

Resource Allocation for Coarse-Grain FPGA Development

Ken Eguro, *Student Member, IEEE* and Scott Hauck, *Member, IEEE*

Abstract—The development of domain-specialized reconfigurable devices, and even the nature of domains themselves, has been largely unexplored. In part this is because the same architectural improvements that allow domain-specialized FPGAs to outperform conventional reconfigurable devices also significantly change the design problem. The migration from completely generic lookup tables and highly connected routing fabrics to specialized coarse-grain functional units and very structured communication resources presents designers with the problem of how to best customize the system based upon anticipated usage. In this paper we establish the nature of many of these problems and illustrate the difficulties of determining the appropriate number and ratio of functional units considering the demands of a diverse domain. We present three algorithms that attempt to balance the hardware needs of the domain while considering design constraints such as the performance and area of the system.

Index Terms—Design automation, field programmable gate arrays, reconfigurable architectures, system on a chip.

I. INTRODUCTION

Although almost limitless flexibility is the hallmark of FPGAs, designers are often unwilling to incur the performance penalties normally associated with reconfigurable devices. Conventional FPGAs, based upon small lookup-tables surrounded by an incredible amount of highly flexible communication resources, are simply too generic to provide high performance in many situations. Although it does not become critical if the reconfigurable device is used only used for small control circuitry, wide computation-heavy datapath operations are often one to two orders of magnitude slower than their equivalent ASIC implementations. Complex computations simply need to be built from too many small logical resources and end up being spread across too general a routing structure to truly be efficient. This phenomenon has clearly caught the attention of commercial FPGA manufacturers. In an attempt to address the most common and most problematic functions, companies such as Xilinx and Altera include dedicated block memory

and multipliers in their newest device families.

However, this approach still has fundamental limitations. First, although hard memory and multiplication resources can greatly improve the performance of circuits that can utilize them, this only represents a small number of the overall complex functions that a design might actually need. Difficult operations, such as floating point arithmetic, must still be implemented using massive numbers of look-up tables. Second, FPGA manufacturers must design devices that offer routing resources that are appropriate for the average case. On an application-to-application basis, this might render much of the silicon unused, due to either an unnecessary surplus of routing or because too sparse a routing structure created unreachable logic.

Domain-specialized FPGAs attempt to address these problems by trading application flexibility for performance. Of course, reconfigurable components are included into a system because their flexibility lends an important aspect to the overall design. Still though, it is likely that the device will only be applied to a limited or related set of applications and much of their overall flexibility will never actually be used. If the intended range of applications for a particular reconfigurable device is known or can be characterized at design time, the logic, memory and routing resources can be specialized to enhance the performance of the device while still providing adequate flexibility to accommodate the anticipated pattern of use. We can identify common complex operations and improve the performance of the device by replacing the classical sea-of-gates structure with application-driven coarse-grain functional units and routing resources.

Although domain-specialized FPGAs can offer improvements over more 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 domain-specialized reconfigurable devices are largely unknown and difficult to quantify.

II. DOMAIN-SPECIALIZED FPGAS

One example of a domain-specialized reconfigurable device is the DSP-oriented RaPiD architecture [14], shown in Fig. 1.

Manuscript received October 29, 2003. This work was supported in part by grants from NSF and NASA. Scott Hauck was supported in part by an NSF CAREER Award, and by an Alfred P. Sloan Research Fellowship.

The authors are members of the ACME Lab in the Electrical Engineering Department, University of Washington, Seattle, WA 98195 USA. (email: {eguro, hauck}@ee.washington.edu)

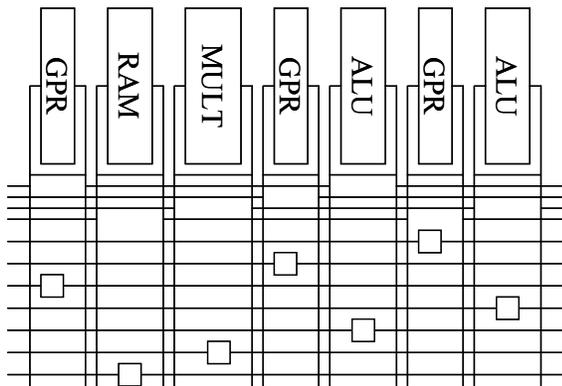


Fig. 1. The RaPiD reconfigurable architecture is built from monolithic multipliers, ALUs and memories. These components are connected through word-width routing segments.

This architecture consists of dedicated 16-bit multipliers, ALUs and RAM modules that are connected through a linear, programmable, and pipelined word-wise data bus. While this architecture clearly lacks much of the flexibility of a more conventional bit-wise LUT-based FPGA, [12] 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 the RaPiD group and others have shown that domain-specialized FPGAs can offer dramatic improvements for a handful of applications, developing architectures for those outside the realms covered by existing architectures is a daunting task.

The development of a conventional LUT-based reconfigurable array to accommodate a specific group of applications is relatively straightforward. Based upon the results of past research, consistently good results can be obtained by providing an array of 4-LUT based logic blocks [2] surrounded by a predetermined mix of short and long routing tracks [5]. 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 logic 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 application 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 structure would be appropriate, what implications this might have on necessary CAD tools, or how any of these factors might affect each other. Clearly, to produce the most effective domain-specialized reconfigurable device, we need to have the iterative exploration and verification abilities offered by automated design tools.

The Totem project [9, 10, 11] is a first step towards such domain-aware tools and offers insight into the issues

surrounding the generation of specialized reconfigurable architectures. However, this work built systems based upon the wealth of existing DSP-centric RaPiD netlists, focusing upon logical unit placement and routing architecture generation to produce better RaPiD-like structures. To produce specialized reconfigurable logic for applications in an unfamiliar domain, an analysis step is needed to identify the coarse-grain functional units to be offered by the architecture. Here, we cannot leverage the pain-staking research performed by the RaPiD group in selecting and developing the logical, arithmetic, and memory components appropriate for the domain. Furthermore, since we are developing the architecture wholly from the ground up, we should consider how different area versus performance implementations of our intended applications interact and fit into a proposed architecture.

III. AUTOMATED ARCHITECTURE EXPLORATION

This analysis starts with a set of target applications that represent our domain. From these we can identify important operations. Crucial parts such as wide multipliers or fast adders should be singled out. In terms of the more familiar tool flow, this might be the equivalent of writing high-level HDL from an algorithm specification.

Next, this preliminary set of operators can be distilled to a smaller set of functional units 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. Notice that this step is unique to coarse-grain architecture development. Normally, the HDL is synthesized and mapped to an existing set of logic blocks. Here, since the architecture is still under development, we are determining what types of coarse-grain functional units the architecture should provide.

Lastly, based upon design constraints, the exact number and relative ratio 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. From here, existing tools [9, 10, 11] can take over to determine the exact placement of the functional units and routing resources.

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 [17] 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. Details of functional units used in our testing can be found in [16]. Furthermore, although beyond the scope of this paper, an automated system could be developed by leveraging a compiler tool such as SUIF [4] to extract dataflow graphs from applications. By beginning with one or, better yet, multiple implementations of a computation and intelligently modulating compiler effort, we can get a wide range of circuits to choose from, each with its own associated hardware requirements and throughput.

IV. 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 need 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. Details of the functional units we considered and the resource requirements for all 15 ciphers can be found in Appendix B of [15]. From this analysis, we demonstrate four main factors that obscure the relationship between the hardware resources that are offered and the potential throughput of the array.

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 five encryption algorithms, as in Table I, we can see the wide variation in throughput and RAM and crossbar requirements among the different algorithms.

TABLE I
BASELINE IMPLEMENTATION REQUIREMENTS

Algorithm (Baseline)	RAM Blocks	XBars	Cycles Per Data
CAST-256 (1x)	16	0	48
DEAL (1x)	1	7	96

Loki97 (1x)	40	7	128
Magenta (1x)	16	0	72
Twofish (1x)	8	0	16
Average	16.2	2.8	72.0
Std. Dev.	14.7	3.8	43.1

Compares the RAM, crossbar and clock cycle 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.

TABLE II
RAM NORMALIZED REQUIREMENTS

Algorithm (Unrolling Factor)	RAM Blocks	XBars	Cycles Per Data
CAST-256 (2x)	32	0	24
DEAL (32x)	32	104	3
Loki97 (1x)	40	7	128
Magenta (2x)	32	0	36
Twofish (4x)	32	0	4
Average	33.6	22.2	39
Std. Dev.	3.6	45.8	51.7

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 both the absolute and relative deviation in crossbars and clock cycles per output increase dramatically as compared to the baseline comparison.

TABLE III
THROUGHPUT NORMALIZED REQUIREMENTS

Algorithm (Unrolling Factor)	RAM Blocks	XBars	Cycles Per Data
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

Illustrates 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.

TABLE IV
IMPACT OF SCALING ON REQUIREMENTS

Algorithm (Unrolling Factor)	RAM Blocks	Muxes	Cycles Per Data
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

Unpredictable nature of hardware demands when unrolling algorithms.

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 Table II. 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 required clock cycles per data block increases sharply.

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 Table III 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 such a wide complexity gap.

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 Table IV for an example of this non-uniform behavior. This generally occurs because we would like to fully utilize as many of the resources as we can during each cycle but they are not necessarily needed evenly. For example, in almost any circuit memories either become a bottleneck, allowing the rest of the logic to be time-multiplexed between multiple RAM units, or sit idle waiting for other calculations to be finished, allowing the memories themselves to be time-multiplexed between multiple groups of logic. In this case we may see a step-wise relationship between memory and logic as we scale the calculation. This also leads to an unpredictable control complexity. This phenomenon makes it difficult to foresee the effect of decreasing the population of one type of functional unit and increasing another.

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.

V. FUNCTION UNIT ALLOCATION

To produce an efficient platform for a diverse group of applications we need an effective solution to the functional unit allocation problem. Such a solution 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.

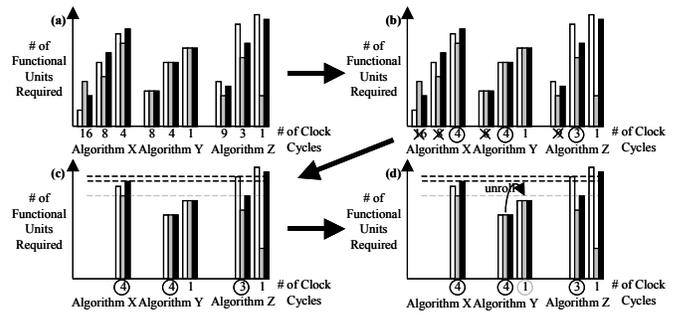


Fig. 2. Illustration of performance-constrained selection algorithm. a) We are given the functional unit requirements of three encryption algorithms in a range of performance levels. We are also given a hard throughput constraint, 4 clock cycles/block in this example. b) Determine the slowest implementation for each algorithm that still satisfies the minimum throughput requirement, then eliminate implementations below the performance threshold. c) Based on this subset of implementations, determine the minimum number of each resource type. d) If possible, further unroll the algorithms to better utilize available resources.

A. 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 and add them to the architecture. Finally, we select the highest performance implementation of each algorithm that fits on this new set of components. See Fig. 2 for an illustrated example of the performance-constrained functional unit selection process. The area estimate that we use for the testing reported in this paper is a simple transistor count model. This does not necessarily reflect the true area impact of adding or subtracting components since it does not capture the non-linear footprint of the associated routing resources necessary to support viable circuits. However, based on this model we can still determine general architectural trends and there is nothing in our methodology that would prevent a more accurate area model from being implemented.

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 an equal or 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.

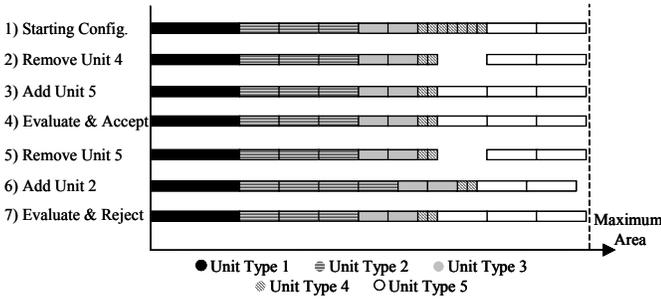


Fig. 3. Illustration of the area-constrained selection algorithm.

B. 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:

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

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 array. In our evaluations, we used a large constant penalty scaling factor (PC) to ensure that we included 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 can create discontinuities in the cost function, and therefore does not necessarily guide the simulated annealing to the best solution since a high throughput implementation may be very close 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 [6]. See Fig. 3 for an illustration of this procedure.

Note that, as described earlier, combinations of other functional units may emulate some operations. 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 by simply expanding the problem space.

C. 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:

$$Cost = \sum_{i=0}^{N-1} CC_i + Area Penalty \quad (2)$$

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

$$Area Penalty = PC * (CA / MA) \quad (3)$$

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

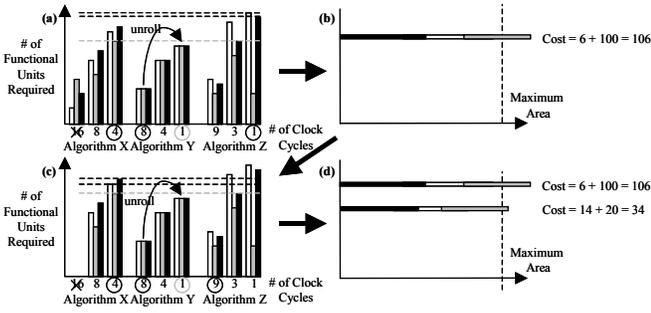


Fig. 4. Illustration of the improved area-constrained selection algorithm. In this example we assume the throughput threshold is set at 10 cycles/block. a) Eliminate any implementations below the given performance threshold, then randomly choose a throughput level for each algorithm and determine the minimum hardware requirements. Unroll the algorithms further, if possible. b) Evaluate the throughput and penalize for any excessive area required by the resulting architecture. c) Randomly choose a new implementation for one algorithm (Z in this case), and determine the hardware requirements for the new configuration. d) Despite the lower performance, the new state will be accepted due to a much lower area penalty.

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 B. See Fig. 4 for an illustration of this process.

VI. 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 15 of the ciphers 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 of the 15 AES encryption algorithms. As expected, the hard throughput constraint of the performance-driven approach has limitations. In Fig. 5 and Fig. 6 we plot the results of all three functional unit selection algorithms over ten area scenarios. Fig. 5 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.

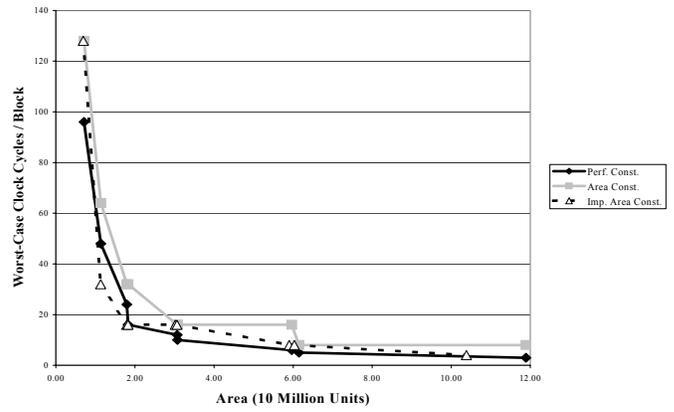


Fig. 5. Graph of maximum number of clock cycles required by any application in the domain as a function of area. Notice that while the improved area-constrained allocation technique occasionally produces better results, the performance-constrained method obtains the best worst-case performance over the majority of the design space.

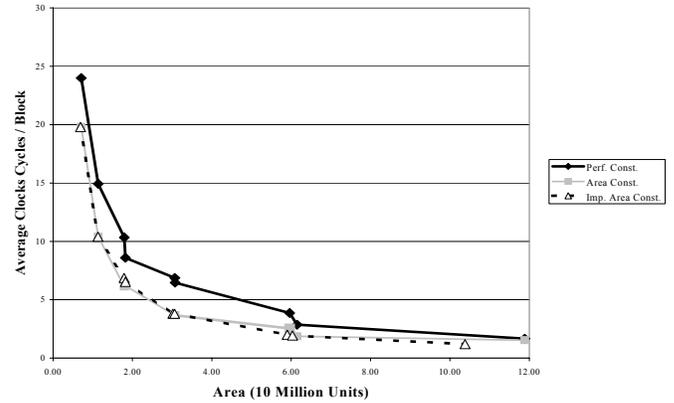


Fig. 6. Graph of the average number of clock cycles required to run all of the applications in the domain as a function of area. Notice that the overall performance of the higher throughput systems produced by the performance-constrained algorithm lag considerably behind that of the architectures generated by either of the area-driven techniques.

In contrast, though, when we plot the average number of clock cycles required by all 15 of the algorithms in the domain as a function of area, as in Fig. 6, we see a completely different picture for the performance-constrained selection method. Fig. 6 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, Fig. 6 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, Fig. 5 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, thus giving us an area benefit. 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, is too complex to solve via such methods. For the general case, given any decent number of applications N and any reasonable number of implementations for each M , a brute force search will be computationally intractable as the solution space is N^M . However, the AES competition also produced a smaller set of five finalist algorithms. We have run this set through each of our algorithms, and compare them to the optimal results found by a brute-force search. Fig. 7 and Fig. 8 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 for the entire 15-algorithm domain at least approach the optimal architecture.

Expanding upon this idea, we can also illustrate some subtleties of the functional unit allocation problem and the sensitivity of our tool by showing how the performance of the array changes depending upon what is used to specialize the architecture. Fig. 9 and Fig. 10 show the results of testing

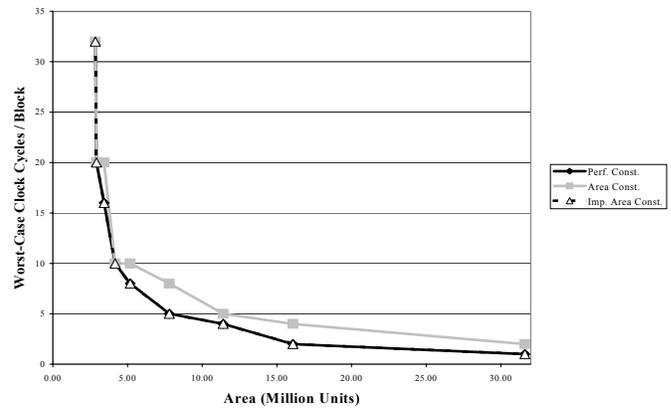


Fig. 7. Graph of maximum number of clock cycles required by any of the five finalist algorithms as a function of area. Although the optimal curve is not shown for clarity, both the performance-constrained and improved area-constrained techniques achieve optimal results for these limited scenarios.

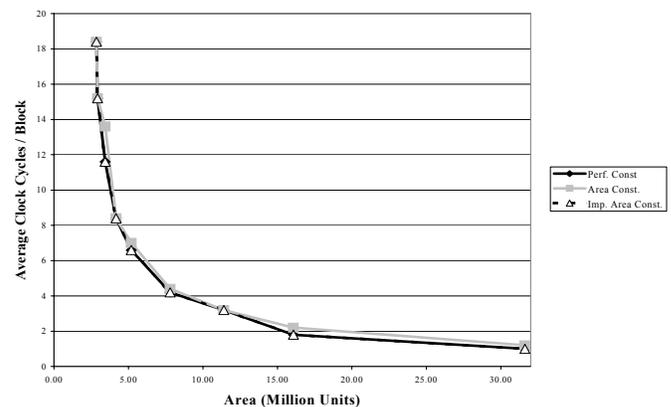


Fig. 8. Graph of the average number of clock cycles required to run all five of the AES finalist ciphers as a function of area. Again, although the optimal curve is not shown for clarity, both the performance-constrained and improved area-constrained techniques achieve optimal results for these limited scenarios.

when we use the five finalist algorithms to determine the composition of an architecture, then attempt to map all of the ciphers to the suggested array. Here we can clearly see that both the worst-case and average throughput of a netlist can be greatly affected depending on whether or not it was considered while the architecture was being developed. The worst-case performance of the ten ciphers not included in the allocation process is up to two orders of magnitude worse than that of the five algorithms considered in the process. Furthermore, the average performance of the unconsidered ciphers is up to one order of magnitude worse than that of the five considered algorithms. As a result, the worst-case performance of the finalist-specialized array when implementing all 15 algorithms is up to 10.6x worse than that of the performance-constrained architectures designed earlier. This also has the result that the average performance of the finalist-specialized array when implementing all 15 algorithms is up to 4.8x worse and that of the improved-area constrained architectures designed earlier.

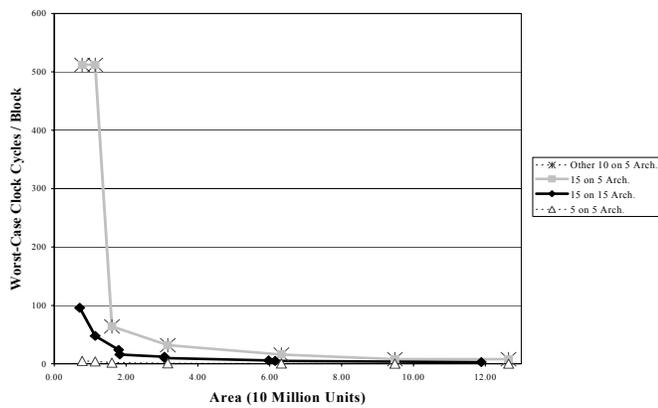


Fig. 9. Comparison of the maximum number of clock cycles required by any of the 15 AES candidates when the architecture is developed based upon the requirements of only the five finalist ciphers. Note that the lines from bottom to top indicate the worst-case performance of finalists on finalist-specialized architecture, all 15 on the original domain-specialized architecture, all 15 on the finalist-specialized architecture and the non-considered 10 on the finalist-specialized architecture.

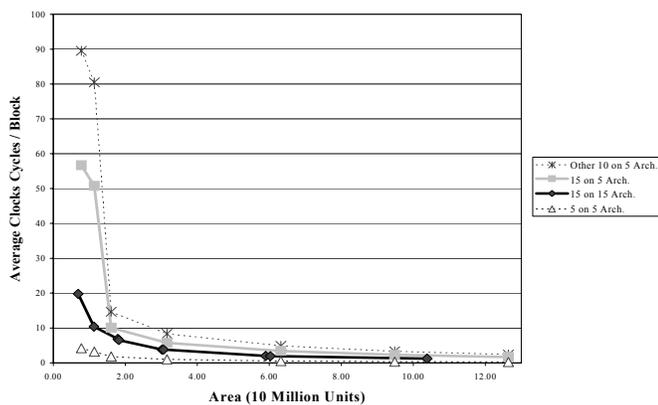


Fig. 10. Comparison of the average number of clock cycles required to run all 15 AES candidates when the architecture is developed based upon the requirements of only the five finalist ciphers versus all 15 algorithms. Again, note that the lines from bottom to top indicate the average performance of finalists on finalist-specialized architecture, all 15 on the original domain-specialized architecture, all 15 on the finalist-specialized architecture and the non-considered 10 on the finalist-specialized architecture.

VII. 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 we have shown that a carefully designed architecture will likely perform well on the limited set of applications that were used to create the design, we have not determined if a special 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? Since we only considered private-key ciphers, how would public-key encryption algorithms fare? How about completely unrelated computation such as DSP applications?

Another issue, related to adaptability, is specialization. While it is expected that an architecture developed from a larger domain of applications will better handle unanticipated netlists, a device centered upon a smaller target domain will be more specific and perform better on known necessary applications. It is vital that domain-specialized reconfigurable device developers understand how these factors affect each other.

We have briefly touched on both of these issues in this paper. If we imagine that the unconsidered algorithms were instead algorithmic updates or unanticipated netlists it can shed light on these phenomenon. For instance, while we see from Fig. 9 and Fig. 10 that there can be a large difference in the composition of an array depending upon how large a portion of the domain is used to generate the architecture, the lines seems to converge when considering larger systems. This is encouraging, as it seems to support one hypothesis that that we began with: there is likely an underlying and exploitable similarity between encryption algorithms. However, there is still a large volume of work to be completed before we can come to any definitive answer considering domains in general. For example, the testing performed in this paper may not necessarily reflect what would happen in actual use. As they were all developed at approximately the same time and mostly independent of each other, they do not share the same design philosophy as true algorithmic updates or genuinely related functions such as DSP filters might.

VIII. 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 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 has the potential to 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.

IX. REFERENCES

- [1] C. Adams and J. Gilchrist. "The CAST-256 Encryption Algorithm." *First AES Candidate Conference*, Aug. 20-22, 1998.
- [2] E. Ahmed 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] R. Anderson, E. Biham and L. Knudsen. "Serpent: A Proposal for the Advanced Encryption Standard." *First AES Candidate Conference*, Aug. 20-22, 1998.
- [4] S.P. Amarasinghe, J.M. Anderson, C.S. Wilson, S.W. Liao, B. R. Murphy, R. S. French, M. S. Lam and M. W. Hall. "Multiprocessors from a Software Perspective." *IEEE Micro*, June 1996: 52-61.
- [5] V. Betz and J. Rose. "FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density." *International Symposium on Field Programmable Gate Arrays*, 1998: 59-68.
- [6] V. Betz and J. Rose. "VPR: A New Packing, Placement and Routing Tool for FPGA Research." *International Workshop on Field Programmable Logic and Applications*, 1997: 213-22.
- [7] L. Brown and J. Pieprzyk. "Introducing the New LOK197 Block Cipher." *First AES Candidate Conference*, Aug. 20-22, 1998.
- [8] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr., L. O'Connor, M. Peyravian, D. Safford and N. Zunic. "Mars - A Candidate Cipher for AES." *First AES Candidate Conference*, Aug. 20-22, 1998.
- [9] K. Compton and S. Hauck. "Totem: Custom Reconfigurable Array Generation." *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.
- [10] K. Compton, A. Sharma, S. Phillips and S. Hauck. "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems." *International Conference on Field Programmable Logic and Applications*, 2002: 59-68.
- [11] K. Compton and S. Hauck. "Track Placement: Orchestrating Routing Structures to Maximize Routability." *International Conference on Field Programmable Logic and Applications*, 2003:121:130.
- [12] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa and C. Ebeling. "Architecture Design of Reconfigurable Pipelined Datapaths." *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999:23-40.
- [13] J. Daemen and V. Rijmen. "AES Proposal: Rijndael." *First AES Candidate Conference*, Aug. 20-22, 1998.
- [14] C. Ebeling, D. C. Cronquist, and P. Franklin. "RaPiD - Reconfigurable Pipelined Datapath." *The 6th International Workshop on Field-Programmable Logic and Applications*, 1996: 126 - 35.
- [15] K. Eguro. "RaPiD-AES: Developing an Encryption Specific FPGA Architecture." Master's Thesis. Dept of Electrical Engineering, University of Washington, Dec. 2002.
- [16] K. Eguro and S. Hauck, "Decipher: Architecture Development of Reconfigurable Encryption Hardware", *University of Washington, Dept of EE Technical Report UWEETR-2002-0012*, 2002.
- [17] National Institute of Standards and Technology. Advanced Encryption Standard (AES) Development Effort. Nov. 11, 2002. <<http://csrc.nist.gov/encryption/aes/index2.html>>.
- [18] R. Rivest, M. J. B. Robshaw, R. Sidney and Y. L. Yin. "The RC6 Block Cipher." *First AES Candidate Conference*, Aug. 20-22, 1998.
- [19] B. Schneier, 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.

Ken Eguro received the B.S. in computer engineering in 2000 from Northwestern University, Evanston, IL and the M.S. in electrical engineering in 2002 from the University of Washington, Seattle, WA where he is currently pursuing the Ph.D. in electrical engineering.

Scott Hauck is an Associate Professor of Electrical Engineering at the University of Washington. He received his B.S. in Computer Science from U. C. Berkeley in 1990, and his Masters and Ph.D. from the University of Washington Department of Computer Science. From 1995-1999 he was an Assistant Professor at Northwestern University. Dr. Hauck's research concentrates on FPGAs, including architectures, applications, and CAD tools. For these efforts he has received an NSF Career Award, a Sloan Fellowship, and a TVLSI Best Paper award.