# Chapter 1

# Whither Reconfigurable Computing?

George A. Constantinides, Samuel Bayliss and David Boland
*EEE Department, Imperial College London, Exhibition Road, London SW7 2AZ, U.K.*
*g.constantinides@ic.ac.uk*

We argue that FPGAs, more than two decades after they began to be used for computational purposes, have become one of the key hopes for extending the performance of computational systems in the era characterised by the end of Dennard scaling. We believe that programmability of future heterogeneous computing platforms has brought a new urgency to bear on several old problems in high-level synthesis for FPGAs. Our focus is on the two areas we believe are most underdeveloped in today's high-level synthesis software: effective utilisation of the numerical flexibility afforded by high-level correctness specifications, and application-specific memory subsystem synthesis. We conclude with our perspective on the likely future evolution of the field.

## 1.1. A Selective Context

We present below a necessarily rather narrow view of the evolution of the FPGA and the microprocessor, highlighting the interaction between the two and the major external drivers.

The Field-Programmable Gate Array (FPGA) was invented in the 1980s, but they developed and matured in the 1990s. Already in the early 1990s, academic conferences started to appear that were largely dedicated to the potential these devices had to implement computation, such as the first FPL, held in 1991 in Oxford. However, the nature of such devices has transformed over recent years. Initially FPGAs were largely homogeneous architectures, consisting of a large number of very fine grain logic cells. Responding to the nature of the new application areas, manufacturers evolved the FPGA architecture, first incorporating larger RAM blocks[1] and then dedicated multiplication logic.[2] Today, modern FPGA architectures are

highly heterogeneous devices, containing logic cells, embedded RAM and DSP functionality, high speed transceiver circuitry and microprocessors. It is worth noting that the majority of these components, present as hard IP within an FPGA, could be implemented using lookup table functionality. However, to do so would be either too large, too slow, or consume too much power to be worthwhile.[3] Thus through one lens we can see the evolution of the FPGA in recent years as a conscious decision to move away from having all area devoted to simple fine-grain units and their fine grain interconnect, towards the *specialisation of circuitry* to perform certain common tasks or classes of tasks.

In a sense, the evolution of the general purpose processor has mirrored the evolution of the FPGA over the same timeframe. Traditional, latency-driven, computer architecture largely consisted of utilising all available silicon in order to keep a single (or small number of) computational units as busy as possible. This resulted in a very large amount of silicon and power consumption devoted to caching, in particular, as well as various micro-architectural innovations to avoid latency-consuming pipeline stalls.[4] These processors have formed the core of general purpose computer design for several decades. The most significant innovation to arise as a result has been the GPGPU, which has delivered major performance improvements in certain domains by explicitly abandoning some of the received wisdom of computer architecture, a process referred to by Bill Dally as 'the end of denial architecture'.[5] GPGPU computing achieves its performance by using an explicitly software-managed memory hierarchy, returning hardware to computation, and using an abundance of threads to hide pipeline stalls. We may therefore view the evolution of the microprocessor as a conscious decision to move away from one complex unit towards dedicating areas to a large number of much simpler units and their interconnect; in a certain sense this is a mirror of the evolution of the FPGA.

It is no accident that the co-evolution of FPGA and microprocessors are now reaching the point of blurred boundaries. On the microprocessor side of the picture, this has largely been driven by the recent failure of Dennard scaling. Dennard scaling[6] provided a road map of how to scale various parameters under manufacturer control, such as supply voltage, in response to the geometric scaling of VLSI given by each processor generation. The recent deviation from Dennard scaling, largely driven by power consumption concerns[7] has forced the general purpose processor industry to look beyond clock frequency as the driver for performance. While high performance for throughput-dominated applications with embarrassing levels of

parallelism can be achieved in a direct way using the GPGPU approach, latency constraints and algorithm bottlenecks mandate a more heterogeneous approach.[8] On the FPGA side of the picture, silicon and power inefficiencies combined with new market opportunities have driven the evolution of FPGA architecture to its present heterogeneous state.

The future of manycore computing using traditional - but simple - microprocessor cores is not rosy. In a landmark paper, Berger[9] *et al.* make a detailed study of the power/area and power/performance tradeoffs available across a spectrum of processor designs. Using predictions of future technologies, even with extremely parallel workloads, the performance to be gained by using more, simpler, traditional processors, is bounded from above by factor of only about 4-8x over the next decade, far slower than the historical trends. This is largely due to power consumption limitations, resulting in the spectre of *dark silicon*, transistors that may be present on a device but cannot be powered on simultaneously without overloading the power limitations. The conclusion is clear: to move beyond such limits, it becomes necessary to improve the Pareto tradeoff itself, rather than simply move towards more, simpler, processors on the Pareto front. The only clear way to do this is through *circuit specialisation*; creating parts of a processor that are specialised to particular commonly occurring tasks, avoiding the energy inefficiency in using general purpose architectures for these tasks. This is exactly the area where the reconfigurable computing community has a head start, and can provide direction to the general purpose architecture community.

It seems inevitable that future computer architecture will therefore be programmable, contain elements of application-specific or domain-specific architecture, and be highly heterogeneous in nature. The major challenge is how to efficiently and effectively compile applications onto such platforms. This is a challenge that must be overcome, but one that is no longer faced by the FPGA community alone, as was often the case in the early efforts of high-level synthesis for reconfigurable computing. The coming industrial turn towards heterogeneous parallel computing opens many doors.

### 1.2. The Promise and the Challenges

High level synthesis for reconfigurable computing has made great strides recently. The Autopilot tool[10] now included within the Vivado design suite is a high quality high-level synthesis environment, using C as the input language. Academic efforts such as LegUp[11] also point to a promising

future for FPGA-based high level design. However, existing solutions for high-level synthesis do not – in our opinion – adequately address memory systems. It should not be up to the programmer to explicitly manage the transfer of data between external memory of various types (SDRAM, SRAM, etc.) and on-chip memory. Equally, we should not squander the potential of FPGA architectures by aping general purpose microprocessor cache schemes within an FPGA. In our view, it is high time that tools for customisation of computational circuitry were matched by aggressive tools for customisation of memory subsystem design. By pushing the complexity into the synthesis tool, we believe that significant performance advantages can be obtained without the area or energy overhead of caching schemes. This is the topic we consider in Section 1.4. We note that the degree of predictability of memory accesses largely defines the potential improvement possible by a customised memory system and that often memory accesses are very predictable in nature, especially for embedded applications, which we believe form the key driver for next-generation computer architecture.

The other area that is poorly covered by existing high-level design flows is the automation of the selection of numerical representation and precision. The designer of any hardware accelerator for a numerically-intensive algorithm knows that this is one of the areas where customised logic can result in huge performance gains, and will naturally ask 'should I use floating point, fixed point, or some more esoteric number system to perform this task', 'how precise do my internal results really need to be', *etc.* These questions remain largely unautomated. As a result, designers will again often ape the systems used in general purpose processor designs, such as IEEE standard floating-point arithmetic as the 'gold standard' of real number representation. There are two problems with this approach. Firstly, it does not work: typically a designer will want to perform operations in a different order to that expressed in the original code, in order to improve hardware efficiency, for example by applying the associative law to regroup addition into a tree structure: $a + (b + (c + d)) = (a + b) + (c + d)$, a law that holds for real numbers *but does not hold for floating-point* thus raising questions of correctness. Usually, whether formalised or not, there will be some notion of an acceptable numerical result, which can be used to drive such decisions; indeed, without such a notion, it becomes impossible to demonstrate that the behaviour of even the original source code is acceptable. We strongly advocate the formalisation of such specifications. This leads us onto the second problem: the designer operates with 'one hand tied behind her back' by being forced to replicate the hardware structures

present in general purpose processors, which may be grossly inefficient for the problem at hand. Once a formal specification of numerical correctness is available, the designer, and the synthesis tool, should be free to produce any hardware structure meeting that specification, playing to the advantages of the underlying architecture. Thus the same algorithmic specification may map automatically to a mixed-precision implementation in a GPU, a double precision implementation in a CPU, and a fixed-point implementation in an FPGA. No two of these implementations may produce the same bit pattern at their outputs, but all should be verifiable with respect to the formal correctness criteria. This is the topic we consider in Section 1.3. We note that, while such freedom can be exploited in all numerical applications, the degree of freedom is particularly great in embedded applications, where specifications of correctness tend to be expressible at very high levels of abstraction, leaving lots of freedom for an advanced design tool to explore, *e.g.* a controller for an aircraft might mandate stability and minimisation of fuel consumption of the aircraft;[12] a much higher level of abstraction than bit-level equivalence to a golden C model!

### 1.3. Numerical Behaviour

When creating digital hardware architectures, one must first select a finite precision number system to represent numerical data. Since this number system can only represent a subset of real numbers, rounding will often occur after an arithmetic operation so as to represent values using the chosen number system. Whilst the error introduced by the rounding of any single value may be small, over the course of an algorithm the accumulation of these errors can cause a significant deviation from the desired result.

A simple tactic to minimise this error would be to err on the side of safety and select a number system that has much greater precision than necessary to obtain the desired quality of output, if such a precision can be determined. However, this will come at a substantial cost in terms of performance. As an example, recent figures for the difference in performance, in terms of peak theoretical FLOPs, between single and double precision is approximately a factor of 2 to 3 for a CPU[13] or 24 for a GPU.[14]

Since arithmetic computation forms the heart of many high-performance digital systems, if we are to create efficient hardware accelerators, we first need to select number systems with the minimum precision necessary to guarantee that our design criteria are met. Unlike CPUs and GPUs, FPGAs offer the freedom to fully customise the precision used throughout

an accelerator. As a result, development of techniques to select an optimised number system have been an extensive research topic for the FPGA community over the past decade.[15,16] In this section, we first describe the state-of-the-art techniques that help us guarantee that a given number system for a hardware accelerator satisfies a numerical correctness criterion. We further discuss how these techniques can be enhanced so that they are applicable to a wide range of algorithms. Finally, we outline some of the future challenges for research in this field.

### 1.3.1. *Bounding Numerical Errors*

The most straightforward way to estimate the error of any hardware accelerator is through simulation; indeed, this is the main technique used by industry. Unfortunately, the size of the search space for the inputs will generally be too large to explore exhaustively; this means simulation may miss corner cases and under-allocate the number of bits for an accelerator. This is unacceptable in any safety-critical system, and in any case only works when there is a trusted, 'golden' reference model or method of certification available.

In contrast, analytical approaches provide guarantees that a design criterion will not be violated. Early analytical approaches were based on Interval Arithmetic (IA),[17] Affine Arithmetic (AA)[18] and LTI Theory.[15] Unfortunately, because IA and AA cannot find tight bounds on the worst case error, they will typically over-allocate bits for any nontrivial example. LTI theory is powerful enough to compute tight bounds, but it is restricted to the LTI domain and this does not include general multiplication, for example.

More recently, new approaches have been created which involve constructing polynomials to represent the worst-case range of intermediate variables throughout an algorithm. Through computing the lower ($\gamma_{lower}$) and upper ($\gamma_{upper}$) bounds of these polynomials, we can select a number system which prevents overflow. Furthermore, if we first construct a polynomial $\hat{p}$ representing the range of every intermediate variable in the presence of finite precision errors and a second polynomial $p$ representing the range in infinite precision, then the extrema of the function $\frac{|p-\hat{p}|}{p}$ represent the worst-case relative error introduced by the use of finite precision arithmetic.

To create these polynomials, we use standard models to represent finite precision errors. When using fixed point, provided there is no overflow, numerical errors are limited to one unit in the last place. If we choose

an $\eta$-bit number system where the maximum value is $2^X$, the worst-case rounding error for any fixed point number $x$ is given by (1.1). It follows that the result of any scalar operation ($\odot \in \{+, -, *, /\}$) is bounded as in (1.2). Similarly, for floating point, provided there is no overflow or underflow, for any real value $x$, the closest floating-point approximation $\hat{x}$ of $x$ can be expressed as in (1.3), where $\eta$ is the number of mantissa bits used. Once again, it follows that the floating-point result of any scalar operation ($\odot \in \{+, -, *, /\}$) is bounded as in (1.4).

$$\hat{x} = x + \delta_1 \qquad (|\delta_1| \leq \Delta, \text{where } \Delta = 2^{X-\eta}). \tag{1.1}$$

$$\widehat{x \odot y} = (x \odot y) + \delta_1. \tag{1.2}$$

$$\hat{x} = x(1 + \delta_1) \qquad (|\delta_1| \leq \Delta, \text{where } \Delta = 2^{-\eta}). \tag{1.3}$$

$$\widehat{x \odot y} = (x \odot y)(1 + \delta_1). \tag{1.4}$$

Through applying these models of error to every computation in an algorithm, we can construct polynomials that represent the potential range of every intermediate variable. This is shown for a simple example in Table 1.1.

| $x, y$ are inputs<br>$\Delta$ is the error bound determined by the precision, so that $\|\delta_i\| \leq \Delta$ | |
|---|---|
| Code | Polynomial Representation of Variable Value (Floating Point) |
| a = x*y; | a = $xy(1 + \delta_1)$ |
| b = a*a; | b = $(xy(1 + \delta_1))^2(1 + \delta_2)$ |
| c = b-a; | c = $[(xy(1 + \delta_1))^2(1 + \delta_2) - xy(1 + \delta_1)](1 + \delta_3)$ |

While constructing these polynomials is straightforward, finding their extrema is computationally intractable. Instead, algorithms focus on finding a computationally tractable lower bound $\hat{\gamma}_{lower} \leq \gamma_{lower}$ and upper bound $\hat{\gamma}_{upper} \geq \gamma_{upper}$. Ideally we wish to find bounds such that $\gamma_{lower} - \hat{\gamma}_{lower}$ and $\hat{\gamma}_{upper} - \gamma_{upper}$ are as small as possible.

One of the latest and most powerful techniques to achieve this is based upon a result from real algebra discovered by Handelman.[19] This states that a polynomial $p$ is non-negative if and only if $p$ has a *Handelman representation* of the form (1.5).

$$f = \sum_{\alpha \in \mathbb{N}^n} c_\alpha \prod_{i=1}^{m} g_i^{\alpha_i}, \tag{1.5}$$

where each $c_\alpha$ is a positive constant, each $g_i$ is a positive inequality and $\mathbb{N}$ is the set of natural numbers.

Using this result, we first re-write the bounds of a polynomial $\hat{\gamma}_{lower} \leq p \leq \hat{\gamma}_{upper}$ as two separate equations $\hat{\gamma}_{lower} - p \geq 0$ and $p - \hat{\gamma}_{upper} \geq 0$. If we can find a Handelman Representation to prove each inequality is non-negative, then we have found the lower and upper bounds of the polynomial. Heuristics which search for these representations have been shown to be able to compute much tighter bounds than IA or AA and enable us to create substantially smaller hardware.[20]

### 1.3.2. *Can we apply these techniques to general code?*

The techniques described in the previous section are powerful and have been shown to result in substantial performance improvements for some simple benchmarks. Unfortunately, the size of the polynomials can grow exponentially in the number of operations, meaning it would become too time consuming to be applicable to real benchmarks.

However, we can simplify large polynomials by replacing all terms that contribute little to the final result with a single term. Table 1.2 analyses a polynomial representing the range of a floating point addition of two variables. It calculates the worst-case range of every individual term in this polynomial. Clearly, several terms such as $100y_1\delta_1$, $100x_1\delta_1$ and $100x_1y_1\delta_1$ will have little impact on the final bounds. As such, if we replace them with a single new bounded variable, we shrink our polynomial with little impact on the final bounds. This simplification technique enables the earlier bounding procedure to be applied to much larger algorithms consisting of straight-line code.[21]

Compute $a = x \bullet y$, where $x = [8; 12], y = [9; 11]$ in 6 bit floating point
let $|x_1| \leq 0.2, |y_1| \leq 0.1, |\delta_i| \leq 2^{-6} \Rightarrow x \in 10(1 + x_1), y \in 10(1 + y_1)$
$a = 10(1 + x_1)10(1 + y_1)(1 + \delta_1)$
$= (100 + 100x_1 + 100y_1 + 100\delta_1 + 100x_1y_1 + 100x_1\delta_1 + 100y_1\delta_1 + 100x_1y_1\delta_1)$

| Term | Potential Contribution | Term | Potential Contribution |
|---|---|---|---|
| 100 | 100 | $100x_1$ | ± 20 |
| $100\delta_1$ | ±0.09765625 | $100x_1\delta_1$ | ±0.01953125 |
| $100y_1$ | ±10 | $100x_1y_1$ | ± -2 |
| $100y_1\delta_1$ | ±0.009765625 | $100x_1y_1\delta_1$ | ±0.001953125 |

However many algorithms cannot be converted into straight-line code; algorithms often contain complex control structures such as 'while' loops. The challenge with these structures is that finite precision errors may cause a 'while' loop to fail to terminate.[16] Interestingly, these polynomial bounding procedures can also be useful in choosing sufficient precision to ensure that 'while' loops terminate.

One technique to prove program termination is based on the following steps:

(1) Construct a *ranking function*,[22] $f(x_1, ..., x_n)$ that maps every potential state within the loop to a positive real number.
(2) Prove that for all potential values of the variables $x_1, ..., x_n$ within the loop body, when the ranking function is applied to the loop variables before and after the loop transition statements, it always decreases by more than some fixed amount $\epsilon > 0$, *i.e.* $f(x'_1, ..., x'_n) \leq f(x_1, ..., x_n) - \epsilon$.

If we note that proving a ranking function decreases ($f(x'_1, ..., x'_n) \leq f(x_1, ..., x_n) - \epsilon$) can be re-written as a question of non-negativity ($0 \leq f(x_1, ..., x_n) - f(x'_1, ..., x'_n) - \epsilon$), then we can apply the same techniques that prove non-negativity to prove termination in finite precision arithmetic.[16]

### 1.3.3. *Next Steps*

The techniques described in this section only touch the surface of research into automatically selecting the minimum precision necessary to meet design criteria. However, crucially they offer substantial progress in answering the following question: given a hardware architecture and word-length specification, will my design satisfy the specification? This will enable further research in this field; this includes delving deeper into techniques to assign the word-length for each individual operator in a large datapath and minimise the total area consumption,[23,24] studying the links between how the order of operations in a hardware datapath can affect the error seen at the output and exploring the relationship between numerical precision and termination of iterative algorithms. Research into numerical behaviour has entered exciting times.

### 1.4. Memory Systems

In the preceding sections, we described how high-level specification of numerical accuracy can enable us to make more efficient use of silicon area. This in turn enables better performance where the number of parallel processing units can be increased, provided those units can be efficiently fed with data.

The most area-efficient technologies in common use for implementing commodity memory today (DRAM and Flash) have optimal process parameters that conflict with those needed to build fast logic. Wherever applications require large amounts of memory, that memory is implemented using a separate memory die. However, because device pin-density and off-chip switching frequencies have not scaled as rapidly as the exponential growth of transistors dedicated to logic datapath implementation, external memory bandwidth has increasingly become a performance bottleneck.

So it is critical to ensure that limited off-chip memory bandwidth is used efficiently. Herein lies a second challenge. DRAM memory structure is arranged in banks, rows and columns. Each row must be 'activated' before data held in columns within that row can be read or written. The row must then be 'precharged' before data in another row can be accessed. Timing parameters determined by physical DRAM memory array architecture constrain the minimum time between successive row activations. Over time, increasing memory clock frequencies mean that there is an ever larger penalty paid for random access to DRAM memory. In practical terms, this means there is a greater than $10\times$ performance difference between the worst case and best case memory bandwidth obtained through different memory address sequences.

This has made it essential to develop memory subsystems which exploit the locality of memory accesses to provide the illusion of fast access to large amounts of memory. In a CPU, caches and dynamic memory controllers buffer and reorder memory requests to help ensure this happens. They typically must assume no prior knowledge of the sequence of memory requests from the datapath. Furthermore, CPUs implement non-deterministic bus interfaces which make memory performance difficult to analyse.

Where a memory system is implemented in reconfigurable hardware, it can be customised for a specific application. Three key benefits can then be realised:

(1) fine grained on-chip memories provide a very large on-chip memory bandwidth to customised datapath,
(2) data buffered in those memories can be reused, reducing off-chip memory bandwidth requirements, and
(3) off-chip memory requests can be reordered to make the most efficient use of limited bandwidth.

The most memory intensive parts of a program tend to be in loops, so we target nested loops in our work.[25] Static analysis to model the sequence of memory accesses which occur in nested loops can be done using the Polyhedral Model.[26,27] From this analysis, automated tools allow us to synthesise a high performance application-specific memory system. In Section 1.4.1, we provide a brief overview of the Polyhedral Model. Section 1.4.2 describes a way in which this model can be used to build high performance application-specific memory systems.

### 1.4.1. *What is the Polyhedral Model?*

The Polyhedral Model represents a set of loop iterations as those integer vectors which satisfy a finite set of affine inequalities. The code in Figure 1.1 shows a two level nested loop. The set of loop iterations is described by upper and lower bounds which are affine expressions of the surrounding loop variables ($x_1$ and $x_2$). The iterations of an $n$-level loop nest can be described implicitly as an integer set $\{\mathbf{x} \in \mathbb{Z}^n \mid A\mathbf{x} \leq \mathbf{b}\}$ where A is a $2n \times n$ integer matrix, $\mathbf{b}$ is a $2n$ integer column vector and the vector inequality is interpreted as $\mathbf{x} \leq \mathbf{y}$ iff $x_i \leq y_i$ for all $i$.

```
int t; int arr[256];
for (x1 = 0 ; x1 <= 15 ; x1++) {
  for (x2 = 0 ; x2 <= 15 ; x2++} {
        ... = function( arr [x1 + 16 * x2] );
  }
}
```

Fig. 1.1. Example code for a two-level nested loop.

These iterations can be scheduled according to a linear mapping function which determines a partial ordering of those iterations. For the example given in Figure 1.1, a mapping function $\sigma(\mathbf{x}) = \begin{pmatrix}16 & 0\end{pmatrix}\begin{pmatrix}x_1 \\ x_2\end{pmatrix}$ describes the iteration ordering. Each iteration $\mathbf{x}$ may access memory via memory accessing function(s) of the form $g_j(\mathbf{x}) = \mathbf{f}_j\mathbf{x} + h_j$.

Code that fits into this form is common in video processing and dense linear-algebra applications. Exact dependence analysis for code which can be described in this way is often tractable using integer linear programming techniques.[28] The Polyhedral Model gives us a formal mathematical representation of the sequence of memory addresses accessed in the program. In section 1.4.2, we show how transformations applied to that formal representation can help build a high performance memory system.

### 1.4.2. *Building high performance memory systems*

In the preceding section, we showed how we could formally characterise the memory access requirements within a nested-loop structure. We can use this information to decouple the off-chip memory accesses from datapath logic using on-chip memory buffers. If we can transform code so that data is reused from the on-chip memory buffer, we can reduce the number of accesses to off-chip memory.

We can represent the specific 'row' and 'burst' accessed in each memory request by adding new dimensions to the loop-nest representation. If the size of each DRAM row is $R$ words, the row accessed by memory address $\mathbf{f}\mathbf{x} + h$ is given by $r = \mathbf{f}\mathbf{x} + h \text{ div } R = \lfloor \mathbf{f}\mathbf{x} + h/R \rfloor$ where $\lfloor \cdot \rfloor$ represents the *floor* function. The columns within each row can be represented as non-overlapping bursts to take advantage of the multi-word burst accesses supported by modern memory devices. These can be represented by $u = \lfloor \mathbf{f}\mathbf{x} - rR)/B \rfloor$ where a burst is $B$ words long.

While neither of these is directly amenable to linear algebraic representation, we may note that from the properties of the floor function:

$$\left\lfloor \frac{\mathbf{f}\mathbf{x} + h}{R} \right\rfloor - 1 < r \leq \left\lfloor \frac{\mathbf{f}\mathbf{x} + h}{R} \right\rfloor, \tag{1.6}$$

and

$$\left\lfloor \frac{\mathbf{f}\mathbf{x} + h - rR}{B} \right\rfloor - 1 < u \leq \left\lfloor \frac{\mathbf{f}\mathbf{x} + h - rR}{B} \right\rfloor. \tag{1.7}$$

We can rewrite (1.6) and (1.7) as linear equalities as shown below in (1.8) and (1.9), without loss of information.

$$\mathbf{fx} + h - R + 1 \leq Rr \leq \mathbf{fx} + h \tag{1.8}$$

$$\mathbf{fx} + h - rR - B + 1 \leq Bu \leq \mathbf{fx} + h - rR \tag{1.9}$$

We can then add these four extra inequalities to those already present defining the loop bounds. This forms, for each memory reference, an augmented system of linear inequalities that completely capture not only the iteration space but also the specific SDRAM rows and bursts accessed within the innermost loop.

Using standard unimodular loop transformations, we can transform this augmented polyhedral representation to expose those occasions where data items are reused by multiple loop iterations. After transformation, those redundant dimensions which *only* represent data reuse can be projected out of the resulting polyhedral representation to produce code which fetches each memory item only once from off-chip memory. From this representation, we can use standard loop reordering transformations to move 'row' dimension to the outer-most level of the loop nest, improving data-locality.

Figures 1.2(a)-(d) illustrate this process for the example code shown in Figure 1.1. The code in Figure 1.2(a) is augmented with the row iterator 'r' and the burst iterator 'u' to form Figure 1.2(b). The 'r' variable can then be hoisted to the outermost loop level to give Figure 1.2(c). Note now that the $x_2$ variable is accessed only once in each loop iteration and can therefore be eliminated to give Figure 1.2(d). The sequence of memory addresses accessed in Figure 1.2(d) accesses the same rows and bursts of the original source code, but now does so in a more efficient order, since there are fewer row-swaps (and their associated timing penalties) incurred.

When this technique is applied to code, it can significantly improve interface bandwidth efficiency. We show this in Figure 1.3 for three benchmarks (Matrix-Matrix-Multiply, Sobel Filter and Gaussian Backsubstitution) parameterised with reuse buffers inserted at different levels of the loop nest. The insertion of the buffer at the outermost level of the loop nest (t=1) allows reordering of all memory accesses and means less than 10% of memory access cycles are spent idle whilst DRAM rows are swapped compared with >75% in the original code. The different levels of parameterisation allow a trade-off between performance and the amount of on-chip memory dedicated to data buffering.

```
char arr[256]; // off-chip memory
for(x1 = 0; x1 <= 15; x1++)  {
  for(x2 = 0; x2 <= 15; x2++) {
    .. = function( arr[x1 + 16*x2] );
  } }
```

(a) Original Code

```
char arr[256]; // off-chip memory
for(x1 = 0; x1 <= 15; x1++) {
  for(x2 = 0; x2 <= 15; x2++) {
  // Note : / is integer division.
  // Note : r and u loops have one iteration.
    for( r = (x1 + 16*x2)/16; r <= (x1 + 16*x2)/16; r++ ) {
      for( u = (x1 + 16*x2 - 16*r)/4; \
                u <= (x1 + 16*x2 - 16*r)/4; u++ ) {
        ... = function( arr[x1 + 16*x2] )
      } }
  } }
```

(b) Augmented Code

```
char arr[256]; // off-chip memory
char buff[16][4][4]; // on-chip memory
for(r = 0; r <= 15; r++)  {
  for(u = 0; u <= 3; u++) {
    buff[r][u][0..3] = burstread(r,u);
    for( x2 = r; x2<=r; x2++) {
      for( x1 = 4*u; x1<=4*u+3; x1++){
        ... = function( buff[x2][x1/4][x1%4] );
      } }
} }
```

(c) Intermediate Code

```
char arr[256]; // off-chip memory
char buff[16][4][4]; // on-chip memory
for(r = 0; r <= 15 ; r++) {
  for(u = 0; u <= 3; u++) {
    buff[r][u][0..3] = burstread(r,u);
    for(x1 = 4*u; x1 <= 4*u+3; x1++ ) {
      ... = function( buff[r][u][x1-4*u] );
    } }
}
```

(d) Transformed Code

Fig. 1.2. Transformation Steps in Improving Memory Bandwidth Efficiency

s

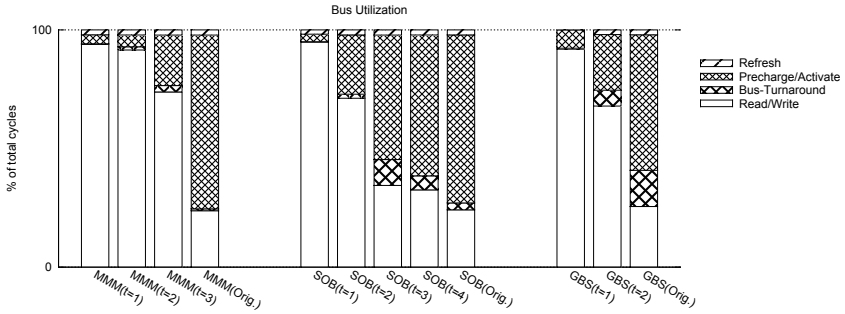


Fig. 1.3. Memory Bandwidth Improvements : Breakdown of interface commands by type.

### 1.4.3. *What might this enable us to do in the future?*

Looking beyond our existing work, the formal model of memory access provided by the Polyhedral Model is a promising representation for enabling other application-specific memory transformations. One emerging area of research is the exploration of how the Polyhedral Model enables the overlapping of off-chip memory operations with on-chip computation. This work makes use of mathematical advances[29] which allow us to count the exact number of integer points contained with a polyhedron without enumerating them.

Knowledge of the exact lifetime of variables fetched into on-chip memory can enable more compact mapping of those variables into limited on-chip memory. Exploratory work on how to better utilise the multiple independent banks within a DRAM also seems like a promising direction, allowing us to further improve the efficiency of off-chip memory accesses.

Exact dependency analysis allows auto-parallelisation, but to support this, we need to ensure that enough on-chip memory ports are available to avoid contention. Emerging automatic array partitioning techniques[30] ensure that contention for on-chip memory ports in minimised. This allows efficient use of the large on-chip bandwidth provided by block RAM resources which are ubiquitous in modern heterogeneous FPGAs.

The key theme is that there are significant opportunities opened up by expanding our synthesis tools to target complete reconfigurable systems including off-chip memory. The formal representation of memory access sequences provided by the Polyhedral Model allows tools to automatically produce efficient application-specific hardware with tailor-made memory systems.

## 1.5. Conclusion

Our view is that many problems in high level design automation, once of concern to the small group of pioneers of reconfigurable computing, now arise in various guises in the much broader setting of computing generally, and embedded computing in particular. While high level synthesis and compilation tools have progressed significantly over the past decade, we believe that there are two very significant gaps in existing tool flows: customisation of memory systems and auto-generation of finite precision arithmetic implementations. We have described our own approaches to these central problems. Our view is that the FPGA computing community is poised to play a central role in the evolution of computer architecture and compilers over the next decade. We must take up the baton.

## Acknowledgements

We wish to acknowledge the inspirational mentorship of Prof. Peter Y.K. Cheung, one of the small group of pioneers of FPGA-based computing. In addition, we would like to express our thanks to EPSRC for funding received to support the development of the ideas expressed in this paper (grants EP/G03157/1, EP/I012036/1, EP/I020357/1, EP/K034448/1).

## References

1. S. Wilton. *Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, (1997).
2. S. Haynes, A. Ferrari, and P. Cheung. Flexible reconfigurable multiplier blocks suitable for enhancing the architecture of FPGAs. In *Proc. Custom Integrated Circuit Conference*, pp. 16–19, (1999).
3. I. Kuon and J. Rose, Measuring the gap between FPGAs and ASICs, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*. **26**(2), 203–215, (2007).
4. J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach (5. ed.)*. (Morgan Kaufmann, 2012).
5. W. Dally. The end of denial architecture and the rise of throughput computing. In *46th ACM/IEEE Design Automation Conference*, pp. xv–xv, (2009).
6. R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. Leblanc, Design of ion-implanted MOSFETs with very small physical dimensions, *IEEE J. Solid State Circuits*. **SC-9**(5), 256–268 (October, 1974).
7. T. Mudge, Power: A first-class architectural design constraint, *IEEE Computer*. **34**(4), 52–58, (2001).
8. A. Rafique, G. Constantinides, and N. Kapre, Communication optimization

of iterative sparse matrix-vector multiply on gpus and fpgas. submitted to IEEE TPDS.

9. H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, (2011).
10. Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, *AutoPilot: A Platform-Based ESL Synthesis System*, In eds. P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, (2008).
11. A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. Legup: High-level synthesis for fpga-based processor/accelerator systems. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 33–36 (February, 2011).
12. E. Hartley, J. Jerez, A. Suardi, J. Maciejowski, E. Kerrigan, and G. Constantinides. Predictive Control of a Boeing 747 Aircraft using an FPGA. In *Proc. IFAC Nonlinear Model Predictive Control Conference*, pp. 80–85 (August, 2012).
13. A. Vladimirov. Whitepaper: Arithmetics on Intel's Sandy Bridge and Westmere CPUs: not all FLOPs are created equal, (2012).
14. NVIDIA. NVIDIA Tesla Kepler GPU Computing Accelerators. `http://www.nvidia.com/content/tesla/pdf/NV_DS_TeslaK_Family_May_2012_LR.pdf`.
15. G. Constantinides, P. Cheung, and W. Luk. The multiple wordlength paradigm. In *IEEE International Conference on Field Programmable Custom Computing Machines (FCCM)*, (2001).
16. D. Boland and G. Constantinides. Word-length optimization beyond straight line code. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, (2013).
17. R. E. Moore, *Interval Analysis*. (Prentice-Hall, Englewood Cliff, NJ, 1966).
18. L. H. de Figueiredo and J. Stolfi, *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monographs, (IMPA/CNPq, Rio de Janeiro, Brazil, 1997).
19. D. Handelman, Representing polynomials by positive linear functions on compact convex polyhedra, *Pac. J. Math.* **132**(1), 35–62, (1988).
20. D. Boland and G. Constantinides, Bounding variable values and round-off effects using handelman representations, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. **30**(11), 1691 –1704 (Nov., 2011).
21. D. Boland and G. Constantinides, A scalable precision analysis framework, *IEEE Transactions on Multimedia*. **15**(2), 242–256, (2013). ISSN 1520-9210. doi: 10.1109/TMM.2012.2231666.
22. B. Cook, A. Podelski, and A. Rybalchenko, Proving program termination, *Communications of the ACM*. (2009).
23. D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk. Minibit: bit-width optimization via affine arithmetic. In *Proc. Design Automation Conference*, pp. 837–840, New York, NY, USA, (2005). ACM. ISBN 1-59593-058-2. doi: http://doi.acm.org/10.1145/1065579.1065799.
24. D. M. H.-N. Nguyen and O. Sentieys. Novel algorithms for word-length opti-

mization. In *European Signal Processing Conference*, pp. 1944–1948, (2011).
25. S. Bayliss and G. A. Constantinides. Optimizing SDRAM bandwidth for Custom FPGA Loop Accelerators. In *FPGA'12 : Proceedings of the ACM/SIGDA 20th International Symposium on Field Programmable Gate Arrays*, pp. 195–204, (2012).
26. C. Lengauer. Loop Parallelization in the Polytope Model. In *CONCUR '93 : Proceedings of the 4th International Conference on Concurrency Theory*, pp. 398–417. Springer (Aug, 1993).
27. W. Kelly and W. Pugh. A Framework for Unifying Reordering Transformations. Technical Report UMIACS-TR-92-126.1, University of Maryland, College Park, MD, USA, (1993).
28. W. Pugh, The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis, *Communications of the ACM*. **8**, 4–13, (1992).
29. A. Barvinok. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension is Fixed. In *FOCS'93 : Proceedings of the 34th IEEE Annual Symposium on the Foundations of Computer Science*, pp. 566–572. IEEE Computer Society, (1993). ISBN 0-8186-4370-6. doi: http://doi.ieeecomputersociety.org/10.1109/SFCS.1993.366830.
30. P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong. Memory partitioning and scheduling co-optimization in behavioral synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 488–495. ACM, (2012).