MPSoC Design Using Application-Specific Architecturally Visible Communication

Theo Kluter, Philip Brisk, Edoardo Charbon, and Paolo Ienne

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
{ties.kluter,philip.brisk,edoardo.charbon,paolo.ienne}@epfl.ch

Abstract. This paper advocates the placement of *Architecturally Visible Communication* (AVC) buffers between adjacent cores in MPSoCs to provide high-throughput communication for streaming applications. Producer/consumer relationships map poorly onto cache-based MPSoCs. Instead, we instantiate application specific AVC buffers on top of a distributed consistent and coherent cache-based system with shared main memory to provide the desired functionality. Using JPEG compression as a case study, we show that the use of AVC buffers in conjunction with parallel execution via heterogeneous software pipelining provides a speedup of as much as 4.2x compared to a baseline single processor system, with an increase in estimated memory energy consumption of only 1.6x. Additionally, we describe a method to integrate the AVC buffers into the L1 cache coherence protocol; this allows the runtime system to guarantee memory safety and coherence in situations where the parallelization of the application may be unsafe due to pointers that could not be resolved at compile time.

1 Introduction

The memory and communication architectures proposed for current and next generation multi- and many-core MPSoCs do not match the needs of streaming applications. Streaming applications use the *Synchronous Data Flow (SDF)* model of computation [11], in which coarse-grained communication is modeled as a pipeline; this pipeline, in turn, can be viewed as the concatenation of a set of producer/consumer relationships. The performance of streaming applications highly correlates with the ability of the pipeline to effectively (1) overlap computation with communication and (2) balance the workload across the multitude of cores in the system. The non-determinism imposed by packet-based on-chip networks is detrimental; likewise, buses and crossbars do not support concurrent communication and quickly saturate as the number of cores increases. Furthermore, cache-to-cache communication is a bottleneck, as the concurrent transfer of data from producer to consumer leads to an excess of coherence traffic within the memory system. These sources of overhead suggest that alternative MPSoC interconnect is required to support streaming applications.

The ideal communication architecture for a producer/consumer relationship is a double-buffer placed between the producer and the consumer. The buffer is replicated so that the producer can write to one buffer as the consumer concurrently reads from the other; this enforces the safety property that the producer cannot overwrite the consumer's data before it is read. When the producer and consumer are finished writing and reading respectively, they swap buffers and repeat the process. This communication architecture has already been used in Streamroller [10], a high-level synthesis system for streaming applications; it also bears some principle similarities to the FIFO interface of the Tensilica LX2 processor [18].

This paper advocates the instantiation of application-specific double buffers between adjacent cores in MPSoCs in order to enhance memory system performance for stream programs. We refer to each double buffer as a single *Architecturally Visible Communication (AVC)* buffer. AVC buffers can be viewed as a scratchpad memory that is shared between two cores.

AVC buffers offer several distinct advantages over on-chip networks and bus-based communication schemes: (1) communication is deterministic; network congestion and bus saturation are wholly eliminated; (2) computation and communication are effectively overlapped; compiler techniques to optimize load-balancing across cores already exist; (3) each access to the AVC buffer is cheaper than a cache access: since there is no tag array, and only one way (direct mapped), each access to the AVC buffer consumes less energy and takes fewer cycles than a cache access; and (4) cache coherence traffic for producer/consumer data is eliminated, reducing pressure on the memory subsystem.

At present, automatic parallelization methods [14,19] are not safe for streaming applications written in languages such as C that permit arbitrary pointer arithmetic. Profile-based algorithms are dependent on the dataset used, and therefore may not uncover all data-dependencies. It is conceivable that a pointer that could not be resolved by the compiler may attempt to modify the contents of the AVC buffer. Furthermore, the processor that executes this access may not be the producer or consumer of the data in question. This access cannot execute, as the data has been statically removed from the memory system by the compiler. To ensure program correctness, in the rare event that a undiscovered data-dependence does occur during runtime, a safety engine is invoked to dynamically resolve the coherence problem by removing data from the AVC buffer and reinsert it into the memory system. Although this degrades performance, it may be necessary to ensure the correctness of the program across all possible executions.

The remainder of the paper is organized as follows: Section 2 details the related work in the domain. Section 3 discusses the specific problems that one could encounter in the introduction of AVC buffers inside an MPSoC, and brings effective and efficient solutions to all of them. We prove this in Section 4.2 by addressing a complete application displaying all qualitative situations of interest, and by using the experimental environment described in Section 4.1. Section 5 concludes the paper.

2 Related Work

In the Streamroller high-level synthesis system [10], high-throughput pipelines are synthesized for streaming applications written in C; AVC buffers are placed between adjacent pipeline stages. Unlike in our work, computing elements are dedicated loop accelerators rather than processors. The Tensilica LX2 processor [18] allows the user to instantiate FIFOs between communicating processors, similar in principle to our AVC buffers. Our AVC buffers, in contrast, are more general: the producer (consumer) may write (read) the data from its buffer in any order, i.e., FIFO semantics are not imposed on the communication medium.

One of the challenges is to compile sequential applications written in high-level languages, such as C/C++. Rul et al. [14] developed heavyweight, unsafe profile-based methods to identify communication patterns in sequential programs. This method is unsafe, as some communication patterns that are theoretically possible may not be induced to occur by the input data used to collect the profile. The parallelization operates under the assumption that inter-processor communication is expensive, so the authors attempt to map producers and consumers onto the same target architecture. In this work they introduce homogeneous, and heterogeneous software pipelining, where (1) in homogeneous software pipelining, each processor executes all kernels on a subset of the input dataset in data parallel fashion, and (2) in heterogeneous software pipelining, each processor executes one single kernel, and applies it to the complete input dataset. The study of bzip2 in [14] concludes that homogeneous software pipelining is superior to heterogeneous software pipelining; our results show that the inclusion of AVC buffers leads to the opposite conclusion.

Thies et al. [19] described a semi-automated method to identify coarse-grained pipeline parallelism in programs written in C. Similarly to our work, the authors concluded that heterogeneous pipelining is superior to homogeneous pipelining for streaming applications. Their method, however, requires that the programmer annotate the code with explicit pipeline boundaries. In principle, their method could be made aware of AVC-buffers and it could target a system such as ours. As our system is application-specific, we give the programmer/compiler the freedom, in principle, to chose AVC buffers of the appropriate size.

Streaming languages, such as StreamIt [2] are dedicated to streaming applications; due to their favorable semantics, numerous compiler optimizations have been proposed, many of which are relevant to this work. Sermulins et al. [15], in the context of a compiler for a single processor system, argued in favor of heterogeneous pipelining that has many similarities to ours. Consider a simple pipeline with two stages, S and T. One approach is to use an ordering ordering STSTST... for pipeline stage invocation. If the instruction cache (I-cache) is large enough to hold S or T, but not both, then each invocation would cause a miss in the I-cache. On the other hand, a ordering similar in principle to homogeneous pipelining would yield an invocation order of SS....STT....T, where S is invoked N times followed by N invocations of T. The only I-cache miss would occur when the ordering transitions from S to T. The drawback is that S would produce N times as much data before T could start consuming it; in principle, this could lead to data cache misses. Their compiler, heuristically attempts to find the best invocation ordering to minimize the aggregate overhead due to cache misses. What is important to note is that the homogeneous pipeline organization creates the I-cache problem, as each processor must execute each stage of the pipeline; heterogeneous pipelining, in contrast, does not suffer from this drawback, as just one pipeline stage executes on each processor through the duration of the application.

Several processor architectures for streaming applications have been proposed in recent years. The Imagine [1], Storm-1 [9], and Merrimac supercomputer [3] employ wide SIMD pipelines that are fed by local register files (LRFs); a very wide streaming register file (SRF) is placed between the LRFs and memory, leading to an effective overlap of computation and communication. The SODA architecture for software-defined radio [13] contains a 32-way wide SIMD pipeline, a simpler scalar pipeline, and an address generation pipeline; communication between these pipelines is accomplished through scratchpad memories, with shuffle exchange networks to assist with serial-to-parallel and parallel-to-serial conversion, which occurs at critical points in their application domain. The Raw microprocessor [17] uses an a scalar operand on-chip network to route streaming data between adjacent functional units; delays, which vary based on the relative placement of tasks, are exposed to the compiler. Our MPSoC is most similar to Raw, as the processors themselves are simple 5-stage RISCs, without support for SIMD operations and wide register files; however, our use of AVC buffers is similar to the SRF/LRF and scratchpad memories, but for processor-to-processor communication.

Several compiler optimization methods for streaming programs targeting streaming architectures have been proposed in recent years. Das et al. [4] focused on techniques to effectively map data to the SRF, along with strip mining, loop unrolling and software pipelining. Lin et al. [12] described a hierarchical modulo scheduling algorithm targeting SODA, which used a general solver tool based on *Satisfiability Modulo Theory (SMT)*. These methods are tied to specific target processor architectures and attempt to exploit features that are not present in our system.

Gordon et al. [6] developed several compiler optimizations targeting the Raw microprocessor. Their work focuses on splitting and fusing tasks within the streaming application's intermediate representation, and then mapping them onto different processors in the system. Their abstract model of communication is a FIFO, similar in principle to the Tensilica LX processor feature described above [18]; however, it must be mapped onto the resources of the target machine: in this case, Raw's scalar operand network. In a follow-up paper, Gordon et al. [5] refined their splitting and fusing to account for both task and data parallelism, and introduced a new method for coarse-grained software pipelining; they achieved an average improvement of 1.87x over their previous work. In principle, these methods could easily be adapted to exploit the AVC buffers; the development of automatic parallelization and compilation techniques targeting MPSoCs with AVC buffers is left open for future work.

3 The Importance of Inter-processor Communication

This section begins with the assumption that all data dependencies can be resolved at compile-time, including pointers. We show how the proposed ideas can be generalized for the cases where all data-dependencies cannot be resolved.

3.1 Producer/Consumer Relationships

A producer/consumer relationship can be qualified as the sharing of a data structure between two or multiple kernels/functions in a program. The shared data structure is in its simplest form a scalar, but can also be a multidimensional array. In this work we assume the shared data structures to be only a fraction of the size of the data caches. The

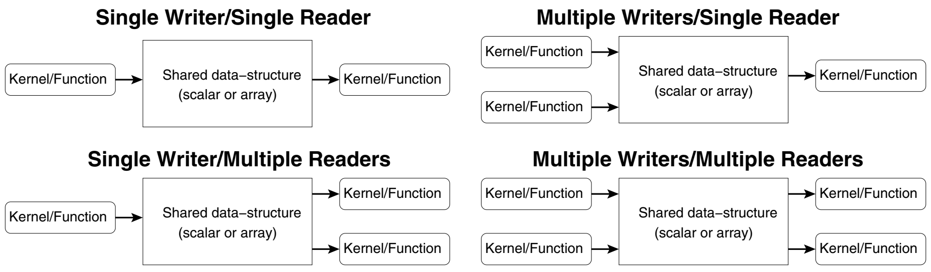

Single Writer/Single Reader
Kernel/Function
Shared data–structure (scalar or array)
Kernel/Function
Multiple Writers/Single Reader
Kernel/Function
Kernel/Function
Shared data–structure (scalar or array)
Kernel/Function
Single Writer/Multiple Readers
Kernel/Function
Shared data–structure (scalar or array)
Kernel/Function
Kernel/Function
Multiple Writers/Multiple Readers
Kernel/Function
Kernel/Function
Shared data–structure (scalar or array)
Kernel/Function
Kernel/Function

Fig. 1. The four different types of producer/consumer relationships

rationale behind this assumption is that (1) excessively large data structures would lead to extensive capacity misses inside the data-cache, thus increasing energy consumption and impeding performance and (2) they would require large buffers (memory), thus potentially increasing the processor's critical path (see Section 3.5), and would consume excessive silicon real estate. Producer/consumer relationships can be divided into four different types as shown in Figure 1. Note, however, that a *producer* may, during its execution, also read from the data structure; a *consumer* may also write to the data structure during its execution. Additionally producer/consumer relationships can be classified based on their access patterns (1) sequential access patterns occur when the elements of the data structure are accessed in increasing/decreasing address order; (2) random access patterns occur when the order of accessing the elements of the data structure is random or cannot be resolved at compile time.

3.2 JPEG Compression Algorithm

For the remainder of the paper we use the JPEG compression algorithm as motivational example and case study. Although JPEG compression is a relatively simple algorithm, it is easy to understand, and representative for streaming applications.

Figure 2 shows the block diagram of the JPEG compression algorithm. The top of Figure 2 shows the four kernels of the JPEG compression algorithm; the bottom shows a schematic data-flow representation of the algorithm. The JPEG compression algorithm contains four major producer/consumer relationships. The first three are between the four kernels, and require a buffer containing 8x8 16-bit values. The fourth one is not explicitly visible in Figure 2, as this relationship is between two consecutive entropy-encoding steps. As the JPEG compression algorithm uses *differential DC-component compression*, the entropy-encoding kernel has a producer/consumer relationship with itself in form of three 8-bit scalars.

The access patterns can be quantified as: (1) sequential writes by the color space conversion and quantization kernels; (2) random reads/writes by the discrete cosine transformation and entropy encoding kernels. All producer/consumer relationships are *single producer/single consumer*, except for the *Discrete Cosine Transformation (DCT)* kernel. As the DCT-kernel performs its operations first column-wise, and then row-wise, it can be viewed internally as a *multiple producer/multiple consumer* relationship.

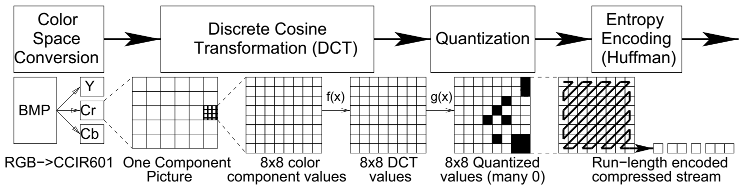

Color Space Conversion
Discrete Cosine Transformation (DCT)
Quantization
Entropy Encoding (Huffman)
BMP
Y
Cr
Cb
f(x)
g(x)
RGB−>CCIR601
One Component Picture
8x8 color component values
8x8 DCT values
8x8 Quantized values (many 0)
Run−length encoded compressed stream

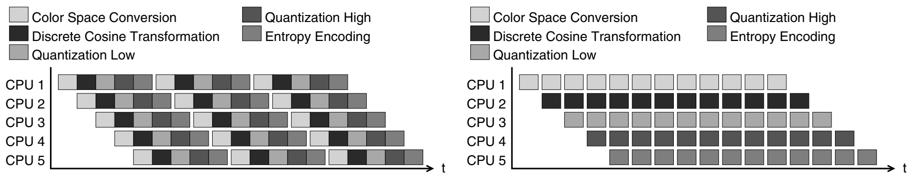
Fig. 2. The Block diagram of the JPEG compression algorithm as motivational example

3.3 Parallelization

In this work we apply synchronous software pipelining with static scheduling to perform parallelization. The synchronous nature of the pipelining is achieved by applying hardware-barrier synchronization between the kernels of the program. A *barrier* can be seen as a global *clock* to the system. Each processor can proceed from the barrier if and only if all other processors have entered it. Contrary to synchronous software pipelining, asynchronous software pipelining synchronizes by *completion detection* in the form of locks/semaphores, and is beyond the scope of this work.

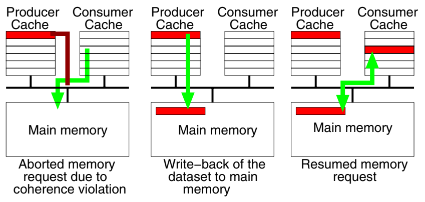
In software pipelining there are basically two styles: homogeneous, and heterogeneous pipelining. Figure 3 shows the two software pipelining styles applied to the JPEG compression algorithm. Homogeneous software pipelining keeps the producer/consumer relationships within the data cache of each processor; each processor, however, must store the complete program in its instruction cache. Heterogeneous software pipelining, distributes the program code across the instruction caches of the different processors without replication; moreover, it exposes the producer/consumer relationship to the memory subsystem. The trade-off of which software pipelining style to use therefore lies in the overhead induced by capacity misses in the cache versus producer/consumer relationships exposed to the memory subsystem. As the cost of exposing producer/consumer relationships is higher than the capacity-miss overhead (as discussed in Section 3.5), most parallelization algorithms tend to prefer homogeneous over heterogeneous software pipelining.

In our system, parallelization and the scheduling of kernels on processors is performed statically at compile time; each kernel will have a direct connection to the appropriate AVC buffer(s).

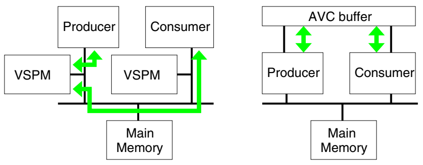


(a) Homogeneous software-pipeline. (b) Heterogeneous software-pipeline.

Fig. 3. The two software pipelining styles applied to the JPEG compression algorithms


Producer Cache
Consumer Cache
Main memory
Aborted memory request due to coherence violation
Write–back of the dataset to main memory
Resumed memory request

(a) The different stages in a memory-subsystem exposed producer/consumer relationship in presence of a coherence protocol.



(b) The Virtual Scratchpad Memory and AVC buffer solution to memory-subsystem exposed producer/consumer relationships.

Fig. 4. The exposure of producer/consumer relationships in three different memory subsystems

3.4 Coherence and Consistence

The introduction of distributed memory elements in a shared-memory system leads to the situation where multiple-copies of the same data are dispersed throughout the system. If these copies are read-only there is no problem; however if one of these copies is assigned a new value then potential coherence and consistence violations may occur.

The sequential consistence model states that all read and write operations are observed as atomic, and in program order. Our programming paradigm is based on this consistence model. The consistence model is primarily enforced in software, but may also be hardware-assisted. In this work we assume software-enforced consistence.

The coherence rule is less strict than the consistence rule, as it only requires all write operations to be seen as atomic operations. In other words, a read operation should always see the latest written value to the shared-memory, regardless of where the write occurs. Coherence, in general, is enforced by the compiler in scratchpad memory-based systems and by hardware in cache-based systems.

3.5 AVC-Buffers, Caches, and ScratchPad Memories

In a single processor system, producer/consumer relationships are generally hidden from the memory subsystem. Although the shared data structures are allocated in main memory, they will almost never leave the data-cache due to (1) explicit locking of the cache-lines in which the shared data structures reside [7], or (2) implicit locking of the cache-lines in which the shared data structures reside by exploiting the temporal locality of the data structure by using, for example, a *Least Recently Used* replacement policy in the cache. In the rare case, when no inter-processor communication exists (e.g., completely data-parallel algorithms), the same technique can be applied to cache-based MPSoCs. Streaming applications, however, require inter-processor communication, and a coherence problem arises (see Section 3.4). An example of inter-processor communication is an exposed producer/consumer relationship between two processors. To insulate the programmer/compiler from the coherence problem, most MPSoCs are provided with a hardware coherence protocol. Most prominent hardware coherence protocols are snoopy protocols with three or more states. The simplest snoopy protocol is the *Modified, Shared, and Invalid (MSI)* states based protocol. More sophisticated protocols utilize *MESI* or *MOESI* states. It is beyond the scope of this work to describe the details of these protocols; however it is important to note that they severely impact on the overhead induced due to exposed producer/consumer relationships. In this work we utilize the most prominent *MESI* states-based hardware coherence protocol.

To understand the cost involved when a producer/consumer relationship is exposed to the memory subsystem, we consider a *single producer/single consumer* relationship as shown in Figure 1. The different stages involved in the coherence protocol are shown in Figure 4(a). The communication starts by the consumer cache making a request (read or write) to the memory subsystem for a shared data structure—as shown to the left in Figure 4(a). The consumer's cache will request the data structure, which is assumed to exist in a modified state in the producer's cache; thus, the data structure is invalid in main memory. Next the producer's cache will abort the consumer's request, as it holds the latest copy. As a reaction to this request/abort action, the producer's cache will write back the shared data structure to main memory as shown in the middle of Figure 4(a). Finally the consumer's cache will resume the memory request and copies the "correct" data structure to itself from main memory, and execution can continue—shown to the right of Figure 4(a). During the write-back stage, the producer is likely to stall (impeding the producer's performance), whilst the consumer is likely to stall during the whole transfer. Furthermore, extra energy is consumed, and bus bandwidth is expended by the write-back/read action to and from main memory. An improved version of on this scheme is to merge the write-back stage, shown in the middle of Figure 4(a), with the resumed memory request phase —shown to the right of Figure 4(a). This optimization gives the consumer the possibility to "read" the values of the data structure during the write-back phase. This optimization will be referred to as a *cache-to-cache copy*.

The scratchpad memory approach is different than that of a cache-base system, as it provides coherence at compile-time. In *Virtual Shared ScratchPad Memory (VSPM)* systems, each processor can access each of the scratchpads at different costs. To deal with shared data structures in conjunction with the coherence problem, the compiler places the shared data structure solely in one scratchpad memory. Coherence is guaranteed, and less energy is consumed, as the data structure is not copied to main memory. Arguably, the impact on performance, in comparison to a cache-based system is equal, or even higher, as each access to a remote scratchpad memory is transmitted on the bus; direct access to a cache is much faster—shown in the left of Figure 4(b). Also the impact of bus bandwidth is at least as large as the cache-based system, as remote scratchpad accesses require bus transactions.

AVC buffers, as shown on the right of Figure 4(b), completely remove the producer/consumer traffic from the memory subsystem. AVC buffers benefit both from this removal, and from the fact that the buffers are moved forward in the processor pipeline using the *Instruction Set Extension (ISE)* interface of the processor. This has a significant impact on the organization of the pipeline, as AVC buffer accesses occur during the execute stage of the pipeline, rather than the write-back stage. This ensures that the AVC buffer load/store operations take a single cycle and are atomic. If the cache access takes multiple cycles (3 cycles for a hit, in our system), then the AVC buffer must spend an extra 3 cycles before it commits. If we have a store to the cache followed by an AVC buffer store, the AVC buffer store would commit and retire itself before the cache store commits and retires, violating consistence. Thus, number of cycles required to access the AVC would need to be the same as the number required to access the cache. This consistence issue is wholly avoided by placing the AVC buffer at the execute stage of the pipeline; although the AVC buffer store occurs before the cache store finishes, the instructions are retired in-order, guaranteeing consistence.

Moving the AVC buffer load/store to the execute stage of the pipeline makes these operations single-cycle and atomic. The disadvantage is that the delay of the AVC-buffer might impact the processor's critical path if the buffer size is large (i.e., the memory read access time exceeds the processor's critical path delay). The AVC buffer does not impact the bus bandwidth, which increases performance, and reduces energy consumption per access compared to a cache and less to equal energy consumption per access compared to scratchpad memories.

3.6 Execution Safety

The discussion in Sections 3.1–3.5 assumed that all the data-dependencies of the shared data structures could be resolved at compile time or by the programmer. In this scenario, we can completely remove the data structure from the memory subsystem and move it into AVC-buffers; however, if we cannot resolve these data dependencies, we must ensure that the correct execution of the program is not jeopardized (we henceforth refer to the correct execution as *safe*).

In a scratchpad-based system all data structures with unresolved data-dependencies cannot be safely allocated to the scratchpad memory. When applying AVC-buffers in a scratchpad based system, we must take a similar approach, as otherwise coherence cannot be guaranteed and safety is jeopardized.

In cache-based systems, unlike scratchpad-based systems, coherence is dynamically enforced by the hardware coherence protocol. As our system contains caches as well as AVC buffers, we can use the hardware coherence protocol to allow data structures with unresolved data-dependencies to be candidates for AVC-buffer allocation. We will refer to these data structures with unresolved data-dependencies as *unsafe structures*. To guarantee the safety of unsafe structures, a three stage approach is taken. First, we allocate unsafe structures to both the AVC-buffers and main memory. Secondly, we transform the AVC-buffer into a coherence protocol-enhanced mini-cache. Finally, to prevent performance losses due to *false sharing*, we cache-block align all data structures.

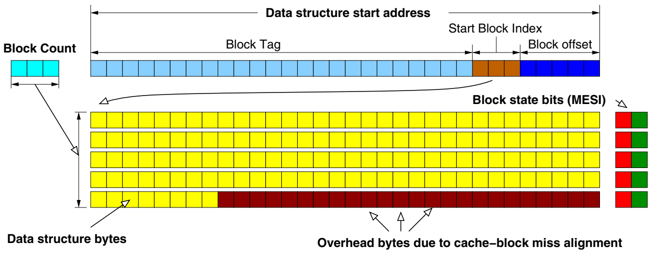


Fig. 5. The AVC buffer converted to a micro-cache

The impact of the redundant allocation of data in both AVC-buffers and memory, is minimal; we lose the advantage of freeing up main memory space by removing the unsafe-structures from it; however, in all other systems this double allocation occurs implicitly: the structure resides in both cache/scratchpad and memory.

The impact of the second action is more severe. The rationale behind the transformation is that in most cases the unsafe structure resides in the AVC-buffer in either the *exclusive* or *modified* state. In the rare case that it is requested by the memory sub-system it has to be reinserted into the memory sub-system by use of the coherence mechanism similar to coherent caches. As the reinsertion into the memory sub-system is rare, we can use a micro-cache architecture similar to a cache with segmented blocks. Our micro-cache, however, only contains one segmented cache line, one tag, and one pair of state-bits per segment (see Figure 5). Each segment of our cache line is the size of one coherence-unit of the applied data caches (i.e., the size of a block inside the data cache). The overhead of this approach is threefold: First, we introduce extra storage in form of a tag, block count, and block-state bits. Second, we must implement a complete cache controller with coherence protocol. Third, we can have overhead due to the fact that the unsafe structures size in the AVC buffer is not a multiple of the size of the coherence unit.

When the compiler allocates memory for all data structures, it may decide to place safe and unsafe data structures continuously in such a manner that the boundary between the data structures is not aligned with the boundaries of the coherence units. This possibly creates situations where the *overhead bytes*, as depicted in Figure 5, are occupied by other safe or unsafe data structures, in which there is a high chance of *false sharing*. *False sharing* is a well-known effect in which the safe data structure invalidates the unsafe structure, or vice versa; as both structures are independent this coherence traffic is redundant, reduces the AVC-buffer performance, and therefore should be avoided. *False sharing* can be avoided by coherence unit aligning all unsafe structures, which can be done automatically. The overhead bytes (as shown in Figure 5) do not require memory elements in the AVC buffers due to cache-block alignment.

It should be noted that this coherent issue does not occur when programs are written using streaming languages, such as StreamIt [2]. These languages do not support arbitrary pointer arithmetic, and as a result, safety violations of this kind cannot occur. Thus, this safety mechanism is only required if the application is written in an inherently unsafe language, such as C.

4 Experiments

4.1 Experimental Setup

We implemented our AVC buffers by augmenting an OpenRISC-compatible platform running on FPGA. The AVC buffers are coupled to the processors utilizing their *Instruction Set Extension (ISE)* interface, where the extended instructions are solely AVC-buffer load/store instructions. Furthermore, for the multi-processor case, we augmented the architecture with a hardware barrier.

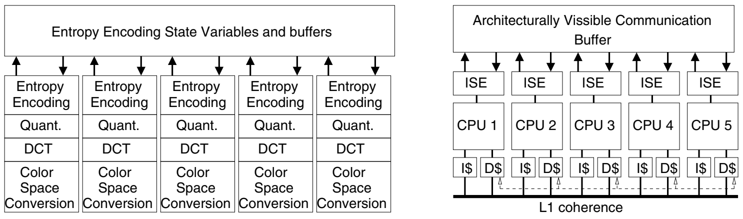
We parallelized the *cjpeg* program from the *EEMBC* denbench suite [8]. The code has been parallelized by hand, while keeping in mind that automatic parallelization algorithms as presented in Section 2 may be able to perform the job as well. Care has been taken to avoid *false sharing* by aligning all data structures on cache line boundaries. The parallelized versions of the JPEG compression are statically mapped onto the 5 processor system. Finally, the complete code-base has been cross-compiled using a "newlib"-based gcc 3.4.4 tool-chain for the OpenRISC.

For all the experiments we used the same 24-bit RGB encoded picture of 1024x768 pixels, similar to the resolution of current high-end web-cams and standard portable phones. For the energy consumption calculations we used CACTI [16] to determine the read/write energy-consumption for different cache configurations in a 90 nm technology The external memory and bus-access read/write energy consumption is estimated to be $792pJ$ per access. The energy values reported here only include the dynamic energy consumed in the memory sub-system; this model does do not include processor and leakage energy.

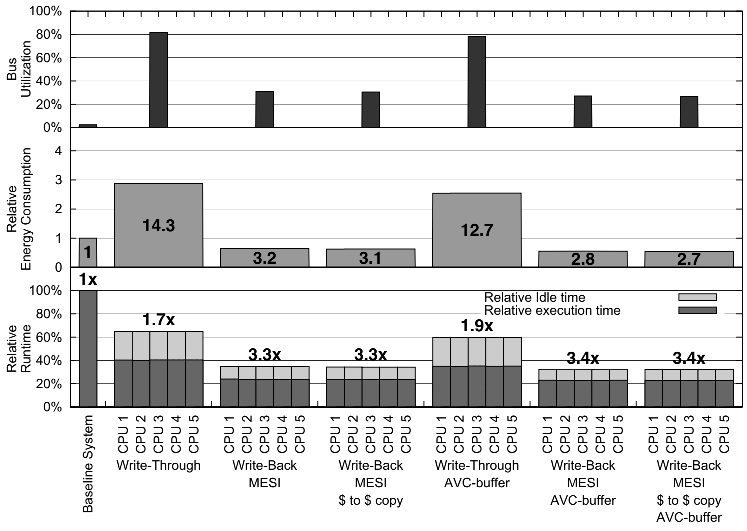
4.2 Experimental Results

To enable a fair comparison, we performed a performance-energy exploration of a single processor-based system running the JPEG compression algorithm. Our baseline architecture was the one that performed best with the minimal energy consumption; it used a 4 kB direct mapped instruction cache, and an 8 kB 2-way set-associative data cache. Next we analyzed the runtime of the different kernels of the JPEG compression algorithm on the baseline architecture. This runtime breakdown guided the creation of a heterogeneous software pipelined five processor architecture shown in Figure 6(a), and a homogeneous version shown in Figure 7(a). Each of the processors of the two systems uses a 4 kB direct mapped instruction cache, and an 8 kB 2-way set-associative data cache with Level 1 MESI-states hardware coherence protocol.

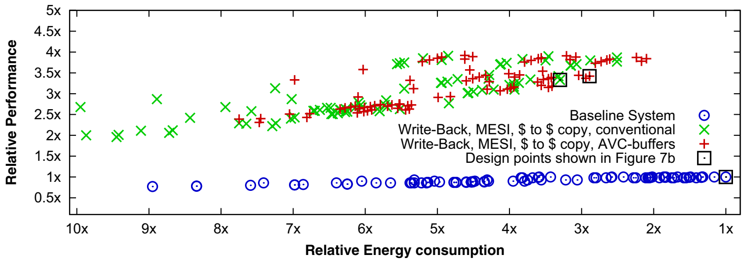
We first accelerate the application by employing a write-through coherence policy, which does not require a hardware coherence protocol implementation. As to be expected, on both the homogeneous and heterogeneous versions the bus is completely saturated—as shown in Figure 6(b) and Figure 7(b), limiting the speedup to a factor of 1.7x compared to the baseline system. Next we accelerate the application on a five-core system that employs a write-back policy with the MESI protocol for cache coherence and snoopy cache-to-cache copies. When homogeneous pipelining is used, this system offers a speedup of 3.3x; the speedup achieved by the same system with AVC buffers is 3.4x compared to the baseline, a meager performance increase. Furthermore, as the inter-processor communication is only three 8-bit scalars (one cache-block), the influence of the cache-to-cache copy enhancement is minimal. When heterogeneous pipelining is used, on the other hand, the speedup of the five core system is 3.2x compared to the baseline, and increases to 4.2x through the addition of AVC buffers. Also the influence of the inter-processor communication is clearly visible. As in the heterogeneous case, the size of the communicated data is three arrays of sixty-four 16-bit values (four cache-blocks); the speedup of the system employing cache-to-cache copies is 3.2x, compared to a 3.0x speedup for the system without cache-to-cache copy; both systems enhanced with AVC-buffers show an equal speedup of 4.2x due to the removal of the inter-processor communication.



(a) Architecture, kernel mapping, AVC-buffer allocation, and MPSoC architecture.



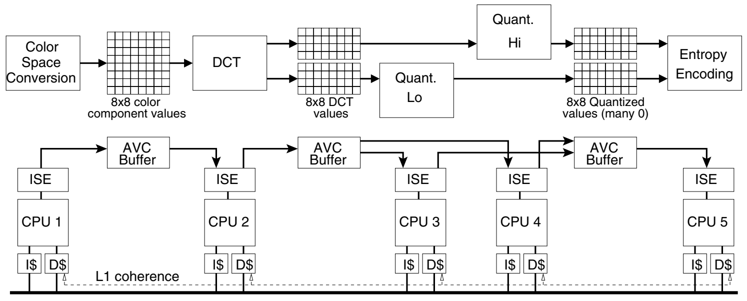
(b) Bus utilization, energy consumption and runtime of the different architectures compared to the baseline system.



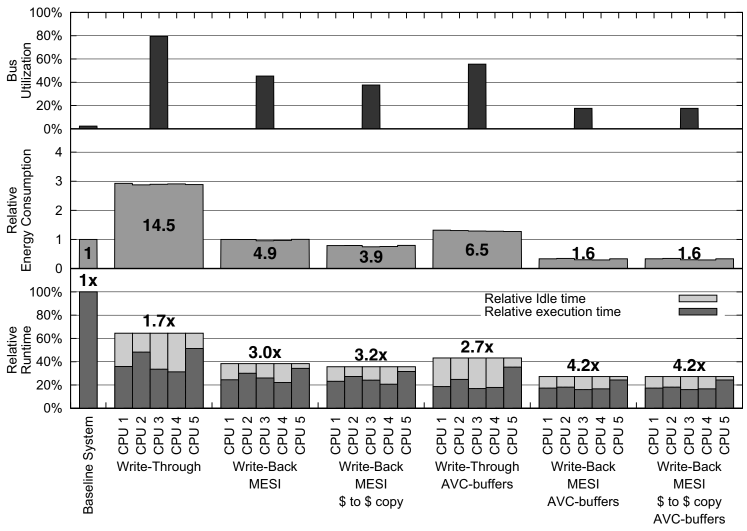
(c) Energy-performance exploration utilizing 2kB, 4kB, 8kB, direct-mapped, 2 way, and 4 way set-associative instruction and data caches.

Fig. 6. Homogeneous Software pipelining of the JPEG compression algorithm on a 5 processor MPSoC

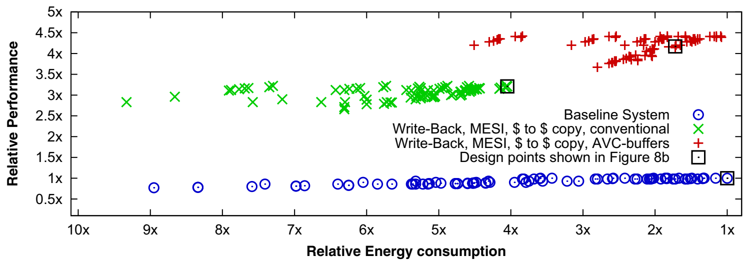
The use of AVC buffers in conjunction with heterogeneous pipelining is also beneficial in terms of its ability to reduce the energy consumption of the memory subsystem. Homogeneous pipelining without AVC buffers increased the memory subsystem energy consumption by 3.1x compared to the baseline single processor system; the inclusion



(a) Architecture, kernel mapping, AVC-buffer allocation, and MPSoC architecture.



(b) Bus utilization, energy consumption and runtime of the different architectures compared to the baseline system.



(c) Energy-performance exploration utilizing 2kB, 4kB, 8kB, direct-mapped, 2 way, and 4 way set-associative instruction and data caches.

Fig. 7. Heterogeneous Software pipelining of the JPEG compression algorithm on a 5 processor MPSoC

of AVC buffers reduced it to 2.7x. Heterogeneous pipelining without AVC buffers increased the memory subsystem energy consumption by 3.9x; however the inclusion of AVC buffers reduced it to 1.6x, which is quite low for a five core system.

Finally to ensure that the presented results are not biased by a poor choice of caches, we performed an exhaustive energy-performance exploration for the cache-to-cache copy enhanced write-back MPSoCs with and without AVC-buffer extension. The results of this exploration is shown in Figure 6(c) and Figure 7(c). Both figures show that: (1) the results are consistent for all cache configurations; (2) the homogeneous software pipelining clearly suffers cache pressure as described in Section 3.3; (3) the heterogeneous software pipelining, originally suffering memory-subsystem pressure due to inter-processor communication, performs consistently better by adding AVC-buffers. Thus we conclude that heterogeneous pipelining with AVC buffers is the best communication architecture for our five core MPSoC implementation of the JPEG compression.

5 Conclusion

This paper discusses a case study using JPEG compression that motivates the use of Architecturally Visible Communication buffers to accelerate producer/consumer communication in MPSoCs for streaming applications. Previous work on automated parallelization has favored homogeneous over heterogeneous software pipelining due to the high cost of core-to-core communication via the memory system. Because of this high communication cost, the most efficient pipelining method mapped producers and consumers of the same data onto the same core; however this approach does not effectively overlap computation and communication, which is of great importance when accelerating streaming applications. Our results show that the inclusion of Architecturally Visible Communication buffers yields the opposite result: providing a fast core-to-core communication mechanism favors heterogeneous over homogeneous software pipelining; these results were consistent and robust over a wide variety of cache configurations.

Automated techniques to parallelize applications written in C are unsafe, as pointer resolution is undecidable in the general case. For our work, the implication of the lack of safety is that data structures that have been removed from the memory subsystem and placed into Architecturally Visible Communication buffers can, theoretically, be accessed by a pointer. Our safety mechanism transforms the Architecturally Visible Communication buffer into a small cache that is connected to the coherence protocol; when an extraneous pointer accesses data in the Architecturally Visible Communication buffer, it is moved back into the memory system; although this implies some performance overhead, the occurrence of such pointer accesses is rare, thus mitigating its impact on the performance of the system, while ensuring correctness and safety.

References

1. Ahn, J.H., et al.: Evaluating the imagine stream architecture. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, Munich, Germany, pp. 14–25 (2004)
2. Amarasinghe, S., et al.: Language and compiler design for streaming applications. International Journal of Parallel Programming 33, 261–278 (2005)

3. Dally, W.J., et al.: Merrimac: Supercomputing with streams. In: Proceedings of the Fifteenth International Conference on Supercomputing, Phoenix, Arizona, pp. 35–42 (November 2003)
4. Das, A., Dally, W.J., Mattson, P.: Compiling for stream processing. In: Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques, Seattle, Washington, pp. 33–42 (September 2006)
5. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, pp. 151–162 (October 2006)
6. Gordon, M.I., et al.: A stream compiler for communication-exposed architectures. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, pp. 291–303 (October 2002)
7. Gummaraju, J., Rosenblum, M.: Stream programming on general-purpose processors. In: Proceedings of the 38th Annual International Symposium on Microarchitecture, Barcelona, Spain, pp. 343–354 (November 2005)
8. Halfhill, T.R.: EEMBC releases first benchmarks. Microprocessor Report (May 1, 2000)
9. Khailany, B.K., et al.: A programmable 512 gops stream processor for signal, image, and video processing, vol. 43, pp. 202–213. IEEE, Los Alamitos (2008)
10. Kudlur, M., Fan, K., Mahlke, S.: Streamroller: Automatic synthesis of prescribed throughput accelerator pipelines. In: Proceedings of the 14th International Conference CODES-ISSS, Seoul, Korea, pp. 270–275 (October 2006)
11. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. 36(1), 24–35 (1987)
12. Lin, Y., et al.: Hierarchical coarse-grained stream compilation for software defined radio. In: Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, Salzberg, Austria, pp. 115–124 (September 2007)
13. Lin, Y., et al.: Soda: A low-power architecture for software-defined radio. In: Proceedings of the 33nd Annual International Symposium on Computer Architecture, Boston, Massachusetts, pp. 89–101 (June 2006)
14. Rul, S., Vandierendonck, H., de Bosschere, K.: Detecting the existence of coarse-grain parallelism in general-purpose programs. In: Proceedings of the 1st Workshop on Programmability Issues for Multi-Core Computers, Goteborg, Sweden (January 2008)
15. Sermulins, J., et al.: Cache aware optimization of stream programs. In: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, Chicago, Illinois, pp. 115–126 (June 2005)
16. Tarjan, D., Thoziyoor, S., Jouppi, N.P.: CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Development Company, Palo Alto, Calif. (June 2006)
17. Taylor, M.B., et al.: Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, Munich, Germany, pp. 2–13 (June 2004)
18. Tensilica. Xtensa LX2: Product Brief (April 2007)
19. Thies, W., Chandrasekhar, V., Amarasinghe, S.: A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In: Proceedings of the 40th Annual International Symposium on Microarchitecture, Chicago, Illinois, pp. 356–359 (December 2007)