Extract from: Reconfigurable Architecture for General Purpose Computing

Special-Purpose Computing We build computing devices to algorithmically transform raw input data into results. Special-purpose computing devices are designed with one particular transformation embedded into their architecture and implementation. Each such device can solve only the particular transformation problem, and that problem is set prior to device fabrication. Conventional fabrication techniques require long turn-around (weeks to months) to produce devices, high up front costs for setup, and large volume sales to amortize out fixed costs for design, tooling, and equipment.

Many of the characteristics which come with special-purpose computing devices are undesirable or untenable in numerous situations.

- Device dedicated to a single function
  - Device can be quickly oboslesced as functional requirements often change, transformations are tuned, algorithms advance, and missions and tasks evolve.
  - When the function needed by a task is time or data dependent the special-purpose devices for functions which are not needed at some point in time sit idle and cannot be used for any other function which may be required by the task.
  - When lower throughput is required from the device than its native capability, the device has spare capacity which cannot be put to productive use.
- High up front cost
- Long delay from concept to delivery
- Economical only in volume

General-Purpose Computing General-purpose devices are our alternative to these fixed function devices. Here, we build computing devices which can be configured to solve a variety of computing problems. Instead of building a device with exactly the computational units and hardwired dataflow necessary to solve a single problem, we build a device with a set of primitive computational elements interconnected via a flexible interconnect. Post-fabrication, we control the behavior of the device with instructions, extra inputs which tell the device what computations to perform and how to route data during the computation. As a result, we:

- Make a single device appealing for a wide-range of tasks. While each, individual task may lack the volume required for a dedicated device to be economical, the general applicability across many tasks provides the volume necessary to make the general-purpose device economical.
- Eliminate the fabrication delay necessary to put a new computational task into use.

- Eliminate the up front cost associated with producing custom hardware for a task.
- Make it possible to customize a single device to perform any of a large number of different computing task, allowing the device to adapt to changes in requirements, or share its capacity among a variety of computing tasks.

The *RP*-space defined here models a large domain of reconfigurable architectures within the general-purpose architecture space.

Reconfigurable Computing Costs Reconfigurable devices gain their breadth of use at a cost in computational density. Reconfigurable devices must add:

1. Flexible interconnect or data flow
2. Instructions to control compute units and data flow

Additionally, the computational units in these devices must be more general than in the special-purpose devices where each compute unit may perform a single, focussed computation.

Replacing fixed interconnect with flexible interconnect is the most costly single addition for reconfigurable architectures. A decent amount of programmable interconnect may add two orders of magnitude in size to the reconfigurable implementation compared to the fully special-purpose implementation of the same task.

Instructions In contrast, the area required to hold a single, device-wide configuration is, itself, an order of magnitude smaller than the interconnect. That is, the area taken by a single instruction is generally an order of magnitude smaller than the active interconnect which it controls. However, if we allocate space to hold tens of instructions per active compute element, the total instruction memory area can easily equal the active compute and interconnect area. By the time we add hundreds of instructions, the instruction memory area can dominate even the flexible interconnect. With this additional order of magnitude in overhead, such a reconfigurable device can easily be three orders of magnitude larger per computational element than its special-purpose counterpart.

Since instruction area can quickly come to dominate even the flexible interconnect, when building reconfigurable computing architectures we often look for structure in typical computational problems which will allow us to reduce the instruction size. One common technique is to control several pieces of interconnect and computational elements with a single instruction. That is, we assemble wide datapaths which are controlled together. This reduces the size of the configuration by reducing the number of instructions required to specify device behavior at any point in time.

Consequently, when we build a reconfigurable computing device, we must make decisions about:

- How many primitive computational elements are directed by each instructions?
- How many instructions are controlled by each controller?
- How many instructions are stored on chip?
- How rapidly can the instructions change, chip-wide?

The answers to these questions place a particular reconfigurable device in the *RP*-space. The answers to each of these questions also determines the size of the reconfigurable device and its efficiency on various tasks.

- If the task has data elements of width of $w_{task} > w_{arch}$, the architecture provides finer instruction control than necessary and pays an overhead for redundant instruction memory.
- If the task has data elements of width of $w_{task} < w_{arch}$, the architecture does not allow control over the compute element at the fine granularity of the task, and computational capacity in the architecture goes to waste.
- If the task needs to cycle through only a few different instructions, but the architecture provides large instruction memories, the reconfigurable device is unnecessarily large for the task, wasting area in unused memories.
- If the task needs to cycle through a large number of different instructions at different times but the architecture provides small instruction memories, the reconfigurable device will not be able to store all the instructions logically associated with each computational element. Extra computational elements will be required simply to hold all of the task's instructions, but these extra computational elements will effectively sit idle during computation.
- If the task requires more independent control of computing resources than provided by the architecture, either resources will go unused since they cannot be controlled or memory requirements will increase greatly to compensate for the lack of control independence.
- If the task requires less independent control than the architecture supplies, the additional controllers and resources are redundant and add to device overhead.
- If the task requires rapidly changing instructions, but the architecture does not meet the required bandwidth, computational resources sit idle, paced by task description bandwidth not the availability of computing resources.
- If the task can handle slowly changing instructions, but the architecture dedicates significant area to providing high instruction delivery bandwidth, much of the dedicated area is overhead making the device larger than necessary for the task.

Interconnect In devices where the ratio between instructions and compute elements is low, flexible interconnect will remain the dominant area feature in reconfigurable devices. Here, a device must decide how richly to interconnect the compute elements. Rich interconnect makes the routing area even greater, while inadequate interconnect can make it impossible to make use of the available computing elements. The choice in interconnect richness determines where the architecture will be most efficient.

- If the interconnect is richer than needed by the task, the device will be larger than necessary.
- If the interconnect is not as rich as required by the task, the task must be laid out sparsely on the architecture. Portions of the interconnect and compute resources are wasted as they cannot be used.

In all computing devices there are two components associated with routing data between producers and consumers:

1. Spatially routing intermediates from the compute element which produced them to those which consume them

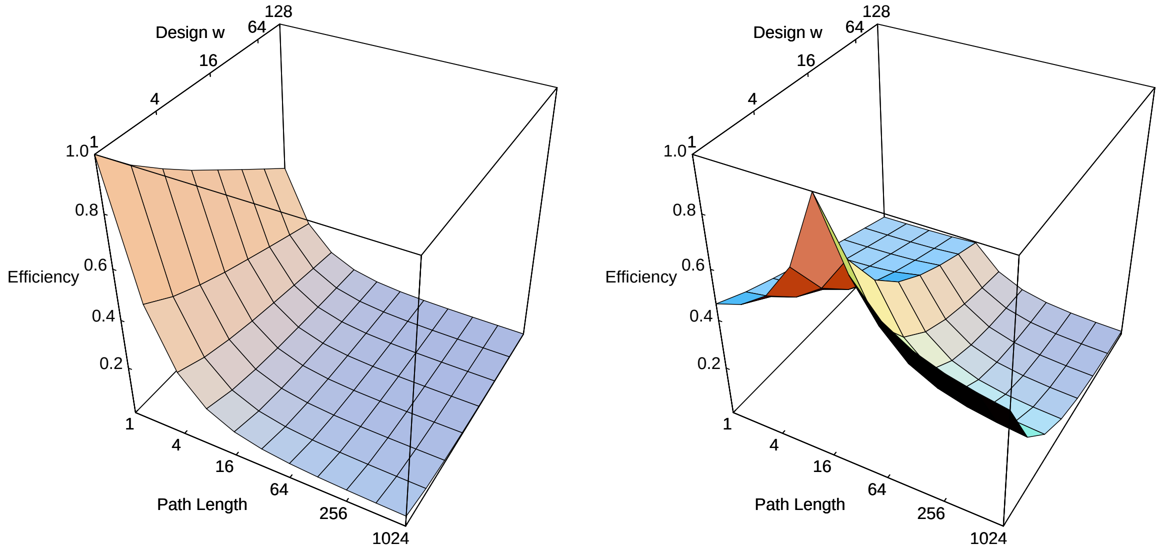
2. Retiming the intermediates for the time when the consumer is ready to use them

Particularly, in reconfigurable devices with expensive, flexible interconnect, memories can hold values for retiming more cheaply than active interconnect.

Degrees of Generality and Reconfigurability There are, of course, degrees of "generality" between fully special-purpose devices and general-purpose devices. Some special-purpose devices are given limited configurability to broaden there use – *e.g.* a typical UART can be configured to handle different data sizes, data rates, and parities. Some devices are targeted at being "general" within very specific domains. Digital signal processors are one of our most familiar examples of a general-purpose, domain-optimized device. The domain may dictate the typical data element size or desirable instruction depth. Further, the domain may allow a more structured programmable interconnect to suffice. Nonetheless, to the extent that we have post-fabrication control over the computations which a device performs, the device will have some form of instructions and will generally have some level of flexible interconnect. With these features it exhibits reconfigurable characteristics, and many of the the architectural characteristics, relations, and issues we have identified in our, more ideal, *RP*-space.

FPGAs Conventional FPGAs fall at a moderately extreme point in our *RP*-space with single bit wide datapaths and single instruction deep instruction memories. At this point, they are efficient on the highest throughput, fine-grained computing tasks and their efficiency drops rapidly as the task throughput requirements diminishes and the word size increases.

Beyond FPGAs in the Reconfigurable Computing Space Beyond FPGAs there is a rich reconfigurable architecture space. Our DPGA represents one different point in this architectural space (See Figure 14.1). The DPGA retains the bit-level granularity of FPGAs, but instead of holding a single instruction per active array element, the DPGA stores several instructions per array element. The memory necessary to hold each instruction, is small compared to the area comprising the array element and interconnect which the instruction controls. Consequently, adding a small number of on-chip instructions does not substantially increase die size or decrease computational density. The addition does, however, substantially increase the device's ability to efficiently handle lower throughput, more irregular computational tasks. At the same time, a large number of on-chip instructions is not as clearly beneficial. While the instructions are small, their size is not trivial – supporting a large number of instructions per array element (*e.g.* tens to hundreds) would cause a substantial increase in die area decreasing the device efficiency on regular tasks. Consequently, we see that we can achieve a design point which is moderately robust across a wide range of throughput variations by balancing the instruction memory area with the fixed area for interconnect and computational units.



FPGA $c = d = 1$

DPGA $c = d = 16$

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

Figure 14.1: FPGA and DPGA efficiency in *RP*-space

The importance of efficiently supporting retiming of intermediates was most clearly demonstrated in the context of the DPGA design. Here, we saw that the benefits of deeper instruction memories were substantially reduced if we forced retiming to occur on active interconnect. However, when we provided architectural registers so that retiming could take place in registers, DPGAs were able to realize typical computing tasks in one-third the area required by conventional FPGAs.

While we did not detail them in this thesis, multiple context components with moderate datapaths also come down essentially in this reconfigurable architectural space. Pilkington's VDSP [Cla95] has an 8-bit datapath and space for four instruction per datapath element. UC Berkeley's PADDI [CR92] and PADDI-II [YR95] have a 16-bit datapath and eight instruction per datapath element. All of these architectures were originally developed for signal processing applications and can handle semi-regular tasks on small datapaths very efficiently. Here, too, the instructions are small compared to the active datapath computing elements so including 4-8 instructions per datapath substantially increases device efficiency on irregular applications and robustness to throughput variations with minimal impact on die area.

Flexible Deployment of Instruction Resources While architectures such as these are often superior to the conventional extremes of FPGAs, any architecture with a fixed datapath width, on-chip instruction depth, and instruction distribution area will always be less efficient than the architecture whose datapath width, local instruction depth, and instruction distribution bandwidth exactly matches the needs of a particular application. Unfortunately, since the space of allocations is large and the requirements change from application to application, it will never make sense to produce every such architecture and, even if we did, a single system would have to choose one of them. Flexible, *post fabrication*, assembly of datapaths and assignment of routing channels and memories to instruction distribution enables a single component to deploy its resources efficiently, allowing the device to realize the architecture best suited for each application. Our MATRIX design represents the first architecture to provide this kind of flexible instruction distribution and deployable resources. Using an array of 8-bit ALU and register-file building blocks interconnected via a byte-wide network, our focus MATRIX design point has 3$\times$ the raw computational density of processors and can yield 10$\times$ the computational density of conventional processors on high throughput tasks.

15. Projections

In Parts III and IV, and Chapter 14, we focussed on reconfigurable, general-purpose computing devices roughly characterized by *RP*-space. In that focussed domain, we were able to look closely at area costs, computational density, and efficiencies. General-purpose devices, more broadly, also share many of the characteristics (*e.g.* instruction depth and width, interconnect richness, data retiming) which we identified as key architectural parameters in *RP*-space and in the more detailed architectural studies. In this chapter, we speculate more broadly on what the relationships developed while focusing on reconfigurable devices in *RP*-space might tell us about general-purpose architectures, in general. We emphasize that these extrapolations may overly trivialize important architectural aspects which did not arise in *RP*-space, and we attempt to identify those aspects during the discussion.

15.1 Role of Memory in Computational Devices

In our computing architectures, we have seen memory show up in two roles:

1. instruction storage
2. data retiming

Neither appears to be really fundamental for computing, but both are of pragmatic value as they facilitate resource sharing and reuse which allows us to implement computing functions in less area when throughput requirements are limited. In special-purpose computing architectures we did not need instructions. For ease of construction, we often use clocked registers to tolerate variable delays through primitive blocks, but otherwise memory for retiming arises primarily from serialization and reuse of common resources.

15.1.1 Memory for Instructions

Instruction memories reduce hardware requirements in two ways by allowing:

1. a fabricated resources to perform any of several functions
2. a resource to be shared among several different functions during a single computation

Select Function In our general-purpose devices, a single resource can perform any of a number of different functions. This allows us not to have a single, dedicated piece of hardware for every possible function ever desired. For an application or device requiring n primitive component computations, this realizes an important compression from "all possible computing functions made of n primitives" to "all n primitive computing functions required by this application." Here, each primitive computing element needs a configuration memory to tell it what computation to perform and where its inputs are produced among the computing elements. The per computing element overhead we pay for this reduction is high, mostly in terms of flexible interconnect, but this quickly balances the exponential reduction realized by only having to implement the n functions required by this task.

We can return to our pedagogical 4-LUTs to see this reduction more concretely. There are $O\left(\left(2^{\left(2^4\right)}\right)^n\right)$ different functions which can be implemented with n, four-input gates. So, even with the $100\times$ area overhead per gate required to support flexible interconnect, our programmable, n 4-LUT device is significantly smaller than implementing all possible n input functions for anything other than trivial values of n.

Shared Function Our general-purpose devices also allow us to share each piece of hardware among multiple functions within a single computing task. This aspect allows us to compress area requirements further from "all n primitive computing functions required by this application" to "all m computing functions required at one point in time in order for this application to achieve the requisite computational throughput." Here, we take advantage of the fact that the configuration memory to describe a computing function is smaller than the active area required to route its inputs and compute the result. In the extreme, this allows us to reduce the area required for a computation from the area required for n programmable compute primitives and their associated interconnect to the area required to store the description of the computation and interconnect performed by n programmable compute elements.

What we trade for this reduction is computational throughput. With only m active computing functions, it requires us, at least, $\frac{n}{m}$ cycles to perform the computation of the n primitive computing functions in the original task. Sometimes, the original task already had a dependency structure such that this reduction comes for free or at minor costs. Other times, we are trading increased evaluation time for reduced implementation area. In the limit, where we have a single computing element with instruction memory to hold n instructions, the task can take n cycles to evaluate.

We often talk about *virtualizing* hardware resources. The virtualization really substitutes a less expensive resources (*e.g.* an instruction in memory, state in memory, cheaper forms of memory) for a more expensive one (*e.g.* a piece of hardware to actually perform a function, fast access memory). Behind all of these virtualizations, we must ultimately have some form of physical memory to hold the description of the virtualized resources and their state.

Notice that we can continue to push technology and structure in order to reduce this last limit, but it cannot be avoided. We can apply aggressive memory technology, such as DRAM or flash memory, to reducing storage cell size. We can store data on different media, such as magnetic disks or tape. We can exploit structure in the task description to compress the number of bits required down to the Kolmogorov complexity limit. *In the limit, however, we ultimately require sufficient area to store the description of the computing task and no further reduction is possible.*

We noted in Section 4.4 that memory can be used as a general-purpose computing element. That role of memory is a special case of role of memory as instructions. The memory contents act as an instruction which configures the memory array to provide the desired computational transformation between the address inputs and the data outputs. In Section 4.5, we saw that computational portion of conventional FPGAs, the LUTs, were programmed in exactly this way.

15.1.2 Memory for Retiming of Intermediate Data

Once we begin to reuse primitive compute functions for different roles at different times, we introduce the need to assure that the right data arrives at the inputs of the function at the right time. This need is particularly acute when we serialize execution and use a single primitive to perform multiple different functions, but it also appears when we reuse a primitive to perform exactly the same function on logically different data. Since programmable interconnect is expensive, we use memories as an inexpensive way to provide the temporal retiming necessary for correct execution.

The use of memory for retiming is pragmatic. We could get away with little more than pipeline registers on interconnect. However, it is cheaper to transport data forward in time through memory than over interconnect. If we do not take advantage of this, much of the area savings potentially associated with serializing execution and sharing primitive compute elements cannot be realized.

The requirements for data retiming depend on the interconnect structure of the problem, not the number of compute elements in the task. The amount of retiming does depend on the amount of serialization. With more parallelism, more data can be consumed as soon as it has been spatially routed avoiding the need for retiming. As we compress size requirements by converting task compute primitives into instructions sharing a small number of physical compute elements, we must ultimately have space to store all computation intermediates at the widest point in the computation flow. That is, we ultimately need space for all the live intermediates in a computation. The number of such intermediates depends on the task and its mapping. The mapping should try to minimize the number of such intermediates.

Note that all non-instruction uses of memory fall into this category.

- Register File – We have already seen that register files perform the same functions as our input retiming registers, transporting results in time from the point of production to the point of consumption.
- Main memory, including data caches – All the data results stored in memories are being transported between the point of production and consumption.
- Buffers an FIFOs – These are explicitly retiming the arrival of data to a time when a portion of the system is ready to consume the data.

If we had not sequentialized execution and shared computational resources among multiple tasks, we would not need these memories.

Even special-purpose devices often sequentialize their processing of data so that a few, fixed compute elements can serve to process data with nominally different roles. The most common example of this is in audio, video, or image processing. Rather than dedicating a separate computational unit to each pixel in a frame, many pixels are processed on the same computational unit. The pixel data stream is serialized into and out of the special-purpose device. The pixels within the frame often need to be retimed so that the right pixel values are presented to the compute elements at the right time. For example, when pixels are fed in by rows, it is often necessary to perform row-wise retiming on data so that the compute element can calculate column-wise interactions between pixel elements. If all the data necessary for the computation were presented simultaneously and all of the output was produced at once, this retiming would not be necessary. However, serialization and reuse is often necessary to make the amount of hardware resources, including component input and output bandwidth, tractable. The serialization allows us to share all of the hardware resources, but requires that we provide unique storage space for intermediate data so that we perform the correct computation on the shared resources.

15.1.3 Implications

There are two important ideas to take away from these observations on the role of memory:

1. Memories in computer architectures facilitate the sharing and reuse of expensive resources. It is the pragmatic fact that the memory necessary to hold an intermediate or an instruction is smaller in conventional technologies than the active computing and interconnect elements which process the data according to the instruction which makes it worthwhile to use memory to reduce implementation area requirements.

2. As we go to heavier sharing, each doubling of our sharing factor does not result in a halving of implementation area because we always leave behind a memory residue composed of (1) instructions and (2) intermediate data. In the limit, the size of our computing element for a task is dictated by the area to hold the instructions to describe the task and the intermediate, live data which must be stored as the task computes.

15.2 Reconfiguration: A Technique for the Computer Architect

Device architects are often faced with the dilemma of balancing semantic expressiveness with instruction distribution bandwidth. In processors, only a few bits are allocated to instruction specification limiting (1) the number of different computations which can be selected and (2) the number of different sources which can be expressed. The latter manifests itself as limited address space and limited size register files, while the earlier is often taken for granted. Architects are reluctant to increase instruction width because it entails added costs in (1) on- and off-chip storage space for all instructions, (2) distribution bandwidth, and (3) power for instruction distribution. However, limited semantic expressiveness can force the processor to issue a large number of instructions to perform the desired computation, resulting in even great losses in time and power efficiency.

Conventional processors generally support an ALU which performs basic operations on 2 or 3, word-wide data inputs. Today we see typical word sizes of 32 and 64 bits. Conventional processors further limit their instruction size to the word size to limit instruction bandwidth requirements. As a consequence of this limitation, it can often take a large number of instructions to specify an operation which is not inherently difficult for the active silicon to perform.

To appreciate the magnitude of the semantic disparity here, we notice that there are:

$$N_{alu2ops} = 2^{\left(w2^{(2w)}\right)}$$

functions from two w-bit wide inputs to one w-bit wide output. If we limit the specification of our function to w bits, we can only address 2^w functions with this instruction. Thus, if all of the $N_{alu2ops}$ were equally likely, on average, it would take at least $2^{(2w)}$ cycles to compute a function.

In practice, a good fraction of the w bits are dedicated to operand selection, increasing the severity of the instruction limitation. While all operations are not equally likely, in practice, this disparity demonstrates that conventional processor design makes an *early binding*, pre-fabrication time, decision on the effective cost of basic operations. *Many applications cannot use the active silicon area on conventional processors efficiently since they cannot directly issue the instructions native to the task.*

Reconfiguration is a technique which allows us to find a resolution to this dilemma. Reconfiguration allows us the semantic expressiveness of very large instructions without paying commensurate bandwidth and deep storage costs for these powerful instructions. What we give up in this solution is the ability to change the entire instruction on every cycle. Rather, the rate of change of the full instruction is determined by the instruction bandwidth we are willing to expend. The distinction between instruction bandwidth which delivers all the semantic content on every cycle and instruction bandwidth that can be used to load a larger semantic instruction is an important one because configured instruction bits which can be used for many operational cycles do not require additional instruction bandwidth once loaded. Returning to our simple calculation above, it may take us $2^{(2w)}$ cycles to load a specification for an instruction the first time it is encountered. However, if this value is loaded into configuration memory, subsequent uses can operate using the loaded data, avoiding the time required to redundantly specify the operation. An architecture without configuration would require the $2^{(2w)}$ cycles each time the computation is required. Reconfiguration thus allows us to compress instruction distribution requirements in cases where the instruction changes slowly or infrequently.

Reconfiguration opens a middle ground, or an intermediate binding time, between ‘behavior which is hardwired at fabrication time’ and ‘behavior which is specified on a cycle by cycle basis.’ This middle ground is useful to consider in the design of any kind of of computing device not just conventional FPGAs. When designing a device with any general-purpose capabilities, the architect’s decision can extend beyond what expressiveness to include or omit based solely instruction size and bandwidth. Rather, the architect should consider the expressiveness which may be required for efficient task implementations and the rates at which various parts of the task description change. Characteristics of the task which change infrequently can be configured rather than broadcast.

15.3 Projecting General-Purpose Computing onto *RP*-space

Our *RP*-space model articulated in Chapter 9 provided architecture implementation area estimates based on a few major parameters. Instruction depth (c), data width (w), interconnect richness (p), and intermediate data retiming support (d) have been the focus of our discussion in Parts III and IV. More broadly, these parameters have rough analogs in all general-purpose architectures. One can, thus, generally project a general-purpose architecture into a point in *RP*-space by identifying these parameters and abstracting away architecture characteristics not covered in the *RP*-space model.

15.3.1 General Hazards

The more general projection to *RP*-space may be hazardous as it ignores many detailed characteristics of real architectures in the broader general-purpose architecture space, such as:

- No special-purpose capacity – we explicitly assumed only general-purpose building blocks for *RP*-space. Most nominally "general-purpose" architectures include blocks of special-purpose logic. The special-purpose blocks do not provide general-purpose capacity, but can provide high density to applications when the specialized structures match the task requirements of the application. The multiply example reviewed in Chapter 5 is the most common instance of a special-purpose block added to general-purpose architectures.

- Homogeneous processing arrays – we explicitly assumed homogeneous arrays. Because of the mixed processing requirements in most computing tasks, a hybrid array which mixes processing blocks with different parameters may be quite interesting. Extending the model to reasonably encompass mixed architectures is an interesting direction for future work.

- No boundary effects – we assumed single chip implementations for all of our comparisons, in effect, assuming that full task implementations for all alternatives fit onto a single die. Since we are looking at 10's to 100's of Gλ^2 of silicon area in the near future, a large class of computing tasks or primary subtasks can be reasonably placed on a single die such that the assumption seems reasonable. However, we also saw that inefficient design points can easily be two orders of magnitude larger than efficient points. With this much area variation, it is not really reasonable to assume that both implementations are single die implementations. The larger implementation is likely to require a multiple-chip solution and will suffer further degradation in latency and bandwidth due to chip crossings.

  Another consequence of ignoring boundary effects is that the model trivializes limited device i/o effects between different components that might make up the core of a general-purpose processing system. Notably, systems have traditionally placed bulk memory on different ICs from the processing. As a result, care must be taken to prevent the limited boundary i/o between compute and memory devices from being the performance limiting bottleneck. This care often shows up as additional mechanism and memories on the processing chips to make most effective use of the limited interchip i/o bandwidth and high interchip i/o latency.

15.3.2 Processors, FPGAs, and *RP*-space

For years, microprocessors have been our canonical example of single-chip, general-purpose computing devices. It is tempting to try to understand the relation between processors, FPGAs, and *RP*-space. In Part II, we took a broad, empirical look at these devices and made a few, high-level observations on their relative efficiencies. In this section, we revisit this comparison projecting both architectures into *RP*-space.

Conventional processors have:

1. moderately wide datapath which have been growing larger over time (*e.g.* 16, 32, 64 bits)
2. support for large on-chip instruction caches which have also been growing larger over time and can now hold hundreds to thousands of instructions (contexts)
3. high bandwidth instruction distribution so that one or several instructions may be issued per cycle at the cost of dedicating considerable die area for instruction distribution
4. a single thread of computation control

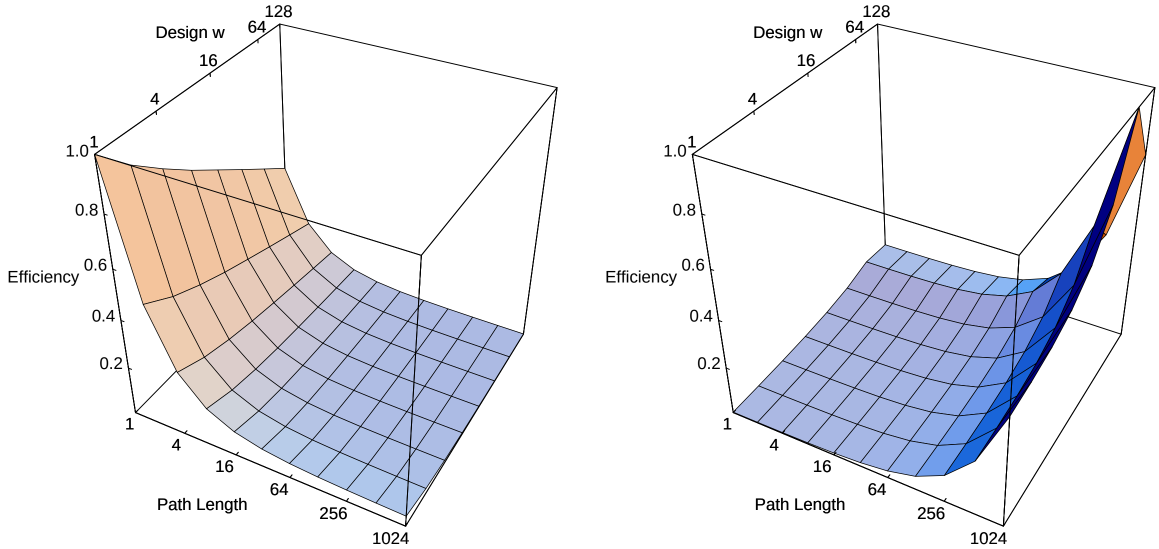
As a consequence these devices are efficient on wide word data and irregular tasks – *i.e.*, those tasks which need to perform a large number of distinct operations on each datapath processing element. On tasks with narrow data items, the active computing resources are underutilized, wasting computing potential. Processors pay overhead for their deep instruction memories. On very regular computational tasks, the on-chip space to hold a large sequence of instructions goes largely unused. Processors exploit wide datapaths to reduce the cost per instruction, but even so, with instruction stores typical supporting thousands of instructions, instruction and retiming memories dominate, leaving their peak general-purpose computational density three orders of magnitude lower than special-purpose devices and one order of magnitude below FPGAs.

Looking at modern scalar, superscalar, and VLIW, processors, then, we might abstract a modern processor as: $k = 2$, $w = 64$, $c = 1024$. Processors use ALU bit-slices in lieu of lookup tables. Each ALU bit-slice takes in two data inputs and a carry bit. As such, they provide less than a full 2- or 3-LUT's capacity per ALU bit, in general, but can provide an add, subtract, or compare operation per bit which would require a pair of 3-LUTs. Processors also include:

- special-purpose capacity (*e.g.* multipliers, floating-point units)
- complicated flow control (*e.g.* branch prediction, bypassing)
- memory controllers to deal with boundary bottlenecks between compute and bulk memory components (*e.g.* cache-controllers, TLBs)

These items tend to make the area of a processor larger than that predicted by the model in Chapter 9. As we have seen in Table 4.1 and Section 4.1, when performing traditional ALU ops, processors generally provide less bit operations per ALU bit than a small LUT. These effects will tend to make the *RP*-space projection of the processor optimistic in terms of area; that is, the real processor will be larger and provide less computational capacity per unit area. On the other hand, the specialized capacity in processors allow them to handle fixed and floating point arithmetic operations more efficiently than would be predicted by the *RP*-space projection.

We have already seen that conventional FPGAs have:



FPGA
$k = 4, c = d = 1, w = 1$

RP-space mapped Processor
$k = 2, c = d = 1024, w = 64$

$n = 2, p = 0.5, N_{ctrl} = 0, N_p = 16384$

Figure 15.1: Comparing efficiency of FPGA and Processor idealizations in *RP*-space

- narrow datapath (*e.g.* almost always one bit)
- on-chip space for only one instruction per compute element – *i.e.* the single instruction which tells the FPGA array cell what function to perform and how to route its inputs and outputs
- minimal die area dedicated to instruction distribution such that it takes hundreds of thousands of compute cycles to change the active set of array instructions

As a consequence these devices are efficient on bit-level data and regular tasks – *i.e.*, those tasks which need to repeatedly perform the same collection of operations on data from cycle to cycle. On tasks with large data elements, these fine-grain devices pay excessive area for interconnect and instruction storage versus a coarser-grain device. On very irregular computational tasks, active computing elements are underutilized – either the array holds all subcomputations required by a task, but only a small subset of the array elements are used at any point in time, or the array holds only the subcomputation needed at each point in time, but must sit idle for long periods of time between computational subtasks while the next subtask's array instructions are being reloaded. The peak computational density for FPGAs is two orders of magnitude lower than special-purpose devices because they pay overhead primarily for the flexible interconnect.

Figure 15.1 shows graphically this idealized comparison projected into *RP*-space in the style used in Section 9.5. As noted before, the FPGA is less than 1% efficient at the cross point of wide task data words and long path lengths. Similarly, the modeled processor is less than 1% efficient processing single bit data items at a path length of one. Certainly, if the processor needs to perform bit operations that do not match its special-purpose support, the inefficiency will be at least this large – and may be greater due to the effects noted above which make the real processor larger than the model.

15.3.3 General-Purpose Computing Space

We have already noted that *RP*-space is large such that we can see two or more orders of magnitude in efficiency loss when the application requirements are mismatched with the architecture structure for fixed instruction architectures. Our comparison in the previous sections underscores that the general-purpose architectural space is even larger making it paramount that one understand the realm of efficiency for each "general-purpose" computing device when selecting a device for an application. They underscore the room for intermediate architectures such as the DPGA, PADDI, or VDSP to cover parts of the space which are not covered well by either conventional extremes of processor or FPGAs. They also underscore the desirability of architectures like MATRIX which allow some run-time reallocation of resources to provide more robust yielded performance across the computational space.

Hybrid Since many tasks have a mix of irregular and regular computing components and a mix of native data sizes, a hybrid architecture which tightly couples arrays of mixed datapath sizes and instruction depths along with flexible control may be able to provided the most robust performance across the entire application. While this thesis focussed on characterizing the implications of each pure architectural point, it should be clear from the development here how a hybrid architecture might be better suited to the mix of datasizes and regularities seen in real applications. In the simplest case, such an architecture might couple an FPGA or DPGA array into a conventional processor, allocating the regular, fine-grained tasks to the array, and the irregular, coarse-grained tasks to the conventional processor. Such coupled architectures are now being studied by several groups (*e.g.* [DeH94] [Raz94] [WC96]).

15.4 Trends and Implications for Conventional Architectures

In summary, we see that conventional, general-purpose device architectures, both microprocessors and FPGAs, live far apart in a rich architectural space. As feature sizes shrink and the available computing die real-estate grows, microprocessors have traditionally gone to wider datapaths and deeper instruction and data caches, while FPGAs have maintained single-bit granularity and a single instruction per array element. This trend has widened the space between the two architectural extremes, and accentuated the realm where each is efficient. A more effective use of the silicon area now becoming available for the construction of general-purpose computing components may lie in the space between these extremes. In this space, we see the emergence of intermediate architectures, architectures with flexible resource allocation, and architectures which mix components from multiple points in the space. Both processors and FPGAs stand to learn from each other's strengths. In processor design, we will learn that not all instructions need to change on every cycle, allowing us to increase the computational work done per cycle without correspondingly increasing on-chip instruction memory area or instruction distribution bandwidth. In reconfigurable device design, we will learn that a single instruction per datapath is limiting and that a few additional instructions are inexpensive, allowing the devices to cope with a wider range of computational tasks efficiently.

15.4.1 Microprocessors

Over the past two decade, microprocessors have steadily increased their word size and their cache size. While these trends allow larger tasks to fit in on-chip caches and allow processors to handle larger word operations in a single cycle, the trends also make processors less and less efficient in their use of die area. While some large word operations are required, a larger and larger fraction of the operations executed by modern processors use only a small portion of the wide datapath. The computationally critical portions of programs occupy only small portions of the instruction and data cache.

We can continue to improve aggregate processor performance by using more silicon in this manner, but the performance per unit area will steadily decrease. To the extent that silicon area is inexpensive, task recompilation is hard or unacceptable, and various forms of parallelism are difficult to achieve, the current trends have their value.

However, to the extent we wish to engineer better silicon system which do more with less resources, these trends are now yielding diminishing returns. We can manage more programmable compute elements than a single, central word-wide, ALU on modern IC dies. Reconfiguration allows us to do this without paying a prohibitive costs for increased instruction distribution as we go to more, independently controlled computing units.

15.4.2 Multiprocessors

The conventional view of multiprocessing is that we replicate the entire microprocessor and place these replicas on the same board or die. At best, this allows aggregate performance to improve with additional area dedicated to additional processors. However, it entails a large amount of unnecessary cost, replicating entire processors when many portions of the processor may not need to be replicated. Further, coupling between processors is poor, at best, entailing 10's to 100's of cycles of latency to move data from one processing element to another and significant overhead to coordinate the activities of multiple processing units.

Most of the task which have generally been “good” multiprocessor applications are very regular computing tasks for which configured, systolic dataflow can provide more area efficient implementations. For the sake of intuition, consider an image processing task where we need to perform 100 operations on each pixel. We can divide this task among n conventional processors, where each processor must have memory to hold the 100 operations and must pay overhead cycles for communication, as necessary, among the n processors. Alternately, we can configure a hardware pipeline to process the data. If we allocate 100 compute elements, each compute element in the configured pipeline needs to only execute its one operation. Direct connections between computing elements transport data avoiding additional overhead cycles. To get the same throughput as the 100 element systolic design, the multiprocessor implementation would need, at least, 100 processors. In terms of instruction memory alone, the multiprocessor implementation requires memory area to hold 9900 more instructions than the systolic implementation, making it significantly larger just to support the same throughput.

The traditional strength of microprocessors has been their ability to pack large computations into small area by reusing central computing resources. This tight packing of functionality comes at the cost of a decrease in computational density as we saw in Chapter 4 and Section 15.3.2. When we are willing to pay area to increase throughput, the traditional microprocessor architecture is not efficient since it brings with it the baggage of a large investment in instruction distribution, instruction memory, and control which are unnecessary for highly regular tasks. Further, the i/o structure of conventional processors is designed around heavy sequentialization, creating an interconnect bottleneck which makes high throughput usage impractical.