Issues and Approaches to Coarse-Grain Reconfigurable Architecture Development

Ken Eguro and Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, WA 98195 USA
{eguro, hauck}@ee.washington.edu

Abstract

Although domain-specialized FPGAs can offer significant area, speed and power improvements over conventional reconfigurable devices, there are several unique and unexplored design problems that complicate their development. One source of these problems is that designers often opt to replace more universal, fine-grain logic elements with a specialized set of coarse-grain functional units to improve computation speed and reduce routing complexity. One issue this introduces is that it is not obvious how to simultaneously consider all applications in a domain and determine the most appropriate overall number and ratio of the different functional units. In this paper, we illustrate how this problem manifests itself during the development of an encryption-specialized FPGA architecture. We present three algorithms that solve this problem by balancing the hardware needs of the domain while considering performance and area requirements. We believe these concerns need to be addressed by future CAD tools in order to develop more sophisticated application-specialized reconfigurable devices.

1 Introduction

While flexibility is an important feature of reconfigurable devices, conventional FPGAs are simply too generic to provide high performance in many situations. General-purpose reconfigurable devices, while well suited to small or irregular functions, typically suffer a stiff penalty when implementing wide and complex arithmetic operations. These types of functions need to be built from too many small logical resources and end up being spread across too general a routing structure to be efficient.

However, if the range of applications that a device is intended for is known beforehand, a designer can specialize the logic, memory and routing resources to enhance the performance of the device while still providing adequate flexibility to accommodate all anticipated uses. Common and complex operations can be implemented much more efficiently on specialized coarse-grain functional units while routing and memory resources can be tuned to better reflect the requirements. An example of such a specialized reconfigurable device is the RaPiD architecture [10], which was originally designed to implement applications in the DSP domain. This architecture consists of dedicated 16-bit multipliers, ALUs and RAM modules that are connected through a programmable and pipelined word-wise data bus. While this architecture clearly lacks much of the flexibility of a more conventional FPGA, [8] has shown that it successfully improves performance while minimally affecting usability since it provides significant speed, area and power advantages across a wide range of DSP applications.

Although domain-specialized FPGAs such as RaPiD can offer great advantages over general-purpose reconfigurable devices, they also present some new and unique challenges. While design choices that affect the performance and flexibility of classical FPGAs are relatively clearly defined and well understood, the effects that fundamental architecture decisions have on specialized reconfigurable devices are largely unknown and difficult to quantify.

2 Implications of Domain-Specialized Devices

The development of a conventional reconfigurable array that can accommodate a group of applications is relatively straightforward. Based upon the results of past research, consistently good results can be obtained using an array consisting of 4-LUT based CLBs [2] surrounded by a predetermined mix of short and long routing tracks [4]. Using existing CAD tools, the only real parameters that would need to be determined are the overall size of the array and the number of routing tracks in the system.

However, when developing domain-specialized reconfigurable devices, the sea of fine-grained logical resources found in conventional FPGAs is typically replaced with a clearly defined set of coarse-grained function units. While this can greatly improve the performance of the system, developers need to closely consider the specific ways in which each algorithm uses

the provided resources since the logical elements are no longer universally flexible. Merely given a domain of applications it is not obvious what the most appropriate set of functional units would be, much less what routing architecture would be appropriate, what implications this might have on necessary CAD tools, or how any of these factors might affect each other.

The first challenge we mentioned, the selection of functional units, can be subdivided into three steps. First, the applications in a domain must be analyzed to determine what operations they require. Crucial parts such as wide multipliers or fast adders should be identified. Next, this preliminary set of functional units can be distilled to a smaller set by capitalizing on potential overlap or partial reuse of other types of units. Different sizes of memories, for example, may be combined through the use of multi-mode addressing schemes. Lastly, based upon design constraints, the exact number of each type of unit in the array should be determined. For example, if the applications are memory-intensive rather than computation-intensive, the relative number of memory units versus ALUs should reflect this. This paper will primarily focus on the problem of functional unit allocation – determining the most appropriate quantity and ratio of functional units across the domain. We will use the 15 candidate algorithms of the Advanced Encryption Standard (AES) competition [12] to illustrate the issues that make component allocation difficult.

While operator identification and optimization are both complex problems unique to coarse-grain architectures, we will not address these issues here since the algorithms in the target domain often provide an obvious starting point. Normally, algorithms are described using strongly typed functions such as multiplications or Boolean operations. The logical optimization and technology mapping for such strongly typed operators is relatively simple to perform by hand. Although this may overlook subtle optimizations, such as the exact tradeoffs between having a separate multiplier and adder versus creating a dedicated multiply-accumulate unit, manual analysis does provide an acceptable working set and offers a starting point for our discussion concerning functional unit allocation.

3 Difficulties of Functional Unit Allocation

Although it is relatively straightforward to establish the absolute minimum area required to support a domain of applications, determining the best way to allocate additional resources is more difficult. While developers of conventional reconfigurable devices can determine the best overall size of the array for any set of design constraints by building performance versus area curves for each algorithm in the domain, these curves do not offer enough information to assist designers of coarse-grained reconfigurable devices. When developing domain-specialized FPGAs, the individual functional unit needs of every algorithm in the domain contends for control over what the overall architecture will look like.

To illustrate the complicated relationships between functional unit demands across a domain, we analyzed the 15 candidate algorithms of the AES competition over a range of performance levels. We identified the resource requirements to implement each of the algorithms at natural unrolling points, from relatively small, time-multiplexed elements to completely unrolled implementations. From this analysis, we demonstrate four main factors that obscure the relationship between hardware resources and performance.

First, although the algorithms in our domain share many common operations, the ratio of the different functional units varies considerably between algorithms. Without any prioritization, it is unclear how to distribute resources. For example, if we consider the fully rolled implementations for six encryption algorithms, as in Figure 1, we can see the wide variation in RAM, crossbar, and runtime requirements among the different algorithms.

Algorithm (Baseline)	**RAM Blocks**	**XBars**	**Runtime**
CAST-256 (1x)	16	0	48
DEAL (1x)	1	7	96
HPC (1x)	24	52	8
Loki97 (1x)	40	7	128
Serpent (1x)	8	32	32
Twofish (1x)	8	0	16
Average	**16.2**	**16.3**	**54.7**
Std. Dev.	**14.1**	**21.1**	**47.6**

Figure 1 – Ratio Complications
This table compares the RAM, crossbar and runtime requirements for the baseline implementations of six encryption algorithms. Notice that in all three categories the deviation in requirements is comparable to the average value.

To complicate matters, if we attempt to equalize any one requirement over the entire set, the variation among the other requirements becomes more extreme. This can be seen in Figure 2. In this case, if we consider the RAM resources that an architecture should provide, we notice that *Loki97* requires at least 40 RAM modules. If we attempt to develop an architecture that caters to this constraint and unroll the other algorithms to take advantage of the available memory, we see that the deviation in the number of crossbars and runtime increases sharply.

Algorithm (Unrolling Factor)	RAM Blocks	XBars	Runtime
CAST-256 (2x)	32	0	24
DEAL (32x)	32	104	3
HPC (1x)	24	52	8
Loki97 (1x)	40	7	128
Serpent (8x)	32	32	4
Twofish (4x)	32	0	4
Average	**32**	**32.5**	**28.5**
Std. Dev.	**5.6**	**40.6**	**49.4**

Figure 2 – Equalization Difficulties
This table displays the compounded problems that occur when attempting to normalize the RAM requirements across algorithms. The other algorithms are unrolled to make use of the memory ceiling set by *Loki97*. Notice that the total deviation in crossbars roughly doubles as compared to the baseline comparison and that the deviation in runtime becomes almost twice the new average value.

The second factor that complicates the correlation between hardware availability and performance is that the algorithms have vastly different complexities. This means that the hardware requirement for each algorithm to support a given throughput differs considerably. In Figure 3 we see an example of five different encryption algorithms that are implemented to have similar throughput, but then have a wide variation in hardware requirements. It is difficult to fairly quantify the performance-versus-hardware tradeoff of any domain that has a wide complexity gap.

Algorithm (Unrolling Factor)	RAM Blocks	XBars	Runtime
CAST-256 (2x)	32	0	24
DEAL (4x)	4	16	24
Loki97 (8x)	320	7	16
Magenta (4x)	64	0	18
Twofish (1x)	8	0	16
Average	**85.6**	**4.6**	**22.8**
Std. Dev.	**133.2**	**7.1**	**6.3**

Figure 3 – Complexity Disparity
An illustration of the imbalance that occurs when attempting to equalize throughput across algorithms. In this case we choose *Twofish* as a baseline and unrolled the rest of the algorithms to best match its throughput. Notice that the deviation in RAM and crossbar requirements is well above the average value.

The third problem of allocating hardware resources is that the requirements of the algorithms do not necessarily scale linearly or monotonically when loops are unrolled. See Figure 4 for an example of this non-uniform behavior. This phenomenon makes it difficult to foresee the effect of decreasing the population of one type of functional unit and increasing another.

Algorithm (Unrolling Factor)	RAM Blocks	Muxes	Runtime
FROG (1x)	8	23	512
FROG (4x)	8	72	128
FROG (16x)	8	256	32
FROG (64x)	16	120	8
FROG (256x)	64	30	2

Figure 4 – Scaling Behavior
An example of the unpredictable nature of hardware demands when unrolling algorithms.

The last problem of estimating performance from available resources is that if a particular implementation requires more functional units of a certain type than is available, the needed functionality can often be emulated with combinations of the other, under-utilized units. For example, a regular bit permutation could be accomplished with a mixture of shifting and masking. Although this flexibility may improve resource utilization, it also dramatically increases the number of designs to be evaluated.

4 Function Unit Allocation

To produce an efficient platform for a diverse group of applications, an effective solution to the functional unit allocation problem must have the flexibility needed to simultaneously address the multi-dimensional hardware requirements of the entire domain while maximizing usability and maintaining hard or soft area and performance constraints. In the following sections we propose three solutions to this problem. The first algorithm addresses hard performance constraints. The second and third algorithms attempt to maximize the overall performance given softer constraints.

4.1 Performance-Constrained Algorithm

The first algorithm we developed uses a hard minimum throughput constraint to guide the functional unit allocation. As described earlier, we began the exploration of this domain by establishing the hardware requirements of all of the algorithms for a variety of performance levels. We use this table of results in all of our functional unit allocation techniques to determine the minimum hardware that each algorithm needs in order to support a given throughput.

Our first algorithm begins by determining the hardware requirements to run each algorithm at a specified minimum throughput. We then examine these requirements to establish the maximum necessary number of each type of functional unit. To calculate the overall performance for this superset of resources, we reexamine

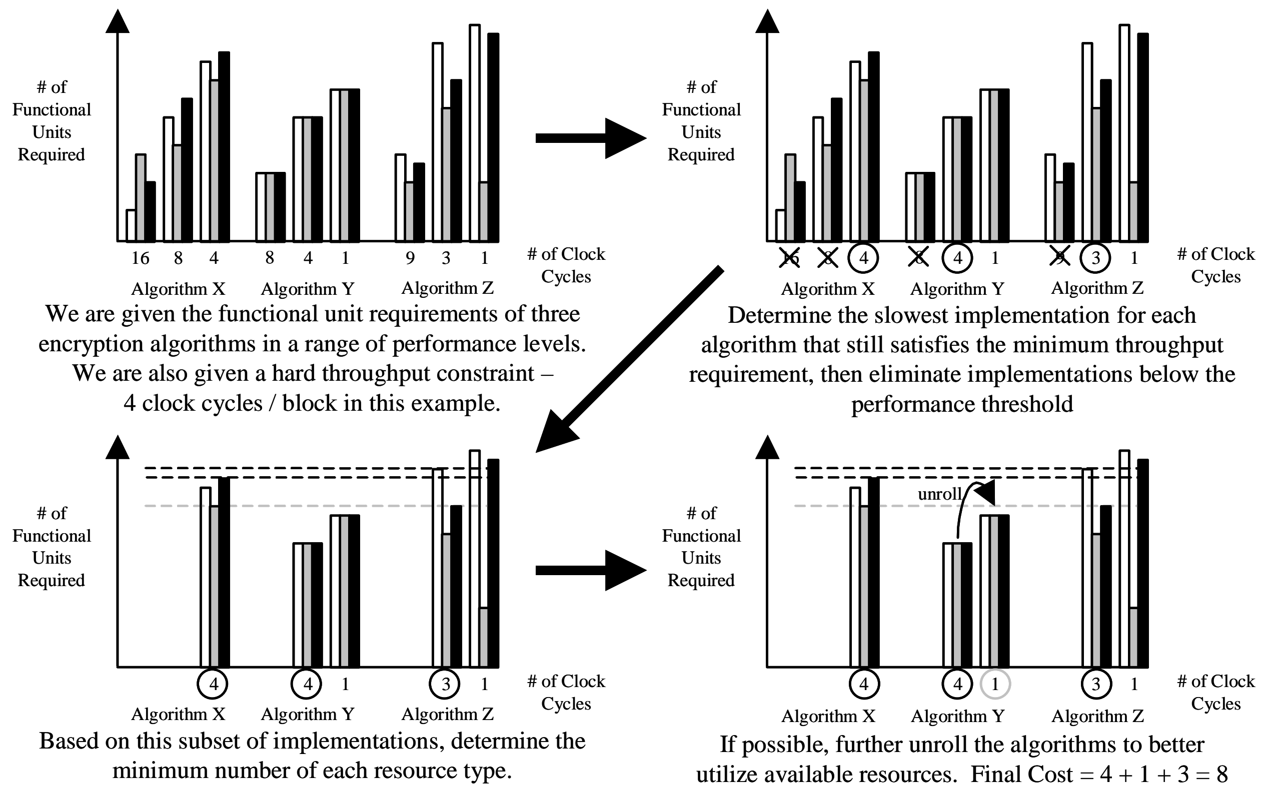

Figure 5 – Performance-Constrained Functional Unit Selection
Illustration of performance-constrained selection algorithm.

each algorithm to determine if there are sufficient resources to allow for greater throughput, then apply the cost function described by this equation:

$$Cost = \sum_{i=0}^{N-1} CC_i$$

In this equation, N is the total number of algorithms in the domain and CC_i is the number of clock cycles required to encrypt a single block of text in the highest throughput configuration of algorithm i that will fit on the architecture. See Figure 5 for an illustrated example of the performance-constrained functional unit selection process.

Note that this is a greedy algorithm and, due to the non-linear and non-monotonic behavior of hardware requirements, does not necessarily find the minimum area or maximum performance for the system. Because the starting point is chosen solely on the basis of throughput without considering hardware requirements, it is possible that higher throughput implementations of a given algorithm may have lower resource demands for particular functional unit types. If that algorithm becomes the limiting factor when determining the number of any resource type, it will likely affect the overall area and performance results.

4.2 Area-Constrained Algorithm

The next two algorithms we developed use simulated annealing to provide more sophisticated solutions that are able to capitalize on softer constraints to improve average throughput. The second algorithm begins by randomly adding functional units to the architecture until limited by a given area constraint. The quality of this configuration is evaluated by determining the highest performance implementation for each algorithm, given the existing resources, then applying the cost function described by this equation:

$$Cost = \sum_{i=0}^{N-1} \begin{cases} CC_i, \text{if algorithm } i \text{ fits on the architecture} \\ PC * A, \text{otherwise} \end{cases}$$

In this equation, N is the total number of algorithms in the domain and CC_i is the number of clock cycles required to encrypt a single block of text in the highest throughput configuration of algorithm i that will fit on the array. However, if an algorithm cannot be implemented on the available hardware, we impose an exclusion penalty proportional to A, the additional area necessary to map the slowest implementation of the algorithm to the

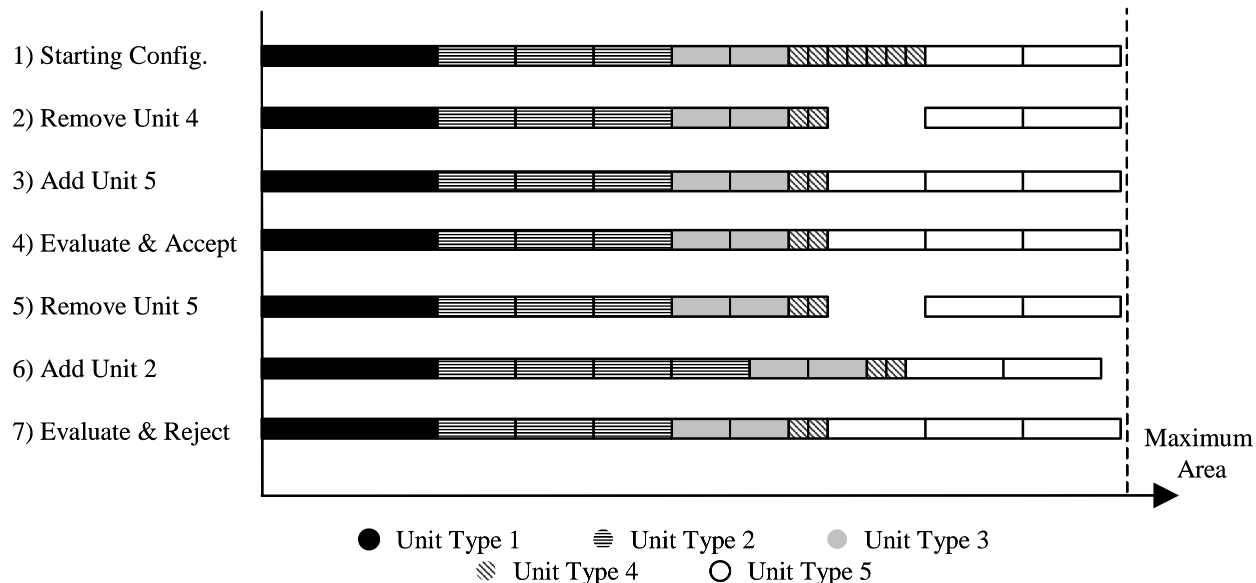

Figure 6 – Area-Constrained Function Unit Selection
Illustration of our area-constrained selection algorithm.

array. In our evaluations, we used a large constant penalty scaling factor (*PC*) since we wanted our system include all of the candidate algorithms. However, this factor is completely application-dependent and must be tuned depending on the size of the functional units, how many algorithms are in the domain, what the average runtime is, and how critical it is that the system is able to implement the entire domain. While this penalty system does not necessarily guide the simulated annealing to the best solution, since a higher throughput implementation may be closer to the existing configuration, it does provide some direction to the tool to help prevent the potentially unwanted exclusion of some of the algorithms in the domain.

After calculating the quality of the configuration we perturb the system by randomly picking two types of components, removing enough of the first type to replace it with at least one of the second, then adding enough of the second type to fill up the available area. Finally, the quality of the new configuration is evaluated in the same manner as before. If the new configuration provides the same or better throughput, it is accepted. If it does not provide better performance, based on the current temperature and relative performance degradation, it may or may not be accepted. This process is based on the simple acceptance function and adaptive cooling schedule described in [5]. See Figure 6 for an illustration of this procedure.

Note that, as described earlier, some operations may be emulated by combinations of other functional units. For simplicity, in this example we do not directly deal with this possibility, but there is no inherent limitation in either of the area-constrained solutions that would prevent this from being addressed with a larger hardware/throughput matrix.

4.3 Improved Area-Constrained Algorithm

Our last functional unit selection algorithm attempts to balance performance and area constraints. First, we eliminate implementations from the hardware/throughput matrix that do not provide enough throughput to meet a specified minimum performance requirement. Then, we randomly select one of the remaining implementations of each algorithm for our current arrangement. We determine the minimum hardware and area requirements necessary to fit all of the algorithms at their current settings, and then establish if any algorithms can be expanded to a higher performance level given the calculated hardware resources. The quality of this arrangement is determined by the number of clock cycles required to run all of the algorithms at their current settings and a penalty based on any excess area needed by the system. The cost function is described by this equation:

$$Cost = \sum_{i=0}^{N-1} CC_i + Area\ Penalty$$

In this equation, N is the total number of algorithms in the domain and CC_i is the number of clock cycles required to encrypt a single block of text in the highest throughput configuration of algorithm i that will fit on the architecture. If the area required for the current configuration is larger than the specified maximum

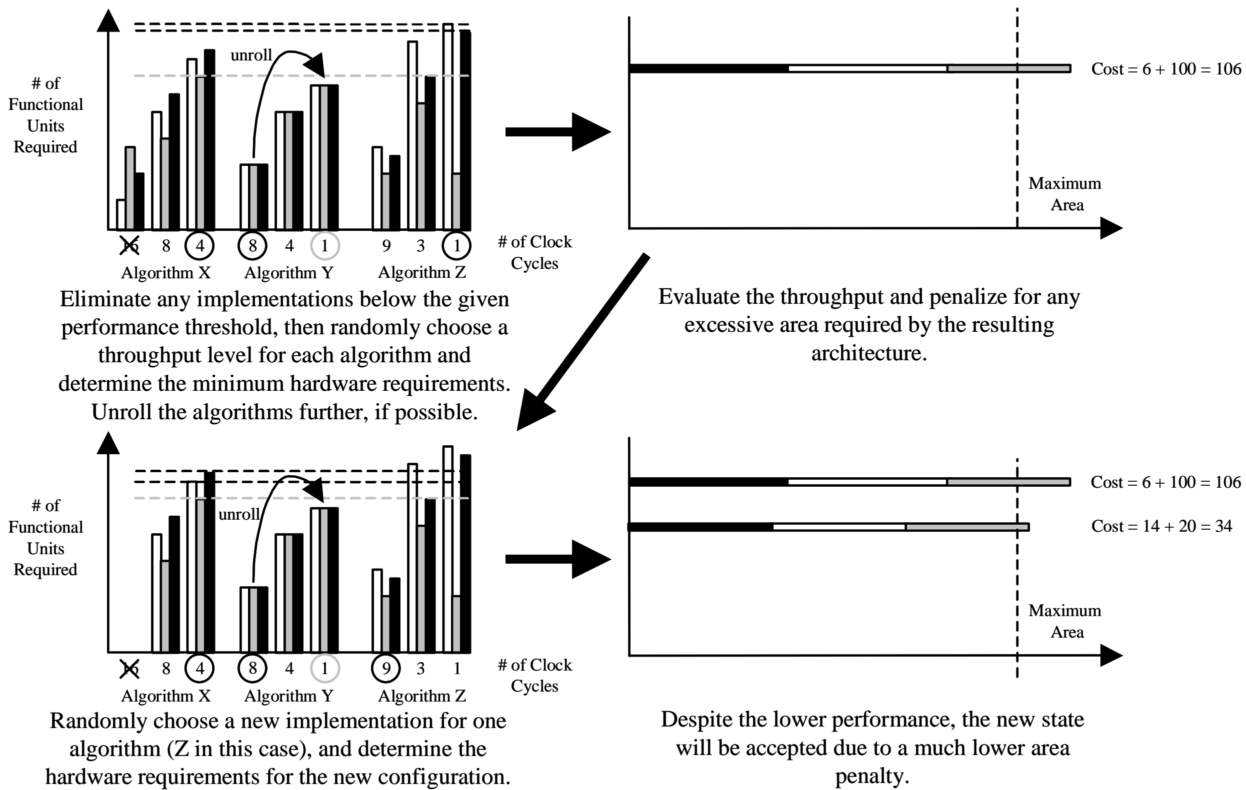

Figure 7 – Improved Area-Constrained Functional Unit Selection

Illustration of our improved area-constrained selection algorithm. In this example we assume the throughput threshold is set at 10 cycles/block.

allowable area, we also add an area penalty that is described by this equation:

$$Area\ Penalty = PC * (CA / MA)$$

In this case, *PC* is a constant penalty scaling factor, *CA* is the calculated area requirement of the current configuration and *MA* is the specified maximum allowable area. Again, since we wanted a hard area constraint for our evaluation, we set *PC* to a large value. However, similar to the previous functional unit selection algorithm, this term is application-specific and must be tuned depending on how hard or soft an area constraint is desired. After calculating the quality of the configuration, we then perturb the system by arbitrarily choosing one algorithm and randomly changing the setting to a different performance level. Finally, the quality is re-evaluated and compared to the original arrangement in the same simulated-annealing manner as described in Section 4.2. See Figure 7 for an illustration of this process.

5 Function Unit Allocation Results

The evaluation of the functional unit allocation techniques began by using the performance-constrained method as a baseline for comparison. We first identified all of the distinct throughput levels between all of the algorithms in the domain. Then, each of these distinct throughput constraints was fed into the performance-constrained functional unit selection algorithm. The area requirements for each were recorded and then used as inputs to the two area-constrained techniques.

The three techniques we developed produce very different results when applied to our example domain. As expected, the hard throughput constraint of the performance-driven approach has limitations. In Figure 8 and Figure 9 we plot the results of all three functional unit selection algorithms over ten area scenarios. Figure 8 shows the maximum number of clock cycles per block required by any algorithm in the domain as a function of the area of the system. Since the number of clock cycles needed to encrypt each block of data is inversely proportional to the throughput, we can see from this graph that, for the majority of the architectures we examined, the performance-constrained algorithm indeed produces the best minimum performance among the three allocation methods. Also, as expected, the limitations of the performance-driven algorithm regarding non-linear and non-monotonic hardware requirements allow the improved area-constrained technique to occasionally obtain somewhat better minimum performance.

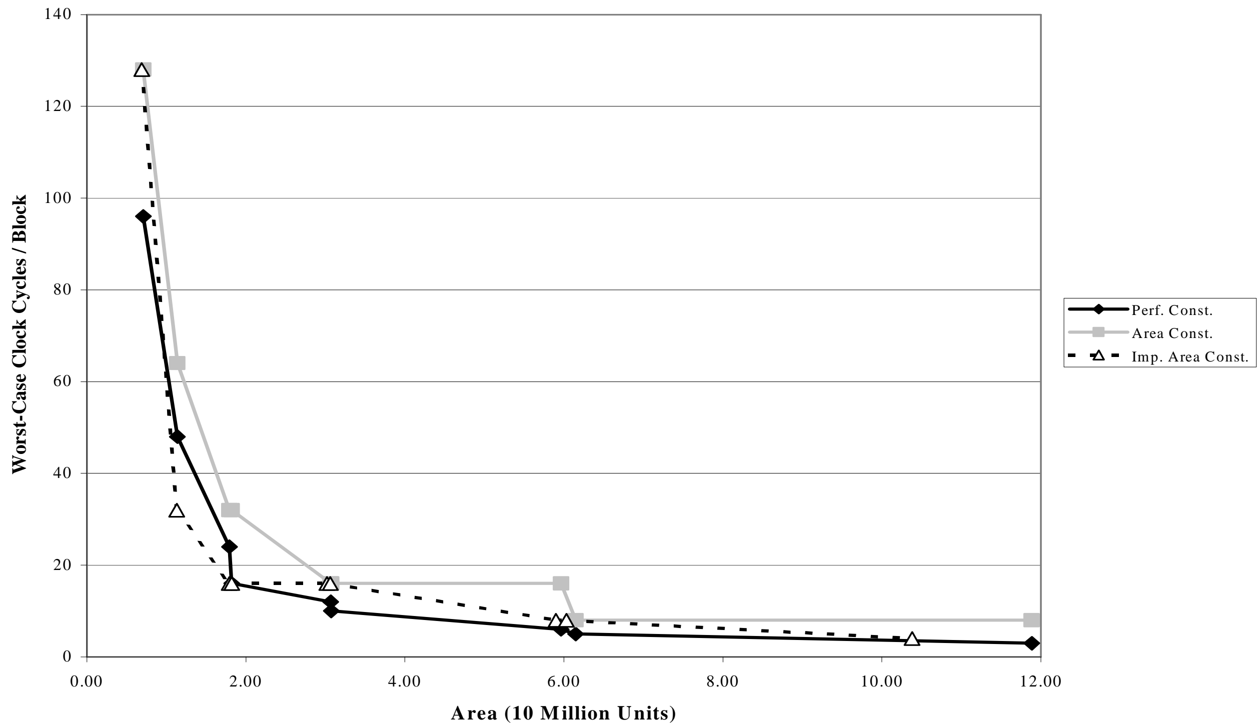

Figure 8 – Minimum Throughput Results of Functional Unit Selection
Graph of maximum number of clock cycles required by any application in the domain as a function of area. Notice that while the improved area-constrained allocation technique occasionally produces better results, the performance-constrained method obtains the best worst-case performance over the majority of the design space.

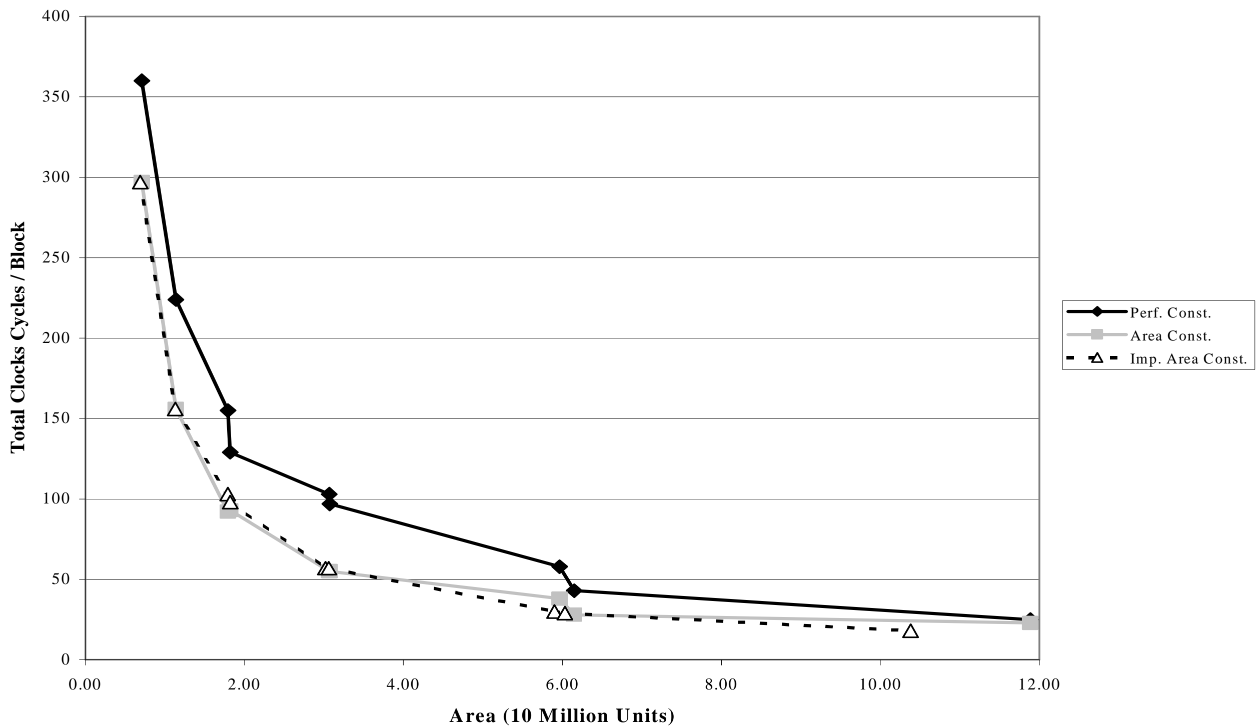

Figure 9 – Performance Results of Functional Unit Selection Across the Domain
Graph of the total number of clock cycles required to run all of the applications in the domain as a function of area. Notice that the overall performance of the higher throughput systems produced by the performance-constrained algorithm lag considerably behind that of the architectures generated by either of the area-constrained techniques.

In contrast, though, when we plot the total number of clock cycles required by all of the algorithms in the domain as a function of area, as in Figure 9, we see a completely different picture for the performance-constrained selection method. The results in this graph directly reflect the average performance of the system for a given configuration. Figure 9 shows that the average performance of the system across the domain is reduced by as much as almost 50% when using the performance-constrained selection method as compared to using either of the area-driven techniques. The poor average throughput is particularly apparent in the larger architectures. This means that if the design constraints allow for some flexibility in terms of the minimum acceptable performance, better average throughput may be obtained by using either of the area driven approaches.

When comparing the two area-constrained techniques, Figure 9 shows that the average performance results of the improved area-constrained technique are marginally better than those from the original area-driven method Furthermore, when we consider the area requirements for the generated architectures, the improved area-constrained method generally produces architectures with equal or smaller area requirements. In addition, Figure 8 shows that the improved area-constrained method consistently produces architectures with an equal or lower maximum number of clock cycles for the worst-case encryption algorithm compared to the basic area constrained technique.

All of these observations can likely be attributed to the same source. Because the original area-constrained functional unit selection algorithm is based upon randomly adding and subtracting different types of components to the system, it is likely that none of the applications in the domain fully utilize any of the functional unit types in the resultant architecture. Conversely, since the improved area-constrained technique is based upon choosing sets of particular implementations, it is guaranteed that at least one application will fully utilize each of the functional unit types. It is likely that this fundamental difference creates more noise in the original area-constrained selection technique and thus makes it more difficult for the algorithm to converge. In addition, even if the original area-constrained technique were to converge on a similar mixture of components as the improved method, it is very possible that there may still be some functional unit types that are not fully utilized by any implementation. Of course, this will result in a larger architecture than is necessary.

To better illustrate the quality of the proposed algorithms we could compare them to the optimal architectures found by brute-force search. Unfortunately, the original set of 15 algorithms from the AES competition, with up to 10 implementations each, is too complex to solve via such methods. However, the AES competition also produced a smaller set of 5 finalists. We have run this set though each of our algorithms, and compare them to the optimal results found by a brute-force search. Figure 10 and Figure 11 show the results of the testing repeated for this smaller domain. In this case, the allocation problem is much simpler and both the performance-constrained and improved area-constrained techniques find the optimal architecture for all area scenarios. In addition to showing that our allocation methods can achieve optimal results, this test also confirms the suspicion that the original area-constrained method does not perform as well as the improved technique. While this domain does not present all of the same challenges as the full set of AES candidates, it is likely that our earlier results at least approach the optimal architecture.

6 Future Work

In addition to the operator identification and optimization problems described earlier, domain-specialized reconfigurable devices also raise several completely new concerns. One interesting question surrounds the issue of adaptability. While a carefully designed architecture will likely perform well on the limited set applications that directly affected the design, what methodology is needed to sufficiently encapsulate the needs of the applications as a domain? Extending this to the example of encryption, would algorithm updates or completely different ciphers also perform well on this system?

Another issue, related to adaptability, is specialization. It is expected that the larger the group of applications used to develop a domain-specialized reconfigurable device, the better it will handle unanticipated netlists. However, the size of the target domain is also connected to the specificity of the array and its overall performance. It is vital that developers understand how these factors affect each other.

7 Conclusions

In this paper we have shown that the development of a coarse-grained reconfigurable architecture raises several unique and un-addressed design problems. We presented three techniques to allocate functional units that attempt to balance performance and area constraints on domains that have vastly different hardware requirements. While taking special consideration for stable, high-performance implementations and the possibility for future flexibility, designers can use these functional unit allocation techniques to develop future coarse-grained reconfigurable devices.

The first algorithm produces architectures under a guaranteed hard performance requirement. The second

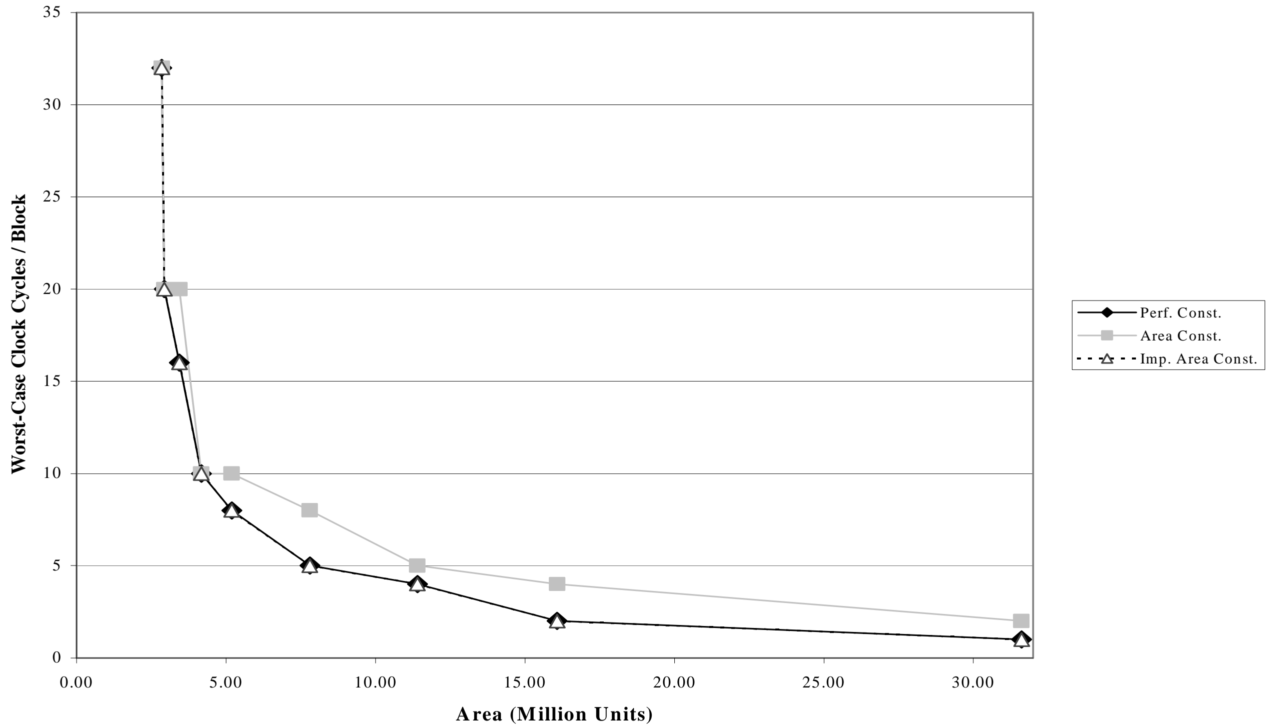

Figure 10 – Minimum Throughput Results of Functional Unit Selection on a Limited Domain
Graph of maximum number of clock cycles required by any of the five finalist algorithms as a function of area. Both the performance-constrained and improved area-constrained techniques achieve the optimal results.

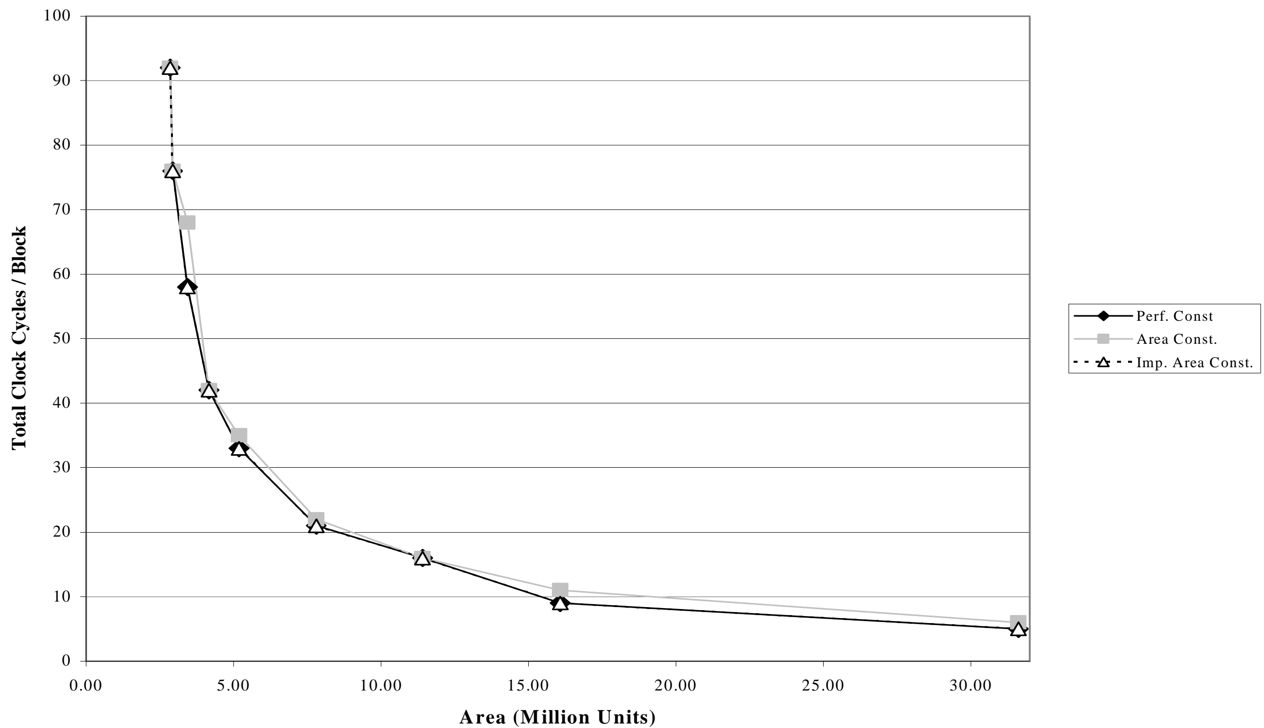

Figure 11 – Performance Results of Functional Unit Selection Across a Limited Domain
Graph of the total number of clock cycles required to run all five of the AES finalist ciphers as a function of area. Again, both the performance-constrained and improved area-constrained techniques achieve the optimal results.

algorithm allows designers to trade versatility for better average throughput. The third algorithm produces efficient architectures that can take advantage of softer area constraints. While the performance-constrained algorithm can be used when designers are only concerned with the minimum performance of a system, the area-constrained algorithms were shown to produce better average performance given similar area. Although the original area-constrained technique allows designers to potentially improve overall performance by excluding very demanding applications, the improved area-constrained technique consistently produced better results when considering the entire domain. It is likely that the improved area-constrained method would be the most appropriate choice unless the minimum performance of the system needs to be absolutely guaranteed.

Although we encountered the difficulties of functional unit selection while exploring an encryption-specific domain, we believe that the causes of the problem are not exclusive to encryption and can be expected to be common in many complex groups of applications. The functional unit selection problem will become more difficult as reconfigurable devices are expected to offer better and better performance over large domain spaces. Increased specialization of function units and growing domain size combined with the need for resource utilization optimization techniques such as functional unit emulation will soon complicate architecture exploration beyond that which can be analyzed by hand. In the future, designers will need CAD tools that are aware of these issues in order to create devices that retain the flexibility required for customization over a domain of applications while maintaining good throughput and area characteristics.

8 References

[1] Adams, C. and J. Gilchrist. “The CAST-256 Encryption Algorithm.” *First AES Candidate Conference*, Aug. 20-22, 1998.

[2] Ahmed, E. and J. Rose. “The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density.” *International Symposium on Field Programmable Gate Arrays, 2000: 3-12.*

[3] Anderson, Ross, Eli Biham and Lars Knudsen. “Serpent: A Proposal for the Advanced Encryption Standard.” *First AES Candidate Conference*, Aug. 20-22, 1998.

[4] Betz, Vaughn and Jonathan Rose. “FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density.” *International Symposium on Field Programmable Gate Arrays, 1998: 59-68.*

[5] Betz, Vaughn and Jonathon Rose. “VPR: A New Packing, Placement and Routing Tool for FPGA Research.” *International Workshop on Field Programmable Logic and Applications*, 1997: 213-22.

[6] Brown, Lawrie and Josef Pieprzyk. “Introducing the New LOKI97 Block Cipher.” *First AES Candidate Conference*, Aug. 20-22, 1998.

[7] Burwick, Carolynn, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shait Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford and Nevenko Zunic. “Mars – A Candidate Cipher for AES.” *First AES Candidate Conference*, Aug. 20-22, 1998.

[8] Cronquist, Darren C., Paul Franklin, Chris Fisher, Miguel Figueroa and Carl Ebeling. "Architecture Design of Reconfigurable Pipelined Datapaths." *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999:23-40.

[9] Daemen, Joan and Vincent Rijmen. “AES Proposal: Rijndael.” *First AES Candidate Conference*, Aug. 20-22, 1998.

[10] Ebeling, Carl, Darren C. Cronquist, and Paul Franklin. "RaPiD - Reconfigurable Pipelined Datapath." *The 6th International Workshop on Field-Programmable Logic and Applications*, 1996: 126 - 35.

[11] Eguro, Ken. “RaPiD-AES: Developing an Encryption Specific FPGA Architecture.” Master’s Thesis. Dept of Electrical Engineering, University of Washington, Dec. 2002.

[12] National Institute of Standards and Technology. *Advanced Encryption Standard (AES) Development Effort*. Nov. 11, 2002. <http://csrc.nist.gov/encryption/aes/index2.html>.

[13] Rivest, Ron, M. J. B. Robshaw, R. Sidney and Y. L. Yin. “The RC6 Block Cipher.” *First AES Candidate Conference*, Aug. 20-22, 1998.

[14] Schneier, B., J. Kelsey, D. Whiting, D. Wagner, C. Hall and N. Ferguson. “Twofish: A 128-bit Block Cipher.” *First AES Candidate Conference*, Aug. 20-22, 1998.