A Memory Interface for Multi-Purpose Multi-Stream Accelerators

Sylvain Girbal
Embedded Systems Lab
Thales Research and Technology
Palaiseau, France
sylvain.girbal@thalesgroup.com

Olivier Temam
ALCHEMY Group
INRIA Saclay
Orsay, France
olivier.temam@inria.fr

Sami Yehia
Embedded Systems Lab
Thales Research and Technology
Palaiseau, France
sami.yehia@thalesgroup.com

Hugues Berry
ALCHEMY Group
INRIA Saclay
Orsay, France
hugues.berry@inria.fr

Zheng Li
ALCHEMY Group
INRIA Saclay
Orsay, France
zheng.x.li@inria.fr

ABSTRACT

Power and programming challenges make heterogeneous multi-cores composed of cores and ASICs an attractive alternative to homogeneous multi-cores. Recently, multi-purpose loop-based generated accelerators have emerged as an especially attractive accelerator option. They have several assets: short design time (automatic generation), flexibility (multi-purpose) but low configuration and routing overhead (unlike FPGAs), computational performance (operations are directly mapped to hardware), and a focus on memory throughput by leveraging loop constructs. However, with multiple streams, the memory behavior of such accelerators can become at least as complex as that of superscalar processors, while they still need to retain the memory ordering predictability and throughput efficiency of DMAs. In this article, we show how to design a memory interface for multi-purpose accelerators which combines the ordering predictability of DMAs, retains key efficiency features of memory systems for complex processors, and requires only a fraction of their cost by leveraging the properties of streams references. We evaluate the approach with a synthesizable version of the memory interface for an example 9-task generated loop-based accelerator.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

General Terms

Design, Performance

Keywords

Memory interface, Accelerators, multi-stream

1. INTRODUCTION

Even though CMPs have emerged as the architecture of choice for most manufacturers, there is a consensus that efficiently exploiting a large number of cores for a broad range of programs will be a daunting task. Moreover, ever stringent power constraints may impose in the future that not all transistors, and thus not all cores, operate at the same time [10].

Consequently, accelerators, i.e., specialized circuits/ASICs, are becoming an increasingly popular alternative. For cost and efficiency reasons, they have been a fixture in embedded systems where SoCs can include tens of accelerators. In high-performance general-purpose systems, they would enable low-power high-performance execution of important tasks. Their footprint is far smaller than a core, allowing to cram a large number of accelerators on a chip, trading some of the cores of a many-core. Such a set of accelerators would become akin to a *hardware library*, and the larger the library the more likely a programmer will find algorithms useful for his/her program. Moreover, the programming support for accelerators is far more simple than parallelization, it is indeed more like a library call. Finally, accelerators can even speed up non-thread parallel tasks thanks to circuit-level parallelism.

While accelerators have many assets, their obvious weakness is flexibility. As a result, a trend is emerging for flexible accelerators: either accelerators which implement the most frequent computational patterns for a set of programs [13], or accelerators which efficiently merge together the circuits for multiple programs [24]. Such flexible accelerators are configurable, but dedicate far less on-chip estate to configuration logic than FPGAs, and are thus much closer to ASICs than FPGAs in terms of cost, power and efficiency.

In the trend towards more customization, loop-based accelerators are becoming especially popular [13, 24, 12, 2] because they not only speed up computations through customization but also achieve high-memory bandwidth by leveraging loop constructs to efficiently stream data into accelerators. As a result, we may soon see complex loop accelerators with a large number of streams to feed. For now, there has been little focus on the memory interface (including the detailed stream implementation) required to achieve the expected memory bandwidth.

Such a memory interface cannot just consist of multiplying the number of DMAs, nor can it correspond to the memory interface used for high-performance processors. A DMA is typically used to feed data into an accelerator, and usually, one DMA handles one stream of data. If the accelerator contains multiple streams, the task of the DMA becomes significantly more complex: it must load balance streams and multiplex the memory bandwidth among the different accelerators. Moreover, as the number of streams scales up, multiple reuse opportunities occur that, if not exploited, would result in sub-par performance. At the same time, it must strictly preserve the ordering of data fed to the accelerator because a custom circuit behaves in a fundamentally different way than a processor: data is *pushed* to the accelerator which expects data to arrive in the right order, as opposed to being *pulled* by the processor when requested (using addresses). Still, the memory systems of general-purpose processors have the desirable property of being designed to achieve both high bandwidth and reuse for multiple concurrent and out-of-order memory accesses, through a combination of non-blocking caches and prefetchers. But this approach is not compatible with accelerators because of its aforementioned pull vs. push mode of operation, and because of its steep cost.

Moreover, the memory interface of multi-stream accelerators is not only key for their performance, it is also the most important part of the accelerator in terms of area and power. For an example 9-task generated loop-based accelerator synthesized using the Synopsys Design Compiler and the TSMC 90nm library, the accelerator streams alone account for more than 8 times the area and 16 times the power of the computational and storage (registers) logic of the accelerator itself.

In this article, we propose a memory interface for multi-stream accelerators which can realize the execution correctness and determinism of DMAs, while retaining many of the performance advantages of general-purpose processors memory systems (reuse, multiple concurrent requests, out-of-order requests), at a small area and power cost. We show how to design streams capable of sustaining 1-word issue per cycle to the accelerator in the presence of complex memory patterns (e.g., short loops, irregular accesses, etc), which is key to expand the scope of such accelerators. We also show how to complement streams with a Stream Table, which has a small area and power footprint compared to streams, but which boosts the average accelerator speedup over a core from 5 to 10 by taking advantage of short-term cross-reference temporal reuse, and by augmenting the apparent memory bandwidth.

2. MEMORY INTERFACE

In this section, we describe the memory interface structure, which includes the streams and a table called the Stream Table, see Figure 1. For that purpose, we go through the different memory access issues raised by multi-stream accelerators and how they are handled by the interface.

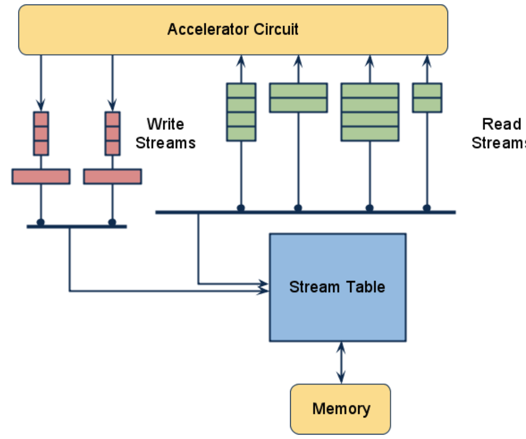


Figure 1: *Overview of the multi-stream memory interface.*

2.1 In-Order accesses and reloads using pre-allocation

As mentioned before, an accelerator can often be considered as a passive circuit which receives data from memory and immediately processes them, it does not "request" data using memory addresses. For instance, it is not possible to use Stream Buffers [18], as proposed for fetching streams into caches, because the circuit does not send an address when it needs a data, and because the circuit cannot filter out speculatively fetched data.

In a single-stream accelerator with a memory ensuring in-order requests, that is a non-issue, data comes back in the order it is requested. If the accelerator is plugged to a NoC, memory requests may no longer arrive in-order. For that reason, the stream controler (i.e., the DMA) must *pre-allocate* an entry in the stream before sending a request to memory. Even if the data comes back out of order, it is stored in the appropriate stream entry. If the top entry (next data to be fed to the circuit) is not ready, the circuit is simply stalled until the appropriate data arrives.

In loop-based accelerators, we denote as a *stream* the control logic required to generate addresses to memory and the fifo used to store data for the circuit. For loops, a stream includes a counter, which is initiated with start and end addresses, and a stride, and it then fetches and feeds the data continuously to the circuit. A handshaking protocol between the stream and the circuit is required to steer data consumption: the stream signals to the circuit that data is available in the buffer with a `ready` signal, and the circuit signals to the stream with a `shift` signal when it can discard the current data and move to the next one; the stream also signals when it has fetched all requested data (`stop` signal), so that the circuit can compute its own global stop signal.

The latency tolerance capability of the stream is naturally correlated to its size. The longer the latency, the longer the time between the pre-allocation and the moment the data is used. If the stream size is properly dimensioned for the latency, then in steady state, a filled buffer can feed the circuit without stalling for the time it takes to fetch data from memory.

2.2 Access patterns

Narrow streams for large strides..

Along the same principles as cache blocks for exploiting spatial locality, it is more efficient to have multi-word stream entries (wide streams) in order to minimize the stream cost. Unlike cache blocks, streams need not all have the same width. And in a multi-stream accelerator, it is possible to offer a mix of streams, best suited to the accelerator. Stride-1 accesses favor the widest possible streams, while large strides favor narrower if not 1-word wide streams. While stride-1 accesses are the most frequent, a quantitative study of loop locality showed that large strides were also frequent [20], typically in array column-wise accesses; small-value strides other than +1/-1, e.g., 2 or 3, are infrequent. The maximum stream width is somewhat constrained by the rest of the memory system. Since memory systems in multi-cores (homogeneous or heterogeneous) are likely to include shared L2 or L3 caches, data is usually fetched by L2 blocks (B words), and thus stream reloads are simpler if the maximum stream width does not exceed B.

Complex access patterns..

As the scope of accelerators expands, the task at hand is more likely to contain complex memory access patterns. For instance, the widespread use of small loops in SpecInt-type programs, in signal processing applications (radio, sound, image, video) or even in scientific applications [20] makes it impossible to restrict accelerators to singly-nested loops. Yehia et al. [24] recently demonstrated the performance benefit of considering multi-loop accelerators in order to compensate for start-up overhead. And 2-deep or more loop nests can already induce non-trivial memory accesses. Consider Figure 2, where a few example access patterns are shown. Assuming an entry width of W words, in case (a), the inner loop stride is 1, so a W-wide stream should be used, but the loop bounds are smaller than the first matrix dimension, resulting in stream entries being partially used; case (b) is a column-wise access in a 2-D array where the first dimension is smaller than the stream width; case (c) is an example of indirect addressing.

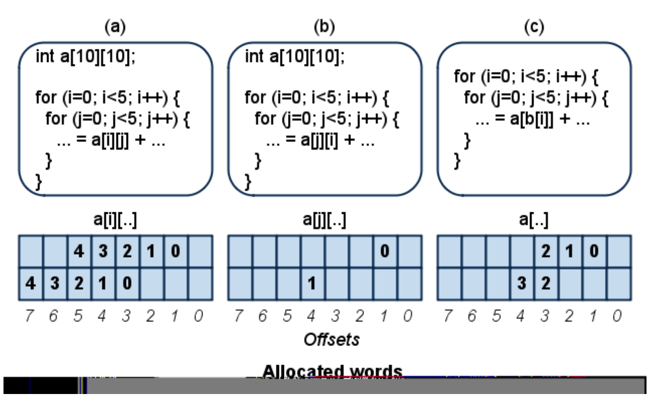


Figure 2: *Stream entries allocation depending on reference patterns.*

All these access cases must be handled by the accelerator streams. The general issue is that not all words within a stream entry may be used (all examples), that all words in a stream entry may not be accessed in a monotonic order (column-wise access and indirect addressing examples), and that some words may have to be accessed multiple times within a short time period ($a[2]$ in indirect addressing example). In order to keep the conceptually simple and fast mode of operation that a word is discarded after being consumed by the accelerator, we forbid the latter case, i.e., each word in a stream entry can only be used once. We explain below how to design the stream to allow all other cases.

Forbidding multiple same-word same-entry accesses..

Because access patterns can be complex, words are allocated one by one in the stream, as soon as the stream controler has computed the next address. If the next word cannot be allocated in the current stream entry, a new stream entry is allocated; each stream entry contains an `allocated` bit and a `tag`, see Figure 3. In order to enforce single usage per word per entry, each stream entry also contains a word mask and a `locked` bit. When a word is allocated, the corresponding bit is set in the mask. If the stream controler wants to allocate the same word a second time, then the entry is locked, and a new stream entry is allocated. Locking the stream entry is necessary to ensure in-order access again. Consider the address reference sequence $a[0]$, $a[1]$, $a[2]$, $a[2]$, $a[3]$, in Figure 2. The first stream entry will be locked upon the second access to $a[2]$, which is allocated in the next stream entry.

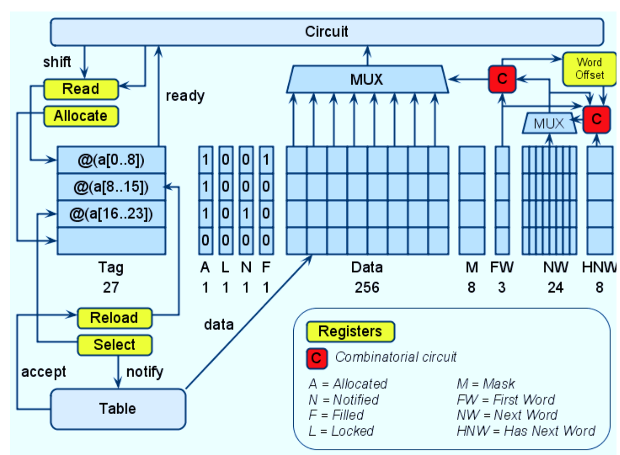


Figure 3: *Detailed design of a read stream buffer.*

Enabling sparse and out-of-order word accesses within an entry..

In order to allow maximum freedom in the order in which words are accessed within an entry, we build the hardware equivalent of a chained list of these words. One major constraint is that word access must be very fast in order to avoid slowing down the circuit; as a result, a two-step indirect access, as usually performed in software, is not tolerable; a word should be issued every cycle to the circuit.

For that purpose, we add a `next-word` field to each entry. Assuming a W-word entry, this field contains W sets of $log_2 W$ bits, each set acting as a *word pointer*. The next-word at position i indicates the offset of the word to be read after word i. Any delay due to the indirection is avoided by reading next-word at the same time as the word is read. Next-Word is then fed to the combinational circuit which drives the multiplexor used to select one word among W for the circuit, see Figure 3.

In order to determine when the last word in the chain for an entry has been reached, a `has-next-word` mask of W bits and a `first-word` set of $log_2 W$ bits are also necessary. When the has-next-word bit is 0, the stream controller knows it has read the last word of an entry, and must shift to the next entry. The first-word bits indicate the offset of the first word to be read in the next entry. The inputs to the multiplexor are thus the current word offset, the corresponding next-word bits, the corresponding has-next-word bit and the first-word bits of the next entry, see Figure 3.

This chained indirect addressing approach induces no timing overhead compared to a direct stride-1 access and enables both sparse and out-of-order word access within a stream entry. For a W-word stream of 32-bit words, the bit storage overhead is $\frac{W \times log_2 W + W + log_2 W}{32 \times W}$, i.e., 13.67% for $W = 8$.

2.3 Concurrent stream accesses at a low cost: readout, allocation, selection, reload

A stream may have to perform all four operations concurrently. For that purpose, there are four registers in each stream, each register pointing to the target entry for one of the aforementioned operations; each register is $log_2 E$-bit large, where E is the number of stream entries. The *readout* register has already been mentioned as used to control the multiplexor to the circuit. The *allocation* register points to the entry being currently allocated. It is used to select the entry where the `allocated`, `next-word`, `has-next-word`, `first-word` and `tag` bits are written. The *select* register points to the entry whose tag will be sent to memory. The *reload* register is actually an E-bit mask because several entries can be reloaded simultaneously, as later explained. It is used to select in which entries the incoming data bits should be written.

Except for the *reload* register which is set by the memory interface, the other three registers behave like fifo pointers: they shift from one entry to the next and back to the top. All these registers are shifted upon different events. The *readout* register is shifted as soon as all words in an entry have been read. The *allocation* register is shifted when a word can no longer be allocated in the currently pointed entry. And the *select* register is shifted as soon as the memory interface has acknowledged the request.

While four different operations normally require four access ports to the storage structure, and are thus exceedingly costly, only one read and write port is actually necessary for each stream storage structure, provided one carefully considers when each bit is being read and written. The bits used for allocation (`allocated`, `next-word`, `has-next-word`, `first-word` and `tag`) are not written in the readout, selection nor reload phase. The only exception is the `allocated` bit which is reset after readout has read all words in the entry; but when the *readout* register points to an entry, it is impossible that the *allocation* register points to that entry as well (we assume a minimum of two entries per stream). Similarly, the `data` bits are only written upon reload. A signaling bit to the memory interface (introduced later) is also written upon selection and readout but again both operations can never occur simultaneously on the same entry (an entry cannot be read to the circuit if it is just being requested to the memory interface). As a result, only a single write port for each sub-structure is required.

2.4 Delaying and load-balancing stream requests to the memory interface

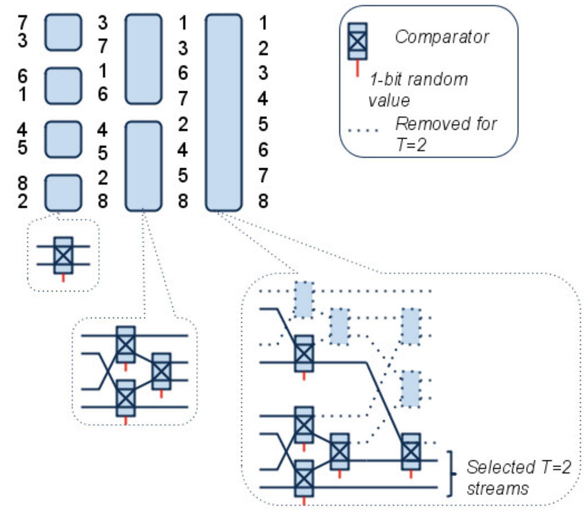


Figure 4: *Combinatorial sorter for $T = 2$ table ports and 8 streams.*

There is a handshaking protocol between the stream and the memory interface. Upon allocation, the `notify` bit of the stream entry is set, signaling a request to the memory interface when the *select* register rotates to that entry. The memory interface can accomodate T requests simultaneously, and it must then pick T streams among the pending ones. We choose to select streams based on the *number of filled words* in each stream, using $log_2(E \times B)$. The number of filled words provides an indication of the stream "needs": if the circuit consumes the stream words slowly, or if this stream has been lately privileged by the memory interface, it will have many words available. This fairness strategy is more robust than round-robin: if a stream is under-privileged, the number of filled words will decrease and its priority will naturally shoot up. And the memory interface randomly selects among the streams with the same number of filled words.

Let us now assume that R streams are sending a request to the Stream Table. If $R \leq T$, then all requests can be handled by the table. If $R > T$, we need to pick the T among R streams with the highest number of filled words. For that purpose, we need to sort the streams according to their number of filled words, and to do so very rapidly and cost-efficiently. We resort to a combinatorial sorter derived from Batcher's odd-even merge sort circuit [7], see Figure 4. This algorithm splits the list to be sorted into pairs of sets of 2^k ordered elements each, with $k = 0$ initially and incremented at each stage, and orders and merges the elements in two sets. Sorting $N = 2^n$ elements requires n merge phases, corresponding in total to $N - 1 + \frac{log_2(N) \times (log_2(N)-1)}{4}$ comparators. We alter Batcher's combinatorial sort circuit in two ways: each comparator is fed with a simple 1-bit pseudo-random number [8] used in case of equality (random decision if two streams have the same number of filled words), and we remove all the comparators of the last (and largest) merge stage which are not necessary to find the top T numbers, see Figure 4. Since we later find that the maximum number of words in a stream is 32 (4 8-word entries), we use 5-bit comparators.

2.5 Stream Table: multiple requests, data reuse

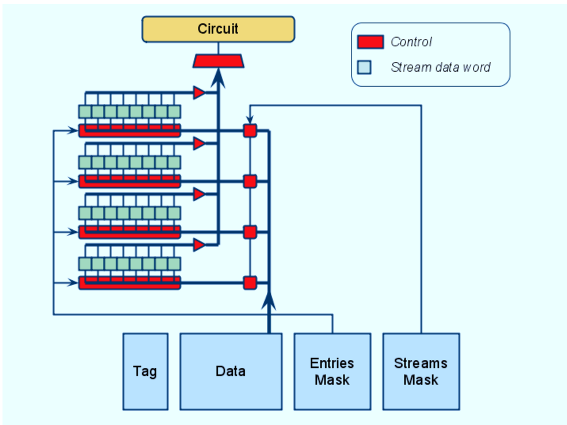


Figure 5: *Table structure and data paths to streams.*

In a multi-stream accelerator, there are usually multiple pending requests. Therefore, we need to implement a table similar to the MSHR (Miss Status Holding Register) of non-blocking caches, with streams, rather than registers, as destinations. Moreover, it can often happen that two streams miss almost simultaneously on the same data. Consider for instance typical references such as `A(i,j)`,`A(i,j+1)`. Rather than issuing two misses, an MSHR would typically record the second request as *hit on pending miss*. We proceed the same way, and add a *stream mask* and a **`pending bit`** to the Stream Table. Whenever a stream hits on a pending miss, the corresponding stream bit is set in the stream mask. When the requested data arrives, it is simultaneously written back to all target streams. Note that writing the same data back to multiple streams simultaneously requires no additional logic or datapath since a bus must anyway connect the Stream Table to all the streams, as shown in Figure 5.

Beyond hits on pending misses, the Stream Table can also fulfill another classic role of exploiting temporal reuse, especially reuse *across* streams. For instance, with `A(i,j)`, `A(i+1,j)`, the reuse distance is too large for a hit on pending miss to occur, but if the matrix dimension of `A` is small enough, `A(i,j)` may still reside in the Stream Table when needed. For that reason, we also allow the table to behave like a cache, and store data along with each entry. That naturally increases the table size, however our goal is not to achieve the same reuse capabilities as traditional cache hierarchies; our focus on streaming data to the circuit makes it unnecessary for many accesses. Our main goal is to take advantage of the frequent short-distance temporal reuse opportunities [20], and that requires only a modestly-sized table, as later shown in Section 4.

Finally, small loops are frequently found in many codes (e.g., SpecInt, signal processing,...), which might result in the same address being requested for several entries of the same stream. Even if the same data appear in two entries in the same stream, these entries should not be merged as data must be delivered in order. Therefore, the table must be able to deliver the data to multiple stream entries. For that purpose, we add an *entry mask* to the Stream Table, besides the stream mask, see Figure 5. Both masks (stream and entry) account for $S \times E$ bits assuming all streams have the same number of entries E. This entry mask also reduces stream cost and speeds up stream reload by saving stream-level tag checks upon reload.

While reloading several distinct data in a buffer require multiple ports, reloading several times the same data in a buffer requires no additional support, the same as for writing the same data to multiple streams. The write port is already connected to all stream entries; the only modification is to allow the simultaneous activation of multiple write signals, see Figure 5.

2.6 Write Streams

Write streams play the same role as write buffers in standard caches, by avoiding to stall the processor or delay miss requests. However, we choose to implement one write stream per circuit output, instead of a common write stream in order to avoid a costly multi-ported stream buffer, and to increase coalescing opportunities (the ability to merge multiple consecutive words in a single write request).

A write stream is composed of two parts: a simple B-word word-wide fifo which buffers incoming write requests, and a B-word latch which also plays the role of a coalescing buffer. The write is sent to memory when a word from the fifo cannot be written in the buffer, because a word at that position is already written, or because the buffer is full. For that reason, the write latch also includes a word bit mask, just like the entries of read streams. The write to memory is delayed until the word fifo is at least half full, in order to find a right balance between coalescing opportunities and not risking to stall the stream. Note that writes can be delayed by misses, hence the half-full threshold precaution.

There is a handshaking protocol between write streams and the memory interface, similar to the one used for read streams. In addition to arbitrate among multiple read streams, the memory interface must also arbitrate between read and write streams. By replacing the "number of filled words" for read streams with "size fifo - number of words in the word fifo" for write streams, we can indifferently consider read and write streams in the load balancing strategy. Indeed, this criterion is the dual of the "number of filled words" criterion: if a word fifo is full, the write stream should be given the utmost priority since the next write will stall the circuit, much like having no filled word in a read stream will stall the circuit.

The Stream Table operates in a write through mode, with one additional tag being used for write streams. Note, though, that there is no hardware support for memory disambiguation, it is considered part of the circuit control task. Most state-of-the-art loop-based circuit generation approaches [13, 24] still do not automatically handle memory disambiguation.

3. EXPERIMENTAL FRAMEWORK

Simulated Architecture. Our architecture is shown in Figure 6(a) and consists of an IBM PowerPC405 [17] core, a simple 32-bit embedded RISC-processor core including a

5-stage pipeline and 32 registers, but no floating-point units. We consider a regular 90nm version running at a frequency of 800MHz, with a 20-cycle memory latency (corresponding to a L2 access). To simulate this architecture, we used the UNISIM [4] infrastructure environment.

(b) *Memory hierarchy parameters*

Figure 6: *Simulated architecture.*

The memory sub-system is composed of two write-back L1 data and instruction caches and a main memory. Their parameters are described in Figure 6(b).

Circuit synthesis. As mentioned before, automatically generating hardware representation from a source code has been previously addressed in research and existing industrial tools [3]. We developed a tool chain which automatically creates loop-based multi-purpose accelerators down to the Verilog HDL. We synthesize all circuits using the Synopsys Design Compiler [1] and TSMC 90nm standard library, with the highest mapping effort of the design compiler (`-map_-effort high -area_effort high` options).

Target accelerator..

We use a 9-task accelerator corresponding to the UTDSP benchmarks of Table 1, as a driving example; the accelerator only includes 32-bit operators for now. To support the fixed-point precision arithmetic which is frequently used in embedded systems for cost and power reasons, we modified all the benchmarks and we chose 12-bit precision for all experiments. The accelerator has been generated using the compound circuit process proposed by Yehia et al. [24]; a similar accelerator could also be obtained using the process proposed by Fan et al. [15]. This compound circuit can be configured to execute each of the individual tasks while having a cost significantly smaller than the union of the 9 circuits; the accelerator is configured for a task through the processor-to-accelerator interface. At any time, only a single task is executed on the accelerator. The number of accelerator operators of each type (adders, multipliers, registers, muxes, read and write streams) are detailed in Table 2; the 32-bit operators are used for computations, while 1-bit operators are usually used for control.

Benchmark	Description
compress	Discrete Cosine Transform
edgedetect	Convolution loop
fft	1024-point Complex FFT
fir	256-tap FIR filter
histogram	Image enhancement using histogram equalization (gray level mapping loop)
iir	4-cascaded IIR biquad filter processing
latnrm	32nd-order Normalized Lattice filter
lmsfir	32-tap LMS adaptive FIR filter
mult	Matrix Multiplication

Table 1: *Benchmark description.*

Operator	Width	Number
read stream	32	15
write stream	32	6
register	32	4
counter	32	1
add	32	3
substract	32	3
multiply	32	4
shift left	32	1
shift right	32	2
mux	32	20
register	1	1
and	1	14
or	1	5
not	1	2
mux	1	31

Table 2: *Operators of the compound circuit.*

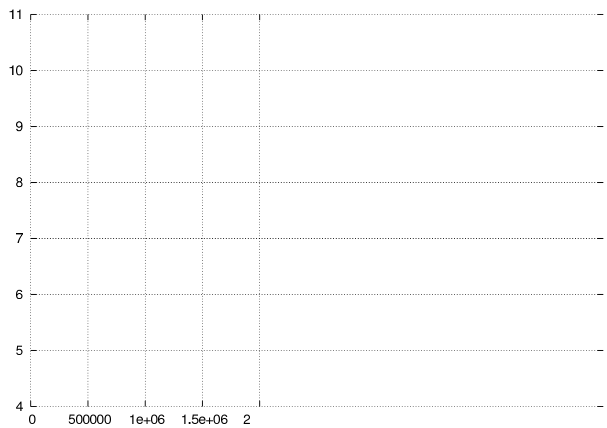
4. PERFORMANCE EVALUATION

We now want to show that it is possible to design a memory interface for our example multi-purpose multi-stream accelerator, using a combination of streams and a table, which achieves high performance at low area and power costs. We first seek the maximal performance that can be achieved with the best possible configuration of the memory interface (stream and table characteristics), independently of cost and power. Then, we minimize area and power by optimizing the different memory interface characteristics without degrading performance.

Maximal performance..

The performance is defined as the average speedup of each individual task over the same task executed on the companion core. While many characteristics come into play (e.g., the ability to simulteanously update or not several streams or entries masks, the arbitration policy for selecting streams which can access the table, etc), we focus on the two characteristics which will most affect execution time, cost and power: the number of stream entries and the number table entries.

The most appropriate number of streams entries is highly correlated to both the latency and the stride of the memory reference mapped to the stream. As mentioned before, words have to be pre-allocated in the streams upon request, and since up to one word can be allocated per cycle, the optimal number of words in a stream depends on the memory latency. The stride further complicates this criterion as not all words within an entry may be useful (e.g., 1 useful word per entry for an 8-word entry and a stride ≥ 8): a stream optimal performance is reached when the number of *useful* words in a stream is greater or equal than the memory request delay. Several issues can further affect the optimal number of streams entries: the task may not always consume one stream word per cycle, or delays incurred by other streams may relieve the pressure on a stream; conversely, a stream may not be able to immediately issue a miss request due to the single memory port (other system issues can naturally have an impact: the variable latency of SDRAM operations, or the presence of an interconnect between the accelerator and the memory, etc).



ber of hits on valid data in Figure 9, shows that the table fulfills a significant role in improving the apparent memory bandwidth.

memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM.

[7] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[8] M.J. Bellido, A.J. Acosta, M. Valencia, A. Barriga, and J.L. Huertas. A simple binary random number generator: new approaches for cmos vlsi. In *Circuits and Systems, 1992., Proceedings of the 35th Midwest Symposium on*, pages 127–129 vol.1, Aug 1992.

[9] Partha Biswas, Nikil D. Dutt, Laura Pozzi, and Paolo Ienne. Introduction of architecturally visible storage in instruction set extensions. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(3):435–446, 2007.

[10] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Over-provisioned multicore processor, September 2009. Patent application, 11/867508.

[11] Nitin Chawla, Roberto Guizzetti, Yan Meroth, Arnaud Deleule, Vishal Gupta, Vinod Kathail, and Pascal Urard and. Multimedia application specific engine design using high level synthesis. In *DesignCon*, 2008.

[12] Nathan Clark et al. OptimoDE: Programmable accelerator engines through retargetable customization, 2004. In *Proc. of Hot Chips 16*.

[13] Nathan Clark, Amir Hormati, and Scott Mahlke. Veal: Virtualized execution accelerator for loops. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 389–400, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Dave Comisky, Sanjive Agarwala, and Charles Fuoco. A scalable high-performance dma architecture for dsp applications. *Computer Design, International Conference on*, 0:414, 2000.

[15] Kevin Fan, Manjunath Kudlur, Ganesh S. Dasika, and Scott A. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*, pages 313–322. IEEE Computer Society, 2009.

[16] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 238–243, New York, NY, USA, 2004. ACM.

[17] IBM. PowerPC 405 CPU Core. September 2006.

[18] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a):364–373, 1990.

[19] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, John D. Owens, and Brian Towles. Exploring the vlsi scalability of stream processors. In *International Conference on High-Performance Computer Architecture, Anaheim, California*, pages 153–164, February 2003.

[20] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. *SIGPLAN Not.*, 31(9):94–104, 1996.

[21] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[22] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 69–80, 2009.

[23] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Sami Yehia, Sylvain Girbal, Hugues Berry, and Olivier Temam. Reconciling specialization and flexibility through compound circuits. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*, pages 277–288. IEEE Computer Society, 2009.