Partition-based exploration for reconfigurable JPEG designs

Philip G. Potter, Wayne Luk, Peter Cheung
Imperial College, London
London, UK
{pgp,wl}@doc.ic.ac.uk, peter.cheung@ic.ac.uk

Abstract

This paper proposes a novel approach for design space exploration by characterizing hardware sharing based on the notion of a partition in set theory. Related designs with different degrees of hardware sharing can be captured concisely by a Hasse diagram, highlighting designs with shared building blocks. Hardware sharing can be implemented in various ways, such as component multiplexing, instruction-set processors, or run-time reconfiguration. We illustrate how the proposed approach can be applied to exploring the design space for FPGA implementations of JPEG image compression.

1 Introduction

Automated hardware design is often expressed as a design space exploration problem. Design space exploration is a particular type of multi-objective optimization problem where a decision vector $\mathbf{x}$ consisting of decision variables x_i is selected to optimize an objective vector $\mathbf{f}$ consisting of objective variables f_i [1]. In general, there may be more than one ideal solution as different objective variables are traded off against each other.

Before an effective design space exploration can happen, the decision variables must be chosen. However, it is not possible to define decision variables which contain the gamut of all possible designs. Examples of previous classifications in design space exploration have been restricted to one particular solution architecture as a result.

This paper attempts to classify the JPEG encoder design space in a way which includes diverse architectures. The aims of the research are to allow a wide variety of designs into the explored space, and to achieve a wide coverage of the objective space from small and slow to fast and large. In particular, we wish to include diverse solutions in structure such as MicroBlaze, a pipelined datapath, and a state machine within the design space classification.

2 Previous work

A number of authors have tried to tackle design space exploration previously. The Platune system [2] exploits independence of decision variables: for instance, reducing voltage reduces both power consumption and maximum frequency monotonically regardless of cache size or instruction encoding. This allows the design space to be pruned by separating independent variables into exponentially smaller subspaces.

Yiannacouras, Steffan and Rose developed a system called SPREE [7] to investigate the design of a soft processor on Altera field-programmable gate arrays (FPGAs). They used a set of benchmark programs to determine the effects of different architectural choices on performance (wall-clock time) and area. Choices included pipeline depth, hardware vs software multiplication, serial shifter vs hardware multiplier-based barrel shifter, and different pipeline lengths. They created a large number of designs which provide a range of solutions of different speeds and areas.

Mohanty et al used two tools, DESERT and HiPerE [3], in a three-phase approach to generate an optimal design based on user constraints. They used their system to target a system consisting of either a StrongARM, a MIPS, or both.

Pimentel et al modelled M-JPEG as a Kahn process network and explored different mappings from Kahn processes to architectural processing elements [4]. Their target architecture was a multiprocessor platform, and they analysed the resultant designs for architecture cost, processing time and power consumption.

Most of the previously mentioned systems represent the design space using a vector of decision variables, and all of them constrain the solution space to a particular architecture template. We propose a different

978-3-9810801-5-5/DATE09 © 2009 EDAA

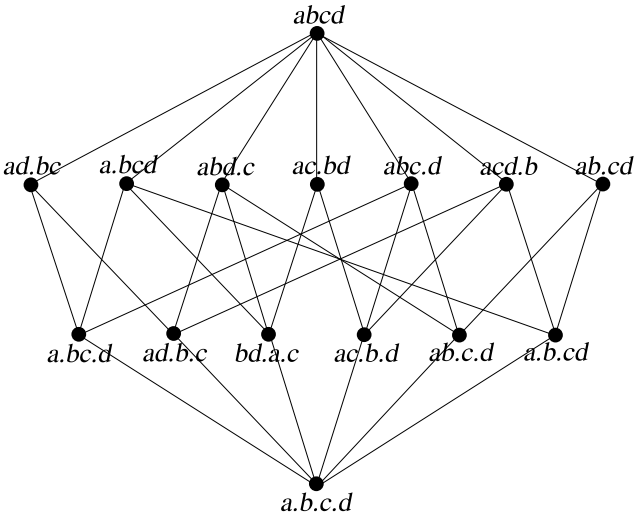


Figure 1. Hasse diagram showing "finer than" partial order of partitions

method of design characterization from those proposed before based on hardware sharing, in order to keep the solution space inclusive of all possible designs. Hardware sharing can take many forms, from simple time-multiplexing of a static resource, to using a soft processor such as MicroBlaze, or using runtime reconfiguration to change usage of hardware [6].

3 Partition-based characterisation

A *partition* of a set S is a set of subsets of S, such that each member of S belongs to exactly one subset. The subsets (or *blocks*) are collectively exhaustive and mutually exclusive with respect to S. For example, two partitions on the set $\{a, b, c, d\}$ are $\{\{a, b\}, \{c, d\}\}$ and $\{\{a, d\}, \{b\}, \{c\}\}$. For ease of notation, these partitions can also be denoted $ab.cd$ and $ad.b.c$.

Partitions can be partially ordered using a *finer than* relation: a partition P is finer than another partition Q if every block of P is a subset of some block of Q. Colloquially, P divides S into smaller blocks than Q does. A diagram of such a partial order is shown in figure 1; finer designs are towards the bottom. At any given 'height' on the diagram, all partitions consist of the same number of blocks; however, the elements have been divided up in different ways between the blocks.

A computational problem can be modelled as a set of tasks. A partition of the set of tasks is created, such that two tasks are placed in the same block in the partition if and only if they share hardware; for example, both use the same multiplier or divider unit on different clock cycles to calculate their results. As a result, each block within the partition is associated with a physical block of hardware in the design.

As one moves through the partition lattice in figure 1 from bottom to top, task-level parallelism decreases, and hardware sharing increases.

Consider an 8-tap finite impulse response (FIR) filter: a circuit which transforms a sequence of input samples x_i into a sequence of output samples y_i based on the following formula:

$$y_i := \sum_{k=1}^{8} a_i x_{i+k-1}$$

We consider each multiplication within the sum as a separate task, labelled from a to h. A pipelined datapath solution would contain 8 separate multipliers, so the partition would be $a.b.c.d.e.f.g.h$. A multiply-accumulator based solution would use only 1 multiplier, taking 8 cycles to produce the output for one sample. Its partition would be $abcdefgh$. A simple processor-based solution would also use only 1 multiplier and have the same partition. In between these extremes there are solutions with varying level of hardware sharing, represented by such partitions as $abcd.efgh$, $ab.cd.ef.gh$ and $abc.def.gh$ using 2, 4 and 3 multipliers respectively.

We use this classification as a method for driving the creation of FPGA designs. In our design approach, we first select classifications which we believe may provide useful designs, and then generate hardware which fits the classification. Currently the exploration is systematic but manual. The choice of which partitions to explore is left to the designer, and once they have been chosen, a number of designs for each partition are created.

The partitions chosen will often share blocks, and so the same subdesign of a block of one partition can be reused within others. Furthermore, the design for a block ab may be reused in the designs for blocks abc and abd; we present an example of this in section 4. Design reuse is a key part of this design process.

The analysis of composite designs for individual computational elements within the original graph is treated recursively as a new graph-partition problem until the underlying computational elements are primitives such as adders and multipliers.

There are many different types of hardware sharing, especially in the FPGA world. A simple way to share a multiplier is to place a multiplexer on the inputs; one can also use the multiplier as a functional unit in a processor, or use runtime partial reconfiguration to change the hardware design around that multiplier.

4 JPEG example

We apply our approach to a JPEG encoding problem. JPEG is a standard for the lossy compression of photographic or continuous-tone images. The process consists of four main stages: colour space conversion (typically from RGB to YUV), 2-D discrete cosine transform (DCT), quantization, and entropy coding.

Overview We divide the JPEG encoding computation into 13 modules: colour space conversion, labelled Y; six one-dimensional DCT modules, labelled from D_{1Y} to D_{2V}; and three quantization modules, labelled from Q_Y to Q_V. We select a number of partitions to choose designs from. The design choices reflect the nature of the design space. The partitions chosen are:

$$\begin{array}{cc} Y.D_{1P}.D_{2P}.Q_P & Y.D_{1S}.D_{2S}.Q_P \\ Y.D_{1P}.D_{2P}.Q_S & Y.D_{1S}.D_{2S}.Q_S \\ YQ.D_{1P}.D_{2P} & YQ.D_{1S}.D_{2S} \end{array}$$

where $D_{1S} = D_{Y1}D_{U1}D_{V1}$, $Q_S = Q_Y Q_U Q_V$, $D_{1P} = D_{Y1}.D_{U1}.D_{V1}$, and $Q_P = Q_Y.Q_U.Q_V$.

The finest-grain partition, $Y.D_{1P}.D_{2P}.Q_P$, is chosen in order to demonstrate the fastest, most parallel design. From this, we consider which elements would most readily share hardware. The DCT blocks are clear candidates to share hardware between themselves, because they each have identical structure; similarly for the quantize blocks. The quantize block seems too computationally insignificant to necessarily require its own hardware block, and so for some designs we combined it with YUV to produce the YQ block.

The blocks needed for implementation are therefore Y, D_{1S}, D_{1Y}, Q_Y, Q_S and YQ. Symmetry between different components allows us to reuse the same designs for Y, U and V streams.

Implementation of Y block We use the RGB-to-YUV definition from JFIF, the JPEG File Interchange Format. Colour space conversion is a simple matrix multiply operation:

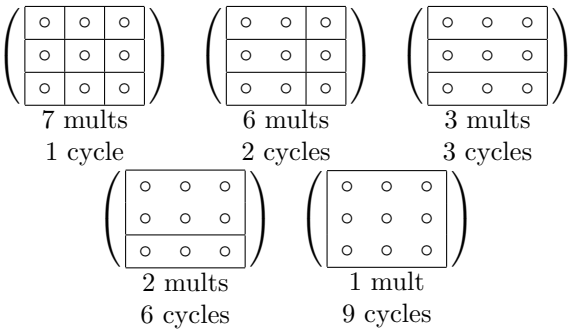
$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & +0.5 \\ 0.5 & -0.4187 & -0.0813 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

The matrix multiply consists of nine multiplications, one for each matrix coefficient.

We recursively characterize this block in terms of set partitions. There are nine multiplies, arranged in three rows of three. The following diagrams illustrate our designs; each dot represents a multiply operation, with multiplies shared between the same multiplier grouped together in the same box.

Multipliers	Cycles		
	YUV	Q	Total
1	9	3	12
2	6	2	8
3	3	1	4
6	2	1	3
7	1	1	2

Table 1. YQ module designs



Note that the cycle-time is equal to the size of the largest block. The completely parallel design requires only 7 rather than 9 multipliers because two of the multiplies are by a coefficient of 0.5 and can be implemented by a bit shift.

Although it is possible that 9 multiplies could be performed in 2 cycles by just 5 multipliers instead of the 6 reported here, such a design would have more complex control flow and so it is omitted.

Combined YUV and quantize For each pixel input, consisting of three colour samples, the YUV section requires 9 multiplies, of which 2 are simply bit-shifts; the quantize section requires 3 multiplies. We consider the set of designs produced when YUV and quantize share hardware by time-multiplexing: in any clock cycle, the hardware is either entirely dedicated to YUV or to quantize. This allows us to base our design for YQ on the previous design for YUV and the design for quantize. This gives the results in table 1.

5 Results

Figure 2 shows the objective space of the designs generated in the previous section. The dotted line shows the *Pareto front*, the line which divides satisfiable design objectives from unsatisfiable objectives given this solution set. The designs at the outside corners of the Pareto front are the *Pareto optimal* solu-

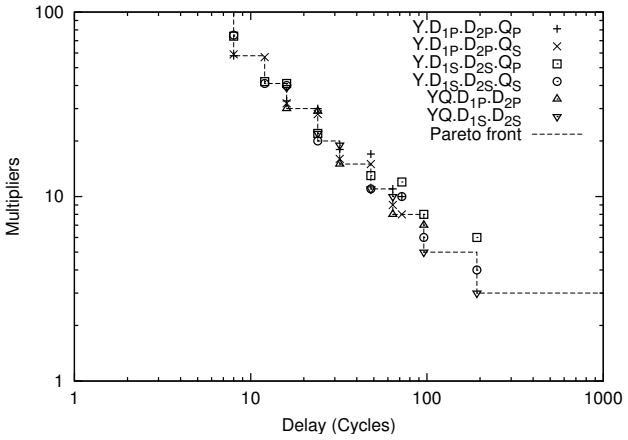


Figure 2. Objective space of JPEG encoders

tions. A design is Pareto optimal if there is no other design which is better in one objective and at least as good in all other objectives.

The fastest design uses 58 multipliers and takes 8 cycles per 8 pixels, while the smallest design uses 3 multipliers and takes 192 cycles. The span between largest and smallest is a factor of 20, while the scale between fastest and slowest is a factor of 24.

In between, there is a cloud of solutions in the objective space. There are regular occurrences of Pareto optimal solutions to satisfy various user constraints. No one partition dominates the results: the Pareto set contains points from four of the chosen partition, while the remaining two partitions have some solutions very close to the Pareto front. This is because the different partitions have different 'sweet spots' where no one constituent block is a significant bottleneck.

For example, the Pareto point with 41 multipliers and 12 cycles per 8 pixels from partition $Y.D_{1S}.D_{2S}.Q_S$, is better than the $Y.D_{1P}.D_{2P}.Q_S$ design which takes 12 cycles with 57 multipliers. The best design, $Y.D_{1S}.D_{2S}.Q_S$, has three of its four blocks with equal throughput, with the Y block running slightly at excess capacity – capable of 8 cycles. The worst design, $Y.D_{1P}.D_{2P}.Q_S$, has a poor match in throughput between its blocks. Both the DCT blocks and the Y block are capable of 8 cycle performance, but the Q block requires 12. There is thus 50% excess capacity in the relatively large Y and D blocks. There is no design available for these blocks which matches the speed of the bottleneck, so a larger design is required.

We expected that finer partitions would be generally faster than coarser partitions, but the results do not show this. This is because the partition does not reflect the *internal* parallelism or hardware sharing within the blocks. A design which is parallel at the high-level but much less so within the blocks may run at a similar speed and with similar area to a design which shares much hardware at a high level but is highly parallel within the blocks.

We have not assessed control-flow requirements. Hardware sharing requires control-flow support structure to route signals to and from hardware resources or as support for runtime reconfiguration.

6 Conclusion

We aim to incorporate this technique into a systematic tool for hardware design. We believe that the generation of component module designs could be automated to some extent, though the high-level choice of partitions to search will require a designer in the loop. The component design can be automated based on the computation/communication graph of the component. Such a tool could take advantage of resource estimation technology [5] to rapidly assess designs without having to fully synthesize them. It would also help in assessing control-flow requirements and give a more accurate impression of the resource usage for each design.

Acknowledgements The support of UK EPSRC and Xilinx is gratefully acknowledged.

References

[1] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, 2001.

[2] T. Givargis and F. Vahid. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(11):1317–1327, Nov 2002.

[3] S. Mohanty et al. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *ACM SIGPLAN Notices*, 37(7):18–27, July 2002.

[4] A.D. Pimentel et al. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Computers*, 55(2):99–112, 2006.

[5] P. Schumacher and P. Jha. Fast and accurate resource estimation of RTL-based designs targeting FPGAs. In *FPL*, 2008.

[6] A. Tumeo et al. A self-reconfigurable implementation of the JPEG encoder. In *ASAP '07*, pages 24–29, July 2007.

[7] P. Yiannacouras et al. Application-specific customization of soft processor microarchitecture. In *FPGA '06*, pp. 201–210. ACM, 2006.