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.

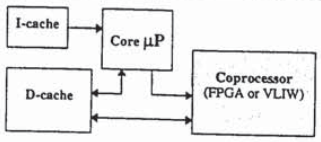

Figure 1. Target coprocessor system organisation.

0-8186-7548-9/96 $05.00 © 1996 IEEE

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_{fpga} 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 $\delta_c \; x \; (N_{vliw}/\lambda_c)$ cycles.

The minimum time to execute the whole loop is $\max\{\delta_c(N_{vliw}/\lambda_c)t^c_{vliw}\} = \rho_{crit} N_{vliw} t^c_{vliw}$ 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:

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

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

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:

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

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

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

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

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

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

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 (Ω*Δ) 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.

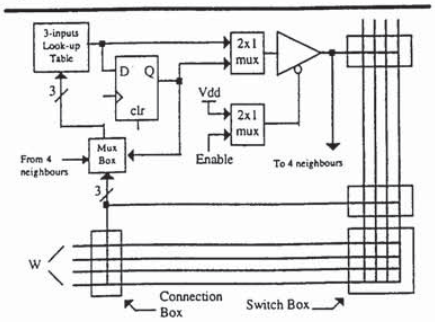

Figure 2. FPGA cell model with a 3-inputs look-up table as a function generator, direct north, south, east, and west neighbour connections, and global horizontal and vertical channel routing.

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 ($N_{rswitch}$) 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.

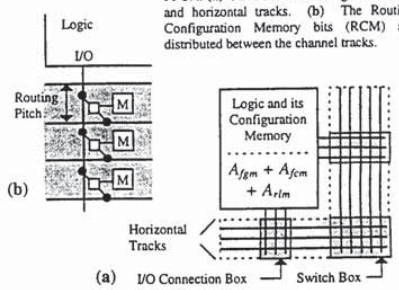

Figure 3. A representation of the total area of an FPGA. (a) Area model showing the vertical and horizontal tracks. (b) The Routing Configuration Memory bits (RCM) are distributed between the channel tracks.

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 $X + (r_{pitch} \cdot W_h) = Y + (r_{pitch} \cdot W_v)$

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

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, τ = 0.5ns and $P_{inv} = 0.6\tau$, while for 0.5u CMOS process, τ = 0.1ns 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:

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

where d^i_{ab} is the delay, in τ units, between input node a and output node b of a cell at circuit depth level i, and d^i_{route} 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^i_{route} 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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.