

A Reconfigurable Approach to Low Cost Media Processing

Igor Kostarnov, Steve Morley, Javed Osmany and Charlie Solomon
[iaklstmljolchas]@hplb.hpl.hp.com

Appliance Computing Department, Hewlett-Packard Laboratories,
Filton Road, Stoke Gifford, Bristol BS12 6QZ, UK

1. Introduction

Future products retailing for \$300 or less will need to support high performance implementations of media algorithms such as image and video decompression. The design of such "media appliances" imposes many constraints, not only on cost and performance, but also on development time and hence time-to-market.

This paper proposes a design approach using an "off the shelf" CPU supplemented by application-specific acceleration implemented in a single Field Programmable Gate Array (FPGA). We describe the operation of the experimental board used to investigate strategies for mapping algorithms onto such systems and, for two representative algorithms, determine the performance that can be achieved with this modest amount of FPGA.

This paper first contrasts existing design approaches with the features of the CPU/FPGA approach. The following section describes the evaluation board, called Riley, which was developed to investigate a few system alternatives and their utility in accelerating media algorithms. The two specific algorithms are then presented in detail and finally some conclusions are drawn.

2. Background

2.1 Existing Design Approaches

The various approaches to low-cost media processing include general purpose micro-processors, augmentation with packed arithmetic instructions, digital signal processors (DSPs), application-specific instruction processors (ASIPs), and application-specific integrated circuits (ASICs). These make significant tradeoffs between processing speed, time-to-market, non-recurring engineering costs (NREs), and flexibility.

2.2 CPU/FPGA Approach

Our approach supplements an "off the shelf" CPU with a modest amount of FPGA for hardware acceleration. This combines the performance benefits of an application-specific hardware solution with the flexibility of a general programmable solution [1].

Consistent with the low-cost criteria only a single FPGA is considered (i.e. a Xilinx XC4013). The advantages offered are:

1. No NREs (with respect to those of an ASIC). The hardware accelerator is a programmable device and development time is significantly reduced.
2. Flexibility. Algorithm changes (such as occur through changes in standards), the can be reprogrammed without having to do expensive hardware modifications.
3. Dynamic reuse. The same silicon (FPGA) can be reconfigured for different modes of use during the normal life cycle of the appliance.

However a number of new design challenges for CPU/FPGA systems remain:

- How much FPGA is required to achieve significant performance improvement? More specifically can we meet our performance targets given the limited FPGA budget? (Note that this contrasts with much existing work that uses arrays of FPGAs for custom computing e.g. [2]).
- How is the scarce reconfigurable resource used effectively? The partitioning of algorithms into hardware and software is obviously crucial to achieving good performance.

Crucial to the effective utilisation of the FPGA is the consideration given to:

1. The relative latencies for implementing the same operation sequences on the CPU versus the FPGA.
2. Exploitation of parallelism both within the FPGA as well as between CPU and FPGA. Both the algorithm and the available FPGA resources will constrain this.
3. Match the data feed and store rates to the particular FPGA configuration's requirements.
4. Reduce the synchronisation overhead between the CPU, FPGA and other system components.
5. Use internal data stores to hold intermediate results. This reduces the requirement for off chip I/O.

We have applied these criteria in the investigation of a few implementations of two representative algorithms an evaluation platform called *Riley*, which we describe in Section 3. The results from these implementations are presented in Section 4.

3. Riley Board Description

3.1 Design Philosophy

Riley is a board designed for the investigation of system architectures comprising a RISC CPU and FPGA. Much of the board space is given over to debug and measurement ports to enable detailed analysis of results.

The Riley hardware permits the major system components to be interconnected in a variety of ways. This also allows experimentation with the way different algorithms are mapped to the board.

3.2 Architecture

Riley comprises an i960J CPU [4], a Xilinx 4013 FPGA [5] and shared system memory. There is also a boot EPROM containing MON960 and an I/O block containing both serial and parallel interfaces. The connectivity is shown in Figure 1. The FPGA is memory-mapped into the i960 address space.

The i960J has a multiplexed address and data bus that is de-multiplexed in a bus control unit. The FPGA has direct access to both the multiplexed and de-multiplexed busses. It can arbitrate for bus mastership using the i960 bus hold/grant protocol or can grab the de-multiplexed busses directly using the “Abus_get” and “Dbus_get” signals. FPGA configurations are stored in the configuration RAM. The FPGA can disconnect this memory from the main system bus and use it as a local working memory completely independent from the main system memory.

Zero wait state SRAM has been used for both the system and configuration memories. This permits the simulation of on-chip storage. The i960 has on-chip I and D caches which can be enabled and disabled as required by an application

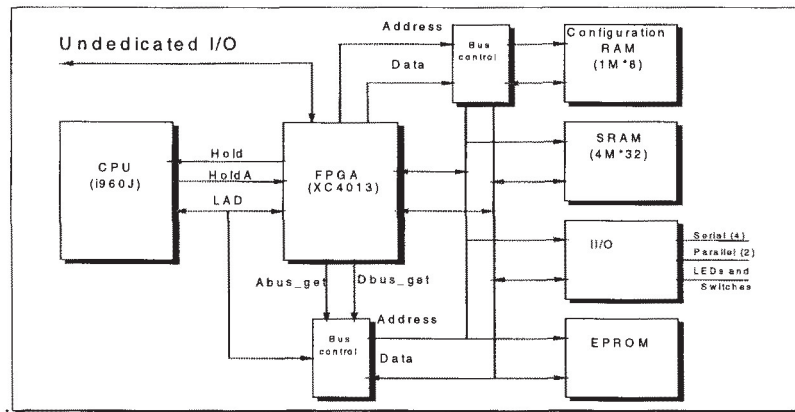


Fig. 1: Riley architecture

3.3 Modes of Operation

We chose to investigate 3 modes of interaction between the CPU, FPGA and memory.

In *functional unit mode* (Figure 2), the CPU is always bus master. The CPU issues instructions to the FPGA, writes operands, and reads results. All data therefore passes to and from the FPGA via the CPU register file. There are two types of instruction issued. 1) An *asynchronous instruction* allows the CPU to continue execution after the

instruction has been issued. A signal back from the FPGA (either an interrupt or a polled memory-mapped register) indicates that the instruction has completed. 2) A *synchronous* instruction tells the FPGA to suspend the CPU bus interface unit by use of a “READY” signal. The FPGA asserts this signal when the instruction has completed.

In *lockstep mode*, the FPGA can read and write directly to the de-multiplexed bus using addresses generated by the CPU. On reads, the FPGA traps the address generated by the CPU and reads data from the de-multiplexed data bus. On writes, the FPGA uses the “Dbus_get” signal to isolate this bus from the CPU. The FPGA writes data directly to the bus during the CPU write cycle.

In *datapath mode* the FPGA can arbitrate to become bus master. The CPU and FPGA execute as two independent execution units and most of the information that passes between them is control and synchronisation.

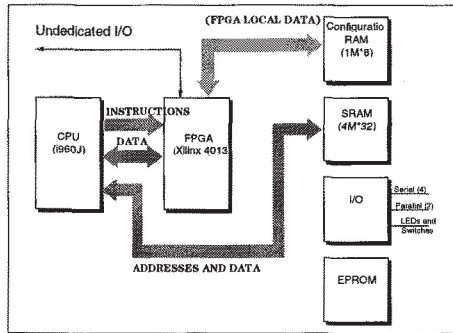


Fig. 2: Functional unit mode

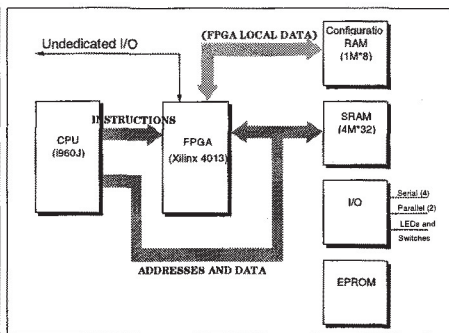


Fig. 3: Lockstep mode

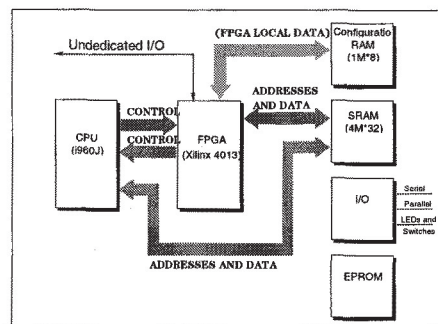


Fig. 4: Datapath mode

The mode of operation strongly influences the partitioning options for a particular algorithm and therefore the performance. An important part of the Riley investigation was to evaluate these modes across a range of algorithms.

3.4 Tools

All applications were implemented using C via the gnu960 compiler and VHDL for the FPGA. Though optimisation switches were used to generate efficient object code, occasionally, inline assembler was used. The VHDL source was synthesised to a Xilinx netlist using Synopsys synthesis tools and Xilinx libraries. Some structural coding was required to instantiate certain components, e.g. on chip RAM.

4. Results

This section describes the findings of two experiments that were conducted using the Riley board. Both performance and partitioning issues are discussed for each experiment.

4.1 Multiply/Accumulate Operations

It is generally possible to build more than one multiplier or accumulator in the FPGA and execute some of these operations in parallel. The capacity of the FPGA dictates how much parallelism can be exploited. In order to make partitioning decisions for the architectures being considered it is necessary to know the cost and performance of MAC operations in both the CPU and the FPGA.

The i960 used on Riley has an on chip multiply divide unit which generates a 32 bit result in 2 to 4 cycles. Addition takes one cycle.

The Synopsys tools synthesise a VHDL “*” operator to a combinatorial multiplier. The delay of a 16*16=32 bit multiplier in the FPGA is 5 cycles worse case at the 20Mhz clock frequency used on Riley. However, this one component uses around half of the available gates. An accumulator in the FPGA will execute in a single cycle. The gate count is insignificant in comparison to the multiplier.

Savings can be made in the FPGA by being efficient on bit-width i.e. only using the precision required. For example an 8*8=16 bit multiplier uses only 1/4 of the gates and executes in 3 cycles worse case. Further savings can also be made if one of the operands is fixed and known at compile time, for example the filter coefficients. Typically, a multiplier with a fixed coefficient uses half as many gates as an arbitrary multiplier.

Table 1 shows the parallel multiply/accumulate operations that fit into the Riley FPGA for various operand precisions. Both fixed and arbitrary multipliers are shown.

No. bits	8	12	16
Arbitrary	5	2	1
Fixed	12	5	2

Table 1: No. of taps in FPGA for filters

To realise the extra performance obtainable by parallelism in the FPGA, the system architecture must support dataflows that ensure the multipliers/accumulators are kept supplied with data. The location of data in the system is key to partitioning.

4.1.1 Partitioning and Results

All the issues just described affect the partitioning process for a DSP algorithm. This is now illustrated using the example of a Finite Impulse Response (FIR) filter.

The absolute performance of a single multiply/add is between 3 and 5 cycles in the CPU and is 6 cycles in the FPGA. In an N-tap filter however, the N multiplies can execute in parallel in the FPGA, so only one extra cycle for the accumulate is added for each tap. Assuming an average of 4 cycles per tap for the CPU, speedups can be achieved if two or more taps are executed in parallel in the FPGA. The speedup increases as taps are added, subject to the resource limits of the FPGA.

The performance of this partition is offset by the interface overhead in reading the new data sample and storing the result. This overhead depends upon the mode of operation. In *functional unit mode*, a load and a store are required for getting the sample from the memory to the FPGA, and a further load and store are required for returning the result. Each CPU load and store takes 3 cycles. In *lockstep mode*, only one load and one store are required as the data goes direct from memory to the FPGA. *Datapath mode* also only requires one load and one store but this time the FPGA can access memory on each cycle. Datapath mode offers the best performance then but requires more gates to implement the address generation.

Figure 6 shows the performance of the three operating modes and of the optimised CPU-only implementation for a 5-tap filter, as measured on Riley. A clock for clock comparison with a DSP, the Motorola DSP56000, is also shown. In all cases, code on the CPU runs entirely from I-cache, the FIR filter code requires only 20 instructions or so. All data is held in D-cache.

The performance in this example is dictated by the CPU/FPGA interface overhead. In functional unit mode, the FPGA is only in use 20% of the time with the CPU working flat out supplying data. In lockstep mode, the FPGA is in use 50% of the time.

When the number of taps in the filter exceeds the number that can be fitted into the FPGA, a different partition must be used. The calculation is broken down into stages so an N-tap filter is calculated in N/M stages (rounded up) of M taps each. An intermediate result must be accumulated between each stage. Fixed multipliers cannot be used in this partition as the coefficients change between sections. This will limit the value of M as shown in Table 1.

The coefficients and filter memory must now be stored either in a memory local to the FPGA or in the main system memory. M-way parallel access to this data is required to exploit the parallelism in the FPGA. This can be achieved for the coefficients by having M memories, one for each multiplier. Each memory has N/M elements. This is

more problematic for the filter memory however as this data must be shifted by one place in memory at each sample interval.

In this case we choose to store the filter memory in main memory. To calculate one filter output therefore, $N-1$ samples are read from filter memory in addition to the new data sample. Figure 7 shows the results obtained for a 100-tap filter running on Riley.

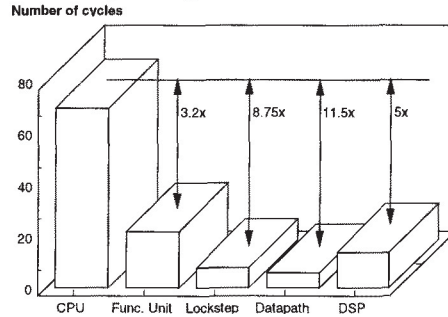


Fig. 6: Performance of a 5-tap filter

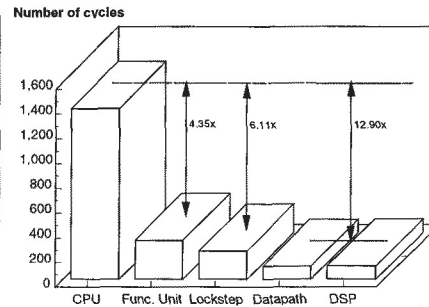


Fig. 7: Performance of a 100-tap filter

Comparing these results, the impact of the extra data marshaling can be seen when considering the DSP result (whose performance increases linearly as a function of the number of filter taps). Datapath mode now only just achieves parity with the DSP. Again, the CPU/FPGA interface limits the performance achieved in each mode.

4.2 The DPMatch Algorithm

Dynamic programming [7] is used in many “pattern matching” media algorithms. The algorithm has many different attributes to the digital filtering previously described.

A Dynamic Programming Match (DPMatch) engine (below) evaluates the “edit distance” between two code strings.

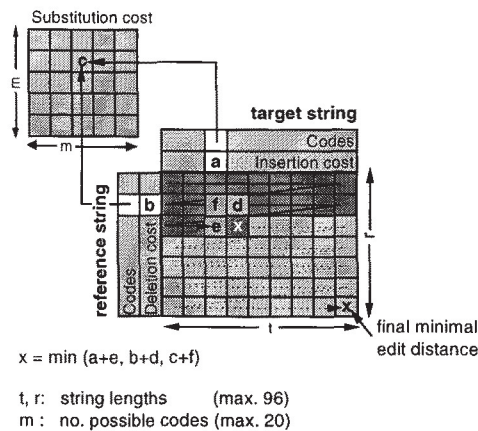


Fig. 8: DPMatch performance

Each cell in the “t” x “r” match-matrix represents the “edit distance” between the strings at that point of the comparison. The cell value is computed by first calculating three “distances” that result from:

1. **inserting** a particular target string code. This is calculated by adding the insertion cost (“a”) of the target string to the combined “distance” thus far (“e”).
2. **deleting** a particular reference string code. This distance is calculated by adding the deletion cost (“b”) to the combined distance (“d”).
3. **substituting** the particular target string code for the relevant reference string code. The cost is defined by the pair-wise mapping of the relevant target and reference codes. The distance value is calculated by adding the substitution cost to the combined string distance *before* the last insertion and deletion operations (“f”).

The resulting distance for the current cell (“x”) is determined by **assigning the lesser** (“min.”) of these three values. The final edit distance is the value that results in the last (lower right-hand) cell. More discussion of this algorithm can be found [8].

4.2.1 The DPMatch Partitions

The DPMatch algorithm was investigated in three different partitions that were compared with the CPU-only version running on Riley. The definition of these partitions was derived from a visual inspection of the “C” code. Starting from the smallest partition and working outward we selected:

1. A cell-by-cell computation. This is a minimal partition in the sense that the computation of one cell in the “t” x “r” matrix in Figure 8 is performed in the FPGA. The CPU performs all the data marshalling.
2. A row-by-row computation. In this partition the FPGA computes one row of the matrix in a single instruction.
3. The full match. Here the FPGA performs a complete match between the target and reference strings.

4.2.2 Cell-by-Cell

In this partition the FPGA hardware consists only of the DPMatch’s cell computation. This is the addition of the three costs and the selection of the minimum. This partition uses *functional unit mode*. The insertion, deletion and substitution costs are all calculated by the CPU and passed as operands to the FPGA.

This minimal partition didn’t quite reach parity with the CPU-only version of the DPMatch. It used 12% of the available FPGA gates. The obvious observation is that the CPU/FPGA interaction overhead is slightly larger than the savings had from the faster computation on the FPGA. Nonetheless, there are a few important observations to be made:

1. Because this partition is close to the computational core, the marshalling is complex.

2. It is difficult to derive much concurrency between the CPU and FPGA owing to the current cell's dependency on the results from the previous one.
3. On Riley, as the bus overhead was significant, operands (i.e. costs) were packed into a single bus write to the FPGA. The extra instructions used for packing in the CPU proved to be more efficient than using separate bus cycles.

4.2.3 Row-by-Row

This partition allows the CPU to load blocks of “pre-costed” target and reference strings to the FPGA at the beginning of the match. “Pre-costing” is the translation of the target and reference strings into their respective insertion, deletion and substitution costs.

As with the cell-by-cell partition, this operates solely in *functional unit mode*. Because on-chip storage is limited in the FPGA, the large matrix of “pre-costed” substitution costs required for a match could not all be stored simultaneously. Therefore the CPU performs an initial load of the strings' insertion and deletion costs, and after each successive computation of a row, the CPU loads the next row of substitution costs. The final result is read back after the loading of the final row. The row-by-row implementation performed 1.6 times faster than the CPU-only implementation and uses 42% of the available FPGA gates. The performance is limited by the overhead of the CPU writing the substitution costs for each row.

The row-by-row partition provides some interesting features over the cell-by-cell:

1. The provision for some internal intermediate state obviated the interactions required to retrieve and re-feed these values.
2. Data marshalling is much simplified because word-aligned blocks of operands were passed hence the interaction overhead per computation was reduced.
3. Some concurrency was obtained while the CPU marshalled for the next row of substitution costs the FPGA calculated the current row.
4. Limited capacity in the FPGA meant that the full block of substitution costs for a match could not be allocated. To simplify the management of this only the capacity for one row of substitution cost was provided.
5. The bottleneck in achieving better performance is the marshalling and transfer of the substitution costs from the CPU to the FPGA.

4.2.4 Full Match

In this partition, as much of the algorithm as possible has been placed in the FPGA. This includes the code-to-cost mapping (“costing”), obviating the row-by-row's requirement to down-load the large cost matrix for each match.

The interface between the CPU and the FPGA is divided into two phases:

1. The initial set-up phase, when the substitution cost look-up tables (LUTs) are downloaded into one the FPGA configuration memory pages. This uses datapath mode. After this phase, all costing is done in the FPGA.
2. The run-time passing of the strings. This uses functional unit mode.

The CPU has two ways of reading the result. It can poll a ready register and read only after the result is ready, or it can read directly via a *synchronous read* instruction (see Section 3.3). In this case the read cycle will be suspended until the DPMatch matrix computation is complete.

This partition achieves a 21x performance increase over the CPU-only implementation. It uses 50% of the available gates in the FPGA. Note that the substitution costs are now held in the configuration memory not in the FPGA; which is why this partition does not use significantly more CLBs than the row-by-row.

The following factors contribute to this result:

1. The costing of the coded strings (i.e. converting the code strings into their representative insertion/deletion cost strings) is being done much more effectively by the FPGA. In the case of the insertion and deletion costs, combinatorial logic is used to generate the cost from the string codes instead of a table lookup.
2. The substitution costs are being accessed via the FPGA's own configuration memory. These accesses are faster than CPU accesses (2 vs. 3 cycles) and the FPGA does not have to arbitrate with the CPU for memory access.
3. Functional unit mode is only used for passing the target and reference strings.
4. All registers and data paths are carefully tailored to the specific minimal requirements of the algorithm.

4.2.5 DPMatch Conclusions

Figure 9 summarises the performance of the CPU-only version of the DPMatch compared with that of the three partitions.

The CPU/FPGA interface overhead limits the performance of the DPMatch algorithm in the cell-by-cell and row-by-row partitions. The important lesson learned from the full match partition is that the main performance gain was achieved *not* because of parallel computation (in fact, there is only one non-pipelined computing unit) but because of the concurrent data access and careful design of the control state machine.

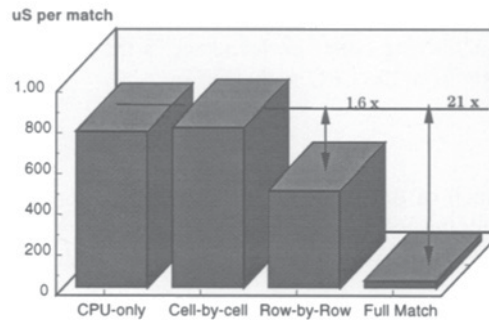


Fig. 9: DPMatch performance

5. Conclusion

In this paper we have proposed a new approach to the design of media intensive appliances using a CPU and a modest amount of FPGA for hardware acceleration. From implementations of representative algorithms (a digital filter and a dynamic programming match), we have demonstrated that a small amount of reconfigurable logic can be used to achieve high performance. For these algorithms the equivalent of around 500 Xilinx CLBs, plus a small local memory, can increase performance 12 to 21 times when compared to an optimised CPU-only implementation.

The FPGA is able to exploit parallelism available in an algorithm. The FPGA can also be “bit-width efficient”, using no more bits of precision than are necessary. Internal store can be used for local data, state and control. The system architecture must however be able to support this increased computation rate. We have explored three architectures that enable this.

In functional unit mode the FPGA is a slave coprocessor to the CPU. This is the simplest mode for partitioning but performance can be limited by marshalling overheads on the CPU. It can be used where FPGA resources do not enable direct memory access or where only portions of an algorithm computation can be accommodated.

Lockstep mode is very similar to functional unit mode but increases performance by having a direct data path between the FPGA and memory.

Datapath mode offers the best potential performance but, as the CPU and FPGA execute as two independent units, there are many more issues that must be considered when partitioning. Datapath mode is particularly effective when the FPGA can process a large independent part of the algorithm without need for complex control and synchronisation with the CPU.

These architectures provide templates for the synthesis of CPU/FPGA systems. They provide contexts within which partitions can be evaluated. Although the process is manual at the moment, we expect the architectures and techniques discussed in this paper to form the basis for future tools which will automate this process. Research at HPLabs Bristol is targeted at providing these design tools.

References

- [1] W.Sharpe, D.M.McCarthy, “Why can’t hardware be more like software?”, *IEEE Colloquium on Partitioning in Hardware-Software Codesigns*, Feb 1995.
- [2] D.A.Buell, J.M.Arnold, and W.J.Kleinfelder, “*Splash 2 : FPGAs for Custom Computing Machines*”, FCCM 95, Los Alamitos, IEEE Computer Society Press, 1995.
- [3] Rahul Razdan, “*PRISC: Programmable Reduced Instruction Set Computers*”, PhD dissertation at the Dept. Of Computer Science, Harvard University, May 1994.

- [4] *"I960[®] Jx Microprocessor User's Manual", Sept.1994, Intel Corp.; ISBN 1-55512-228-0.*
- [5] *"The Programmable Logic Data Book", Xilinx Inc., 1994. San Jose, CA*
- [6] *Steve Morley, "Multiply-Accumulate Intensive Algorithms on the Riley Experimental Board", Hewlett Packard Laboratories, Bristol, November, 1995; HPL-95-127.*
- [7] *Igor Kostarnov, Javed Osmany, Charles Solomon, "Riley DPMatch - An exercise in algorithm mapping to hardware", Hewlett Packard Laboratories, Bristol, November, 1995; HPL-95-128.*
- [8] *Robert Sedgewick, "Algorithms", Addison-Wesley Publishing Company, Reading, Massachusetts, 1983; ISBN 0-201-06672-6; "Chapter 37: Dynamic Programming", pp 483ff.*