The need for instructions to control device operation is one distinguishing feature of general-purpose computing devices. These instructions give general-purpose devices their flexibility to solve a variety of problems. At the same time, instructions require dedicated resources for storage and delivery.

General-purpose computing architectures must address a number of important questions:

1. How are general-purpose processing resources controlled?
2. How much area is dedicated to holding the instructions which control these resources?
3. How many resources are controlled with each instruction?
4. How much bandwidth is provided for instruction distribution?
5. How frequently can instructions change?

There are many, different possible answers to these questions and the answers, in large part, distinguish the various general-purpose architecture categories which we reviewed in Chapter 4 (*e.g.* word-wide uniprocessor, SIMD, MIMD, VLIW, FPGA, reconfigurable ALU). The answers also play a large role in determining the efficiency with which the architecture can handle various applications.

In this chapter, we look at the problem of instruction control and the resources involved. We start by looking at the extreme case where every bit operation is given a unique instruction on every cycle. This example illustrates that instruction distribution resource requirements can be quite large – dominating other areas in a device. To combat these requirements, traditional architectures have placed various, stylized restrictions on instruction distribution in order to contain its resource requirements. Each of these restrictions also limits the realm of efficiency of the architecture. We review these restrictions and their effects on device utilization efficiency in Section 8.3. Of course, the opportunity to compress instruction distribution requirements depends on the inherent *compressibility* of the instruction stream suggesting that some computations will remain description limited while others are compute limited (Section 8.4). We also look at the issue of instruction stream control (Section 8.5). Finally, Section 8.6 organizes the architectural parameters reviewed in this chapter into an expanded taxonomy for multiple data processing architectures.

8.1 General Case Example

Consider that we have N_p bit processing elements. Each of these elements may be a 4-LUT as in the previous section or a one bit ALU. We want to provide a different instruction to each processing element on every cycle of operation. From Section 4, we see that 100 MHz+ operating frequencies are readily achievable today, with many devices already achieving higher frequencies. We will consider a 200 MHz operating frequency as one that will be easily achievable in the very near future. We saw from Section 7.8 that each 4-LUT needs 40-50 bits to describe its network configuration and 16 bits to control the logic function. We will thus assume each LUT requires a 64-bit instruction to control it.

Let us further assume that we distribute the instruction, one per clock cycle, from all four sides of the array of N_p processing elements densely using 2 layers of metal with an 8λ wire pitch. Of course, the assumption that we dedicate two full metal layers to instruction distribution is extreme, but even making this best case assumption, we will see that the resources required for full instruction control can dominate all other concerns. For the sake of easy comparison, we will target a $1024\lambda \times 1024\lambda = 1\mathrm{M}\lambda^2$ array element, which is on par with large, conventional 4-LUT FPGA implementations (Table 7.1).

Across the width of one processing element, we can run instruction distribution busses to control two such elements:

$$2 \times \underbrace{64}_{\text{bits/LUT}} \times \underbrace{8\lambda}_{\text{wire pitch}} = 1024\lambda$$

This means, we can support an array with $2\times$ as many processing elements (N_p) as it has edge widths. That is:

$$2 \times 4 \times \sqrt{N_p} = N_p$$

From which we conclude $N_p = 64$.

At this point, we have fully saturated the i/o bandwidth into the compute array. Any further increase in the number of elements supported in the array must be accompanied by a corresponding increase in LUT size. That is:

$$\begin{aligned}
\frac{\sqrt{A_{LUT}}}{64 \times 8\lambda} \times 4 \times \sqrt{N_p} &= N_p \\
\frac{\sqrt{A_{LUT}}}{128\lambda} &= \sqrt{N_p} \\
A_{LUT} &= 16384\lambda^2 N_p \qquad (8.1)
\end{aligned}$$

Consequently, LUT area increases linearly with the number of processing elements supported in the array. *The instruction distribution bandwidth requirement is the dominante size effect determining the density with which computational elements can be built.* Notice that the LUT area growth rate dictated by instruction bandwidth is larger than the interconnect growth rate for any value of $p < 1$ and, ultimately, both interconnect and instruction distribution compete for the same limited resource – wire bandwidth.

Further, we can calculate the actual instruction bandwidth requirements. At 200 MHz and 64-bit instructions, each LUT requires 1.6GBytes/s of instruction distribution. For the $N_p = 64$ case above, this amounts to over 100GBytes/s.

This kind of bandwidth could not, reasonably, be supported from off chip with any contemporary technology, necessitating on-chip instruction memory. At $1000\lambda^2$ per SRAM cell, 16 64-bit instructions will occupy the same space as each $1\mathrm{M}\lambda^2$ processing element. If more than 16 unique instruction sets are required, instruction memory will occupy more area than the processing elements and interconnect.

8.2 Bits per Instruction

In the previous section we assumed 64 bits per instruction. This seems to be a reasonable, ballpark estimate of the number of bits required to describe a single bit operation, including interconnect.

Processors Modern processors generally employ 32 bits per instruction. However, as we saw in Section 4.1, about half of the instructions issued by a microprocessor are interconnect operations. Particularly, when we looked at gate evaluations, we saw that each processor instruction describes, on average, about 0.5-0.6 gate evaluations.

FPGAs Modern FPGAs use 120-200 bits per 4-LUT. We pointed out in the previous chapter that this was due to non-sparse encoding, and much denser encodings were possible. Simply using the crossbar bound from Section 7.8.1, we see that we can handle a 4000 4-LUT device with 48 bits of interconnect description and 16 bits of logic description. VEGA, a heavily multicontexted FPGA, with 85 bits per instruction, comes closer to this range [JL95].

8.3 Compressing Instruction Stream Requirements

Section 8.1 showed us that we cannot afford to have full, independent, cycle-by-cyle control of every bit operation without instruction storage and distribution requirements dominating all other resource requirements. Consequently, we generally search for application characteristics which allow us to describe the computation more compactly. In this section, we review the most common techniques generally employed to reduce instruction size and bandwidth. We see that every architecture reviewed in Chapter 4 exploits one or more of these compression techniques.

8.3.1 Wide Word Architectures

Processors do not, commonly, operate on single bit data items. Rather, sets of w bit elements ($w = \{8, 16, 32, 64\}$) are grouped together and controlled by a single instruction in SIMD style. This has the effect of reducing instruction bandwidth requirements and instruction storage requirements by a factor of w. This compression scheme takes advantage of the fact that we commonly do want to operate uniformly on multibit quantities. We can, therefore, effectively amortize instruction resources across multiple bit processing elements.

Returning to our opening example, we can support w^2 more processing elements before reaching the same point of wire saturation:

$$\begin{aligned} 2 \times 4 \times \sqrt{N_p} &= \frac{N_p}{w} \\ 8w &= \sqrt{N_p} \\ N_p &= 64w^2 \end{aligned}$$

The utilization efficiency of the resulting architecture depends on the extent to which all operations are w-bit operations, or even multiples thereof. When smaller operations, $w_d < w$, are required, $w - w_d$ bit processing units will sit idle while only w_d units provide useful work.

8.3.2 Broadcast Single Instruction to Multiple Compute Units

SIMD and vector machines take this instruction sharing one step further. They arrange so that multiple functional units operating on nominally different words share the same instruction. This allows them to scale up the number of bit operators without increasing the word granularity or instruction bandwidth. It does, however, increase the operation granularity. To remain efficient now, the application requires n_v w-bit operations, where n_v, is the number of word-wide datapaths controlled by each instruction.

8.3.3 Locally Configure Instruction

Reconfigurable architectures, such as FPGAs, take advantage of the fact that little instruction bandwidth is needed if the instructions do not change on every cycle. Each bit processing element gets its own, unique, instruction which is stored locally. However, this instruction cannot change from cycle to cycle. A limited bandwidth path is used to change array instructions when necessary.

There are two viewpoints from which to approach the efficiency of this restriction:

1. **Frequency of Instruction Change** – Given a lower bandwidth path, I_{BW}, and N_p compute elements with n_{ibits} long instructions, each context reload will take $\frac{n_{ibits} \cdot N_p}{I_{BW}}$. If the inter arrival time between reloads is n_{cycle}, the efficiency of operations is equal to the fraction of time spent computing versus total compute and reload time:

$$\text{Efficiency} = \frac{n_{cycle} \cdot t_{cycle}}{(n_{cycle} \cdot t_{cycle}) + \left(\frac{n_{ibits} \cdot N_p}{I_{BW}}\right)}$$

   This, of course, assumes that every processor is doing useful work on each cycle.

2. **Task Critical Path Length** – Alternately, if we assume the computing array is sufficiently large to perform the task, the efficiency is the fraction of compute elements performing useful work on each cycle. If the device has N_p processors and must perform a task with N_{bit_ops} which has a critical path of length L_{crit}, then the efficiency is the number of useful bit operations divided by the total number of processors and the critical path length:

$$\text{Efficiency} = \frac{N_{bit\_ops}}{L_{crit} \times N_p}$$

For the frequency of change case, *partial reconfiguration* can reduce the reload time by allowing individual processing units to change instructions without requiring an entire reload of all instructions in the array. This can increase reload efficiency if a large fraction of the instructions does not change. Partial reconfiguration can also allow portions of the array to change their instructions while other portions of the array continue to operate. Modern FPGAs from Plessey [Ple90], Atmel [Atm94], and Xilinx [Xil96] support partial reconfiguration for these reasons.

8.3.4 Broadcast Instruction Identifier, Lookup in Local Store

A hybrid form of instruction compression is to broadcast a single instruction identifier and lookup its meaning locally. This allows us to use a moderately short, single "instruction" across the entire array in a manner similar to SIMD instruction broadcast. Each processing element performs a local lookup from the broadcast instruction identifier to generate a full length instruction. DPGAs (Chapter 10), PADDI [CR92], and VLIW machines with an independent cache for each functional unit, exhibit this kind of hybrid control.

This technique is similar to a dictionary compression scheme where the set of entries at each "instruction" address makes up one element in the dictionary. The instruction address is the encoded symbol which can now be transmitted into the array with minimal bandwidth. The key benefit here is that the parallel instruction sets can be tailored to each application in the same way the dictionary can be tailored to a message or message type.

Efficiency in this scheme is similar to the task critical path length case above. The difference being, that each single processor need not be dedicated to a single instruction. With local instruction stores each holding n_{inst} instructions, an array of N_p processing nodes can perform up to $n_{inst} \cdot N_p$ different bit operations. In the single instruction configuration case, a critical path of $L_{crit} > 1$ implied a peak, achievable efficiency of $\frac{1}{L_{crit}}$. In this case, the peak, achievable efficiency is $\min\left(\frac{n_{inst}}{L_{crit}}, 1\right)$.

Viewed in terms of instruction change frequency, the additional local configurations can serve as a cache, diminishing the need to fetch array instruction from outside the array. Like a cache if each instruction set required is unique, it will provide no benefit. However, when an array instruction can be kept in the array and used several times before being replaced, we reduce the required instruction bandwidth. Use of a loaded instruction can occur at the operational cycle rate rather than at the bandwidth limited reload rate.

With multiple configurations it is possible to arrange for instruction reload to occur as a background task operating in parallel with operation. If the reload time, $\frac{n_{ibits} \cdot N_p}{I_{BW}}$, is less than the run time within the balance of the loaded instruction memory, $n_{cycle} \cdot t_{cycle}$, and the next instruction can be predicted sufficiently in advance, reload time can be completely overlapped with computation.

8.3.5 Encode Length by Likelihood

Since it is unlikely that all instructions will be used with equal frequency, one can break instructions into a series of smaller words, giving common instruction short encodings. If we huffman encode our instruction into a series of s-bit words, we can, potentially, reduce the instruction distribution bandwidth by a factor of $\frac{s}{\log_2(|\text{instructions}|)}$; that is, $\frac{s}{64}$ if we assume 64 bit instructions.

The efficiency now depends on the expected number of s-bit words required to construct a single, logical instruction. If the instruction stream entropy is low, this kind of encoding can be very efficient — asymptotically approaching one symbol per instruction. Counterwise, if the instruction stream entropy is high – or even flat, it may take a full $\frac{\log_2(|\text{instructions}|)}{s}$ cycles to built up a single instruction. Worse, if the instruction frequency is substantially different from the instruction frequency for which the encoding was optimized, it can actually take more cycles, on average, to build an instruction. Of course, the huffman encodings could be variable, as well, to avoid this mismatch, but the space required to handle programmable huffman encodings will likely exceed the area of several computational units.

8.3.6 Mode Bits for Early Bound information

All of the bits in an instruction do not always need to change at once — or portions of an instruction may change at different rates. Rather than include the infrequently changing portions of the instruction in the word which is broadcast from cycle to cycle, these portions can be factored out of the broadcast instruction and explicitly loaded with new values only when they need to change. This allows us to describe richer instructions with less bandwidth. These locally configured instructions can be seen as a special case of the previous section on likelihood encoding — but in this case we exploit the low frequency of change rather than simply the low frequency of occurrence of some instructions.

Mode bits such as these are used to define operational modes in several architectures. Floating point coprocessors often use mode bits to define rounding modes. Segmented SIMD architectures such as Abacus [BSV$^+$95] and the dynamic computer groups of [KK79] use mode bits to define segmentation of the SIMD datapaths.

Bandwidth savings depends on the number of bits factored out of the broadcast instruction stream. Efficiency depends on the frequency with which non-broadcast instruction values need to change. Typically, it takes an instruction cycle to load each mode value – which is an instruction cycle which does not serve a purpose towards execution.

8.3.7 Themes

Two major themes emerge from the techniques listed here:

1. **Granularity** – How many resources are controlled by a each instruction? From a resource cost standpoint, this is the motivation behind word-wide datapaths, SIMD, and vector processors. processing

2. **Local Configuration Memory** – How many instructions are stored locally per active computing element? Similarly, this is the motivation behind configurable architectures and local memories.

In the next chapter, we will look effects which these techniques have both on resource requirements and on utilization efficiency.

8.4 Compressibility

Of course, we can only succeed in compressing the instruction bandwidth when there is structure to the task description for us to exploit. If the task descriptive complexity really is as large as implied in Section 8.1, we are instruction bandwidth limited, and instruction distribution does determine achievable, computational density.

This suggests we have two extremes in the characterization of computing tasks:

1. **Descriptive Complexity Limited** – the instruction bandwidth to describe the computation limits the rate of execution.

2. **Compute Limited** – the active computing elements performing the required computation limit the rate of execution.

Regular tasks such as signal and stream processing, systolic computations, and computational inner loops are typically compute limited. Irregular, run once, tasks such as initialization, cleanup, and exception handling are typically descriptive complexity limited. Of course, applications tend to have a mix of both elements. It has long been observed that only a small fraction of the code in a typical application accounts for most of the computational time [Knu71]. The regions composing this small fraction are heavily reused, allowing the computation to be described compactly. The code outside of the heavily used fraction, does not benefit from the heavy reuse amortization and will tend to be more description limited.

As with most compression schemes, the amount of compression achievable, in practice, also depends heavily on the frequency of repetition and storage space available. For example, if a task performs a sequence of one million, unique operations, then restarts the sequence, the stream is very repetitive, and an infinite sequence of such such repetitions contains a constant amount of information. However, unless we have space to hold all one million instructions on chip, we will not be able to take advantage of this regularity and low information content in order to compress instruction bandwidth requirements. Further, holding one million instructions on chip is a large cost to pay for instruction storage, even by today's standards.

8.5 Control Streams

In Sections 8.1 and 8.3, we viewed the set of processing elements as having a single, large, array-wide, instruction. In general, the array-wide instruction context may be decomposed into a number of independent instruction streams. This decomposition does not change the aggregate instruction bandwidth which may be required into the array, but it may change the number of distinct contexts used by the array and hence the requirements for instruction distribution and storage.

Let us assume, as in the case of Section 8.3.4, that each processing element has a local store of n_{inst} instructions. Let us also assume we have a series of m independent tasks, each composed of at most n_{task} instructions. The total number of distinct, array-wide contexts may be as large as $(n_{task})^m$, since the tasks are independent and any combination of instructions is possible. If each of the tasks is controlled separately, we need only $n_{inst} = n_{task}$ instructions to describe and control the tasks. If we must control the m tasks with a single instruction stream, that stream requires all $(n_{task})^m$ contexts and hence a larger number of instructions, $n_{inst} = (n_{task})^m$, are required.

This example demonstrates that there is a control granularity which is a distinct entity from the operation granularity introduced in Sections 8.3.1 and 8.3.2. As with operation granularity, we can compress instruction control requirements by sharing the control among a number of operating units. However, if we control too many units by the same control stream, we are forced to use the device inefficiently. In the worst case, we may pay an efficiency or compaction penalty in proportion to the product of the instruction sets of the independent operations which must be combined into a single control stream.

The separate streams of control are, of course, what distinguishes MIMD architectures (Section 4.9), as well as MSIMD (*e.g.* [Bri90, Nut77]) or MIMD multigauge [Sny85] architectures.

Control Threads (PCs)	Instructions per Control Thread	Instruction Depth	Granularity	Architecture/Examples
0	0	0	n/a	Hardwired Functional Unit (*e.g.* ECC/EDC Unit, FP MPY, Hardware Systolic)
	n	1	1	FPGA, Programmable Cellular Automata
			w	Reconfigurable ALUs Programmable Systolic Datapath Arrays
1	1	c	$n_v \cdot 1$	Bitwise SIMD
			w	Traditional Processors
			$n_v \cdot w$	Vector Processors
	n	c	1	DPGA
		8	16	PADDI
		c	w	VLIW
m	1	c	$n_v \cdot w$	MSIMD
m	1	c	1	VEGA
		8	16	PADDI-2
		c	w	MIMD (traditional)

Table 8.1: Instruction Control Taxonomy

8.6 Instruction Stream Taxonomy

Table 8.1 categorizes the various architectures we have reviewed in Chapter 4 according to the granularity (w,n_v), local instruction storage depth (c), number of distinct instructions per control thread (n), and number of control threads (m) supported. This taxonomy elaborates the multiple data portion of Flynn's classic architecture taxonomy [Fly66] by segregating instructions from control threads and adding granularity.

8.7 Summary

In this section, we have seen that the requirements for instruction distribution and storage can dominate all other resources on general-purpose computing devices, dictating the size and density of computing elements. A major distinguishing feature of modern, general-purpose architectures is the way in which they compress the requirements for instruction control. Traditional microprocessors, SIMD, and vector machines reduce the requirements by sharing a single instruction across many bits or words. FPGAs and programmable systolic arrays reduce requirements by maintaining the same instruction from cycle to cycle. VLIW-like architectures use small, local instruction stores addressed by short addresses so that limited instruction distribution bandwidth can effect cycle-by-cycle changes in non-uniform instructions. Each of the techniques used to reduce instruction control resources comes with its own limitations on achievable efficiency should the needs of the application not meet the stylized form in which the instruction bandwidth reduction is performed. Some instruction sequences are more compressible than others, suggesting we have a continuum of task descriptive complexities such that some tasks are, by nature, instruction bandwidth limited while others are parallel computing resource limited. In this chapter we reviewed both the nature of the resource reductions and the efficiency limits which arise from these techniques. In the following chapter, we will combine these effects with our size and growth observations from Chapters 4 and 7 to model the size and efficiency of reconfigurable computing architectures.

In this chapter, we put together the sizings from Chapter 4 and 7, the growth rates from Chapter 7, and the instruction requirements from Chapter 8 to form a unified area model for *RP*-space, a large class of reconfigurable processing architectures. The area model gives us a first order size estimate for reconfigurable computing devices based on the key parameters identified in the previous chapters. We use this model to estimate peak computational density as a function of granularity and on-chip instruction store sizes. We also use it to characterize the way computational efficiency decreases as application granularity and path lengths differ from the architecture's optimal points.

9.1 Model and Assumptions

We assume an array of homogeneous, general-purpose processing elements. For pedagogical purposes, no special-purpose processing units are included. The area for each bit processing element is taken to include:

- Fixed area for the computational function
- Amortized storage space for instructions
- Storage space for data
- Space for interconnect resources
- Amortized space for control

We compute the area per bit processing element as:

$$\begin{aligned} A_{bit\_elm} &= A_{fixed} + \underbrace{N_{SW}(N_p, w) \cdot A_{SW}}_{\text{interconnect}} + \underbrace{\left(\frac{c}{w}\right) \cdot n_{ibits} \cdot A_{mem\_cell}}_{\text{instruction memory}} + \\ & \quad \underbrace{d \cdot A_{mem\_cell}}_{\text{data memory}} + \underbrace{\left(\frac{N_{ctrl}}{N_p}\right) \cdot A_{ctrl}(c_{total})}_{\text{control area}} \end{aligned} \tag{9.1}$$

Table 9.1 summarizes the parameters used in Equation 9.1.

$A_{mem\_cell} = 1200\lambda^2$ is typical of static memory, which we will assume here. Memory cells packed into large arrays are likely to be denser, on average, than small arrays or isolated memory cells. Dynamic memory cells may be a factor of four smaller in large arrays, where appropriate.

Equation 9.1 assumes that interconnect area is proportional to the number of switches. In Sections 7.6 and 7.9, we saw that switch growth rates match or determine interconnect growth rate. In Section 7.9, we did see that wiring might dominate switch growth for large w, which is not accounted by Equation 9.1. $A_{SW} = 2500\lambda^2$ is a constant of proportionality intended to match the number of switches to the empirical interconnect areas typically seen rather than a model of any particular interconnect geometry. Table 9.2 summarizes the number of switches as a function of

Parameter	Role	Assumed Value
A_{bit_elm}	Area per bit processing element	
A_{fixed}	Fixed area per compute element (LUT mux, output flip-flop, buffers)	20Kλ^2
w	Datapath width – number of bit elements controlled by one instruction	
c	Contexts – number of instructions stored per group of w processing elements	
c_{total}	Total number of instruction or data contexts addressed by controller	
n_{ibits}	Number of bits in each instruction	64
A_{mem_cell}	Area of a configuration or data memory cell	1200λ^2
N_p	Number of bit processing elements in the array	
$N_{SW}(N_p, w)$	Number of switches per bit processing element	[Eq. 7.24]
n	Tree arity in modeled hierarchical interconnect	2
k	Number of LUT inputs	4
p	Rent parameter for network	0.5
A_{SW}	Amortized area of each switch	2500λ^2
d	Number of data bits per bit processing element	
N_{ctrl}	Number of independent stream controllers	
$A_{ctrl}(c_{total})$	Area of instruction stream controller	0.3M$\lambda^2 \cdot \log_2(c_{total})$

Table 9.1: Summary of Area Model Parameters

N_p and w for $p = 0.5$, as will be used here. This is the same data which was plotted in Figure 7.5; for $p = 0.5$, the only difference is that we use $\frac{N_p}{w}$ as the network size when determining N_{SW} (See Equation 7.24).

For devices with multiple contexts, a controller manages the selection and sequencing of instructions in the array. The area we use for A_{ctrl} is a rough estimate based on a sampling of processor implementations (See Table 9.3). We assume that the area in the controller is proportional to the number of instruction address bits, $\log_2(c_{total})$. FPGAs traditionally have a single context, making $N_{ctrl} = 0$, while processors have controllers composing the program counter and branching logic.

FPGA Example Traditional FPGAs have $w = 1$ and $c = 1$. Equation 9.1, for $N_p = 4096$, computes $A_{bit\_elm} \approx 870\text{K}\lambda^2$. Comparing with Table 7.1, we see this is in the range of conventional devices.

PADDI-2 Example PADDI-2 is made from 48, 16-bit units. Each has an 8 instruction memory ($c = c_{total} = 8$) and effectively 6 words of data per compute element, $d = 6$. PADDI-2 has

$\frac{N_p}{w}$	N_{SW}	$\frac{N_p}{w}$	N_{SW}	$\frac{N_p}{w}$	N_{SW}
1	0	32	100	1024	252
2	16	64	131	2048	281
4	31	128	162	4096	311
8	49	256	192	8192	340
16	69	512	222	16384	370

Table 9.2: $N_{SW}(N_p, w)$ for $p = 0.5$, $k = 4$, $n = 2$

Design	**Controller Area**
MIPS-X [HHC+87, Cho89]	8Mλ^2
PA-RISC [YFJ+87]	12Mλ^2
VIPER [GNAB93]	12Mλ^2

Table 9.3: Area for Instruction Control Sampling

3-inputs per EXU, $k = 3$, and an initial convergence of $n = 4$. Equation 9.1 predicts 370Kλ^2 per bit operation or 284Mλ^2 for the entire array, which is about half the size of the prototype PADDI-2 die which is 576Mλ^2.

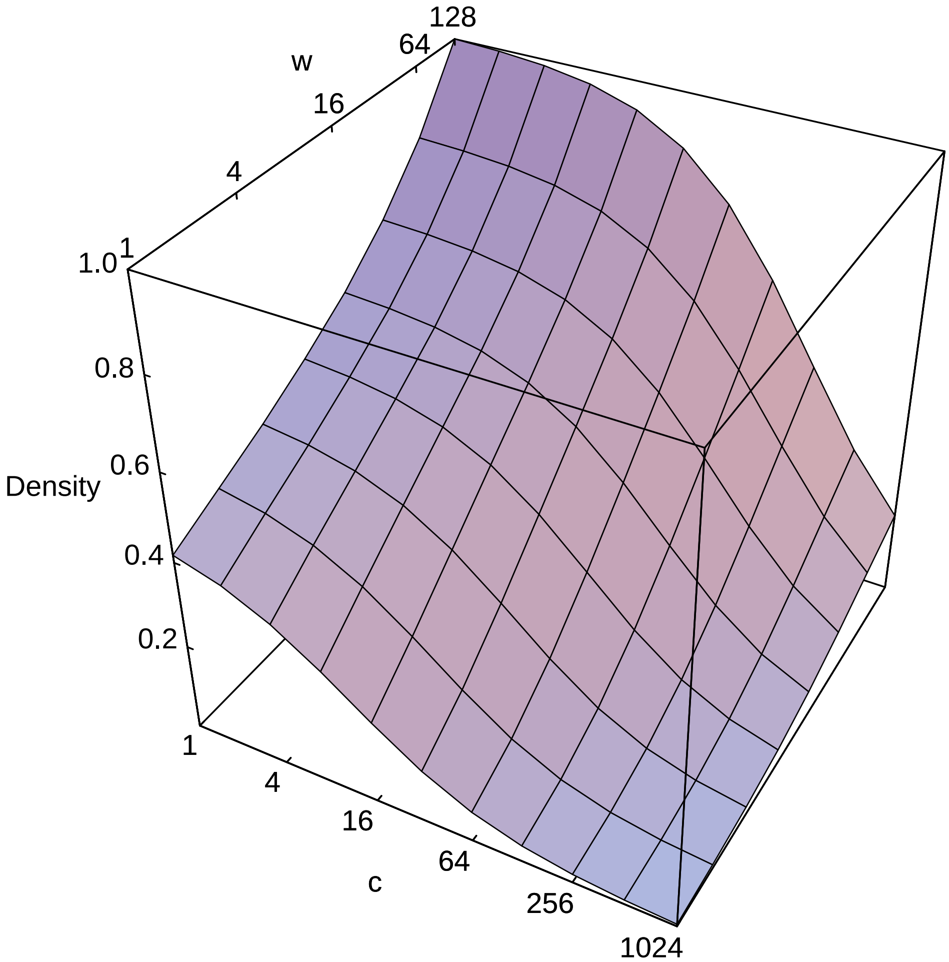
9.2 Peak Performance Density

Using the model, we can examine the peak computational densities from various architectural configurations in *RP*-space. Figure 9.1 plots computational density against datapath width, w, and the number of instructions per function group, c. As w increases there is more sharing of instruction memories and less switches required in the interconnect resulting in smaller bit processing element cell sizes or higher densities. As c increases, there are more instructions per compute element resulting in lower densities. The effect of more instructions is more severe for smaller datapath widths, w, since there are less processing elements against which to amortize instruction overhead.

For single context designs, there is only a factor of 2.5$\times$ difference in density between single bit granularity and 128-bit granularity. At this size, network effects dominate instruction effects, and the factor of difference comes almost entirely from the difference in switching requirements. For heavily multicontext devices at the same number of instruction contexts, the difference between fine and coarse granularity is greater since the instruction memory area dominates (See also Figure 9.2). At 1024 contexts, the 128 bit datapath is 36$\times$ denser than an array with bit-level granularity.

As the number of contexts, c, increase, the device is supporting more loaded instructions; that is, a larger on chip instruction diversity. Figure 9.2 shows how instruction density increases with increasing numbers of contexts alongside the decrease in peak computational density.

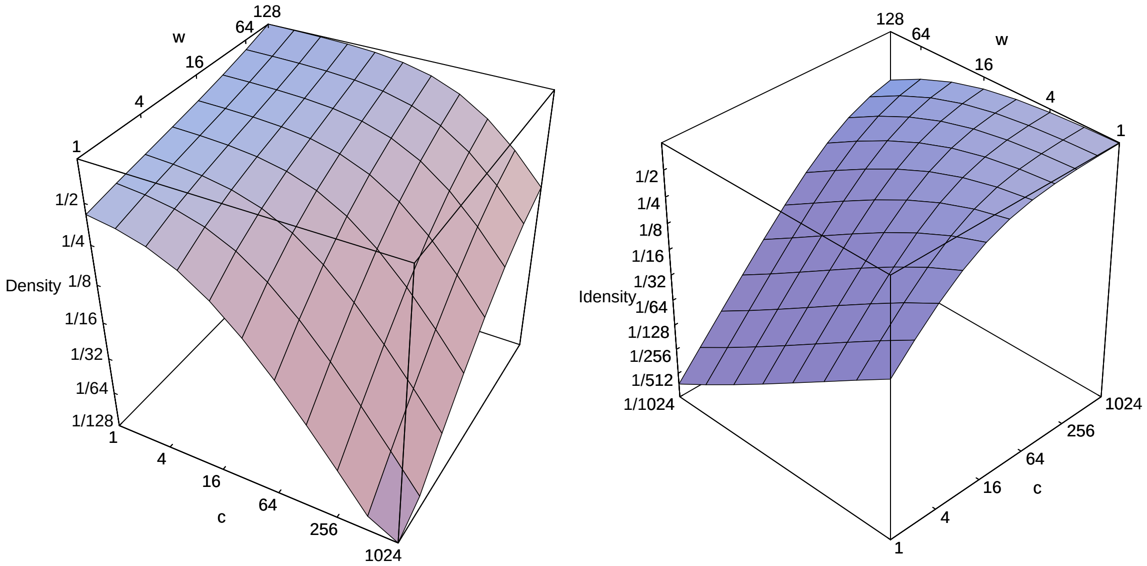
These same density trends hold if we set aside a fixed amount of data memory. The area outside of the data memory will follow the same density curves shown here.



$k = 4$, $n = 2$, $p = 0.5$, $c = d$, $N_{ctrl} = 0$, $N_p = 16384$

Reference Density of 1.0 corresponds to $w = 128$, $c = 1$

Figure 9.1: Peak Computational Density Versus Contexts and Datapath Width



Left – Computational Density; Right – Instruction Density
$k = 4$, $n = 2$, $p = 0.5$, $c = d$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.2: Compute and Instruction Densities Versus Contexts and Datapath Width

9.3 Granularity

As noted in the previous chapter, we can use larger granularity datapaths to reduce instruction overheads. The utility of this optimization depends heavily on the granularity of the data which needs to be processed. As noted in the previous section, the coarser the granularity the higher the peak performance. However, if the architectural granularity is larger than the task data granularity, portions of the device's computational power will go to waste.

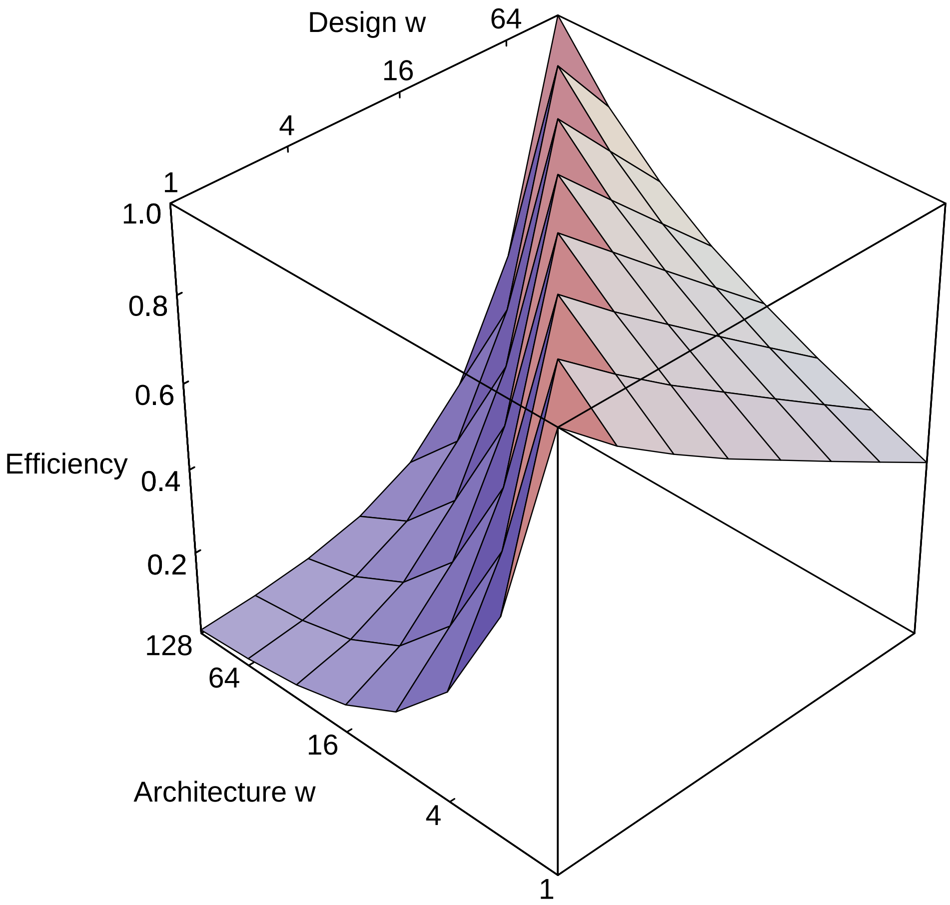
We can model the effects of pure granularity mismatches using the area model developed above. First, we note that the optimal configuration for a given word size will always be the architecture which has the same word size as the task. We can then determine the efficiency associated with running tasks with word size w_{des} on an architecture with word size w_{arch}, by dividing the area required to support the task on a w_{des} architecture by the area required on a w_{arch} architecture. For $w_{des} = C \cdot w_{arch}$, for some integer C, the efficiency is simply the ratio of the bit processing element areas. For $w_{des} > w_{arch}$, the task can run on top of the low w_{des} bit processing elements in the architecture datapath, leaving the remaining processing elements unused. The efficiency here is the ratio of the area of w_{des} bit processing elements from a w_{des} architecture versus w_{arch} bit processing elements from a w_{arch} architecture.

$$\text{Efficiency} = \begin{cases} \dfrac{A_{bit\_elm}|_{w=w_{des}}}{A_{bit\_elm}|_{w=w_{arch}}} & w_{des} = C \cdot w_{arch} \\ 1 & w_{des} = w_{arch} \\ \dfrac{w_{des} \cdot A_{bit\_elm}|_{w=w_{des}, N_p=N_{p_0}}}{w_{arch} \cdot A_{bit\_elm}\Big|_{w=w_{arch}, N_p=\left(\frac{w_{arch}}{w_{des}}\right) N_{p_0}}} & C \cdot w_{des} = w_{arch} \end{cases} \qquad (9.2)$$

Note that a single-chip implementation is assumed for comparison so that there are no boundary effects between components.

Figure 9.3 shows the efficiency for various architecture and task granularities. At $c = 1$, the active switching area dominates. The fine granularity ($w = 1$) has the most robust efficiency across task granularities. The efficiency drops off quickly for large grain architectures supporting fine grain tasks.

Figure 9.4 shows that the robustness shifts as the numbers of contexts increases. For $c = 1024$, the instruction memory space dominates the area. Consequently, the redundancy which arises when fine-grained architectures run coarse-grain tasks is quite large, leading to rapidly decreasing efficiency with increasing task grain size. In this regime, the coarse-grain architectures are more robust, since the extra datapath and networking elements are moderately inexpensive compared to the large area dedicated to instruction memory. For $c = 1024$, $w = 32$, is the most robust datapath width as shown extracted in Figure 9.5.



$k = 4$, $n = 2$, $p = 0.5$, $c = d = 1$, $N_{ctrl} = 0$, $N_p = 16384$

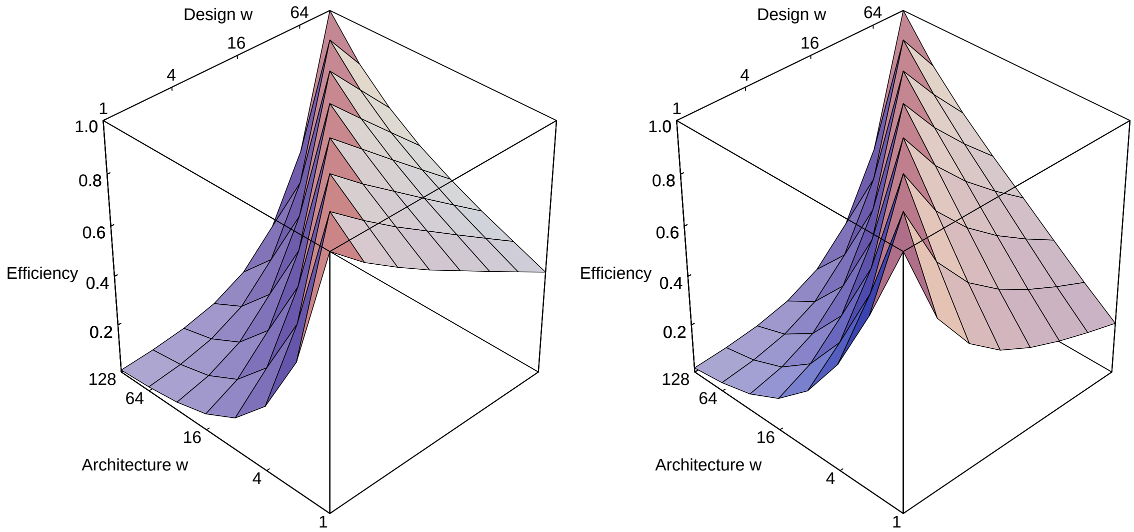
Figure 9.3: Efficiency as a Function of Architectural and Task Granularity for Single Context Architectures

These robust points correspond to the mix where the context memory makes up roughly half the area of the device.

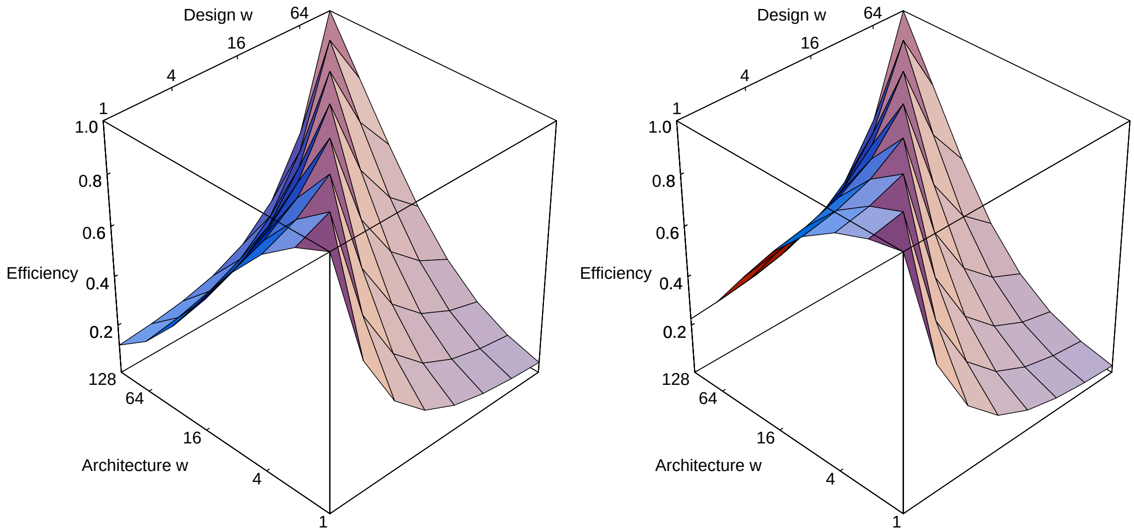
$$A_{bit\_elm}|_{c=0,w=w^*} = \left(\frac{1}{2}\right) A_{bit\_elm}|_{c=c,w=w^*} \tag{9.3}$$

At this point:

- Finer grain devices running coarser granularity tasks waste, at most, a little over half of their area – the memory area plus the switching overhead associated with finer granularity.
- Coarser grain devices running fine-grain tasks waste at most half of their area – the unused datapath area.



Left – $c = 1$; Right – $c = 16$



Left – $c = 256$; Right – $c = 1024$

$k = 4$, $n = 2$, $p = 0.5$, $c = d$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.4: Efficiency as a Function of Architectural and Task Granularity

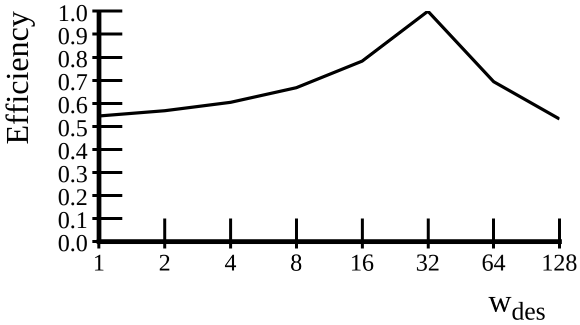


Figure 9.5: Efficiency versus Task Data Width for a 1024-context, 32-bit Granularity Device

9.4 Contexts

We saw in Section 9.2 that the computational density is heavily dependent on the number of instruction contexts supported. Architectures which support substantially more contexts than required by the application, allow a large amount of silicon area dedicated to instruction memory to go unused. Architectures which support too few contexts will leave active computing and switching resources idle waiting for the time when they are needed.

We can model the effects of varying application requirements and architectural support in an ideal setting using the area model. We assume we have a repetitive task requiring N_{op} operations which has a path length l_{path}. In an ideal packing, an architecture with $N_p = \frac{N_{op}}{l_{path}}$ processing units and $c = l_{path}$ instruction contexts can support the task optimally. If $c > l_{path}$, the area per processing element is larger than necessary to support the application. If $c < l_{path}$, it will be necessary to use more processing elements simply to hold the total set of instructions.

$$\text{Efficiency} = \begin{cases} \frac{A_{bit\_elm}|_{c=l_{path}}}{A_{bit\_elm}|_{c=c_{arch}}} & l_{path} < c_{arch} \\ 1 & l_{path} = c_{arch} \\ \frac{c_{arch} \cdot A_{bit\_elm}|_{c=l_{path}, N_p = N_{p_0}}}{l_{path} \cdot A_{bit\_elm}|_{c=c_{arch}, N_p = \left(\frac{l_{path}}{c_{arch}}\right) N_{p_0}}} & l_{path} > c_{arch} \end{cases} \tag{9.4}$$

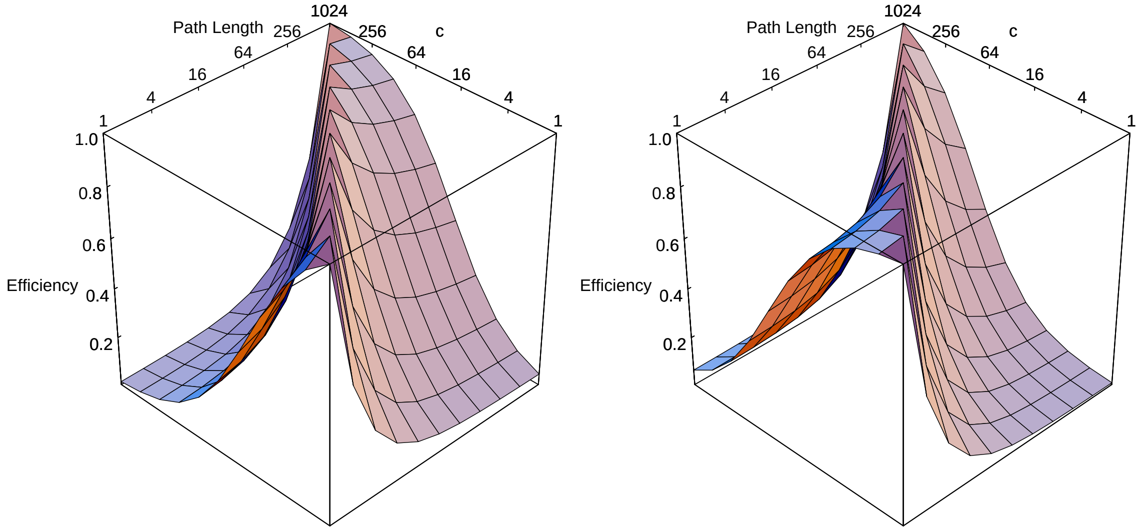
This relation is shown for several datapath widths, w, in Figure 9.6. Again, single chip implementations are assumed for comparison.

The efficiency dropoff for $l_{path} < c$ is less severe for large datapaths, large w, than for small datapaths. Similarly, the dropoff for $c < l_{path}$ is less severe for small datapaths than for large datapaths. This effect is due to the relative area contributed by instructions. In the small w case, the instruction area takes up relatively more area than in the large w case, so costs of extra active area is relatively smaller than in the large w case. In the large datapath case, the instructions make up a lower percentage of the area so the overhead for extra instructions is relatively smaller.

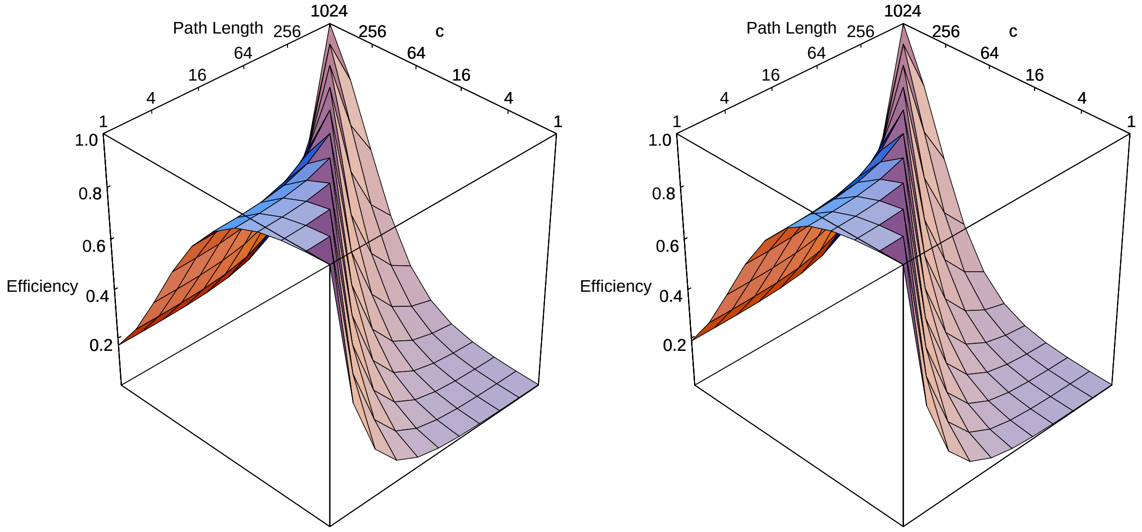
The 16 instruction context case is the most robust across this range for single bit datapaths (See Figure 9.7). Similarly, 256 instruction contexts is the most robust for $w = 128$ (See Figure 9.8). Neither of these cases drops much below 50% efficiency at either the $l_{path} < c$ or $c < l_{path}$ extremes. These "robust" cases correspond to the points where the instruction memory area is roughly equal to the active network and computing area. In either extreme, at most half of the resources are being underutilized. c^*, our robust context selection, can be defined as:

$$A_{bit\_elm}|_{c=0} \approx \left(\frac{1}{2}\right) A_{bit\_elm}|_{c=c^*} \tag{9.5}$$

Remember that the network resource requirements grow with array size. In the $c < l_{path}$ case, where we must deploy more processing elements to handle the task, the total number of processing elements increases causing the switching area per processing element to increase as well. This effects acounts for the fact that the efficiency can drop below 50% and the approximate relation in Equation 9.5.



Left – $w = 1$; Right – $w = 8$



Left – $w = 64$; Right – $w = 128$
$k = 4$, $n = 2$, $p = 0.5$, $c = d$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.6: Efficiency as a Function of Task Path Length and Architectural Contexts

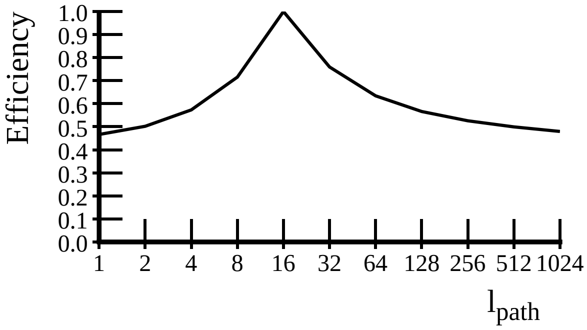


Figure 9.7: Efficiency versus Task Path Length for a 16-context, Single-bit Granularity Device

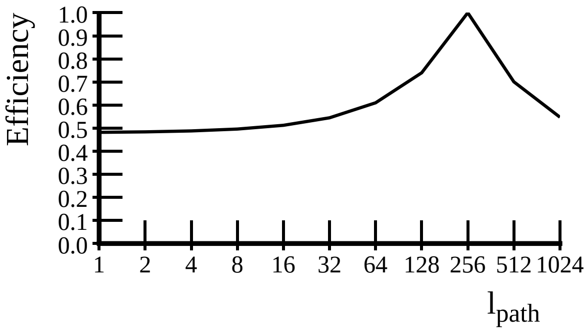


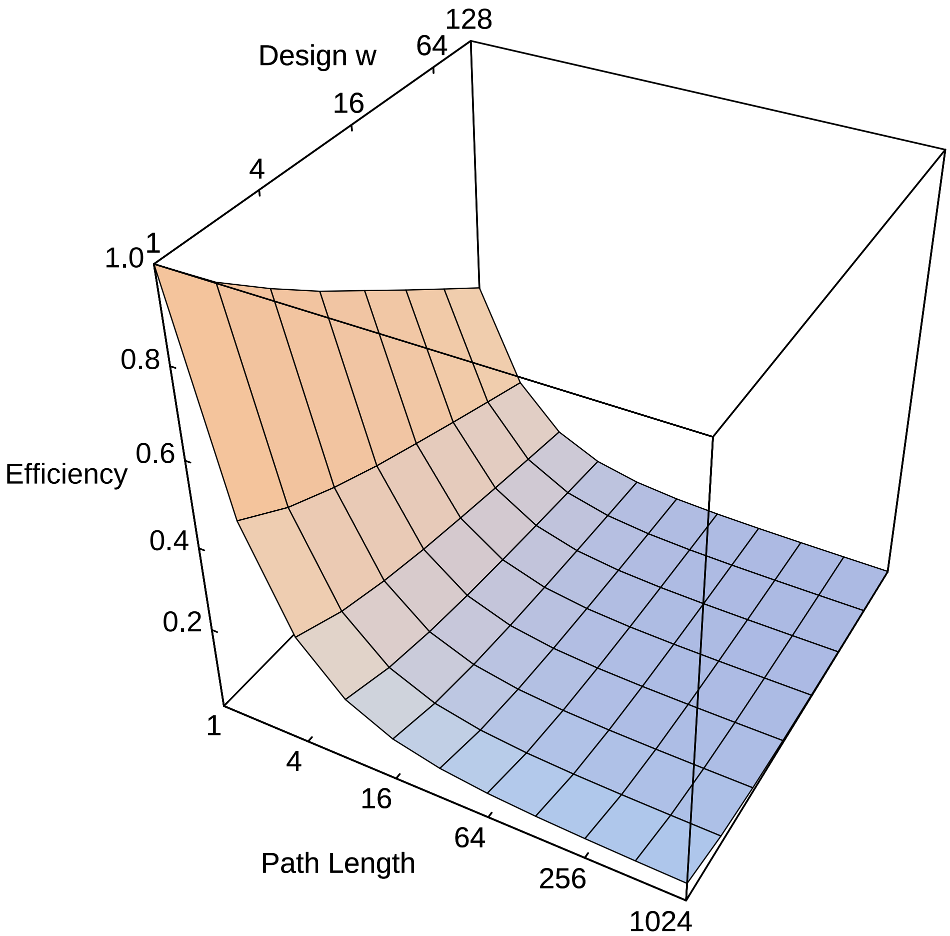
Figure 9.8: Efficiency versus Task Path Length for a 256-context, 128-bit Granularity Device

9.5 Composition

In general, we see cumulative effects of the grain size and context depth mismatches between architecture and task requirements. Figure 9.9 shows the yielded efficiency versus both application path length and grain size for the conventional FPGA design point of a single context and a single bit datapath. The FPGA drops to 1% efficiency for large datapaths with long path lengths. Similarly, Figure 9.10 shows the efficiency of a wide word ($w = 64$), deep memory ($c = 1024$) design point. While this does well for large path lengths and wide data, its efficiency at a path length and data size of one is 0.5%. *Notice here, that the wide, coarse-grain design point is over 100$\times$ less efficient than the FPGA when running tasks whose requirements match the FPGA, and the FPGA is 100$\times$ less efficient than said point when running tasks with coarse-grain data and deep path lengths.*

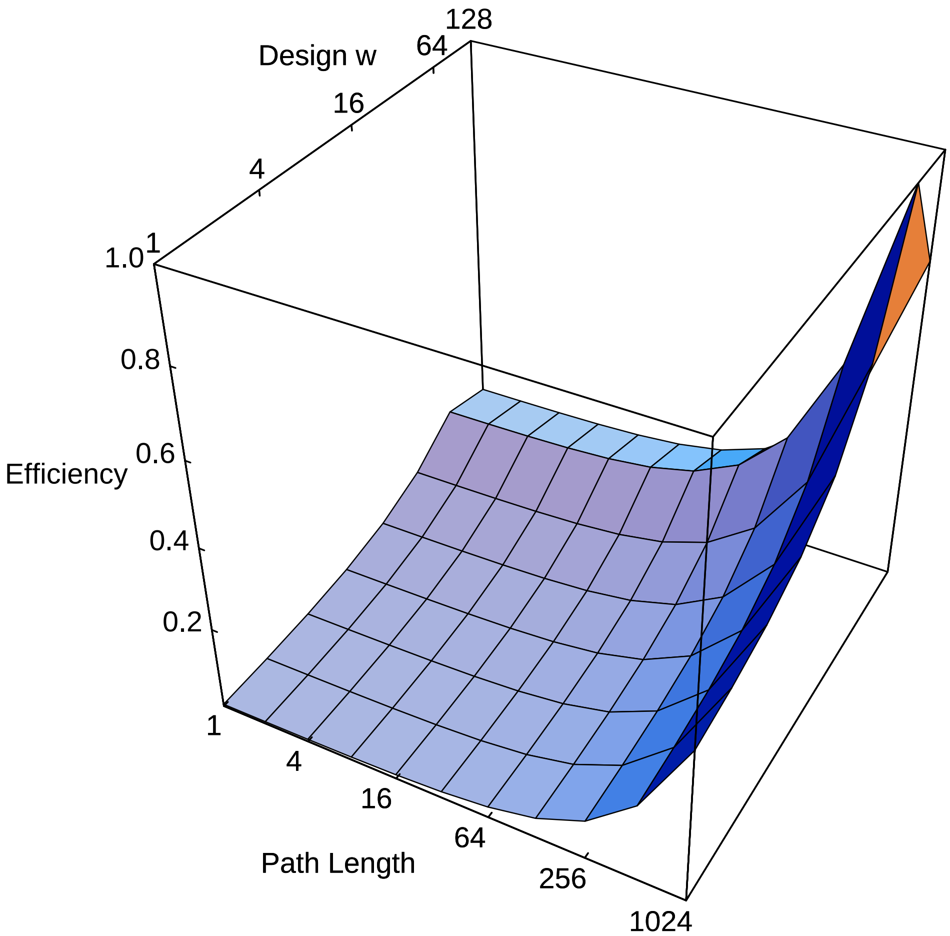
In the previous sections we saw that it was possible to select reasonably robust choices for datapath width or number of instruction contexts given that the other parameter was fixed. We also saw that the robustness criterion followed the same form; that is, the inefficiency overhead can be bounded near 50% if half of the area is dedicated to instruction memory and half to active computing resources. This does not, however, yield a single point optimum since the partitioning of the instructions between more contexts and finer-grain control is handled distinctly in the two cases.

Figure 9.11, for instance, shows the yield for a single design point, $w = 8$, $c = 64$, across varying task path lengths and datapath requirements. While the $w = 8$ and $c = 64$ cross-sections are moderately robust, the efficiencies at the extremas are low. At $l_{path} = 1$, $w = 1$, the efficiency is just under 8%, and at the $l_{path} = 1024$, $w = 128$, the efficiency is just over 8%. This design point is, nonetheless, more robust across the whole space than either of the architectures shown in Figures 9.9 and 9.10.



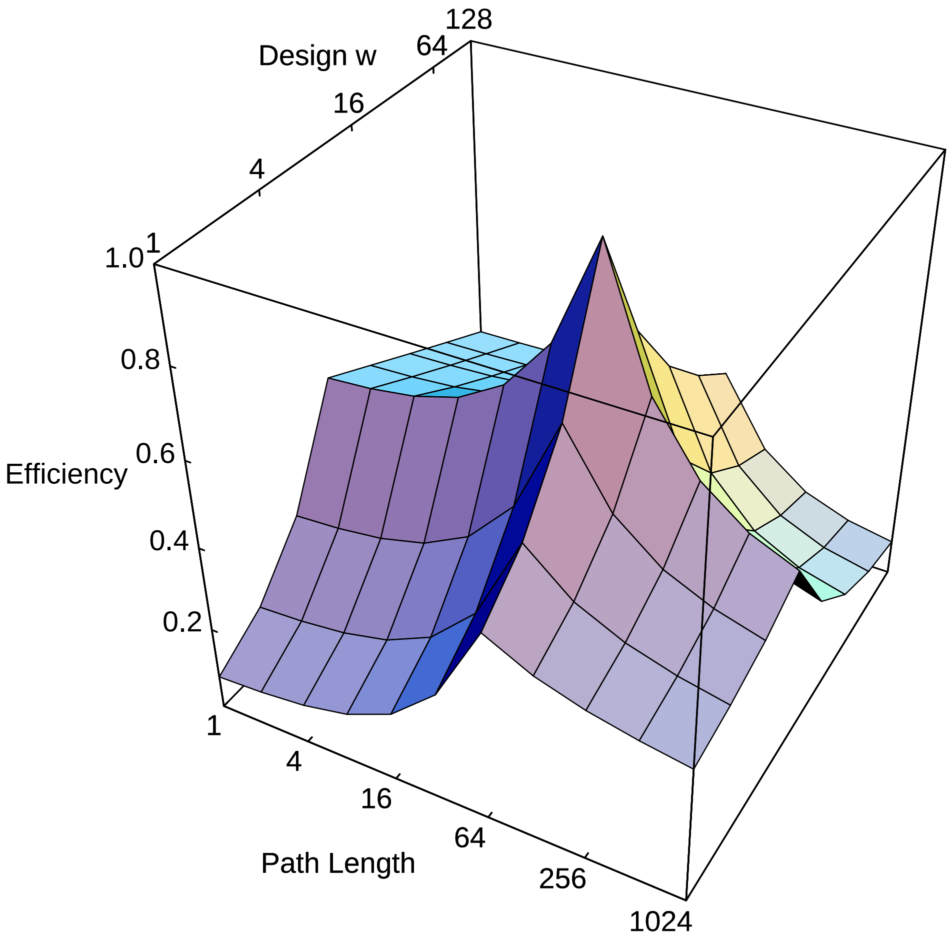
$k = 4$, $n = 2$, $p = 0.5$, $c = d = 1$, $w = 1$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.9: Efficiency for Conventional FPGA Design Point ($w = 1$, $c = 1$)



$k = 4$, $n = 2$, $p = 0.5$, $c = d = 1024$, $w = 64$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.10: Efficiency for Coarse-Grain, Deep Memory Design Point ($w = 64$, $c = 1024$)



$k = 4$, $n = 2$, $p = 0.5$, $c = d = 64$, $w = 8$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.11: Efficiency for Fixed $w = 8$, $c = 64$

9.6 Summary

The area model shows us how peak capacity depends on granularity organization and instruction support. We see that the penalty for fine-granularity is moderate, 2.5$\times$ difference between $w = 1$ and $w = 128$, in the configurable domain where there is only instruction memory for a single context. The penalty is large, 36$\times$, in the heavy multicontext domain. We also looked at the effects of application granularity and path length. In both cases, we found that, given *a priori* knowledge of either the task granularity or context requirements, we could set the other parameter such that the efficiency did not drop significantly below 50% for any choice of the unknown parameter. This is significant since the peak performance densities across the range explored differed by roughly a factor of 200$\times$. For both of these cases, the robust selection criterion is to choose the free parameter such that instruction memory accounts for one half of the processing cell area. We saw that the effects of granularity and path length mismatches were cumulative and that FPGAs running tasks suited for deep memory, coarse-grained architectures can be only 1% efficient. If we must select both the datapath granularity and the number of contexts obliviously, we cannot obtain a single design point with as robust a behavior as when we only had one free parameter. A good design point across this region of the *RP*-space suffers a 13$\times$ worst-case overhead.