A Video Compression Case Study on a Reconfigurable VLIW Architecture

Davide Rizzo Osvaldo Colavin
AST San Diego Lab, STMicroelectronics, Inc.
{davide.rizzo, osvaldo.colavin} @st.com

Abstract

In this paper, we investigate the benefits of a flexible, application-specific instruction set by adding a run-time Reconfigurable Functional Unit (RFU) to a VLIW processor. Preliminary results on the motion estimation stage in an MPEG4 video encoder are presented. With the RFU modeled at functional level and under realistic assumptions on execution latency, technology scaling and reconfiguration penalty, we explore different RFU instructions at fine-grain (instruction-level) and coarse-grain (loop-level) granularity to speedup the application execution. The memory bandwidth bottleneck, typical for streaming applications, is alleviated through the combined adoption of custom prefetch pattern instructions and an extent of local memory. Performance evaluations indicate up to 8x improvement, with loop-level optimizations is achieved under various architectural assumptions.

1. Introduction

In the last few years, Reconfigurable Computing [1][2] has been investigated as a way to bring the computational density of spatial computing (hardwired operators) to programmable platforms and the versatility and the abstraction of the latter to the former. Architecturally, this is usually achieved through hybrid systems by coupling a Reconfigurable Unit to a CPU or DSP. The reconfigurable unit microarchitecture varies from an FPGA-like sea of bit-level reconfigurable logic elements to an array of fixed size ALUs where only the interconnect is reconfigurable (Rapid [3]).

Streaming applications, with their high computation and bandwidth requirements, are typically used as benchmarks because they stand to benefit the most from this reconfigurable computing.

Previous work shows that approaches which map entire kernels to the reconfigurable hardware (GARP [4], Morphosys [5]) achieve significant speedups (5-10x) while approaches which simply extend an instruction set (Chimaera [6]) see only a marginal improvement (1-2x), although this conclusion is more qualitative than quantitative since to our knowledge no comparison of the two options was done on the same platform.

Another conclusion is that tightly coupled architectures are more amenable to a single abstract programming paradigm (usually C) than loosely coupled ones, which typically expose to the programmer unwieldy low-level hardware features and even sometimes two programming paradigms (e.g. C and RTL).

The work presented in this paper was undertaken with the idea of exploring various architectural options on the same platform, and testing the practical limits of reconfigurable computing.

The benchmark application is an MPEG4 video encoder [7] with a particular focus on the motion estimation kernel which happens to be where the application spends most time in the reference code.

The simulation platform is an internal configurable VLIW tool set (compiler and simulator) [8] to which we added a reconfigurable execution unit. We did not consider a loosely coupled reconfigurable co-processor in our study, because we decided to limit ourselves to architectures which lend themselves to a generic programming abstraction, with the idea that compilers are capable to generate code for these targets as demonstrated by [9][10][11]. On this platform, we accelerated the motion estimation kernel first by extending the instruction set with application specific operators, then by mapping increasing portions of the kernel to the reconfigurable unit, up to the entire kernel. We made various hypothesis on latency and memory bandwidth to evaluate the effects of these parameters. Because we focus on architectural trade-offs, we make only general assumptions on the architectural details which do not influence the outcome of the experiments. For example, we do not describe the exact fabric of the reconfigurable unit, rather we only characterize it by its functionality, throughput and latency, and we assume that it can implement relatively arbitrary portions of code of the original application.

The main contribution of this paper is the exploration of a relatively large architectural space on the same platform and the quantitative comparison of various architectural trade-offs.

The rest of the paper starts with a description of the reference architecture, methodology, tools and benchmark

used for this work. The results of the experiments are then presented and analyzed and finally some preliminary conclusions are drawn.

2. Architectural model

Our Reconfigurable VLIW architecture, shown in Figure 1, is composed of a VLIW core processor tightly coupled with a Reconfigurable Functional Unit (RFU). The processor is the *ST200* (also known as *Lx* [8]), a clustered VLIW core, jointly developed by Hewlett-Packard Labs and STMicroelectronics. Its scalability and customizability reflect in the multi-cluster organization of the resources, supported by a toolchain based on an aggressive ILP compiler technology. The ST200 architecture consists in its simplest form of one cluster with a 4-issue VLIW datapath (4 32-bit integer ALUs, 2 16x32 multipliers, 1 Load/Store Unit and one Branch Unit). The cluster includes 64 32-bit General Purpose registers and 8 1-bit branch registers (to store branch conditions, predicates and carries). We considered 128KB direct mapped Instruction cache and 32KB 4-way Data cache (with an 8-entry Prefetch Buffer). The SIMD computing model is supported through sub-word parallelism with instructions working on four 8-bits operands or two 16-bits operands. At this preliminary stage the RFU is modeled at functional level, without details on the underlying microarchitecture. We assume that the RFU has available, some form of local memory, which can have a regular structure or being distributed along the computational resources to act as a local distributed register file. We furthermore assume that architectural assists are available to tackle the reconfiguration penalty problem (i.e. multicontext configuration memory, configuration cache [12][14][15]).

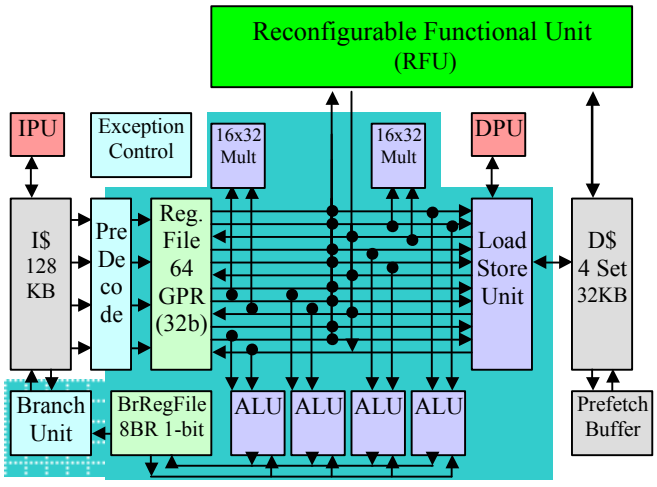

Figure 1: the modified ST200 1-cluster architecture with the support of a Reconfigurable Functional Unit (RFU).

3. Methodology and Programming Model

We compile the whole application using the ST200 compiler (based on Multiflow[8]) and profile the performances on the ST200 compiled simulator, which also embeds I and D cache models. An inspection of the code representing the bottlenecks then drives the manual selection of the RFU instructions. The RFU is modeled at functional level, with no detailed microarchitecture or hardware mapping considered at the moment. At this early stage, the major hypothesis needed to get meaningful performance assessment regards the compiler-visible static latency of the RFU instructions. The flexibility of the ST200 toolchain allows to extend the instruction set to define new custom instructions (with up to eight input and one output operands), through which the RFU is simulated. Furthermore the simulator libraries have been extended with data structures to model the RFU state. The RFU is then programmed by manually inserting the custom instructions in the C-code. As a general rule, to avoid constraints on the number of operands needed, we usually adopt a mixed approach with explicit and "implicit" operands, with the latter specified in the instruction configuration. In such a case, "custom load" instructions are defined and used to load the operands in the RFU, relatively to a specific configuration. The generic RFU operational behavior relies on three steps (*#x* stands for configuration *x*):

a. *Initialization* for *#x* — *RFUINIT(#x)*
b. *Load* data for *#x* — *RFUSEND(#x, op1,…,opN)*
c. *Execute #x*, *Write* destination register — *Dest = RFUEXEC(#x, op1,…,opN)*

We assume the RFU to have no reconfiguration penalty. This gives an upper-bound performance assessment, which should be approached in real cases, by developing "smart" reconfiguration strategies, based on configuration prefetch and management, to hide the reconfiguration penalties [12][14][15]. We furthermore assume to generate RFU instruction configurations which fit the RFU size and available resources. The preliminary way we propose to use the reconfigurable unit resembles the methodology for embedded systems, where the requirements usually force the designer to adopt mixed solutions (with coprocessors or dedicated accelerators) or completely dedicated (ASIC). The programming model is much closer to a software approach, with the advantages of flexibility and fast programmability, allowing to speedup the architecture exploration and the application partitioning between conventional resources and the Reconfigurable Unit. We can envision the RFU as an added set of tightly coupled dedicated accelerators in the CPU architecture, whose availability in time depends on the RFU size and on the usage pattern shown by the

application. Different architectural solutions can be explored with as many dedicated components as needed, in a highly extensible, customizable architecture. The result of the exploration is a partitioned application between the conventional processor and the dynamically reconfigurable portion added to the datapath. This work focuses on experiments related to an early architectural exploration stage and it represents a preliminary step to define a platform architecture suitable to meet the performance requirements of domain-specific applications.

4. Case study: Motion Estimation algorithm

The application chosen is the MPEG4 video encoding [7], simple profile, on a QCIF (Foreman) sequence of 25 frames, with constant quality factor (fixed quantization parameter *Q=10*). The reference C code (in the following referred as "*ORIG*") has been optimized to get advantage of a basic SIMD subset of instructions. The frames are allocated, aligning on 32 bytes boundaries. The Instruction Cache size (128KB) holds completely the application code and makes negligible the related stalls. The SIMD model and the code optimizations represent a reasonable assumption to guarantee confidence to the results achieved, preventing us from ending-up with unrealistic speedups and conclusions. The initial profile shows a *25.6%*, of the execution time spent in the *GetSad()* hot spot. This routine is part of the motion estimation (ME) stage and it computes the Sum of Absolute Differences (SAD) between a given reference macroblock (16x16 pixels) and a candidate predictor macroblock (up to 17x17 pixels). A sub-task implemented is the predictor macroblock interpolation in case this latter is specified by a sub-pixels motion vector. Given the SIMD model, each 8-bit pixel is accessed through the 32-bit word in which it has been packed, but the computation could require sets of pixels that move across consecutive packed words. Therefore the alignment problem has to be addressed. Figure 2 shows an example of the data set required with an alignment 3 and diagonal interpolation. Listing 1 shows a pseudo-code of the operations performed. We choose the ME kernel loop as the case study of our experiments.

```
SAD_value=0;
Read the first predictor macroblock row
For each macroblock row {
  Read the 5 32-bit words (20 pixels) in the
      predictor macroblock row
  Align the required 17 pixels, if needed
  Interpolate the 16 pixels, if sub-pixels
      motion vector components are specified
  Read the 16 pixels from the reference
      macroblock row
  Compute the SAD and accumulate to
      SAD_value
}
Return SAD_value
```

Listing 1: **the *GetSad*() pseudo-code.**

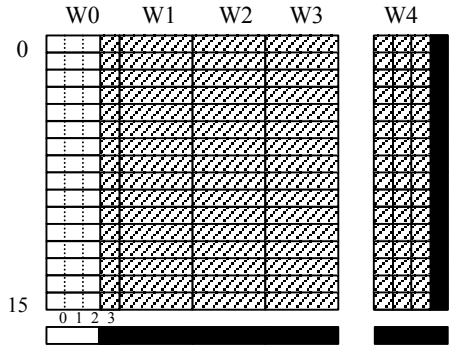

Figure 2: **the data set for a predictor macroblock with alignment 3 (shaded) and for the diagonal interpolation (black); each rectangle represents a 32-bit word, containing 4 packed 8-bit pixels.**

5. RFU exploration and Experimental results

We carried out a set of experiments to explore RFU instructions with increasing granularity from instruction-level to loop-level (from simple 1-cycle 3-operands custom instructions to a long latency instruction implementing the entire ME kernel loop). An RFU unit having the capability to autonomously access the memory subsystem is a further distinction between the loop-level and the instruction-level experiments. In the following we will detail the different scenarios explored. The relative speedups and stall cycles reductions presented in the tables are always relative to the optimized C-code version (*orig*). The purpose of these experiments is to evaluate how the RFU can be exploited to speedup the execution, with a quantitative comparison of various alternatives on the same platform. We expect the results to provide insights on the "best" granularity of the RFU instructions, to drive the investigation on other parts of the application or even for similar applications, targeting the highest speedup.

a) Instruction-level optimizations

We modified the diagonal interpolation exploring three types of new RFU instructions with increasing complexity and data-parallelism. In the first case (A1) we reformulated the computation to use a different set of 1-

cycle latency SIMD instructions than the one already exploited in the reference code; without entering too much in details, we simply rewrote the diagonal interpolation to get advantage of intermediate horizontal and vertical interpolations with some extra rounding adjustments on the result. This scenario is a perfect example of how the RFU can be used to extend the ISA of the processor. The current reference processor already supports a set of SIMD instructions, but the detailed analysis of the case study highlights the need for similar (but less generic) missing instructions, which specificity prevents them to be economically considered in the basic supported ISA. We assume to be able to issue up to 4 instructions per cycle. In the second case (A2), we defined an instruction to implement the diagonal interpolation over a set of 4 pixels. We assume that a small set of registers in the RFU is loaded with the needed data set (4 32-bitwords), prior to execute the diagonal interpolation. In the last case (A3) we further increment the exploited data-parallelism, computing the diagonal interpolation over a set of 16 pixels from a macroblock row. A load phase of the required data set (10 32-bit words) precedes the instruction execution. We considered a single cycle latency also for the latter two cases.

Results and Analysis. Table 1 shows the results of these experiments. The new SIMD instructions (case A1) grant a 14% of improvement, while a dedicated diagonal interpolation instruction sorts out higher speedups (28% and 31% improvement respectively for A2 and A3). We observe that these results are strongly data-dependent and, as Amdahl's law states, the percentages would be higher on a sequence requiring to diagonally interpolate every predictor macroblock and zero, if no interpolation is required at all. In the chosen test sequence, the diagonal interpolation is executed only in 18% of the cases. The magnitudes of the speedups are someway expected and in line with similar approaches [6].

	CYCLES	S.Up	%Improv.
Orig	12096536	1.00	
A1	10342596	1.17	14.50%
A2	8717762	1.39	27.93%
A3	8325476	1.45	31.17%

Table 1: instruction-level optimizations results.

b) Loop-level optimizations

We carried out a set of experiments to implement the ME kernel loop as a long latency RFU instruction. We assume the RFU able to autonomously access the memory subsystem. Different loop implementations are possible with trade-offs in terms of overall latency, hardware utilization and data bandwidth requirements. We consider three scenarios corresponding to choose the data bandwidth available to the RFU, among one 32-bit (case "1x32"), one 64-bit ("1x64") or two 64-bits ("2x64") data accesses per cycle. Having a fixed set of pixels to be accessed, it is clear that the resulting static loop latency will be shorter when more bandwidth is available. We are interested to evaluate the best theoretical performance with the kernel loop approach, removing the potential bottlenecks to achieve the fastest loop implementation with the highest hardware utilization possible under the bandwidth constraint. The loop implementation is pipelined on *load*, *computation* and *write* stages, with initiation interval *N* and we try to instantiate enough operators so to get the lowest *N* (possibly 1). On data cache misses, the whole machine stalls as usual.

We will consider a technology scaling factor ***b*** on the loop latencies in order to account for a presumably slower RFU technology when compared to the processor speed. The kernel loop implementation strategy we adopted relies on two key points:

1. RFU prefetch instructions
2. local storage memory

An RFU instruction could be defined to perform the prefetch of a complex pattern of accesses with this latter hardwired in the instruction configuration. In this specific case study, the macroblock is a natural candidate pattern. We therefore inserted custom prefetch instructions "sufficiently ahead" in the code, with respect to the effective usage of the prefetched macroblock. After being issued, the RFU instruction execution continues as a separate thread (non-blocking behavior) and it generates to the cache controller a cache line request with the starting address for each of the macroblock rows (16 for the reference or 17 for the predictor macroblock). If any macroblock row crosses the data cache line, an extra prefetch is also issued. We prefetch both the reference and the candidate predictor macroblocks. Furthermore in order to try to guarantee a wider time window for the predictor prefetches to complete, we issue a prefetch for the next required predictor macroblock before starting the computations over the current predictor macroblock.

Regarding the local storage memory, we can observe that each reference macroblock is compared with a set of predictor candidate macroblocks to compute a SAD value. It may have sense to store it "locally" with a two-fold benefit: avoid risks of conflicts in data cache and release the bandwidth to early access the predictor required for the currently computed SAD. A solution inspired by a line buffer [13] has been considered to locally store an entire macroblock (*16x16* pixels). We will refer to this local storage as "line buffer", meaning essentially a form of level-0 cache, whose detailed organization and placement is out of the scope of the current analysis. The RFU prefetch instruction can be modified to also gather and store the reference macroblock in the line buffer, when each memory access

is completed. A flag *Done*, initially set to zero, is turned to one on each macroblock row prefetch completion. The Line Buffer size is *16*16=256* bytes plus *2* bytes for the flags information and it can be seen as a register file of *16* registers, each of *16 8-bit* pixels. The replacement can adopt a circular indexing scheme with LRU policy, naturally following the row indexing in the macroblock. We assume a 2-cycle latency, with throughput 1 to access the entire content of a row in the line buffer. If the data is missing (flag *Done* is 0), the RFU stalls the processor and issues a corresponding cache access. The Prefetch Buffer size has been extended to 64.

The address of the reference macroblock is stored in local registers in the RFU after the prefetch. The RFU loop instruction has as input operands the address of the candidate predictor macroblock (image pointer, offset) and information about the alignment and interpolation required.

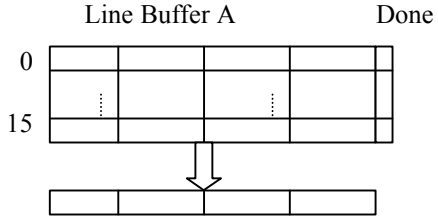

Figure 3: the Line Buffer A to store a reference macroblock and the *Done* flags.

Technology scaling considerations. To account for a potentially slower technology in the RFU (based on programmable logic and interconnects), we consider a technology scaling factor ***b*** in the instruction latencies. As up today, the worst case speed factor existing among FPGA and semi-custom can be considered around 5 (the same circuit implemented on FPGA is about 5 time slower than on standard cells). We aim to implement the RFU as "FPGA-like" but at coarser granularity level: this should allow gains in terms of speed, compared to FPGA, since less programmable points are driven and the building blocks are datapath structures other than at bit-level; a technology scaling factor of 5 can be therefore considered a reasonable worst case, and it essentially takes into account delays in the propagation along the programmable interconnect. We observe that the scaling should influence only the computational stages of the loop pipeline. Indeed the read/write stages are constrained by the external architecture and therefore they are unchanged. The RFU clock cycle is the same CPU one, and we assume that the scaled number of computational stages can still be pipelined at CPU clock cycle granularity (the interconnect provides pipelining support).

Results and analysis. Table 2 shows the results obtained for this set of experiments. For each bandwidth scenario explored, the static loop latency ("*Lat*"), in clock cycles, is reported with the total execution cycles for the two technology scaling factors ***b*** considered. In the 32-bit bandwidth scenario, the speedup achieved is considerable (*3.18*) also when the technology scaling is taken into account in a slower RFU (*2.74*). The other scenarios show an even greater speedup, due to reducing the I/O bottleneck on the predictor macroblock loads; indeed the ME algorithm is bandwidth-bound, once a sufficient computational parallelism (on 8 pixels) is exposed in the RFU. One 64-bit load access allows a *4.26* speedup, while two 64-bit allow a *5.29* speedup.

ME	*b* =1			*b* = 5		
	Lat	Cycles	S.Up	Lat	Cycles	S.Up
Orig		12096536	1.00		12096536	1.00
1x32	90	3807932	3.18	102	4416707	2.74
1x64	56	2836640	4.26	68	3446383	3.51
2x64	39	2286782	5.29	51	2895329	4.18

Table 2: loop-level optimizations results with various bandwidth options (on rows) and technology scaling factors b**.**

The technology scaling seems to degrade more the speedup, with the increase of bandwidth: for example the speedup reduces around 0.4 in the *1x32bit* scenario, while it is around 1.1 in the *2x64 bit*; this can be explained by the fact that the loop latency increase is fixed among the three cases (it is *12* cycles) but it represents a higher relative increase for the faster cases (with higher bandwidth). In fact Table 3 shows the relative percentage of the static loop latency increase and the corresponding relative speedup reduction, when scaling the technology.

ME	*b* =1	*b* =5	%Increased Latency	% SpeedUp Reduction
	Lat	Lat		
1x32	90	102	13.33%	-13.78%
1x64	56	68	21.43%	-17.69%
2x64	39	51	30.77%	-21.02%

Table 3: relative percentage of the static loop latency increase and corresponding relative speedup reduction when scaling the technology, in the loop-level optimizations experiments.

Cache results. Table 4 shows that the cache stalls are greater in the 64-bit bandwidth cases than in the 32-bit one. This comes from the fact that with more bandwidth available the static scheduled length of the loop has been shortened, narrowing the time window between the starting cycle of the custom prefetch instructions for the candidate macroblocks and the loop execution cycle

where the data is needed. This implies that most of the candidate macroblock prefetches cannot complete in time to provide the data in cache.

D$	*b* =1		*b* = 5	
	Cycles	**%Red**	**Cycles**	**%Red**
Orig	1325790	0.00%		
1x32	675294	49.06%	644791	51.37%
1x64	790827	40.35%	761294	42.58%
2x64	784359	40.84%	753627	43.16%

Table 4: ME cache stalls adopting one line buffer, with various bandwidth options and % of reduction.

This consideration is true on both the technology factor cases. In the 64-bit cases, the number of load operations has been reduced by half, but the data is not guaranteed to be in data cache. However it can be noted that the bandwidth available is still sufficient to allow to complete the custom prefetch on the reference macroblocks to store them in the Line Buffer. Indeed the experiments show that the number of late and incomplete prefetch operations is relatively low (<1%) for the sequence considered. When scaling the technology the cache stalls diminish because of a wider time window available to complete the custom prefetch on the candidate predictor macroblocks. Table 5 shows the relative percentage of cache stalls with respect to the total execution time. We can observe that the stall cycles are a relatively small component of the total ME execution time.

	b =1	*b* = 5
	%ofTotal	**%ofTotal**
Orig	10.96%	
1x32	17.73%	14.60%
1x64	27.88%	22.09%
2x64	34.30%	26.03%

Table 5: relative percentage of cache stalls with respect to the total ME execution time.

Theoretical Speedups. In Table 6 we computed for each experiment the theoretical speedup based on the loop static execution time ("*StaticCycles*" as static loop latency "*Lat*" times the total number of executions) without taking into account the cache effects (ideal 100% hit). We can observe that the achieved experimental result ("*S.Up*") is always above *57%* of the theoretical one ("*Th.S.Up*") and that its accuracy gets lower when more bandwidth is available: this is essentially another way to see the data cache stalls relevance in the overall execution time (as already observed from Table 4).

		StaticCycles	Th.S.Up	S.Up	Ratio
b=1	**1x32**	2940412	4.11	3.18	77.37%
	1x64	1853738	6.53	4.26	65.24%
	2x64	1310401	9.23	5.29	57.31%
b =5	**1x32**	3579632	3.38	2.74	81.07%
	1x64	2492958	4.85	3.51	72.37%
	2x64	1949621	6.2	4.18	67.42%

Table 6: loops theoretical speedups ("Th.S.Up", no cache effects) vs experimental speedups ("S.Up"), in case of one line buffer.

Two Line Buffers. A further improvement is possible when increasing the size of the local line buffer, to store also the current predictor macroblock. Indeed the prefetch of this latter, even if completed successfully gives no guarantee by itself that the data will be available in cache when needed, because of potential conflicts.

Given the strategy we chose to overlap a prefetch for the next required predictor macroblock with the computation over the current required predictor macroblock, we adopt a double buffering scheme. Furthermore we try to exploit the data locality between consecutive candidate predictor macroblocks, to reduce the traffic to the external bus and implement the local line buffer with full associativity. When executing a prefetch macroblock instruction, the RFU sequences, issues and stores the 17 cache line addresses (and the at most 17 potentially crossed cacheline); a tag derived from each of the row addresses is also stored for each cacheline. For every cacheline prefetch the RFU checks if a previous same request is already pending in the line buffer, in such a case the new entry gets the same status (and possibly the data, if available) as the found one and no new request is issued to the data cache controller. The line buffer B size is *17* cache line entries, to be considered four times (see Figure 4) because of the double buffering scheme and the potential cacheline crossings with few extra bytes for tags and flags (*68x23 + 68* bits = *204* bytes). The total line

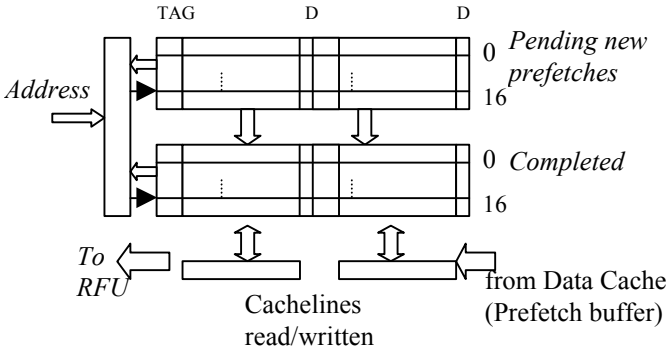

Figure 4: the Line Buffer B to store candidate predictor macroblocks, adopting a double buffering scheme.

buffer size is therefore *4*17=68* cache lines (*2176* bytes + *204* bytes). The full associativity (comparison logic) increases the resource requirement. We assume again a 2-cycle latency, with throughput 1, to access a cacheline (and the potential crossing) at once. The bandwidth is 1x32, in case of cache misses.

Results and Analysis. Table 7 shows the reduced loop latency and related speedup, for both the technology scaling factors. The speedup further increases to *8.00* and *5.40*. The cache stalls reduction improves to at least *60%*, due to a lower cache pressure because of the added Line Buffer B for the predictor macroblock. Column *%Rel* shows the contribution (in percentage) of the ME SAD kernel loop to the total application execution time: from the initial profiling of *25.6%* it has been reduced to respectively 4.14% and 6.01% with technology factors *1* and *5*.

ME	Lat	ExCycles	S.Up	%Rel	Stalls	%Red
Orig		12096536	1.00	25.6%	1325790	0.00%
b=1	23	1512950	8.00	4.14%	521954	60.63%
b=5	39	2241067	5.40	6.01%	482948	63.57%

Table 7: ME results with two line buffers.

6. Conclusions and future activity

The runtime reconfigurability added to computing devices has the potential to represent the next architectural innovation in the processor architecture field. In this paper we showed results of a preliminary exploration on a reconfigurable VLIW architecture in a video compression case study, focusing on the motion estimation stage. As a preliminary conclusion the reconfigurability assist gives benefits when kernel loops are implemented as RFU instructions. When exploiting the algorithm parallelism in the RFU, the I/O becomes the limiting factor. We explored different bandwidth options to seek for the theoretical performance allowed by this approach and showed that the I/O bottleneck can be partially solved when using complex prefetch techniques and some extent of local memory. The technology scaling of a slower RFU can be mitigated by aggressive pipelining. The results obtained in this subset of the application are significative: a speedup up to *8x* with loop-level optimizations is achieved under various architectural assumptions. Future work will extend the analysis to other parts of the application and consider the dynamic interaction between a greater set of configurations competing for resources in the RFU; we will explore reconfiguration management techniques to hide the reconfiguration penalty, and refine a more detailed microarchitectural model of the Reconfigurable Unit, validating the hypothesis on size and instruction latencies. The VLIW compiler support to automate the analysis and extraction of the configurations is a research topic that will be taken into future consideration. The final outcome will be the definition of a flexible platform architecture suitable to meet the performance requirements of domain-specific applications.

7. References

[1] W. Mangione-Smith, et al. "Seeking solutions in configurable computing," *IEEE Computer*, Dec 1997, pp. 38-43.

[2] K. Compton, S. Hauck "Reconfigurable Computing: A survey of Systems and Software", Nothwestern University, Dept. of ECE Technical Report, 1999.

[3] D.C. Cronquist, P. Franklin, S.G. Berg, C. Ebeling "Specifying and Compiling Applications for RaPiD," Proc. IEEE Symp. FCCM, 1998.

[4] J. Hauser, J. Wawrzynek "Garp: a MIPS Processor with a Reconfigurable Coprocessor," Proc. IEEE Symp. FCCM, 1997, pp.24-33.

[5] H.Singh, et al. "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *IEEE Trans. on Computers*, Vol.49, No.5, May 2000, pp.465-481.

[6] Z. Ye, P. Banerjee, S. Hauck, A. Moshovos, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled RFU," Proc. 27th ISCA, 2000.

[7] ISO-IEC 14496-2 "Final Draft of international standard", Dec 18 1998.

[8] P. Faraboschi, G. Brown, J. Fisher, G. Desoli, F. Homewood "Lx: A technology Platform for Customizable VLIW Embedded Processing," Proc. 27th ISCA, 2000, pp.203-213.

[9] T. Callahan, J. Hauser, J. Wawrzynek "The GARP Architecture and C compiler," *IEEE Computer*, vol.33, no.4, April 2000, pp. 62-69.

[10] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, J. Stockwood, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", Proc. DAC, 2000, pp.507-512.

[11] http://www.proceler.com

[12] Z.Huang, S.Malik, "Managing Dynamic Reconfiguration Overhead in Systems-On-A-Chip Design Using Reconfigurable Datapaths and Optimized Interconnection Networks", Proc. DATE, 2001, pp.735-740.

[13] K.M. Wilson, K. Olukotun "High Bandwidth On-Chip Cache Design," *IEEE Trans. on Computers*, vol. 50, no. 4, April 2001, pp. 292-307.

[14] Z.Li, K. Compton, S. Hauck "Configuration Caching Techniques for FPGA", Proc. IEEE Symp. FPGA for Custom Computing Machines, 2000.

[15] R. Maestre et *al.* "A Formal Approach to Context Scheduling for Multicontext Reconfigurable Architectures," *IEEE Trans. on VLSI Systems*, Vol.9, no.1, Feb.2001, pp.173-185.