On the Viability of FPGA-Based Integrated Coprocessors

Osama T. Albaharna[†] , Peter Y. K. Cheung, and Thomas J. Clarke

Information Engineering Section
Department of Electrical and Electronic Engineering
Imperial College of Science, Technology and Medicine
Exhibition Road, London, SW7-2BT, UK

Abstract

This paper examines the viability of using integrated programmable logic as a coprocessor to support a host CPU core. This adaptive coprocessor is compared to a VLIW machine in term of both die area occupied and performance. The parametric bounds necessary to justify the adoption of an FPGA-based coprocessor are established. An abstract Field Programmable Gate Array model is used to investigate the area and delay characteristics of arithmetic circuits implemented on FPGA architectures to determine the potential speedup of FPGA-based coprocessors.

Our analysis shows that integrated FPGA arrays are suitable as coprocessor platforms for realising algorithms that require only limited numbers of multiplication instructions. Inherent FPGA characteristics limit the data-path widths that can be supported efficiently for these applications. An FPGA-based adaptive coprocessor requires a large minimum die area before any advantage over a VLIW machine of a comparable size can be realised.

1. Introduction

The ever increasing spare transistor capacity has only been absorbed so far into a limited number of architectural features. Integrated programmable logic has emerged as one of the very few novel architectural ideas with the potential to exploit this abundant resource. A custom coprocessor can directly exploit the concurrency available in applications, algorithms, and code segments. An FPGA-based coprocessor can further adapt to any demands for special-purpose hardware by mapping an algorithm onto run-time configurable logic. These versatile "adaptive" coprocessors can be used to augment the instruction set of a core CPU or as special purpose custom computing engines. Real-time applications can also *swap* multiple functions and subroutines directly onto the reconfigurable hardware during execution [1]-[5].

[†] e.mail: a.osama@ic.ac.uk
   http://www.ee.ic.ac.uk/research/information/www/aosama/aosama.html

The adaptive coprocessor model challenges the more established general purpose techniques that exploit fine-grain instruction level concurrency. We ask, *Under what architectural conditions can the integration of a core CPU and an FPGA-based coprocessor on a single die outperform the possible alternative of using a Very Long Instruction Word engine (VLIW) on that same die area?*

This paper addresses this question through four stages. First, in Section 2, the cost and performance bounds of both computational models, the VLIW and the FPGA coprocessing, are examined and a set of critical parameters is determined. Section 3 describes the experimental methodology used to establish the characteristics of arithmetic computation on FPGAs and Section 4 summarises the results of this investigation. In Section 5, we explore the implications of these results on the achievable cost and performance limits of FPGA-based coprocessors. Finally, in Section 6 we apply the ideas and conclusions presented in earlier section to a typical computational example to determine its suitability for FPGA-based adaptive coprocessor implementation.

2. Computational Models

An adaptive coprocessor uses silicon real-estate to integrate more programmable logic. This can then be used to implement larger custom circuits or exploit more concurrency. On the other hand, a VLIW machine will use this same die area to increase the number of ALUs and execute more instructions per cycle. In this section, we examine the cost and performance of implementing an algorithm on both computational models organised as in Figure 1.
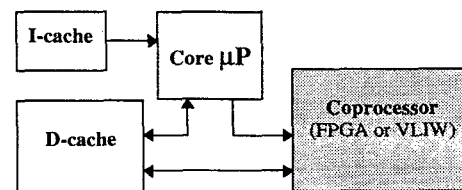

Figure 1. Target coprocessor system organisation.

2.1 FPGA-based coprocessor organisation

To achieve a high coprocessor throughput, we assume a pipelined implementation of all algorithms. This means that the performance of the FPGA-based coprocessor depend on the cycle time of the pipeline (t^c_{fpga}), the number of iterations the circuit is used (N_{fpga}), the number of concurrent copies of the circuit mapped onto the FPGA (k_{fpga}), and the number of cycles needed to fill the pipeline (c^{fill}_{fpga}). The total number of cycles is then

$$T_{fpga} = \left[\frac{N_{fpga}}{k_{fpga}} + c^{fill}_{fpga} \right] \times t^c_{fpga}$$

The area cost is the sum of the areas for all arithmetic nodes in a design. We assume integer nodes and floating-point nodes are used. All other operator nodes are expressed as a percentage (c_{fpga}) of the area. If a^i_{fpgs} is the area of node type i and n^i_{fpga} the number of nodes of type i used in the circuit, the cost of a circuit implemented on an adaptive coprocessor can be expressed as,

$$A_{circuit} = \left(1 + c_{fpga} \right) \times k_{fpga}$$

$$\times \left[\sum_{int-i} \left(a^{int-i}_{fpga} \times n^{int-i}_{fpga} \right) + \sum_{fp-i} \left(a^{fp-i}_{fpga} \times n^{fp-i}_{fpga} \right) \right]$$

2.2 VLIW machine organisation

We assume the VLIW utilises integer and floating-point ALU units rather than single operation functional modules and that they constitute most of its area. All other area is expressed as a percentage (c_{vliw}) of the total area. If a^i_{vliw} is the area of a function node of type i and n^i_{vliw} the number of nodes of type i used, the cost of a VLIW machine can be expressed as,

$$A_{vliw} = \left(1 + c_{vliw} \right)$$

$$\times \left[\left(a^{int-alu}_{vliw} \times n^{int-alu}_{vliw} \right) + \left(a^{fp-alu}_{vliw} \times n^{fp-alu}_{vliw} \right) \right]$$

In addition to available resources, the performance of a VLIW machine is limited by two types of dependencies [6]. The data dependencies within an iteration and the ones between iterations. A VLIW program can be viewed as a dependence graph, as in Figure 4, which must be repeated N_{vliw} times. Data and iteration dependencies are represented by the bold and dashed edges respectively. Since our VLIW processor model uses piplined ALUs, each node, or operation, in the graph takes a single time unit to execute (t^c_{vliw}). The iteration distance, attached to dashed edges, is the number of loop iterations after

issuance of S_i that S_j can begin execution. The number of time units it takes to execute a cycle within a dependency graph (δ_c), given maximum resources, is the sum of all nodes along this cycle path. The number of iterations (λ_c) it takes the pattern in a cycle to repeat execution is the sum of all iteration distances along this cycle's dependency path. Therefore N_{vliw}/λ_c repetitions of a given cycle will be executed requiring $\delta_c \times (N_{vliw}/\lambda_c)$ cycles.

The minimum time to execute the whole loop is $\max[\delta_c(N_{vliw}/\lambda_c)t^c_{vliw}] = \rho_{crit} \cdot N_{vliw} \cdot t^c_{vliw}$ where ρ_{crit} is the critical iteration period bound. Using software pipelining [7] and other advanced compiler transformations, it is possible to overlap the execution of several different iterations of the loop. If k_{vliw} iteration can be unrolled and then scheduled, the iteration interval t_{iik} is the time units needed to execute an entire iteration of k unrolled loops. It must satisfy both types of data dependencies as well as resources dependencies. If q^i_k is the number of operations a resource of type i must be used in k_{vliw} iterations we can estimate lower bounds on the iteration interval and the maximum VLIW performance as follows:

$$T_{vliw} = \left[\frac{N_{vliw}}{k_{vliw}} \times t_{iik} + c^{fill}_{vliw} \right] \times t^c_{vliw}$$

$$t_{iik} \geq \max\left[t_{iik} (resources), t_{iik} (dependence) \right]$$

$$t_{iik} (resources) \geq \max\left[\left\lceil q^i_k / n^i_{vliw} \right\rceil \right]$$

$$t_{iik} (dependence) \geq \max\left[\left(\delta_c / \lambda_c \right) \right]$$

Although the problem of finding an optimal schedule using software pipelining is NP-complete, it has been shown that near optimal results can often be obtained for loops with both intra- and inter-iteration data dependencies. It has also been shown that hierarchical reduction allows software pipelining to be applied to complex loops containing conditional statements. Program restructuring using loop transformations and optimising data locality using space tiling techniques can also be applied to increase both the fine-grain and coarse-grain parallelism available in nested loops.

2.3 Effects of memory access

The power of a custom circuit often lies in its use of a custom address generator to reduce explicit address calculation instructions. This option can be successfully exploited by both coprocessor models, the FPGA-based and the VLIW. Furthermore, both models can gain from customising memory access to fit the data bit width. We

expect these and other memory access optimisation techniques to produce equivalent benefits for both models and are therefore not directly included in the analysis.

2.4 Best-case comparative analysis

We can now compare the performance of both models, neglecting the pipeline fill cycles, by determining the speedup:

$$SU_{fpga} = \frac{T_{vliw}}{T_{fpga}} = \frac{k_{fpga}}{k_{vliw}} \times t_{iik} \times \frac{t^c_{vliw}}{t^c_{fpga}} = \frac{k_{fpga}}{k_{vliw}} \times \frac{t_{iik}}{\Delta} \quad \textbf{Eq(1)}$$

The speedup is effected by the number of concurrent copies of the circuit (k_{fpga}) mapped onto the FPGA. Since the areas of both models have to be the same, we can determine k_{fpga} in term of the equivalent number of integer ALUs as follows:

$$k_{fpga} = \frac{n^{int-alu}_{vliw} + \alpha \times n^{fp-alu}_{vliw}}{\sum_{int-i}\left(\Omega^{int-i} \times n^{int-i}_{fpga}\right) + \sum_{fp-i}\left(\alpha \times \Omega^{fp-i} \times n^{fp-i}_{fpga}\right)}$$

where $\Omega^{int-i} = \dfrac{a^{int-i}_{fpga}}{a^{int-alu}_{vliw}}$, $\Omega^{fp-i} = \dfrac{a^{fp-i}_{fpga}}{a^{fp-alu}_{vliw}}$, and $\alpha = \dfrac{a^{fp-i}_{vliw}}{a^{int-alu}_{vliw}}$

Using t_{iik}(resources), t_{iik}(dependence), and the speedup equation we can determine the conditions for which an FPGA-based coprocessor is virtually guaranteed to have better performance than a VLIW engine:

$$\rho_{crit} \geq \frac{k_{vliw}}{k_{fpga}} \times \Delta \quad \textbf{Eq(2)}$$

$$max\left[\left\lceil\frac{q^{int}_k}{n^{int-alu}_{vliw}}\right\rceil, \left\lceil\frac{q^{fp}_{vliw}}{n^{fp-alu}_{vliw}}\right\rceil\right] \geq \frac{k_{vliw}}{k_{fpga}} \times \Delta \quad \textbf{Eq(3)}$$

We can further simplify Eq(1), Eq(2), and Eq(3) by considering integer arithmetic only and substituting k_{fpga} to get:

$$SU = \frac{t_{iik}}{k_{vliw}} \times \frac{n^{int-alu}_{vliw}}{\sum_i n^{int-i}_{fpga}} \times \frac{1}{\Omega^{int}_{ave} \times \Delta} \quad \textbf{Eq(4)}$$

$$\rho_{crit} \geq \Omega^{int}_{ave} \times \Delta \times \frac{\sum_i n^{int-i}_{fpga}}{n^{int-alu}_{vliw}} \times k_{vliw} \quad \textbf{Eq(5)}$$

$$q^{int}_{k_{vliw}=1} \geq \Omega^{int}_{ave} \times \Delta \times \sum_i n^{int-i}_{fpga} \quad \textbf{Eq(6)}$$

We refer to Ω and Δ as the area and delay overheads, respectively, of a particular circuit implementation compared to an ALU's area and delay. They are inherent characteristics of the implementation platform (FPGA in this case) and limit its maximum achievable speedup. To sense how much speedup an adaptive coprocessor can deliver for a given fixed area and whether an algorithm has the necessary criterion that would make it suitable for adaptive coprocessor implementation, we need to estimate the minimum value of $(\Omega*\Delta)$ for arithmetic circuits implemented on FPGA platforms.

3. Experimental Methodology

To examine how efficiently FPGAs implement arithmetic circuits we need to eliminate technology and design variations and create an "even level" for comparison. This section describes how this even playing field is established. We first describe the cell architecture that is used throughout this paper and detail our models for estimating the area and delay of any FPGA cell. Then, our choices for arithmetic test circuits and implementation procedure are explained. In all discussion to follow, we consider only SRAM programmable FPGAs since only they provide the flexible platform necessary for field re-programmability.

3.1 FPGA cell architecture

We examined 15 different FPGA cell architectures that span the range of current research and commercial arrays. The function generators included a 2-input NAND gate, a 2-input Universal Logic Module (ULM.2) capable of implementing any of 16 2-input Boolean logic functions [8], look-up tables (LUT) of input sizes 3, 4, 5, and 6, and finally, the cell architectures of both the Altera FLEX-8000 [9] and the Xilinx XC5000 [10] which include specialised hardware to speedup carry propagation and wide gate implementation. All cells also incorporated D-type flip-flops (FF). The cells interconnection capabilities examined included extensive neighbour connections with 8, 12, and 16 possible neighbours, channelled 2D arrays with 4-neighbour connections, or channelled arrays with fully, or partially, connected clusters of cells similar to the Altera FLEX-8000 and the Xilinx XC5000 array architectures.

Of all these cell types, the 3-input LUT cell proved the best overall for arithmetic circuit implementations. We elect to use it and a 2D channelled array architecture for communication as the example cell throughout this paper. A neighbour interconnection only array may also be used and will give similar numerical results. The chosen cell is based on a look-up table design similar in functionality to other look-up table model proposals [11]. It incorporates 4-nearest neighbour connections as a vital way to reduce delay and improve routability. Figure 2 gives a conceptual

diagram of this cell. The routing channel width, W, is assumed to be the same for both the vertical and the horizontal channels. For LUTs with 3, 4, 5, and 6 inputs, the average minimum channel widths necessary for routing has been observed to be 9, 11, 11, and 12 respectively [11]. We therefore adopt a channel width of 9 for this model cell although the actual channel width should probably be slightly higher.
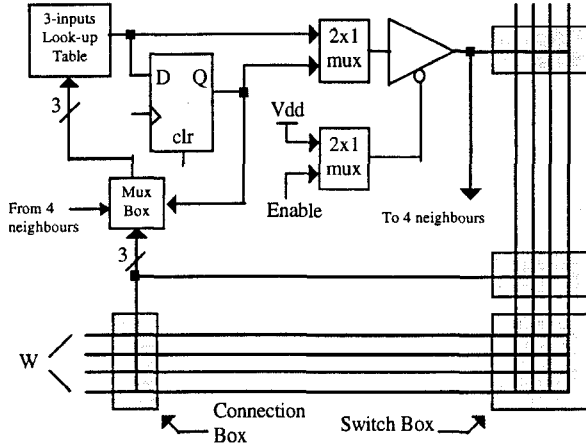

Figure 2. FPGA cell model with a 3-inputs look-up table as a function generator, direct north, south, east, and west neighbour connections, and global horizontal and vertical channel routing.

Limitations. We do not account for all the factors effecting the implementation and performance. Specifically, we leave issues such as external access, programming, testability, clock and control signals distribution, clock skew, and power consumption for future work. Of the global programming logic and network we only include the cost of the communication channel network and the number of SRAM configuration bits within a cell as part of the cost of the cell. These limitations bias Ω in favour of the FPGA-based coprocessor model.

3.2 Area measurement

The area of an FPGA cell is approximated using a transistor density coefficient metric (α) in μm^2/transistor. This density coefficient is dependent on the fabrication process technology, layout methodology, and the circuit logic structure used. It is obtained by averaging layout area per transistor over all cells available in a library or over samples or real designs. We assume a normalised function generator logic density coefficient of α_f, a configuration memory normalised density coefficient of α_m, and a routing pitch normalised density coefficient of α_r.

Figure 3a is a representative model of the total cell area showing also the routing pitch between the physical channel tracks. We assume that the Routing Configuration Memory (RCM) bits used for the channels' switch and connection boxes are distributed between the channel tracks as shown in Figure 2b. It is therefore reasonable to assume that α_m equals α_r. Other similar models also assume a distributed RCM [11]. The number of memory bits distributed within the channels (N_{switch}) depend on the connection and switch boxes. The connection box flexibility F_C is defined as the number of channel tracks each input and output can be connected to. The switch box flexibility F_S is defined as the number of possible tracks each incoming track can be connected to.
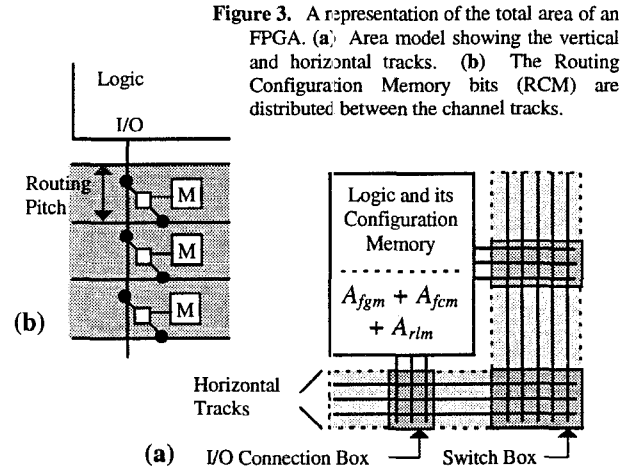
Figure 3. A representation of the total area of an FPGA. (a) Area model showing the vertical and horizontal tracks. (b) The Routing Configuration Memory bits (RCM) are distributed between the channel tracks.

It has been show [12] that F_C has to be greater than half the number of tracks for 100% routing completion to be possible. Additionally, only a small F_S value is needed to achieve a 100% routing completion. In our model, we choose $F_C = 0.75W$ and $F_S = 3$. The routing pitch is determined by a five-transistor SRAM bit (a_m) and a single pass-transistor PIP (a_p) and is defined as

$$r_{pitch} = \sqrt{\alpha_r \cdot (a_m + a_p)} = \sqrt{6 \cdot \alpha_r}$$

The FPGA cell is modelled as a square die area having the following characteristics:

$$A_{cell} = A_{func} + A_{mem} + A_{route}$$

$$A_{func} = \alpha_f \times (N_{fgm} + N_{rlm})$$

$$A_{mem} = (\alpha_m \cdot a_m) \times (N_{fcm} + N_{rcm})$$

$$A_{route} = \left[(r_{pitch}^2 \cdot W_h \cdot W_v)\right] + \left[r_{pitch} \cdot (X \cdot W_h + Y \cdot W_v)\right]$$

$where\ X \cdot Y = A_{func} + A_{mem}\ and\ X + (r_{pitch} \cdot W_h) = Y + (r_{pitch} \cdot W_v)$

$$A_{comm} = (\alpha_f \cdot N_{rlm}) + (\alpha_m \cdot a_m\ N_{rcm}) + A_{route}$$

209

where,

A_{cell} = the area of an FPGA cell
A_{func} = logic area used for function generation
A_{mem} = memory area used for configuration
A_{route} = the area of the routing channels within a cell
A_{comm} = the area of the cell used for communication
N_{fgm} = # of transistors used for function generation
N_{rlm} = # of transistors used for routing logic & muxs
N_{fcm} = # of memory bits for LUTs and control
N_{rcm} = # of mem. bits used for routing configuration
a_m = # of transistors in a memory bit = 5
W_h = # of routing tracks in each horizontal channel
W_v = # of routing tracks in each vertical channel

The total area of a circuit implementation depends on how the mapping from logic equations to FPGA cell functions is performed and how they are placed onto the cell array. If N_{cell} is the number of FPGA cells used to implement the circuits, the total circuit area is

$$A_{circuit} = N_{cell} \times A_{cell}.$$

3.3 Delay measurement

The delay of an FPGA cell is approximated using the method of "logical effort" proposed by Sutherland and Sproull [13] [14]. The method is based on a simple RC model for transistors and provide a first order approximation of a circuit delay. It defines τ as the actual time, for a fabrication process, that corresponds to a delay unit. The value of τ can be measured from the frequency of oscillation of a ring oscillator. For each type of logic gate, the method assigns delay unit values based on the topology of the circuit element, the difficulty that an element has in driving capacitive loads, and the parasitic capacitance exhibited by the gate. The delay of an ideal inverter that drives another identical inverter is the sum of a single unit delay (τ) and the parasitic delay value P_{inv}. Typically, for 3u CMOS process, $\tau = 0.5$ns and $P_{inv} = 0.6\tau$, while for 0.5u CMOS process, $\tau = 0.1$ns and $P_{inv} = 0.5\tau$. All other gate delays are measured relative to that of an ideal inverter.

Logical effort is used to arrive at delay value for each type of FPGA cell. Separate values are determined for each cell input to output, the set-up time, and the synchronous clock to output delay for each cell type. These delays also include the effects of internal fan-outs.

After a circuit is mapped onto an array, its delay depends on the number of cells (N_{depth}) along the longest path from an input to an output as well as the routing delay between these cells. The routing delay between neighbour

cells is accounted for by the explicit loading on that cell's output. The routing delay between non-neighbouring cells in a channelled array is more difficult to estimate specially without knowledge of the exact placement and routing information and the capacitive loading on each level due to the programmable routing switches along the path. The total execution time of a circuit, in τ units, can be determined as the sum of all delays along the longest path as follows:

$$T_{circuit} = \sum_{i=1}^{N_{depth}} \left(d_{ab}^i + d_{route}^i\right)$$

where d_{ab}^i is the delay, in τ units, between input node a and output node b of a cell at circuit depth level i, and d_{route}^i is the routing delay, in τ units as well, between the output of the cell at level i and the input of another cell at level $i+1$. In this investigation, we will assume d_{route}^i to be zero. This assumption will bias Δ in favor of the FPGA-based coprocessor model.

3.4 Implementation Procedure

We determine the number of cells needed to implement a circuit (N_{cells}) and the depth of the implementation (N_{depth}) by a structure preserving direct hand mapping from the original circuit designs. Automated mapping and routing results vary significantly with different tools and for different optimisation criterion. They also significantly alter the overall high level organisation resulting in low area utilisation even for regular circuits structures. We can also assume that with improved FPGA cell architectures, mapping, placement, and routing technologies, the routing structure is sufficient to complete the mapped network interconnections and give a very high array utilisation.

The results will therefore provide a lower bound on the cost and performance of different implementations which is exactly what we are looking for. Different designs are compared based on their implementation efficiency defined as the area times delay product 'AT', or cost*performance, for that circuit. The less 'AT' is, the more efficient is the implementation.

3.5 Choice of arithmetic circuits

We mapped 10 different integer addition circuit designs representing several delay and area optimisation techniques. They included, serial, carry-ripple, carry-skip, several one-level and two-levels carry-lookahead, conditional-sum, carry-select, and pyramid adders. For integer multipliers, we only considered 2's complement multipliers with 1-bit Booth recoding. We also mapped 6

210

different multiplication circuit designs including serial, sequential, sequential with one-level and two-levels carry-save logic, parallel array, and bit systolic. For the sequential and array multipliers which require an adder, we tried all the integer adders above to determine the ones that produce the best 'AT' results. All integer circuits were examined for bit widths varying from 4-bits to 64-bits.

For floating-point numbers we implemented a subset of the 32-bit and 64-bit IEEE specification standard and mapped both addition and multiplication circuits. Not all options referred to in the standard were included. Particularly, we assumed truncation is performed after addition or multiplication rather than rounding to eliminate one normalisation step. Ten mantissa adders and six mantissa multipliers were tested corresponding to the types of integer units above. The exponent adders used were chosen to minimise the 'AT' value for that design. Finally, we also implemented a pipelined version of those integer and floating-point circuits above that are actually pipelineable without incurring any increase in area cost.

Limitations. We did not include the control-path in either the cost or performance calculations. Furthermore, we did not examine any division algorithms or combination functions such as multiply-accumulate. We also restricted the investigation by not considering table-lookup techniques, distributed arithmetic, or the potential of other number systems.

4. Implementation Efficiency Gap

In an earlier study [15], we demonstrated that there is a large implementation efficiency gap between FPGA and VLSI platforms. We showed that a system implemented on FPGAs will require as much as 100 times more die area than a custom VLSI implementation and would be about 10 times slower. The result demonstrates how much worse an FPGA implementation is compared to a VLSI implementation. In this section we estimate the implementation efficiency gap between an FPGA platform and a programmable ALU as defined in section 2. The ALU parameters we use are based on estimates for the area and delay of complementary logic CMOS circuits which represent upper bounds on VLSI implementation cost and performance. Therefore, Ω and Δ, defined as the ratios of the area and delay results respectively of an FPGA-based implementation to those of an ALU, represent lower bounds while actual area and delay overheads could certainly be much higher. We will now compare the efficiency of pipelined, and non-pipelined, FPGA-based arithmetic circuits to that of ALUs.

We select two pipelined integer ALUs, 32-bit and 64-bit, capable of addition, subtraction, multiplication, shift to the left, arithmetic shift to the right, and finally, logical AND, OR, and NOT operations. They use a fast array multiplier to achieve small multiplication delays at the expense of high die area. We also select two pipelined floating-point ALUs, 32-bit and 64-bit, capable of addition, subtraction, and multiplication which also use an array multiplier as their mantissa multiplier/adder. For integer ALUs, we assume that addition takes a single cycle while multiplication needs 2 cycles on a 32-bit ALU, and 3 cycles on a 64-bit ALU. For the floating-point ALUs, we assume 4 execute stages for a 32-bit FP ALU and 6 stages for a 64-bit ALU. The result of the comparison is given in Table I which gives the 'best case' results for both pipelined and non-pipelined FPGA implementations. Note that multiplying Ω and Δ in the table may not correspond to the $\Omega*\Delta$ entry because all results have been rounded.

By limiting pipelined integer circuits to only those pipelinable without additional area, multiplication is restricted to array multipliers which have high area costs. Similarly, floating-point multiplication is not able to make use of the compactness of sequential multipliers and incurs a large area cost as well. The effects of these restrictions can be observed in Table I. The first thing to notice is that pipelined integer addition implementation give reasonably efficient results even though the area overhead is still large. Of course, this is an improvement over the non-pipelined results, and is mostly due to the low cycle time of the FPGA implementations and, subsequently, the low delay overhead. This suggests that addition on FPGAs should be pipelined.

The second observation is that pipelined integer multiplication efficiency is better than the non-piplined case but at the expense of very high area cost, up to 60 times from a maximum of 4.1 times in the non-piplined case. Unlike addition, however, this result suggest that multiplication is not suitable at all for FPGA implementation. If required, however, but with real-estate at a premium, as in our case, multiplication should not be pipelined so as to save on area. Finally, the overhead for floating-point is much larger than that of integer implementation. The large overhead magnitude raises doubt regarding the usefulness of floating-point operations on FPGAs under any circumstances.

Serial Arithmetic. Serial arithmetic can deliver low area overheads for low operand width values. However, for larger widths, 32-bits and 64-bits for example, the area overhead is still considerably high because the registers used in accumulating the result have to be included in the cost. It is also evident from our results that the low

Characterising FPGA-based Arithmetic Circuit Implementations

			32-bit ALU				64-bit ALU									
			Integer				FP	Integer								FP	
FPGA circuit bit widths			8	16	24	32	32	8	16	24	32	40	48	56	64	32	64
pipelined FPGA circuits	ADD	Ω x Δ	0.15	0.49	0.57	0.8	31.4	0.03	0.11	0.13	0.18	0.23	0.29	0.35	0.4	4.19	17.3
		area (Ω)	0.76	2.5	2.87	4.05	15.0	0.21	0.68	0.78	1.1	1.44	1.79	2.14	2.46	3.79	9.15
		delay (Δ)	0.2	0.2	0.2	0.2	2.1	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	1.63	1.89
	Mult	Ω x Δ	0.76	3.05	6.42	11.3	108	0.17	0.69	1.44	2.54	3.94	5.65	7.67	9.68	21.3	106
		Ω	3.87	15.5	32.6	57.3	51.5	1.05	4.22	8.88	15.6	24.3	34.8	47.2	59.6	13.0	56.3
		Δ	0.2	0.2	0.2	0.2	2.1	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	1.63	1.89
non-pipelined FPGA circuits	ADD	Ω x Δ	0.6	2.1	3.53	5.65	119	0.14	0.47	0.79	1.27	1.64	2.02	2.41	2.76	15.9	67.2
		area (Ω)	0.32	0.61	3.73	5.19	16.7	0.09	0.17	1.02	1.42	1.83	2.25	2.69	3.08	4.23	10.2
		delay (Δ)	1.88	3.45	0.95	1.09	7.13	1.55	2.85	0.78	0.9	0.9	0.9	0.9	0.9	5.54	6.56
	Mult	Ω x Δ	4.26	16.2	35.8	63.1	280	0.96	3.64	8.04	14.2	22.0	31.6	42.7	53.6	55.1	253
		Ω	1.07	2.07	3.08	4.08	14.2	0.29	0.57	0.84	1.11	1.38	1.66	2.37	2.67	3.59	8.37
		Δ	3.97	7.81	11.6	15.5	19.7	3.28	6.43	9.59	12.6	15.9	19.1	18.0	20.1	15.3	30.2

performance of serial designs make their use on FPGAs unacceptable for high throughput computations.

Summary. The results given here highlight the main problem with FPGA-based arithmetic circuits. They cannot compete with other programmable devices such as ALUs, at least at the circuit level. Pipelining can be used to bring the cycle time of an FPGA implementation close to the delay of an ALU to reduce the delay overhead. But, the intractable problem is the area overhead. Consider, from Table I, that m arithmetic nodes of type and data width i have total die area equivalent to $m*\Omega_i$ ALUs. Serial and sequential arithmetic can reduce the implementation cost but at the expense of decreasing the performance and increasing the delay overhead. Since on-chip real-estate is valuable and the area cost of any FPGA implementation is already high, FPGA design trade-offs should be optimised for area. Indeed, neither floating-point arithmetic nor integer multiplication have shown any implementation attributes favourable to FPGA platforms. For the case of integer multiplication, this was also reported by a study on commercial FPGAs [16].

5. Viability of Adaptive Coprocessors

From the results of the previous section and the 6 numbered equations in Section 2, we can evaluate the viability of the adaptive coprocessor proposal. For clarity, we assume either integer or floating-point operations only. Table II examines the critical iteration period bound (ρ_{crit}) and the minimum number of instructions in an unrolled iteration ($q^i_{k=1}$) as given by Eq(5) and Eq(6). These are the conditions necessary for a VLIW coprocessor machine to have better performance than an FPGA-based coprocessor; i.e, speedup is larger than one. Values of both parameters

are determined for a total coprocessor die area equivalent to approximately 5M and 20M transistors and assuming a hypothetical algorithm requiring total node count $\Sigma n^i_{fpga} \cong$ 10 with a reasonable value of loop unrolling $k_{vliw}=10$.

We examine four cases of the algorithm. All nodes are addition operations, all are multiplication operations, 9 are adds and one is a multiply, and finally, 5 are adds and 5 are multiply.

Total area vs. speedup. Although a die area equivalent to 5M transistors may seem to be large for such a small number of function nodes, already some data-path widths, blanked out in Table II, will either not fit onto an FPGA of this size or will not achieve better performance than the VLIW model under any ρ_{crit} and $q^i_{k=1}$ conditions. Also, the smaller die area forces the use of the more compact, but slower, non pipelined circuits giving worse results than the larger die area. On the other hand, from Eq(1) and Eq(4), as the area available, or number of ALUs, increase, the iteration period (t_{iik}) decrease reducing the potential speedup.

Minimum instructions per iteration ($q^i_{k=1}$). Considering the small die area first, rows 1 and 2 show that the number of compiled VLIW instructions per iteration, before loop unrolling, that would guarantee FPGA superiority is relatively small. It can be easily satisfied by many algorithms that only require a limited number of multipliers. Rows 3 and 4 show that as the number of multiply operations increase, the number of instructions per iteration has to be very high for an FPGA to beat a VLIW engine particularly for data bit widths larger than 16-bits. This suggests that highly concurrent algorithms of limited data-path widths, such as some image processing algorithms are suited for implementation on integrated

TABLE II

Conditions determining the implementation viability of an FPGA-based Coprocessor

For $\Sigma n^i_{fpga}=10$ and $k_{vliw}=10$, an FPGA-based coprocessor is better if			32-bit ALU Integer				32-bit ALU FP	64-bit ALU Integer								64-bit ALU FP		
			8	16	24	32	32	8	16	24	32	40	48	56	64	32	64	
$q_l \geq$	Small die area	all Adds	2	5	6	9		1	2	2	2	3	3	4	4			1
		9 adds and 1 mult.	3	8	12	19		1	2	3	5	6	9	11	14			2
		5 adds and 5 mult.	5	18	346	631		2	4	78	140	225	330	406				3
		all Mults	8	162	358	633		2	37	81	140	220	318	427	537			4
	Large die area	all Adds	2	5	6	9	315	1	2	2	2	3	3	4	4	159	670	5
		9 adds and 1 mult.	3	8	12	19	392	1	2	3	5	6	9	11	14	193	1940	6
		5 adds and 5 mult.	5	18	36	62	699	2	4	8	14	225	330	406	516	596	2805	7
		all Mults	8	31	66	633	2798	2	7	15	140	220	318	427	537	550	2528	8
$\rho_{crit} \geq$	Small die area	all Adds	1	1	1	1		1	1	1	1	1	2	2	2			9
		9 adds and 1 mult.	1	1	2	2		1	1	1	2	3	4	4	5			10
		5 adds and 5 mult.	1	2	33	60		1	2	29	52	83	122	151				11
		all Mults	1	16	34	61		1	14	30	52	82	118	158	199			12
	Large die area	all Adds	1	1	1	1	8	1	1	1	1	1	1	1	1	15	62	13
		9 adds and 1 mult.	1	1	1	1	10	1	1	1	1	1	1	1	2	18	180	14
		5 adds and 5 mult.	1	1	1	2	17	1	1	1	2	21	31	38	48	31	260	15
		all Mults	1	1	2	16	67	1	1	2	13	21	30	40	50	51	234	16

FPGA arrays. Rows 5 to 8 show that as the total die area increase, bigger and faster multipliers can be used for even larger operation widths (16 to 32 in this case) reducing the number of instructions per cycle for those data-path widths and allowing more applications to satisfy the instruction count condition.

Iteration period bound (ρ_{crit}). The critical iteration bound is the minimum time to execute a loop and is shown in rows 9 to 12 of Table II for the small die area. Similar observations to those of the q^i_k parameter can be made. Again, many algorithms can easily satisfy this condition and will therefore perform better on an FPGA-based coprocessor. Restrictions still have to be made to the number of multipliers used. If the number of multipliers required is large however, the condition can still be satisfied for small data-path widths. Furthermore, increasing the total area available for the coprocessor, rows 13 to 16, means that more multipliers can be accommodated and larger data-path widths can be used.

Summary. We have numerically demonstrated that FPGA arrays are suitable as coprocessor platforms for realising algorithms requiring a limited number of multipliers across a wide range of data-path widths particularly below 32-bits. From Eq(2) and Eq(3), or Eq(5) and Eq(6) for integer numbers only, we can deduce that as the silicon area available increase, the algorithm domains for which the FPGA-based coprocessor is better than a VLIW machine improve in favour of the adaptive coprocessor. The programmable array would be able to handle algorithms with more function nodes and more multipliers.

For any given algorithm, maximum speedup is achieved with lower die area sizes above an absolute minimum area. The FPGA, however, would still not be able to effectively deal with algorithms dominated by floating-point operations.

Finally, note that the parametric limits reported in this section regarding the suitability of one algorithm or architecture over another are indicative of general trends for a broad range of operand width values and arithmetic functions and should not be interpreted as precise statements of comparison for the specific models and circuits investigated.

6 Example

Consider the implementation of the following code fragment from a hydrodynamic simulation (1st Loop of the Livermore Kernels):

```
for (j = 0; j<n; j++)
  { X[j] = q + (Y[j] * ((r * Z[j+10]) + (t * Z[j+11])))}
```

A pipelined custom implementation of this code segment requires three floating-point multipliers, two floating-point adders, and three integer adders. Assume 64-bit operations. Figure 4 gives the compiled generic instructions (before scheduling and allocation) and their dependency graph. The speedup ranges gained by using an FPGA-based coprocessor model over a VLIW model are summarised in Table III. We apply the static characteristic of FPGA implementations (Table I) to derive the best possible speedup ranges.

213

```
        R1 ← adr(x)              ; get base address of array x
        R2 ← adr(y)              ; get base address of array y
        R3 ← adr(z)              ; get base address of array z
        R4 ← n                   ; get number of iterations
        F1 ← q                   ; get floating number q
        F2 ← r                   ; get floating number r
        F3 ← t                   ; get floating number t

   Start:   R5 ← R0              ; k = 0
        F4 ← M[ 10 + R3 ]        ; variable z[10]
        F5 ← M[ 11 + R3 ]        ; variable z[11]
        F10 ← M[ R2 ]            ; variable y[0]
        R6 ← R3 + 12             ; base address for z[11]   ; z[12]

S1   Itr:   R7 ← R6 + R5(k-1)    ; base addr for var z[11 + k]; z[12 + k]
S2          F6 ← M[ R7 ]         ; load floating var. z[11 + k]; z[12 + k]

S3          F7 ← F3 * F5(k-1)    ; t * z[11 + k]
S4          F8 ← F2 * F4(k-1)    ; r * z[11 + k - 1]

S5          F9 ← F8 + F7         ; ( r * z[k+10] ) + ( t * z[k+11]   )

S6          F4 ← F5(k-1)         ; F4 = z[11 + k]
S6x         F5 ← F6              ; F5 = z[11 + k + 1 ]

S7          R8 ← R2 + R5(k-1)    ; base address for variable y[k]
S8          F10 ← M[ R8 ]        ; load floating var. y[k]
S9          F11 ← F10(k-1) * F9  ; y[k] * ( r * z[k+10] ) + ( t * z[k+11] )

S10         R9 ← R1 + R5(k-1)    ; base address for variable x[k]
S11         F12 ← F1 + F11       ; x[k] = q + S9
S12         M[ R9 ] ← F12        ; store floating var. x[k ]

S13         R5 ← R5(k-1) + 1     ; k++
S14  End:   if (R4 != R5) PC ← Itr   ; branch to itr if k < n
```
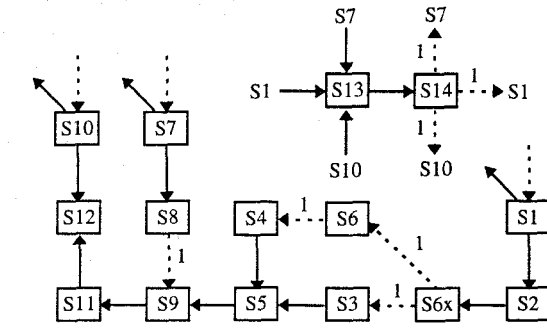

Figure 4. Dependence graph for the 1st Livermore Loop and a generic VLIW compilation.

k_{vliw}		1	2	4	6	25
number of operations (q^i_k)	int add	4	4	4	4	4
	fp add	4	7	13	19	76
	fp mult	3	6	12	18	75
	brn-jmp	1	1	1	1	1
	I/O	3	6	12	18	75
Maximum ALU resources		12	18	30	42	156
Iteration period bound (ρ_{crit})		3	3	3	3	3
$A_{circuit}$ (k_{fpga} = 1) in equ. ALUs		min = 51 & max = 195				
SU_{fpga}	max area	1.59	0.8	0.4	0.27	0.06
	min. area	0.1	0.05	0.025	0.017	0.004

equivalent to the maximum and minimum areas needed to implement the function on a programmable array.

Table III shows that the best speedup is 1.59 assuming the VLIW machine uses the maximum resources and that the VLIW compiler does not utilise loop unrolling. Comparing $A_{circuit}$ to A_{vliw} we see that even with $k_{fpga} = 1$, the adaptive coprocessor will use enough area to allow the VLIW machine maximum resources and to unroll about 25 loops per iteration. With these resources, the VLIW machine has better performance than the adaptive coprocessor. In fact the only time that the FPGA may have any advantage is if no loop unrolling is done at all which is an unrealistic restriction on the VLIW machine. Finally, it is possible to deduce that this code fragment, which is dominated by floating-point operations, is not suitable for FPGA-based custom computing because the VLIW is limited by $\rho_{crit} = 3$ which is much smaller than the entries in Table II suggest for floating-point circuits with a similar number of operations.

Integer arithmetic. The hydrodynamic simulation example requires both integer and floating-point resources. However, many algorithms need only use integer arithmetic. In Table IV we re-examine the code segment using only integer arithmetic for both machines. We restrict the VLIW to 32-bit or 64-bit ALUs while the adaptive coprocessor implementation can be as small as an 8-bit data-path. The first row gives the area of the custom implementation of a single iteration. The second row is the number of custom iterations needed to make available enough area for maximum VLIW resource needs. The third row is the number of loops that can be unrolled by the VLIW to expose as much concurrency as there is area. Now, the speedup ranges obtained are much higher than in Table III. The FPGA-based model can maintain better performance up to 24-bits compared to a 32-bit VLIW machine and up to 48-bits compared to a 64-bit VLIW

The first group in Table III include 5 rows that show the number of operations for each type of resource required by the code segment as k_{vliw} loops are unrolled. The second group show two parameters. First, the maximum number of ALUs that could be utilised by the VLIW machine for this code segment. Second, the minimum iteration period ρ_{crit}. The third group gives the area of the custom circuit implementation (FPGA-based coprocessor) normalised to that of an ALU. The minimum and maximum values, corresponding to non-pipelined and pipelined implementations respectively, are obtained using the data in Table I. Note that the maximum area, which will also give the minimum FPGA delay, is larger than the maximum VLIW resources required. Finally, the fourth group is the speedup results for two possible die areas

TABLE IV

Implementation Results for 1st Livermore Loop Assuming Integer Operations Only

FPGA bit widths	32-bit ALU Integer				64-bit ALU Integer							
	8	16	24	32	8	16	24	32	40	48	56	64
A_{fpga} in equivalent # of ALUs ($k_{fpga}=1$)	15	59	112	192	3	13	30	52	80	113	152	191
max k_{fpga} (A_{vliw}[max] $\cong k_{fpga} \times A_{circuit}$[min])	1	1	1	1	5	2	1	1	1	1	1	1
max k_{vliw} (given max k_{fpga})	1	7	14	25	1	3	3	6	9	14	20	25
Speedup using — max VLIW resources	15	2.14	1.07	0.6	94	12.5	6.25	3.13	2.08	1.34	0.94	0.75
Speedup using — min VLIW resources	30	4.29	2.14	1.2	188	25	12.5	6.25	4.17	2.68	1.88	1.5

engine. We can see that high speedups are obtained for the same data-path widths predicted in Table II.

7. Conclusions

We have investigated the cost and performance of two coprocessor models; a VLIW engine and an FPGA-based coprocessor. We also examined the characteristics and parameters that determine the suitability of one model over the other. We established best case numerical limits for these parameters beyond which one machine would be more suited than the other to a specific application domain.

We conclude that the inherent characteristics of FPGAs and arithmetic circuits limit the algorithms suitable for implementation on an integrated adaptive coprocessor to those with a limited number of multipliers and with no floating-point operations. When multipliers are an essential part of the algorithm, only small data-path widths can be expected to produce any performance gains over a VLIW machine. The range of algorithms that would benefit from an FPGA-based coprocessor increase as the silicon area available increase. However, for each algorithm, the implementation efficiency gap between the FPGA platform and ALUs necessitates that a large minimum die area is used, equivalent to tens of ALU areas, before positive speedup in favour of the integrated adaptive coprocessor can be achieved. Maximum speedup is obtained at or close to this absolute minimum area requirement.

References

[1] P. Athanas and H. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *IEEE Computer*, Vol. 26, No. 3, pp. 11-18, March 1993.

[2] J. P. Gray and T. A. Kean, "Configurable Hardware: A New Paradigm for Computation," in *Advanced Research in VLSI*, Ed. C. L. Seitz, MIT Press, pp. 279-295, Proc. of the 1989 Decennial Caltech Conference, March 1989.

[3] N. Hastie and R. Cliff, "The Implementation of Hardware Subroutines on Field Programmable Gate Arrays," *Proc. Custom Integrated Circuits Conf.*, pp. 31.4.1-31.4.4, 1990.

[4] Alberto Sangiovanni-Vincentelli, "Some Considerations on Field Programmable Gate Arrays and Their Impact on System Design," *Proc. of the 2nd Int'l Workshop on Field-Programmbale Logic and Applications*, FPL'92 (Lecture Notes in Computer Science # 705, H. Grunbacher and R. Hartenstein, Ed.), Vienna, pp. 26-34, August/September 1992.

[5] S. Trimberger, "A Reprogrammable Gate Array and Applications," *Proc. of the IEEE*, Vol. 81, No. 7, pp. 1030-1041, July 1993.

[6] David J. Lilja, "Exploiting the Parallelism Available in Loops," *IEEE Computer*, Vol. 27, No. 2, pp. 12-26, February 1994.

[7] Monica Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. of the ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pp. 318-328, 1988.

[8] X. Chen and S. L. Hurst, "A Comparison of Universal-Logic-Module Realizations and Their Application in the Synthesis of Combinatorial and Sequential Logic Networks," *IEEE Transactions on Computers*, Vol. C-31, No. 2, pp. 140-147, February 1982.

[9] Altera Corporation, *FLEX 8000 Handbook*, May 1994.

[10] Xilinx Inc, *XC5000 Logic Cell Array Family*, Advanced Information (v2.0), February 1995.

[11] Jonathan Rose, Robert Francis, David Lewis, and Paul Chow, "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency," *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 5, pp. 1217-1225, October 1990.

[12] Jonathan Rose and Stephen Brown, "Flexibility of Interconnection Structures for Field-Programmable Gate Arrays," *IEEE Journal of Solid-State Circuits*, Vol. 26, No. 3, pp. 277-282, March 1991.

[13] Ivan E. Sutherland and Robert F. Sproull, *Logical Effort: Designing Fast MOS Circuits*, Internal Report SSA #7893, Sutherland, Sproull, & Associates Inc., 1990.

[14] Ivan E. Sutherland and Robert F. Sproull, "Logical Effort: Designing for Speed on the Back of an Envelope," in *Advanced Research in VLSI*, Proc. of the 1991 University of California at Santa Cruz Conference, Ed. by Carlo H. Sequin, pp 1-16, MIT Press, 1991.

[15] Osama T. Albaharna, Peter Y.K. Cheung, and Thomas J. Clarke, "Area & Time Limitations of FPGA-based Virtual Hardware," *Proc. of the IEEE Int'l Conf. on Computer Design: VLSI in Computers & Processors*, ICCD'94, pp. 184-189, October 1994.

[16] Russell J. Peterson and Brad L. Hutchings, "An Assessment of the Suitability of FPGA-Based Systems for Use in Digital Signal Processing," *Proc. of the 5nd Int'l Workshop on Field-Programmbale Logic and Applications*, FPL'95 (Lecture Notes in Computer Science # 975, Oxford, pp. 293-302, August/September 1995.