A RECONFIGURABLE APPROACH TO LOW COST MEDIA PROCESSING

Igor Kostarnov, Steve Morley, Javed Osmany and Charlie Solomon
[iak|stm|jo|chas]@hplb.hpl.hp.com

Appliance Computing Department, Hewlett-Packard Laboratories, Bristol, UK

1. INTRODUCTION

Future embedded systems will need to process media data at low cost. Products retailing for $300 or less will need to support high performance implementations of media algorithms such as image and video decompression. These products are termed "media appliances". Their design imposes many constraints on design teams, not only cost and performance, but also engineering development time and hence time to market.

This paper proposes a new approach to processing in such products using an "off the shelf" CPU supplemented by custom hardware acceleration implemented in a Field Programmable Gate Array (FPGA). In the paper we investigate the mapping of algorithms onto such a system and, for two representative algorithms, determine the performance that can be achieved with a modest amount of FPGA.

The CPU/FPGA approach suggests a combined hardware/software solution for implementing media algorithms. We have therefore investigated the characteristics of the algorithms in order to understand the partitioning tradeoffs in CPU/FPGA solutions. Traditionally, this process is done manually by designers and decisions are made on experience and prior domain knowledge. Future tools will undoubtedly help automate this process. This work provides a basis for our future work in developing these tools.

The paper begins with an overview of existing design approaches and describes and contrasts the CPU/FPGA approach. A number of system architectures are suggested. The following section describes an evaluation platform, called Riley, which was developed for investigating these ideas. Two specific investigations are then presented in detail and finally some conclusions are drawn.

2. BACKGROUND

2.1 Existing design approaches

The ideal solution for a low cost media appliance would be to use an "off the shelf" processor, if one could be found with the required cost/performance. This solution can leverage existing skills, code and tools to achieve fast time to market. The solution is also flexible in that algorithm changes can be incorporated with firmware upgrades. Generally however, the performance of low cost CPUs is inadequate for the demands of media algorithms so some hardware acceleration is required.

Digital Signal Processors (DSPs) are used extensively in low cost embedded systems that process media data e.g. modems. Algorithms must be written and optimised (generally in assembler) for a specific DSP platform however. This can lead to long development times and means that the final implementation is not portable. Performance can also be limited by the sequential processing model of a DSP. A number of manufacturers (e.g. Texas Instruments, TriMedia and MicroUnity) have developed "media engines" to address this. These give the required performance but at a cost too high for the $300 appliances we are considering.

As an alternative, it is possible to design a custom processor for the specific algorithm or application. The characteristics of the algorithm define the architecture of the processor, so this solution can obviously yield very high performance. There are many disadvantages however, including development time, Non-recurring engineering charges (NREs), engineering risk and compatibility in terms of porting of existing application code and tools.

The most common design approach is to use an "off the shelf" CPU with an ASIC to provide

the algorithm acceleration. This solution can also give very high performance but again suffers from high NRE charges and long engineering development time. Furthermore, the ASIC may have to be redesigned if the algorithm specification changes.

An approach used by some of the mainstream processor vendors is to extend the instruction set of their existing processors to support media algorithms. This approach typically uses subword operations where a number of operands are operated on in parallel. Examples of the types of instruction added include subword arithmetic, scatter/gather operations and data type conversion. This approach is used in the HP PA-RISC architecture, the Sun UltraSparc VIS and the Intel MMX instruction set.

This approach can achieve good speedup over the basic (unenhanced) CPU implementation and it is also flexible as algorithm changes can be incorporated by upgrading firmware. There is however restricted scope for adding to the instruction set of a processor without impacting performance for existing operations. This limits the performance achievable with this approach.

2.2 CPU/FPGA approach

Our approach is to supplement an "off the shelf" CPU with a modest amount of FPGA for hardware acceleration. This combines the performance benefits of a hardware solution with the flexibility of a programmable solution [1]. As the intended target is a low cost embedded system only small amounts of FPGA are considered, the amount provided by a single commercially available device. (The actual device used in our investigations is a Xilinx XC4013). This approach has a number of advantages:

1. As the hardware accelerator is a programmable device, there are no NRE charges (c.f. ASIC). The development time also decreases significantly.
2. If the algorithm changes (for example an update to a standard), the FPGA can be reprogrammed by firmware without having to do expensive hardware modifications.
3. The FPGA can be reprogrammed for different modes of use of the appliance i.e. hardware reuse.

There are however a number of new design challenges for CPU/FPGA systems:

How much FPGA is required to achieve significant performance improvement? More specifically for the type of systems being considered here, can we meet our performance targets given the limited FPGA budget? (Note that this contrasts with much existing work which uses arrays of FPGAs for custom computing e.g. [2]).

Given the modest amount of hardware acceleration, how is this scarce resource used effectively? The partitioning of algorithms into hardware and software is obviously crucial to achieving good performance. Some of the partitioning criteria relevant to the CPU/FPGA system are:

1. The cost of performing a computation in the CPU and the FPGA. For the FPGA, the number of gates used and the latency for the computation is used. For the CPU the latency of the instruction (or sequence of instructions) is used.
2. Opportunities for parallelism. Many of the performance improvements in a CPU/FPGA system will come from exploiting parallelism. Both the algorithm and the amount of FPGA resource available will constrain this.
3. The system architecture required to support the dataflows for computation in the FPGA.
4. Interaction between the CPU, FPGA and other system components.
5. The need for internal store in the FPGA to hold intermediate results.

We have investigated these criteria by implementing a number of representative algorithms on an evaluation platform called Riley, described in detail in Section 3. The results from these implementations are presented in Section 4.

2.3 Architectures

The interaction between the CPU and the FPGA is key. This will drive the partitioning process and hence determine the performance achievable for an algorithm. We have explored three architectures, or modes of interaction, which define the way in which the CPU and FPGA interact both with each other and with other system components such as memory and I/O. The following descriptions may be read in conjunction with Figures 2, 3 and 4 which

show how the modes have been implemented on Riley.

In *functional unit* mode the FPGA operates much like a traditional coprocessor. It is always slave to the CPU. The CPU issues instructions and writes operands to the FPGA. Results are always returned to the CPU. More than one function can be implemented in the FPGA. This mode is very similar to the work in [3]. The CPU is responsible for all address generation (to memory and I/O) and data marshaling i.e. the management of dataflows to and from the FPGA and memory. This mode represents the simplest model for partitioning. Typically, frequently executed compute intensive sections of the algorithm are compressed into single FPGA instructions. It must be noted however that all data passes between the FPGA and memory via the CPU register file. The CPU can therefore spend a significant amount of time marshaling data

Lockstep mode is a variant of functional unit mode. It overcomes some of the drawbacks of that mode by having a direct datapath between the FPGA and memory. The FPGA traps certain addresses as they are issued by the CPU and reads and writes data accordingly. There is an implicit "instruction" in the FPGA associated with the CPU address. This mode enables the addressing power of the CPU to be used but data no longer moves to the FPGA through the CPU register file thus improving performance. An important feature of this mode is that the execution of the CPU and the FPGA remain synchronised.

In *datapath* mode the CPU and FPGA operate as two independent execution units. The FPGA is itself responsible for addressing memory and I/O and must arbitrate with the CPU for control of system busses. The CPU and FPGA interact mainly to synchronise and exchange control information. This mode potentially offers the best performance as the addressing and computation datapaths can be optimised directly for the algorithm, rather than being limited by what the CPU can provide as in the previous modes. Partitioning to realise this performance can be difficult however. The algorithm must be analysed in great detail to ensure that dependencies between the CPU and FPGA are minimised. Other issues such as coherency of shared data in memory also have to be addressed.

3. RILEY BOARD DESCRIPTION

3.1 Design philosophy

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

The Riley hardware is very flexible to allow a wide range of architectures and algorithms to be implemented. The major system components can be interconnected in a number of ways. As far as is possible on a board design, Riley places few constraints on the architectures that can be implemented.

3.2 Architecture

The major components on Riley are an i960J CPU [4], a Xilinx 4013 FPGA [5] and shared system memory. There is also a boot EPROM 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 which is demultiplexed 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 demultiplexed 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 gives single cycle access for both code and data, largely eliminating the effects of memory accesses on performance. A large amount of memory has been provided for storing the large data sets typically used in media algorithms (e.g. images). The i960 has on chip I and D caches which can be enabled and disabled as required by an application.

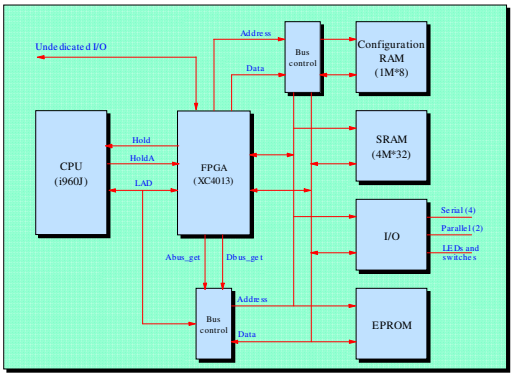

Figure 1 - Riley architecture

3.3 Modes of operation

Riley can support the 3 modes of operation described in section 2.3. The concept of a "wrapper" was developed for implementing the modes on Riley. The wrapper embodies the mode as a set of "C" drivers (running on the i960) and VHDL code (synthesised in the FPGA) which handle the interaction between the CPU, FPGA and the rest of the system. Applications work under the wrappers to ensure they obey the "rules" of the mode. This approach has a number of benefits:

1. The wrapper provides a model which can be used for partitioning algorithms and synthesising implementations.
2. Consistency between designs: The performance of the wrapper (both in terms of speed and size of circuit) can be determined independently from any application and factored out of comparative results.
3. Portability: C and VHDL code could be reused across different algorithm implementations thus saving development time.
4. Protection: Given the connectivity of Riley, it is possible to generate FPGA designs that can damage the board. The wrappers were developed and tested first in order to provide some protection against this.

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. In an asynchronous instruction, the CPU can continue execution after the instruction has been issued. A signal back from the FPGA (either an interrupt or a polled signal) indicates that the instruction has completed. In a synchronous instruction, the FPGA suspends 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 demultiplexed 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.

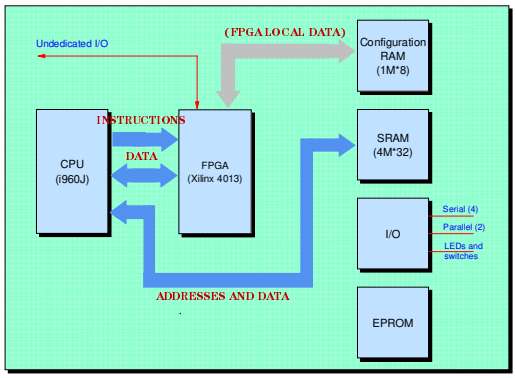

Figure 2 - Functional unit mode

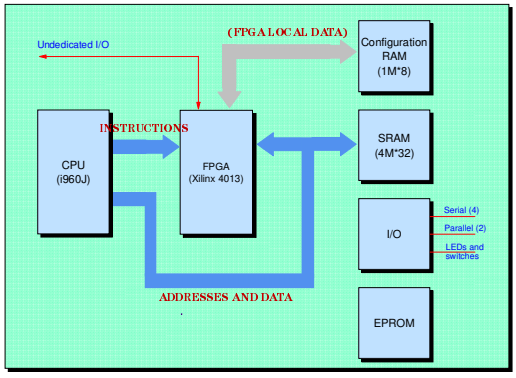

Figure 3 - Lockstep mode

.

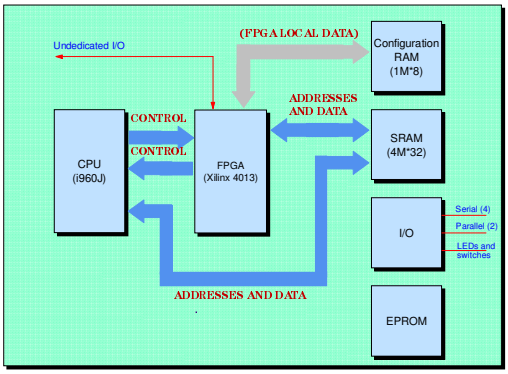

Figure 4 - Datapath mode

The mode of operation strongly influences the partitioning options for a particular algorithm and hence the results obtained for that algorithm. An important part of the Riley investigation was to evaluate these modes across a range of algorithms. This will drive the design and synthesis of this type of system in the future.

3.4 Tools

All applications were implemented using C for i960 code and VHDL for FPGA circuits. The C source was compiled using the GNU GCC compiler and associated tools. Optimisation switches were used to attempt to generate the most efficient object code. Occasionally, inline assembler was used to improve performance beyond that achievable using the compiler. The VHDL source was synthesised to a Xilinx netlist using Synopsys synhthesis tools and Xilinx supplied libraries. Some structural coding was required to instantiate certain components, e.g. on chip RAM.

3.5 Measurements

Timing statistics for long algorithm runs were collected using monitor program timing routines which count system clock ticks. Typically, the results were averaged over many runs. More detailed information was obtained using a logic analyser attached to one or more measurement ports.

4. RESULTS

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

4.1 Multiply/Accumulate

4.1.1 Background

DSPs are used extensively today in embedded systems. Their architectures are optimised for the mathematical operations found in many media algorithms The work described in this section was aimed at evaluating the CPU/FPGA architectures for tasks traditionally executed on DSPs (see [6] also)..

4.1.2 Multiply/Accumulate operations

A multiply/accumulate (MAC) unit is at the core of every DSP architecture. All algorithms that execute on DSPs use this unit extensively.

The complexity of a DSP algorithm can be estimated in terms of the number of MAC operations required. 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 and this obviously effects the partitioning. 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 up 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 bitwidth 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 number of parallel multiply/accumulate operations that fit into the FPGA on Riley 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.

4.1.3 Partitioning and results

The issues described in the previous section all drive the partitioning process for a DSP algorithm. This is now illustrated using the example of an Finite Impulse Response (FIR) filter.

Figure 5 shows a flowgraph of an FIR digital filter. Each tap computes the product of a data sample and a coefficient. The filter output is the sum of all the taps. The data samples form a shift register (the filter memory) which clocks once per sample period. Filter lengths can range from a few taps (e.g. simple low/high pass filters) to hundreds of taps (e.g. echo cancellers, windowing functions).

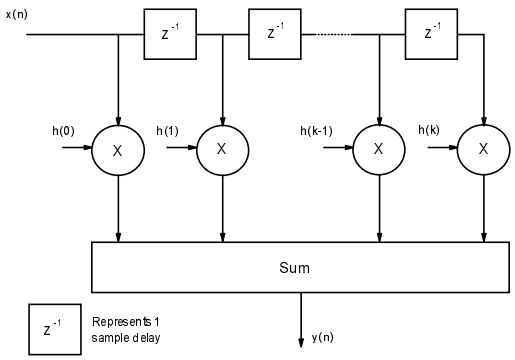

Figure 5 - FIR filter flowgraph

An N-tap FIR filter requires N multiplies and N-1 additions for each output. Assume that all N taps, including store for the coefficients and the filter memory, can be synthesised in the FPGA. In this partition, a new data sample must be read from memory and the result sample stored back to memory at each sample period.

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, this implies that speedups can be achieved if two or more taps are be executed in parallel in the FPGA. The speedup increases as taps are added. A limit will obviously be reached when no more taps fit in the FPGA (see Table 1).

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 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 .

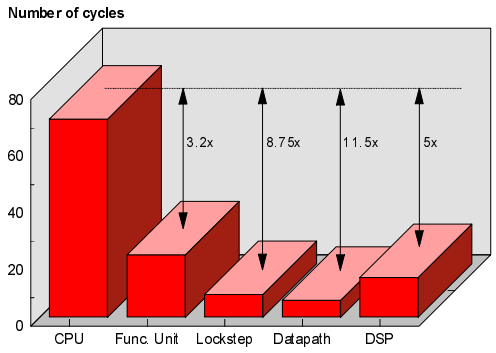

Figure 6 - Performance of a 5-tap filter

In all cases, all 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. Datapath mode achieves almost 100% efficiency, so all the parallelism in the FPGA is being exploited to give the best performance. Note that, clock for clock, both lockstep and datapath modes perform better than the DSP (which used 9+N cycles for an N-tap FIR filter).

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 the results presented here the filter memory was stored in main memory. To calculate one filter output therefore, N-1 samples are read from filter memory in addition to the new data sample.

In functional unit mode, there is one read and one write (each 3 cycles) for each filter memory read. This total of 6 cycles is greater than the delay of a single multiply/accumulate, so any parallelism in the FPGA cannot be exploited. The solution used in the results presented here is to use the i960's burst accesses, which read or write 4 memory locations in 6 cycles. In functional unit mode then, 4 samples can be read and written in 12 cycles. In lockstep mode there is just one read per filter memory read. Again burst accesses can be used to read 4 samples in 6 cycles. One sample can be read every cycle in datapath mode to achieve the best performance. Figure 7 shows the results obtained for a 100-tap filter running on Riley.

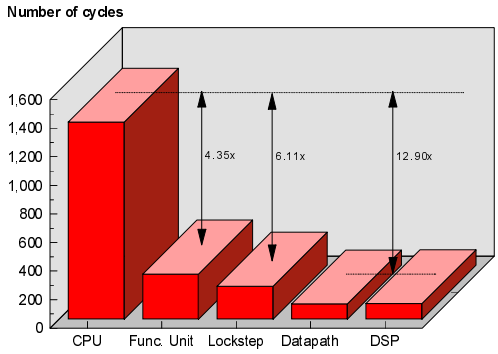

Figure 7 - Performance of a 100-tap filter

As with the results for the 5-tap filter, all code is in D-cache and all data is in I-cache. I960 burst accesses have been used in the results for functional unit and lockstep modes .

In comparing these results to the 5-tap filter, 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 Dynamic Programming

4.2.1 Background

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

4.2.2 The DPMatch algorithm

A Dynamic Programming Match (DPMatch) engine (Figure 8) evaluates the "edit distance" between two code strings.

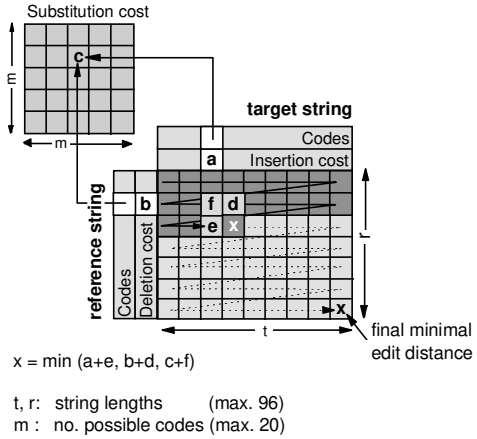

Figure 8 - DPMatch engine

Each cell in the "t" x "r" match-matrix represents the "edit disance" 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, represented by ("e").
2. **deleting** a particular reference string code. This distance is derived 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 which results in the last (lower right-hand) cell.

It should be noted, for completeness, that an initial row and column are included in the matrix to initialise the computation at the "boundaries" of the matrix. The initial row consists of a cell-by-cell accumulation of the insertion costs and the initial column consists of a cell-by-cell accumulation of the deletion costs.

More discussion of this algorithm can be found [8].

4.2.3 The DPMatch Partitions

The DPMatch algorithm was investigated in three different partitions which 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.

In terms of FPGA circuit complexity, the cell-by-cell represents the simplest possible partition and the full match represents the most complex. Each partition is explained below.

4.2.4 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.

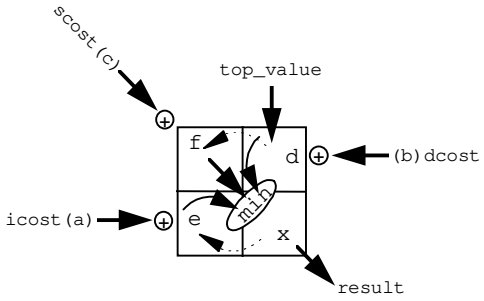

Figure 9 - Cell computational element

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 observation 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.

4.2.5 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 substituion costs (see Figure 10). The final result is read back after the loading of the final row.

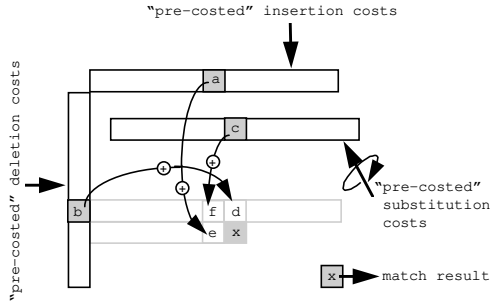

Figure 10 - Row by row computation

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 marshaling is much simplified because word-aligned blocks of operands were passed hence the interaction overhead per computation was reduced.
3. Some concurrency was obtained by the CPU marshalling for the next row of substitution costs while 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 simplfy the management of this only the capacity for one row of subtitution cost was provided. Use of the configuration RAM for the local stroage of substitution costs was not considered.
5. The bottleneck in achieving better performance is the marshaling and transfer of the substitution costs from the CPU to the FPGA.

4.2.6 Full Match

In this partition, as much of the algorithm as possible has been placed in the FPGA (the complete engine as shown in Figure 8). 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.

Though unused in the measurements, a further optimisation of the run-time interface allows any piece of a string (target or reference) to be loaded into the FPGA. In this way, assuming lexical ordering of the test set strings, large amounts of replicated information from one string to the next would not require reloading and common elements of previous computations could be reused.

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 however that the substitution costs are now held in the configuration memory on Riley not in the FPGA. This explains 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 and 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 cycles as opposed to 3) and the FPGA has sole access to this memory i.e. it does not have to arbitrate with the CPU.
3. CPU/FPGA 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.

4.2.7 DPMatch conclusions

Figure 11 summarises the performance of the CPU-only version of the DPMatch compared with that of the three partitions.

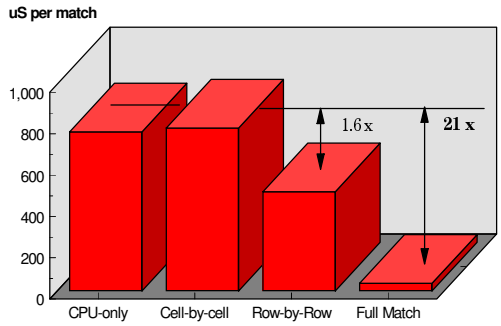

Figure 11 - DPMatch performance

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 highly parallelised data access and careful design of the control state machine.

5. CONCLUSIONS

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 the algorithm to increase the rate of computation over that available in the CPU. The FPGA can also be "bitwidth 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 which 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 the overhead of passing data via the CPU register file. 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 then provide a model for the synthesis of CPU/FPGA systems. They direct the process of partitioning a particular algorithm. Although this process is a manual one at the moment, we expect the architectures and techniques discussed in this paper to form the basis of future tools which will help automate this process. Research at HPLabs Bristol is targeted at providing these design tools.

ACKNOWLEDGEMENTS

We wish to acknowledge the contribution by the other members of the Appliance Computing Department in HPLabs Bristol, who reviewed this paper.

REFERENCES

[1] W.Sharpe, D.M.McCarthy, "Why can't hardware be more like software?", IEE 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