost-effective when for instance all media data is read through the first task port, and the task only occasionally reads some meta data from its second task port. Sharing the cache row then avoids

the hardware cost of a full row of cache locations for the second task port.



**Figure 2. Addressing cache locations through port\_id and address.**

Figure 2 depicts a direct-mapped cache organization. This means that every port ID and address combination can only map to a single cache location. Within a cache row selected through the port ID, a cache block is indexed through the lower bits of the I/O address. Thereto, the number of cache blocks in the row is restricted to a power of two. This results in a simple and cost-effective cache implementation in the processor shell. Clearly, such a scheme can be extended to more general set-associative cache organizations, where the stream ID selects a cache row and the lower bits of the address select a set of cache blocks. The actual data word is then further located through tag matching on the address.

## 4.2 Cache coherency through synchronization

For processing streaming data, several groups work on processors with special stream cache architectures to improve the data transport to/from memory [3-7]. In any multiprocessor system that deploys caches to access shared memory, cache coherency must be enforced to ensure that each processor reads properly updated data values from shared memory. When a processor reads data from a stream buffer through a private cache, the processor needs to ensure validity of the read data.

Synchronization between tasks is required for intertask signalling of delivery or consumption of data. The inquiry/commit synchronization scheme operates at byte granularity. A major responsibility of the cache is to hide the global interconnect data transfer size and data transfer alignment restrictions from the processor. As a result, the same memory word may be stored simultaneously in the caches of different processors, and invalidate and dirty information must be handled in each cache at byte granularity.

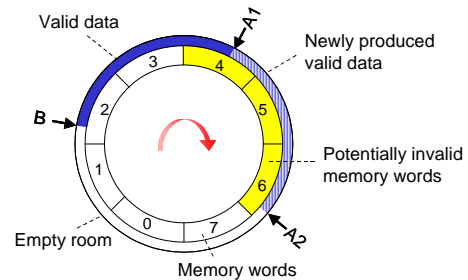
In a multiprocessor system designed for streaming data, these cache coherency issues can be solved in a specialized and efficient way. The inquiry/commit synchronization mechanism explicitly controls cache coherency transparently to the processor. The shell's cache coherency mechanism builds on three key observations:

1. The access window on stream data, which is granted to a task port by a successful inquiry action, is guaranteed to be private. Thus, read/write operations in this area are safe and do not require immediate intra-processor communication.
2. Additional inquiry requests extend the access window, obtaining new memory space from a predecessor in the cyclic buffer. Data in the cache that corresponds to this new memory space possibly needs invalidation. A subsequent read action

on such a cache location then results in a cache miss, upon which the cache loads fresh valid data from the cyclic buffer.

3. Local commit requests reduce the access window, leaving new memory space to a successor in the cyclic buffer. Dirty data in the cache that corresponds to the memory space in the reduction interval needs to be flushed to the cyclic buffer to make the local data available for other processors. Sending the 'commit message' to another processor must be postponed until the cache flush is completed and safe ordering of memory operations can be guaranteed.

Figure 3 depicts the fixed-size cyclic memory space used as communication buffer. The rotation arrow in the center shows the direction in which a producing task *A* and a consuming task *B* move their access points ahead with each commit action. A commit action by the producer reduces the access window on empty room in the buffer, while the producer extends this access window with a successful inquiry action. Equivalently, the consumer extends its access window on valid data through inquiry actions, and reduces the access window by committing already consumed data.



**Figure 3. Basic stream mapped to a finite cyclic buffer.**

The inner circle in Figure 3 depicts the memory words in the buffer. The producer moves its access point (its *write pointer*) from *A1* to *A2* by committing newly written data. The consumer may subsequently extend its access window into this new data range. To ensure cache coherency, the producer's write cache must have flushed memory words 4, 5, and 6 to memory, and the consumer's read cache must invalidate cached memory words with these same addresses.

Generalizing the situation of Figure 3 leads to the following implementation. On a commit action, the write cache flushes all cached data words whose tag addresses overlap with the address range of the producer's previous write pointer to its updated write pointer after the commit action. On a consumer's inquiry action, the read cache invalidates all cached data words whose tag addresses lie between the current write pointer and the write pointer at the last inquiry action. The consumer's shell computes the write pointers from its local buffer administration. Clearly, discrepancies in buffer administration between the producer and consumer shells due to synchronization messaging delays do not affect functional correctness.

## 4.3 Prefetching on dismissed cache locations

Traditionally, prefetching algorithms decide to fetch data that is predicted to be needed in the direct future, disregarding the value of data already present in a fully loaded cache. Thus, these algorithms may victimize valuable data. The processor shell addresses this problem by carefully selecting when to execute a prefetch. Instead of predicting future I/O operations, the shell predicts dismissing of cached data that is not expected to be fur-

ther used. Subsequently, it fetches data to replace the dismissed data in the cache. Thereby, the shell reduces the risk of overwriting cached data that is still needed in the cache.

Prefetching in the processor shell is initiated by read and invalidate requests. Apart from sporadic random accesses within the acquired window of valid data, processors are expected to access data in a streaming fashion. Thus, we assume that subsequent reads belonging to the same stream address a contiguous range in memory in linear order. This streaming behavior allows the shell to cost-effectively embed prefetching caches.

If a read action within a stream buffer accesses the last data word in a cache block, the shell assumes that all data of the block has been read and can be dismissed. At this event, the shell prefetches a data block from a new location in memory that fits to the cache location of the dismissed cached data. The shell prefetches the next higher address from the address of the dismissed data that fits the cache location. As result of this choice, once the prefetched data arrives, it will be stored automatically at the location of the dismissed data. As stream accesses occur in linear order, the stream is expected to access the prefetched data in the near future.

Invalidate requests on cached data to control cache coherency are triggered by processor inquiry actions. These invalidate events mark locations in the cache to be considered as empty. Invalidates are caused by a task that produces new data on the stream. By inquiring if a certain amount of data is available for consumption, the reading processor indicates that it expects to access this new data in the near future. Therefore, the processor shell issues a prefetch for those cache locations marked invalid.

Additionally, the shell prefetches new data for all cache locations within the selected cache row that fall outside the range of valid data in the stream buffer. The latter is of special importance to reduce the latency of updating a cache row immediately after a task switch in case that cache rows are shared between stream buffers of different tasks. For these dismissed cache locations, the shell prefetches from the memory addresses that fit the dismissed cache location and are closest to the current point of access.

The write cache applies a similar strategy by flushing cached data as soon as it predicts that this data will not be accessed anymore. The write cache initiates such a *preflush* when a write access moves to a next word. With a streaming write behavior, the previous word will not be further accessed, and its cache location can be made available for expected future.

## 5. IMPLEMENTATION

We deploy the caching techniques in a multiprocessor architecture where a number of function-specific processors together execute a dataflow-style application. Processors communicate through stream buffers allocated in shared embedded SRAM. Each processor has its own processor shell that handles all data access and buffer administration of the streams associated with the application tasks that are mapped onto the processor.

Figure 4 depicts the internal architecture of the read module inside the processor shell. In addition, the processor shell includes a similar write module, a synchronization module that maintains administration of stream buffer filling, and a task scheduler. The shell implements a control interface to allow access to programmable registers—such as the stream and task table entries in Figure 4—for system configuration.

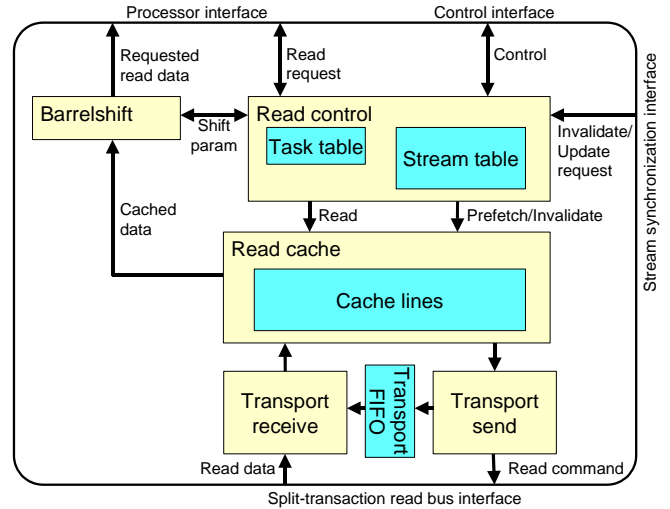


Figure 4. Read unit inside the processor shell.

We deploy the read and write caches in a multiprocessor architecture that targets dual-stream, high-definition MPEG-2 encoding and decoding [10]. In this setup, the processor shell is targeted to execute at around 150 MHz, providing an average 4-cycle latency for single word read and write transactions on a dual 128-bits bus to on-chip SRAM. Initial synthesis results indicate that a typical instantiation of the processor shell template occupies 0.2 mm<sup>2</sup> in CMOS12 technology, out of which the read and write modules absorb approximately half of the total shell size.

### 5.1 Read requests

The Read Control component in Figure 4 handles the read requests from the processor. The processor communicates the port ID and the requested number of bytes to read through a handshake protocol. The Read Control component subsequently indexes the task table with the port ID and obtains the index into the stream table. The stream table administers the current byte-level access point (the *read pointer*) into the communication buffer, as well as the size of the buffer for cyclic addressing. Based on the current access point and the requested number of bytes, the Read Control component generates parallel requests to the cache at the level of memory words. A memory word matches the bus and memory data width, and is for instance 128 bits wide.

The Read Cache component indexes the cache using the port ID, and performs a tag match on the address passed by the Read Control component. On a cache miss, the cache issues a read request to the Transport Send component. The Transport Send component places the cache location of the requested data in a queue, to be accessed by the Transport Receive component upon receiving the data from the memory. Received data words are passed through the cache to the Barrelshift component. The barrel shifter combines data from potentially multiple cache words and left shifts the result before acknowledging the processor read request.

The Transport Send and Receive components are separated to enable split-transaction requests, in which the Transport Send component can issue a new request every bus cycle without waiting for the requested data to return. This pipelining of bus requests greatly reduces the latency for multiple transactions, e.g. on unaligned read requests from the processor that span multiple memory words, or invalidate actions that triggers multiple prefetch requests.

## 5.2 Update/Invalidate requests

The synchronization module in the shell signals update/invalidate events to the cache, triggered by inquiry and commit actions from the processor. Update events consist of a port ID and the size in bytes of a corresponding commit action by the processor. The Read Control component uses this information to update its current read pointer in the stream table.

An invalidate event is accompanied with the port ID and the current buffer filling of the requested stream. Upon such invalidate requests, the Read Control component computes an address range of cache words that potentially need to be invalidated, based on the current buffer filling and the filling at the previous inquiry action—as maintained in the stream table. The cache matches the tag address of all cache locations of the indicated stream and invalidates cached data words that lie within the specified address range.

## 5.3 Prefetch requests

The shell initiates prefetch requests on both read and inquiry (invalidate) events from the processor. To this end, the Read Control component passes an address range of available data words in the communication buffer to the cache through the Prefetch/Invalidate interface. The cache subsequently issues prefetch requests for each cache location that is invalidated, or has a tag address behind the current read pointer or outside the range of valid data.

## 6. RESULTS

This section gives two experiments to show the effectiveness of the proposed caching techniques on a heterogeneous multiprocessor architecture [10]. The experiments execute on a cycle-accurate, bit-true SystemC [13] model of the processor shell and communication network. The application tasks execute on highly abstract models of function-specific processors. While the generic caches of Section 5 allow arbitrary access patterns, the simulation results are based on our multiprocessor MPEG implementation that exhibits a largely linear access pattern for inter-task communication.

### 6.1 Dual-task discrete cosine transform

We present an example study into the behavior of the read and write caches of a hardwired processor that computes the inverse discrete cosine transform (IDCT) for two different MPEG-2 streams. The IDCT processor executes the two IDCT tasks in a time-shared fashion. Each task reads coefficient data from its input stream, and produces blocks of pixels on its output stream.

The IDCT read and write caches connect to separate read and write buses of 128 bits wide to on-chip memory. The processor strictly reads and writes in a streaming fashion. The IDCT processor has two 32-bit data ports to its shell, on which it issues variable-length read and write requests. It is the task of the shell to perform address generation. The processor has no idea on memory alignment issues—the read and write requests may occur on an unaligned address. In this setup, the shell read cache is sized to contain four bus words of 128-bit, which are shared by the input streams of both IDCT tasks. The write cache contains only two bus words, shared by the output streams of both tasks. Thus, total cache size of the shell is 96 bytes.

Table 1. Dual stream DCT read/write cache behavior.

	No cache		Cache, no pre-fetch / preflush		Cache, pre-fetch / preflush	
	Rd.	Wr.	Rd.	Wr.	Rd.	Wr.
# Misses (x 10 <sup>3</sup> )	-	-	466	274	0	0
# Transfers (x 10 <sup>3</sup> )	1117	994	466	428	532	428
Average latency	11	3	7	3	2	3

Table 1 shows the actual cache behavior. The experiment shows that the caches dramatically reduce the bandwidth requirement on the communication network. Additionally, the prefetch mechanism reduces the average latency from 11 to 2 cycles. The reduction in write latency by issuing preflush actions is not visible as the simulation model immediately acknowledges a write request—even before the data is written in the cache. Even without a write cache, the shell contains a one-word write buffer to hide the latency of accessing the write bus from the processor.

Despite the tiny cache sizes, the (shell of the) IDCT processor experienced *not a single* read cache miss as result of the automatic prefetch, although some reads had to wait a few cycles because the prefetch did not yet complete. The write cache allocates on a write-miss (does not fetch). Correspondingly, the preflush empties dirty cache words to memory, so that the allocate *not once* had to delay for first flushing a dirty cache word.

### 6.2 Multiprocessor MPEG-2 decoding

To show the effectiveness of the caches as part of the generic processor shell, we decode a number of MPEG-2 streams on the simulated architecture. The architecture consists of six hardwired processors: DMA, variable-length decoding, picture and slice decoding, run-length decoding and inverse quantization, inverse discrete cosine transform, and motion compensation. Each processor has its own shell and communicates through stream buffers in on-chip memory. The reference pictures for motion compensation are accessed from off-chip memory and do not pass through the shell. The cache sizes vary with the number of input and output streams to the processors. Each input stream is assigned a cache line of maximally 4 bus words, while each output stream is assigned to a cache line of 2 bus words.

Table 2. Cache influence on MPEG-2 execution time.

	No caches	Caches, no pre-fetch / preflush	Caches, pre-fetch / preflush
tennis	100	75	55
teeny	100	73	52
tech	100	74	64
oslo	100	73	61

Table 2 gives the normalized execution time for three standard-definition and one high-definition (oslo) MPEG-2 streams of 8, 19, 31, and 30 frames, respectively. The table shows that the prefetching caches—despite their tiny sizes—significantly reduce the overall execution time.

## 7. CONCLUSION

This paper presents three innovative techniques for the design of data caches that are optimized for streaming multiprocessor architectures. The primary technique uses a generic dataflow synchronization scheme to accomplish cache coherency and automatic prefetching/preflushing. These properties are transparent to the application tasks.

Secondly, the cache organization reserves non-overlapping cache locations for each data stream accessed by the processor. The processor selects a set of cache locations through a stream identifier that is unique for the dataflow structure. This greatly reduces the complexity of tag matching hardware, while avoiding cache contention with respect to conventional associative caches.

Thirdly, the cache employs a new technique for data prefetching that prevents overwriting valuable cached data. The method identifies cache locations for which it predicts that the data content is not expected to be further used and can be dismissed. Information on the data availability in stream buffers in shared memory—provided by the synchronization scheme—ensures that no invalid data words are prefetched.

Even though the techniques can be used for larger cache designs, our specific implementation focuses on tiny caches of around a 100 byte. Simulation experiments show that the portrayed cache architecture effectively reduces utilized bandwidth and access latency to on-chip memory. We aim to deploy this cache architecture in a range of heterogeneous multiprocessor SoC subsystems for consumer electronics devices.

## REFERENCES

- [1] Doug Burger, James R. Goodman, and Alain Kägi, “Limited Bandwidth to Affect Processor Design”, *IEEE Micro*, vol. 17, no. 6, pp. 55-62, Nov./Dec. 1997.
- [2] Alan J. Smith, “Cache Memories”, *ACM Computing Surveys*, vol. 14, no. 3, pp. 473-530, Sept. 1982.
- [3] Daniel F. Zucker, Ruby B. Lee, and Michael J. Flynn, “Hardware and Software Cache Prefetching Techniques for MPEG Benchmarks”, *IEEE Trans. Circuits and Systems for Video Technology*, vol. 10, no. 5, Aug. 2000.
- [4] Z. Wu and W. Wolf, “Study of Cache Systems in Video Signal Processors”, *IEEE Workshop on Signal Processing Systems*, pp. 23-32, Oct. 1998.
- [5] Norman P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers”, *Proc. 17<sup>th</sup> Int. Symp. Computer Architecture*, pp. 364-373, May 1990.
- [6] Tien-Fu Chen and Jean-Loup Baer, “Effective Hardware-Based Data Prefetching for High-Performance Processors”, *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609-623, May 1995.
- [7] J. Fu, J. Patel, and B. Janssens, “Stride directed prefetching in scalar processors”, *Proc. 25<sup>th</sup> Int. Symp. Computer Architecture*, pp. 102-110, Dec 1992.
- [8] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publ., San Mateo, CA, 1990.
- [9] R.K. Arimilli et al., *Extended Cache State with Prefetched Stream ID Information*, US Patent US6360299, March 19, 2002.
- [10] Martijn J. Rutten, et al., “Eclipse: A Heterogeneous Multiprocessor Architecture for Flexible Media Processing”, *IEEE Design and Test of Computers: Embedded Systems*, pp. 39-50, July/Aug. 2002.
- [11] David E. Culler, Jaswinder Pal Singh, with Anoop Gupta, *Parallel Computer Architecture, A Hardware/Software Approach*, Morgan Kaufmann Publ., San Francisco, CA, 1990.
- [12] O. P. Gangwal, A. Nieuwland, and P. Lippens, “A Scalable and Flexible Data Synchronization Scheme for Embedded HW-SW Shared-Memory Systems”, *Int. Symp. System Synthesis (ISSS)*, pp. 1-6, Oct. 2001, Montréal, Canada.
- [13] *SystemC User’s Guide*, version 2.0, [www.systemc.org/](http://www.systemc.org/), 2001.