Exploiting Configuration Dependencies for Rapid Area-efficient Customization of Soft-core Processors

Deshya Wijesundera, Alok Prakash, Siew Kei Lam, Thambipillai Srikanthan
School of Computer Engineering,
Nanyang Technological University, Singapore 639798
deshyase001@e.ntu.edu.sg, {alok, assklam, astsrikan}@ntu.edu.sg

ABSTRACT
The large number of possible configurations in modern soft-core processors make it tedious and time consuming to select the optimal configuration for a given application. In this paper, we propose a framework for rapid area-efficient customization of soft-core processors that exploits the dependencies between the various configuration options to prune the design space. Additionally, the proposed technique relies on rapid and accurate estimation models instead of the time consuming synthesis and execution techniques proposed in the existing work. Experimental results based on hand-coded applications and applications from the popular *CHStone* benchmark suite show that the proposed framework can rapidly and reliably select the best processor configuration for a given application and save an average of 47.58% area over the processor with all the configuration options enabled while achieving similar performance.

CCS Concepts
• **Hardware~Software tools for EDA**

Keywords
design space pruning; application-specific customization; soft-core processors; area constraints; performance-area trade-off; CSoC; FPGA.

1. INTRODUCTION
Field Programmable Gate Array (FPGA) based Configurable System-on-Chip (CSoC) has been the attractive alternative over Application Specific Integrated Circuits (ASICs) for embedded system implementation in the recent past due to fast Time-to-Market (TTM) and low Non-Recurring Engineering (NRE) cost [14]. The re-configurability and heterogeneous real estate of FPGA provides reusability and shorter development cycles. A CSoC, typically consists of a processor, hardware accelerators and a memory subsystem.

In a CSoC, the developer has an option to select either a hard-core processor, soft-core processor or both. A soft-core processor, unlike a hard-core processor, offers immense flexibility for further customization through micro-architectural modifications as well as instruction set extension. Soft processors continue to be the choice in providing a level of programmability, allowing non-experts to configure FPGAs [2], [3], [15]. Soft processors are used in auxiliary functions of the system, such as managing non-critical data movement, providing an interface for configuration, or implementing the cognitive functions in an adaptive system [3].

SCOPES '16, May 23-25, 2016, Sankt Goar, Germany

DOI: http://dx.doi.org/10.1145/2906363.2906385

The domain of improving processor architectures continue to grow with FPGA architecture-tailored soft processor designs being proposed [1], [2], [3], [4], [7], [15], [31], [32]. Moreover, FPGAs are used to implement full applications rather than merely being used as accelerators, hence soft-cores dominate due to their ability to be application specific [7]. Cheah et al. [7] further state that soft-processors continued to dominate since earlier attempts in introducing hard processors in the *Virtex II Pro* [28] and *Virtex 4 FX* [27] were not entirely successful as a particular fixed processor may not always suit the wide range of applications which can be implemented on FPGA. The increasing popularity of partial reconfiguration also makes soft-core processors the preferred choice over hard-cores [23]. At the same time, soft processors are increasingly being used in FPGAs for reliable computing systems [25]. Cheah et al. [3] state that in recent work, soft processors have been demonstrated as a viable abstraction of hardware resources, allowing multi-processor systems to be built and programmed easily.

However, the process of finding the optimal configuration tailored for a particular application is often tedious and time consuming due to the large number of configuration options available in modern soft-core processors. Hence, a methodology that reduces the runtime and effort for application specific customization of a soft-core processor is a necessity in the embedded systems domain with stringent TTM constraints. Further, tight area constraints imposed on embedded devices require the embedded designer to create area-efficient designs that achieve high performance. Growth of a multitude of soft-processor architectures makes it imperative that the customization methodology remains adaptable to a given processor architecture.

In this paper, we propose a methodology for rapid design space exploration and pruning for soft-core processors under area constraints without performing lengthy synthesis and simulation runs. The main contributions of this paper are,

1. Analysis of the dependencies for a given processor architecture
2. Propose a tree based heuristic to prune the design space in a time-efficient manner
3. A methodology for rapid application specific soft-core processor customization under area constraints

The rest of the paper is organized as follows. Section 2 discusses the existing literature. Section 3 describes the proposed methodology, the identified bottlenecks and the implementation details. Section 4 presents the experimental results and discussion. Finally, we conclude in section 5.

2. RELATED WORK

Methods for micro-architectural modification as well as instruction set customization have been proposed by researchers in view of exploiting the benefits of a soft-core processor. Leading FPGA vendors *Altera* [12] and *Xilinx* [29] provide different versions of their soft-core processors and also support instruction set customization and extension. For example, *Altera* provides 2 versions of the *Nios II Gen2* processor [12] with different pipeline depths, customization options and custom instructions. Instruction subsetting is defined as creating an application specific instruction set processor from a more general processor, by removing the support for unused instructions [5]. The myriad of configuration options in Commercial FPGA-based soft-core processors [12], [29] facilitate instruction subsetting. Instruction subsetting provides significant improvements in performance while reducing area utilization and power consumption [14], [21].

Application specific customization of processors has been a hot research topic in the recent past due to its inherent benefits. [17], [20] and [26] propose methodologies for soft-core processor customization for specific types of applications. Yiannacouras et al. [30] and Padmanabhan et al. [17] perform application specific customization on *Nios* and *LEON* processors respectively, with significant gain in performance and area. However, these approaches necessitate synthesis and execution of the code thus, incurring exponentially increasing time with increasing number of configurable parameters. For example, an exhaustive search for a suitable configuration in *MicroBlaze* could take up to 11 hours [19], [21]. Sheldon et al. [21] propose a synthesis-in-the-loop approach using impact ordered tree search heuristic. This approach reduces the search to a fraction of the solution space but requires synthesis and execution of the code. The application-specific impact ordered tree based search requires a run time of 3.33 hours. The fixed ordered tree based approach can run in 1.5-2 hours but behaves poorly in certain scenarios. Further, this approach does not consider dependencies between configurations. Sheldon et al. propose improvements on the work in [21] using a design of experiments (DoE) approach [22]. However, this approach also requires synthesis runs of 2-3 hours and, manual analysis and exclusion of infeasible configurations. Since the DoE tool does not analyze infeasible combinations, this methodology can have scenarios where the considered design space has a significant component of infeasible combinations which can have a direct negative impact on the results. Further, the DoE approach is implemented through manual intervention of the DOE PRO XL tool [24].

Prakash et al. [19] claim to significantly reduce run time using an estimation approach. However, this method uses exhaustive estimation for each possible configuration and thus, will require significant effort with increasing number of configurations, even though only 12 configurations of *Nios II* is used for analysis purposes in this work. The paper mentions that the *LEON* processor has $5*2^{10}$ configurations. Therefore, such a processor would increase the run time claimed in [19] by approximately 2^9 times. Moreover, the work does not analyze dependencies between configuration options, which can result in the selection of configurations which have repetitive functionality or are not feasible. For example, the work mentions that dependency between the Floating Point unit and Floating Divider unit is not considered in the analysis.

3. METHODOLOGY

In this section we describe the proposed methodology for rapid application specific design space exploration for soft-core processors. Soft-core processors offer user configurable functional units (FUs) which accelerate the performance by implementing a given set of instructions directly on hardware. If the mentioned FU is not selected for the processor, the corresponding instruction generates an exception which is then served and executed by the processor similar to other instructions from its Instruction Set Architecture (ISA). Our approach leverages this fact to identify the configurable FUs which would benefit the given application and as a further step we try to optimize the performance-area usage by finding the most suitable FUs in an area-constrained design. Our approach can be abstracted into 4 phases namely, processor characterization, architecture-aware dependency analysis, application profiling & analysis and selection heuristic. Even though we use the *Nios II* soft-core processor for verification of the proposed methodology, the proposed approach is generic and can be extended to any soft-core processor. The framework is shown in Figure 1.

3.1 Processor Characterization

Processor characterization is a one-time process for a given processor. This phase involves identification of the configurable FUs of the target processor, the corresponding instructions and the relevant performance-area characteristics for each FU. For example, in the *Nios II* processor, we identified 5 configurable FUs, namely 32-bit Integer Multiplier, 64-bit Integer Multiplier, Integer Divider, Shifter and the Floating Point unit. It is important to mention that the Floating Point unit is provided as a custom instruction interface in the *Nios II Gen2* processor design but it could be analyzed as an additional FU [8]. Certain FUs can be implemented with different configurations. This further increases the complexity of the analysis in phase 2 as each configuration provides a different performance-area metric.

We need to analyze the area occupied by each FU in the FPGA real estate as our aim is to identify the configuration which maximizes the performance, given an area constraint. We initially derive the area of each configurable unit in hardware using the *Altera Quartus* [10] tool. It was observed that certain configurations utilize the on-chip digital signal processing (DSP) blocks. In order to compare the results across all the configurations we replace DSP with the equivalent adaptive logic modules (ALMs). To derive the equivalent number of ALMs for a DSP, we switch off the functionality of DSPs in the *Altera Quartus* tool when generating the processor.

The next step identifies corresponding instructions which benefit from the relevant FUs. This data is extracted from the datasheet or obtained through simulations performed under test conditions. This process is required as we later map the corresponding LLVM [12], [18] instructions to the FUs during application profiling in phase 3. Also, it is important to identify the execution times of these instructions for both hardware and software based implementations. Our approach utilizes the gain of these instructions in order to predict the best possible configuration of a processor tailored for an application.

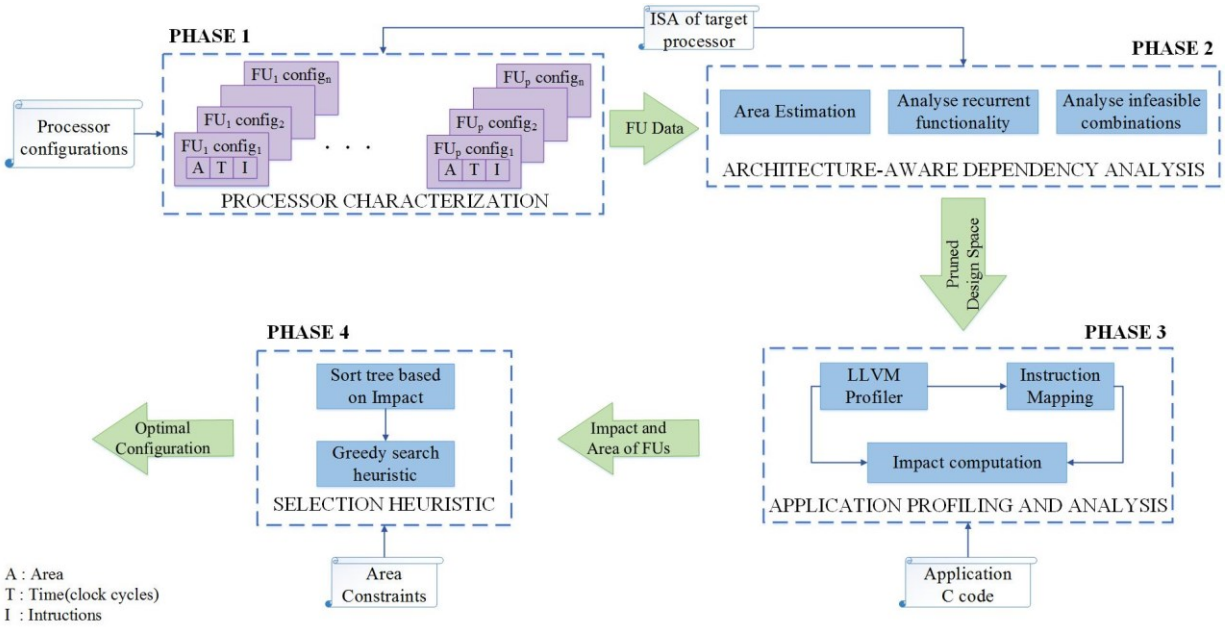


Figure 1. Design Space Pruning Framework

3.2 Architecture-aware Dependency Analysis

This is a one-time phase for a given processor. Initially, we analyze performance-area data for all possible configurations for each FU and identify the best suited configuration/configurations. This step removes repetitive functionality and selects the configuration with optimal performance and minimal area in the case of configurations with comparable area.

Then, we analyze the dependencies between configurable FUs to eliminate configurations which are either impossible combinations or have repetitive functionality. For example, the 64-bit Integer Multiplier cannot be configured as a standalone FU. It can be configured only in conjunction with the 32-bit Integer Multiplier. Also, the functionality of the Shifter FU is automatically included when the 64-bit Integer Multiplier is configured. Such architecture-specific dependencies are analyzed in this phase of the framework.

Our algorithm analyzes all possible configurations for the processor to derive the best configuration. Hence, we need to accurately estimate the area utilization of configurations with multiple FUs. Based on the results of exhaustive synthesis runs for different configurations in phase 1, we postulate that area utilization for such configurations has an additive effect. For example, area estimation for a configuration with the Base Processor, Integer Divider and 32-bit Integer Multiplier should be equal to the addition of the individual areas for the Base Processor, Integer Divider and 32-bit Integer Multiplier. In section 4, we verify our approach with the observed results.

The benefits of the analysis are twofold. It prevents the designer from analyzing unnecessary configurations and also identifies the most profitable configuration among configurations with repetitive functionality. This step is critical as it provides significant gain to the run-time of the application by pruning the design space.

3.3 Application Profiling & Analysis

In this phase, the application is compiled and profiled using the LLVM open-source complier. This process is independent of the target processor architecture. The application C code is initially compiled using LLVM to produce the LLVM Intermediate Representation (IR). We use the LLVM IR to extract function names, basic blocks, instructions in the basic blocks, the arguments and data types. We have written a LLVM pass for this purpose. The LLVM-profiler is used to extract execution frequencies of basic blocks. The extracted details with the information obtained in phase 1 are further processed to derive the execution count of each type of instruction which is then aggregated to the abstraction level of FUs.

We utilize the extracted instruction counts as well as the area information to compute the gain and impact of each FU for a given application using equation (1) and equation (2) respectively.

$$G_{FU} = \sum_{i=0}^{n} (CC_{iSW} - CC_{iHW}) \times N_i \qquad (1)$$

$$I_{FU} = G_{FU} \div A_{FU} \qquad (2)$$

Where,

G_{FU} : Gain of FU,
I_{FU} : Impact of FU,
A_{FU} : Area of FU,
n : Number of instruction types,
CC_{iSW} : Clock cycles in software,
CC_{iHW} : Clock cycles in hardware,
N_i : Instructions/type$_i$

It is important that we use the derived dependencies among the FUs from phase 2 in this phase of the methodology. For example, the 64-bit Integer Multiplier when instantiated, performs 32-bit integer multiplication and shifting in addition to the 64-bit multiplication. Similarly, we have utilized the other dependencies

and redundancies for our analysis. Finally, we provide the impact and area of each FU to the selection heuristic.

3.4 Selection Heuristic

The final phase of the proposed methodology is the area constrained selection heuristic. The output of phase 3 and user specified area constraint are the inputs to this phase. In this phase, our main goal of optimizing performance under area constraints for a given application is approached using a greedy search heuristic which is based on a sorted tree structure. This methodology is simple to implement yet has considerable accuracy with minimal time complexity.

We have experimented with two variants of the greedy search heuristic. We call the first approach as the 1-greedy approach while the second method is called the 2-greedy approach. In the 1-greedy approach, a tree sorted by the impact is pruned if the local area constraint is not met by a FU. For example, in Figure 2 for the *ADPCM* algorithm, if the area constraint is 1700 ALMs the element at the root of the tree *IM64*, cannot be accommodated. The base processor accommodates 1304 ALM and the remaining area of 396 ALM is insufficient to incorporate the *IM64* FU (requires 434 ALM). Thus, we prune the root and move to the next node of the tree that has the second highest impact. This element, *S* could be accommodated as the area constraint is met. This leaves us with 352 ALMs and the next node of the tree that has an area constraint of 324 can be accommodated. Similarly, we move to the bottom of the tree until either the area constraint is met or the leaf node is reached. Finally, we analyze the chosen configurations with the provided data from previous phases to eliminate any impossible configurations and derive the final output.

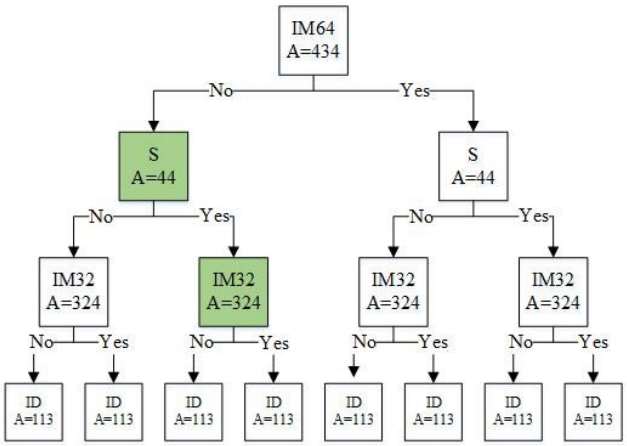


Figure 2. Ordered tree for 1-greedy search heuristic

In the 2-greedy approach we initially combine all possible combinations of two FUs. We refer to this as a node in the tree. Similar to 1-greedy search heuristic we sort them according to the impact and subsequently prune using the area constraint until the leaf node is reached. The pseudocode for 1-greedy and 2-greedy algorithms are given in Algorithm 1 and 2 respectively.

4. RESULTS AND DISCUSSION

This section presents the results of our work. We used the *Nios II fast* soft-core processor from *Altera* [12] for the experiments. *Nios II Gen2* processors provide 2 different variations of processor cores, i.e. economy (e) and fast (f). The fast core has been selected as it provides better performance-area than the economy core [30]. In order to verify the results of our algorithm we exhaustively synthesized the *Nios II* processor with various combinations of the configuration options using the *Altera Quartus* development environment and *Qsys* tool [10]. The different benchmark applications were then executed on these processors under a *ModelSim* [16] simulation environment. The selected target FPGA platform is *Altera Cyclone V* [9]. However, it should be noted that the proposed techniques in this work are independent of the platform. We used all the applications from the popular *CHStone* benchmark suite [6] for experimentation. The *CHStone* benchmark suite provides a balanced distribution of workloads with a variation in their compute intensity. However, these applications do not include single precision floating point operations. Hence, we have also used 2 hand-coded applications that contain several single precision floating point operations to exercise the single precision floating point FU.

4.1 Area Estimation

Area estimation is performed in phase 2 of the methodology and the estimated area is utilized in calculating the impact of FUs in phase 3. Further, the selection heuristic in phase 4 also utilizes the area in pruning the ordered tree. The area of the FUs is calculated in terms of ALMs and DSPs. In order to use a common metric for our calculations we derive the area in equivalent ALMs as described next.

Firstly, we obtained the area utilization of each FU from the synthesis results after synthesizing the processor twice for every FU, once with DSP and once without DSP support. In the case of DSP disabled configuration, the corresponding functionality is implemented in the ALMs. The area results are shown in Table 1. Next, we deduce the area for each FU, by deducting the area of the base processor from the relevant configurations. The results for this process are given in Table 2. Comparing the area results for the DSP enabled and disabled runs, the equivalent ALMs required per DSP for each FU is obtained as shown in Table 2. It is interesting to note that the equivalent ALM utilization per DSP is dependent on the FU. We attribute this fact to organization of the DSP blocks and allocation of resources within the blocks. We rely on the numbers in Table 2 to estimate the area utilization of configurations with multiple FUs.

An additive approach is used to estimate the area of configurations with multiple FUs. Figure 3 presents the results of area estimation for the pruned design space in phase 2 compared to the actual values. The maximum error in area estimation is less than 1.2% and the average error in estimation is 0.55%. Hence, the additive method could be reliably deployed to estimate the configurations with multiple FUs.

Algorithm 1: Pseudocode for 1-greedy Search Heuristic

```
begin:
    sort the functional units according to impact;
    eliminate functional units with 0 impact;
    RA = RA – Area BP;
    For each node i :
    {
        if (Area Ni < RA) :
        {
            accommodate node i;
            RA = RA – Area Ni;
        }
        else prune node i;
    }
end
```

Algorithm 2: Pseudocode for 2-greedy Search Heuristic

```
begin:
    generate all possible combinations of 2 configurations;
    sort the new combined node array according to impact;
    eliminate combinations with 0 impact;
    RA = RA – Area BP;
    For each combined node j :
    {
        if (Area Nj < RA) :
        {
            accommodate node j;
            RA = RA – Area Nj;
            break;
        }
        else prune node j;
    }
    Apply 1-greedy for the rest of the individual nodes
end
```

Where,

RA: Remaining area,
Area BP: Area of base processor,
Area N_i: Area of i^{th} node,
Area N_j: Area of node for j^{th} combination

Table 1. Area characteristics of configurations

Configuration	Area		
	DSP Enabled		DSP Disabled
	ALM	DSP	ALM
Base Processor (BP)	1304	0	1304
BP + Integer Divider (ID)	1417	0	1417
BP + 32-bit Integer Multiplier (IM32[1])	1349	3	1628
BP + 64-bit Integer Multiplier (IM64)	1330	3	1737
BP + Shifter(S[1])	1348	0	1348
BP + Floating Point Hardware 2 (FP2)	2369	5	2750

Table 2. Equivalent ALMs of FUs

FU	Area			
	DSP Enabled (Configuration – Base)		DSP Disabled	
	ALM	DSP	ALM (Configuration – Base)	ALM per DSP (Configuration – Base)/DSP
ID	113	0	113	0
IM32[1]	45	3	324	93
IM64	26	3	433	136
S[1]	44	0	44	0
FP2	1065	5	1446	76

[1] The values for the selected method of implementation in Table 3 is mentioned in this case. Similar analysis has been done for all methods of implementation.

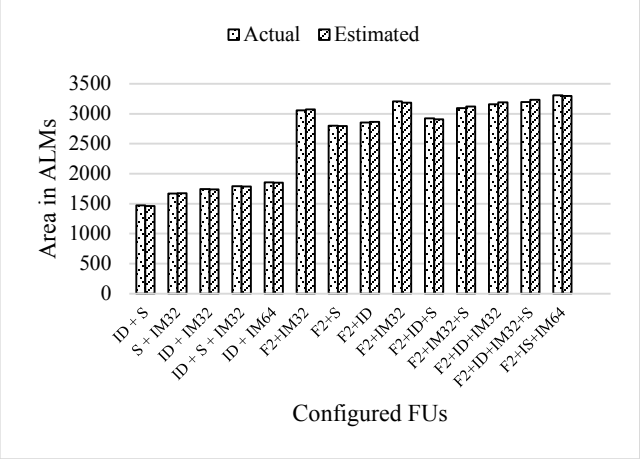


Figure 3. Comparison between actual and estimated area

Table 3. Selected configurations of FUs

Configurable FU	Implementation Methods	Selected Method
32-bit Integer Multiplier	Logic elements 3 * 16 Multiplier 1 * 32 Multiplier	3 * 16 Multiplier
Shifter	Pipelined Non-pipelined	Pipelined

Table 4. Timing characteristics of configurations

FU		Execution Time (Clock cycles)		Gain (SW-HW)
		Software (SW)	Hardware (HW)	
BP		N/A	N/A	N/A
ID		56	4 - 66	24
IM32		140	1	139
IM64		150	1	149
S		1 - 32	1	15
FP2	add	196	5	191
	sub	218	5	213
	div	517	16	501
	mul	774	4	770
	sqrt	494	4	490
	float to int	69	2	67
	int to float	115	4	111
	min	151	1	150
	max	160	1	159
	abs	11	1	10
	compare	91	1	90

4.2 Dependency Analysis

As mentioned in section 3, some configurable FUs provide several methods of implementation, thereby increasing the complexity of exploring the design space. In phase 2, we select one/several methods for each FU, based on performance optimization under area constraints. The respective FUs, implementation methods and selected method are presented in Table 3.

Our observations in phase 2 of the methodology are given below.

- *Nios II* provides the 64-bit Integer Multiplier unit as a separate configurable FU. The implementation of this unit requires the implementation of the 32-bit Integer Multiplier as 3 * 16-bit or 1 * 32-bit multipliers
- The execution time of each FU depends on the method used for implementation. For example, the 32-bit Integer Multiplier implemented as 3 * 16-bit multipliers, 1 * 32-bit multipliers or using logic elements will have different performance metrics.
- Area utilization depends on the configuration.
- Double precision floating point instructions are always implemented in software.

The proposed algorithm is based on the gain achieved by implementing instructions on dedicated hardware. Table 4 presents the execution time in clock cycles and the corresponding gain for each configuration, considering the selected method of implementation in Table 3. All execution times in software have been obtained through experimental results, except for the Shifter FU. The execution times in hardware and the software execution time for Shifter FU have been obtained from datasheets available online for *Nios II* [8]. The gain is calculated as a difference between the software and hardware execution time and is used for calculation of impact in phase 3.

Dependency analysis has the ability to rapidly prune the design space. A total of 8 independent implementation methods are considered. Theoretically, there are 256 different configurations. The design space is first pruned to 96 configurations by removing the infeasible combinations. For example, multiple instances of the same FU using different methods of implementation cannot be configured in the same configuration. Selecting the best configuration for the remaining combinations as given in Table 3, further prunes the design space to 32 configurations. Dependency and redundancy analysis further reduces this to 20 configurations in the case of the selected processor. We achieve 79.17% reduction in the total number of configurations which require analysis.

4.3 Performance Estimation

The results of profiling in phase 3, for the benchmarks and the hand-coded applications are presented in Table 5. It is observed that applications such as *ADPCM* and *JPEG* contain all types of instructions under consideration except floating point operations. Also, it is important to note that the benchmark suite consists of applications that use floating point instructions. However, it was revealed that all of them are double precision instructions. As discussed earlier in section 4.2, the *Nios II* core processes double precision floating point operations in software, thus these instructions were not accelerated using the Floating Point Hardware 2 FU. Hence, as mentioned in section 4, we use two hand-coded algorithms to prove the scalability of our methodology for single precision floating point arithmetic.

Figure 4 presents the impact of each FU for the test applications, computed in phase 3. The graph indicates that the impact of each FU varies with the application and as a designer, the complexity of manually choosing the best possible configuration is tedious. Further, an area constrained design space increases the complexity as the optimization goal is based on a performance-area trade-off.

Figure 5 presents the results obtained for the pruned design space in the exhaustive technique for SHA, MOTION, AES, JPEG, FLOAT_HW1 and FLOAT_HW2. A similar process has been carried out for the other applications. These results indicate that

Table 5. Instruction counts for test applications

Application	Instruction Count				
	ID	IM32	IM64	S	FP
ADPCM	3776	3306	56229	36590	0
MIPS	0	0	0	2926	0
GSM	0	0	1868	1955	0
BLOWFISH	0	0	0	70302	0
SHA	1044	0	0	96349	0
MOTION	0	0	0	141	0
AES	259	7023	0	5892	0
JPEG	86884	812018	1232352	1902149	0
DFADD	0	0	0	6325	0
DFDIV	27	0	1104	3471	0
DFMUL	0	0	836	2552	0
DFSIN	802	804	50184	225435	0
FLOAT_HW1	0	0	0	0	105
FLOAT_HW2	3776	3306	56229	36590	46725

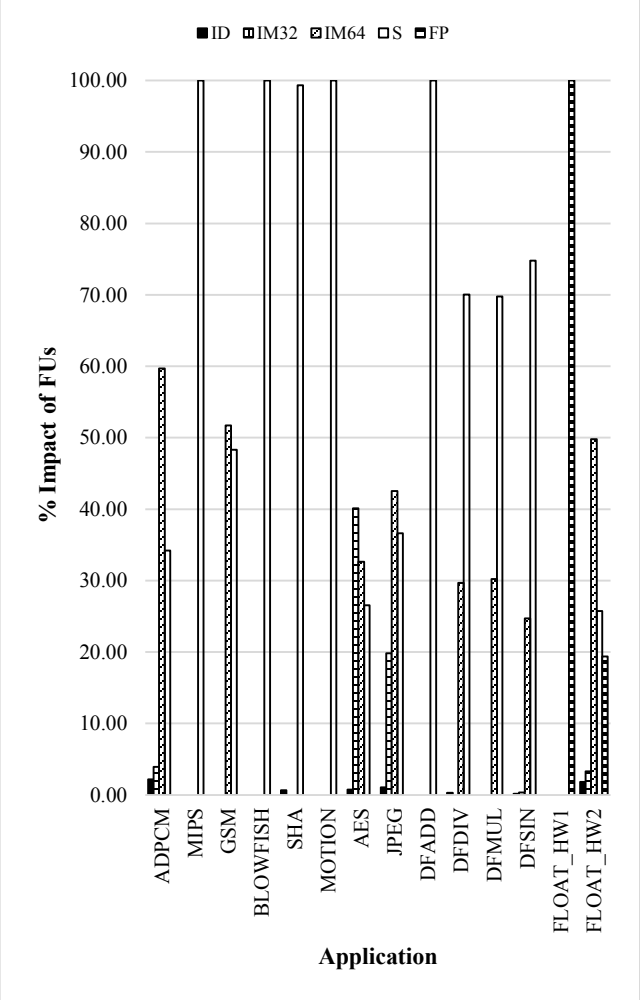
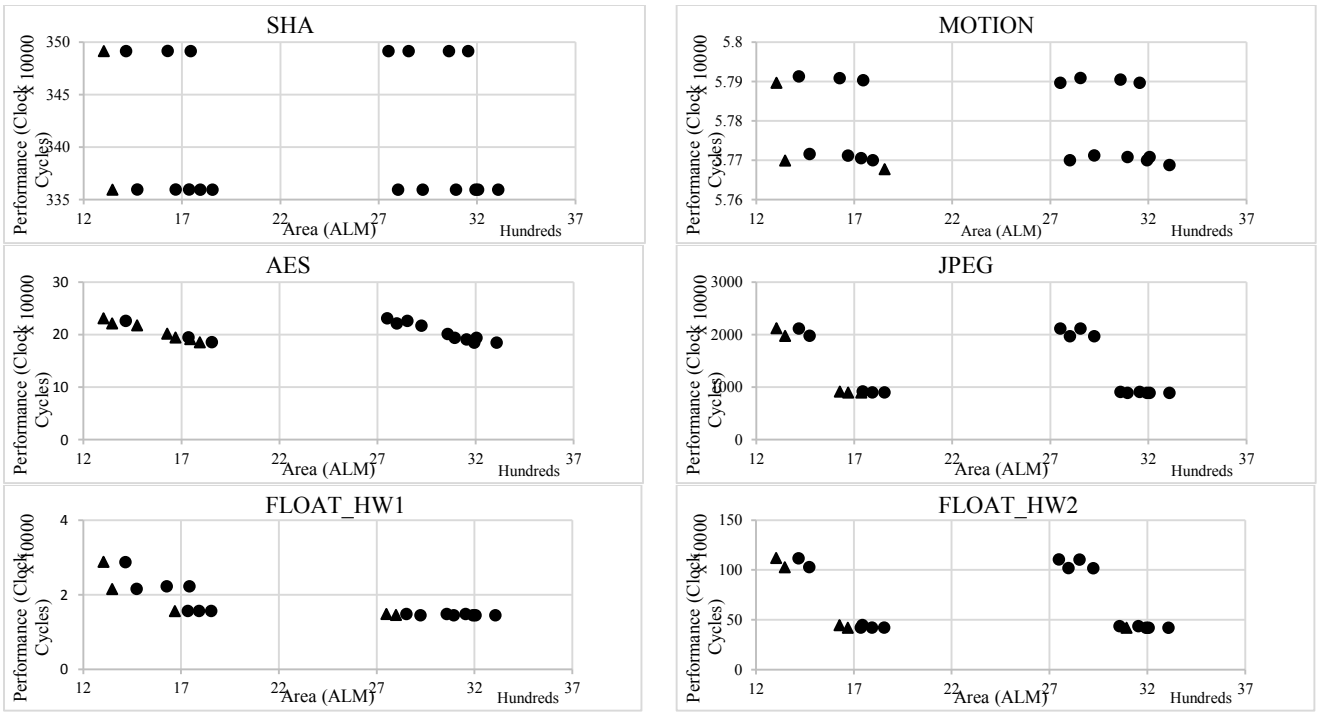


Figure 4. Distribution of FU Impact

the configuration which gives the best performance depends on the application. Exhaustive runs will be time consuming and the results have to be further analyzed to find the optimal configuration meeting specified area constraints.



The dots respresent the design space while the triangles represent the pareto points.

Figure 5. Results of exhaustive technique

Table 6. Comparison of results

Application	Selected FUs								
	ALM Constraint < 1700			ALM Constraint < 1800			ALM Constraint < 3400		
	E*	1-G*	2-G*	E*	1-G*	2-G*	E*	1-G*	2-G*
ADPCM	S+IM32	S+IM32	S+IM32	S+IM32	IM64	IM64	IM32+S	IM64 +ID	IM64 +ID
MIPS	S	S	S+ID	S	S	S+ID	S	S	S+ID
GSM	S+IM32	S	S+ID	IM32+S	IM64	IM64	IM32+S	IM64	IM64
BLOWFISH	S	S	S+ID	S	S	S+ID	S	S	S+ID
SHA	S	S+ID	S+ID	S	S + ID	S+ID	S	S+ID	S+ID
MOTION	S	S	S+ID	S	S	S+ID	S	S	S+ID
AES	IM32+S	IM32+S	IM32+S	IM32+S+ID	IM32+S+ID	IM32+S+ID	IM32+S+ID	IM32+S+ID	IM32+S+ID
JPEG	IM32+S	IM32+S	IM32+S	IM64	IM64	IM64	IM64	IM64 +ID	IM64 +ID
DFADD	S	S	S+ID	S	S	S+ID	S	S	S+ID
DFDIV	S+IM32	S+ID	S+ID	S+IM32	IM64	IM64	S+IM32	IM64+ID	IM64+ID
DFMUL	S+IM32	S	S+ID	IM64	IM64	IM64	IM64	IM64	IM64
DFSIN	S+IM32	S+IM32	S+IM32	S+IM32	IM64	IM64	S+IM32	IM64+ID	IM64+ID
FLOAT_HW1	S+IM32	BP	BP	S+IM32	BP	BP	FP2+S	FP2	FP2+S
FLOAT_HW2	S+IM32	S+IM32	S+IM32	S+IM32	IM64	IM64	FP2+IM32+S	IM64+FP+ID	IM64+FP+ID

E* = Exhaustive, 1-G* = 1-Greedy, 2-G* = 2-Greedy

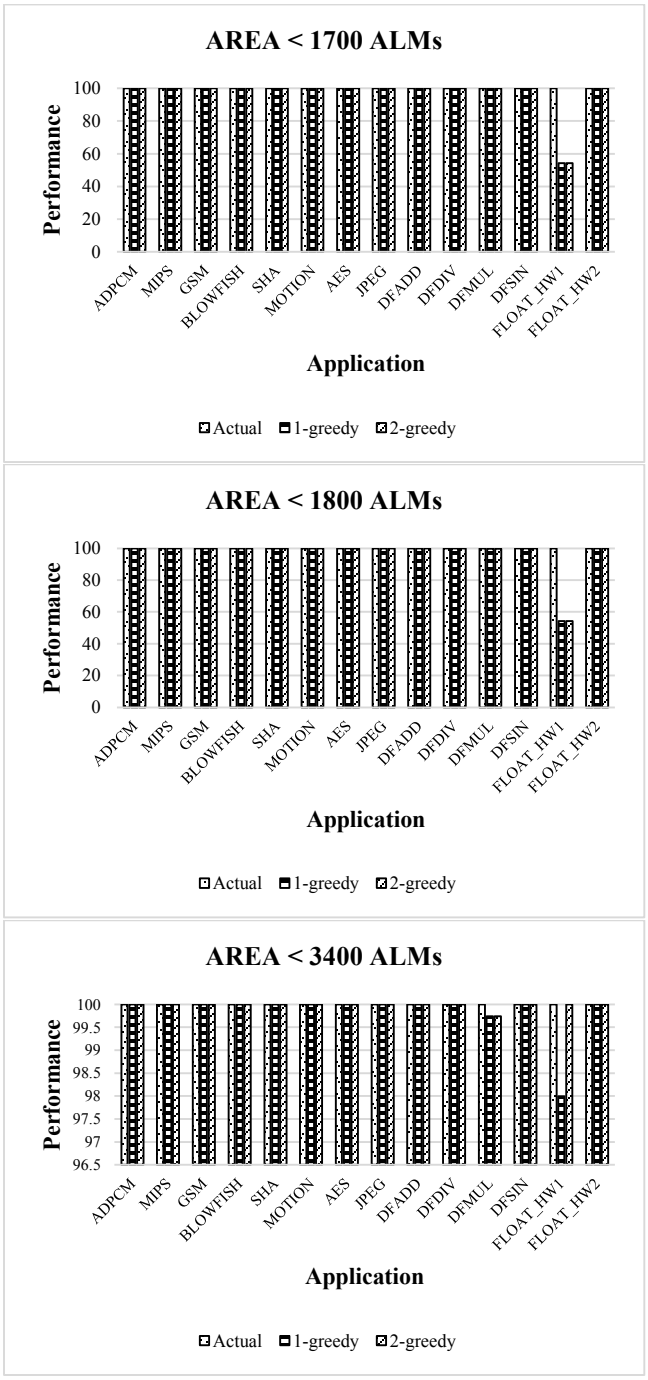


Figure 6. Comparison of accuracy of estimated performance

The final goal of the proposed approach is to find the optimal processor configuration under given area constraints. We have used exhaustive implementation to obtain the performance and area values for verification of the proposed methodology. Results of the proposed methodology and those of exhaustive technique, given area constraints of 1700, 1800 and 3400 ALMs are presented in Table 6. The area constraints have been selected such that 1700, 1800 and 3400 ALM could accommodate 1 FU, 2-3 FUs and all FUs respectively. This represents a balanced distribution of the design space.

Table 7. Percentage reduction in area compared to enabling all configurations

Application	% Area Reduction	Application	% Area Reduction
ADPCM	49.516	JPEG	49.516
MIPS	59.25	DFADD	59.25
GSM	49.516	DFDIV	49.516
BLOWFISH	59.25	DFMUL	49.516
SHA	59.25	DFSIN	49.516
MOTION	59.25	FLOAT_HW1	16.868
AES	49.516	FLOAT_HW2	6.469

The results in phase 4 prove that the 1-greedy search heuristic produces accurate results in most scenarios. The results are accurate for MIPS, BLOWFISH, DFADD, MOTION and AES. The datasheets indicate that the 32-bit Integer Multiplier does not support 64-bit multiplication instructions. However, the experimental results do not confirm this fact. The exhaustive approach indicates that the impact of the Integer Divider is negligible in most cases. However, the datasheet indicates a significant gain. Further investigation revealed that the gain is dependent on the bit width of the operands. We attribute the deviation of results in Table 6 to this unexpected behavior of the 32-bit Integer Multiplier and Integer Divider FUs. Changing the methodology to consider these two facts yields nearly 100% optimal results as shown in Figure 6. The instruction profile of *FLOAT_HW1* does not map to any of the integer instructions indicated in the datasheets. However, exhaustive results reveal that performance improves for some integer configurations. Further exploration with different handwritten codes provided similar results. Even though it is not mentioned in the datasheets this indicates that Floating Point Hardware 2 unit utilizes some functionality of the integer units if these FUs are configured. Thus, the presence of these FUs inevitably improves performance for most applications but this will be at the cost of area.

2-greedy heuristic produces sub optimal results in most scenarios for the test applications. This is due to the fact that the algorithm initially checks for combinations of two configurations in selecting the best configuration. Several applications do not utilize all FUs. Thus, 2-greedy approach produces poor results. However, when multiple FUs are utilized by the algorithm the 2-greedy method also produces accurate results. Thus, we expect the accuracy of the 2-greedy approach to increase with the complexity of the algorithm. *FLOAT_HW2* justifies our understanding.

The ALM constraint of 3400 could accommodate a processor with all configurations enabled. The proposed methodology provides maximum, median and average area reductions of 59.25%, 49.52% and 47.58% respectively, compared to a design where all configurable options are enabled. Table 7 depicts the reduction in area for each application compared to the configuration that has all configurable options enabled.

Another factor we need to consider with the exhaustive estimation approach in [19] is the time complexity of the algorithm. Considering a scenario with n possible configurable methods, using dependency analysis we eliminate several configurations which effectively yields m configurations ($m << n$). Our approach has a time complexity of $O(m)$ while exhaustive estimation approach claims $O(2^n)$.

The 1-greedy and 2-greedy approaches have computation times of 204 µs and 224 µs respectively for the *Nios II* processor under

consideration. This is a significant improvement compared to the computation time of 512 s for the exhaustive estimation approach discussed in [19]. Further, when the number of configurable units increase the runtime of exhaustive estimation increases considerably as each addition of a configurable unit doubles the runtime. For the proposed algorithm runtime analysis reveals that the overhead is less than 20% for the *Nios II fast* processor. For example, a processor like *LEON* which has $5*2^{10}$ configurations, will require a run time of almost 3 hours in the exhaustive approach whereas the proposed method would take less than a second. Thus, considering the significant gain in runtime and time complexity the proposed approach is superior to the exhaustive estimation approach.

5. CONCLUSION

In this paper, we proposed a rapid design space pruning technique for soft-core processor customization under user specified area constraint by exploiting dependencies between the various configuration options. The proposed methodology provides maximum, median and average area reductions of 59.25%, 49.52% and 47.58% respectively compared to a processor core with all configuration options enabled. The runtime complexity of our approach is linear, which yields better results when the design space becomes prohibitively large. The 1-greedy and 2-greedy heuristics used in the proposed methodology have computation times of 204 µs and 224 µs compared to 512 s in an existing exhaustive estimation approach. Lastly, the proposed methodology is generic and therefore, could be applied to any soft-core processor with similar configurable functional units on an FPGA.

6. REFERENCES

[1] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. 2014. Soft vector processors with streaming pipelines. In Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays (FPGA '14). ACM, New York, NY, USA, 117-126.

[2] Charles Eric LaForest and John Gregory Steffan. 2012. OCTAVO: an FPGA-centric processor family. In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '12). ACM, New York, NY, USA, 219-228.

[3] Cheah Hui Yan, Suhaib Fahmy, and Nachiket Kapre. 2015. On Data Forwarding in Deeply Pipelined Soft Processors. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15). ACM, New York, NY, USA, 181-189.

[4] Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G.F. Lemieux. 2011. VEGAS: soft vector processor with scratchpad memory. In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11)

[5] Dougherty, W.E., Pursley, D.J., and Thomas. D.E., 1998. Instruction subsetting: Trading power for programmability. In Proceedings of the International Conference on VLSI Technology and Circuits (VLSI '98), pages 42-47, 1998.

[6] Hara, Y., Tomiyama, H., Honda, S.; Takada, H., and Ishii, K., 2008. CHStone: A benchmark program suite for practical C-based high-level synthesis. In Proceedings of the International Symposium on Circuits and Systems (ISCAS '08), pages 1192-1195, May 2008.

[7] Hui Yan Cheah, Fredrik Brosser, Suhaib A. Fahmy, and Douglas L. Maskell. 2014. The iDEA DSP Block-Based Soft Processor for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 7, 3, Article 19 (September 2014), 23 pages.

[8] INTEL CORPORATION. 2015. Nios II Floating Point Hardware 2 Component User Guide. https://www.altera.com/content/dam/altera-www/global/en_US/others/literature/ug/ug_fph2.pdf

[9] INTEL CORPORATION. 2015. Nios II Gen2 Processor Reference Guide. https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf

[10] INTEL CORPORATION. 2016. Altera Design Software. https://www.altera.com/products/design-software/overview.html

[11] INTEL CORPORATION. 2016. CYCLONE V FPGAS & SOCS. https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html

[12] INTEL CORPORATION. 2016. Nios II Processor: The World's Most Versatile Embedded Processor. https://www.altera.com/products/processors/overview.html

[13] Lattner, C. and Adve, V. 2004. The LLVM Compiler Framework and Infrastructure. http://llvm.org/pubs/2004-09-22-LCPCLLVMTutorial.pdf

[14] Lewis, B. and Ramamoorthy, G. 2009. Market Trends: ASIC Design Starts. http://www.gartner.com/DisplayDocument?id=919712

[15] Madhura Purnaprajna and Paolo Ienne. 2012. Making wide-issue VLIW processors viable on FPGAs. ACM Trans. Archit. Code Optim. 8, 4, Article 33 (January 2012), 16 pages.

[16] MENTOR GRAPHICS CORPORATION. 2016. ModelSim ASIC and FPGA Design. https://www.mentor.com/products/fv/modelsim/

[17] Padmanabhan, S., Cytron, R.K., Chamberlain, R.D. and Lockwood, J.W., 2006. Automatic application-specific microarchitecture reconfiguration. In Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS '06), pages 25-29, Apr. 2006.

[18] Plessl, C. 2014. Static LLVM Compilation Tool flow. http://homepages.uni-paderborn.de/plessl/teaching/2014-Codesign/slides/02-Compiler-LLVM.pdf

[19] Prakash, A., Siew-Kei Lam, Singh, A.K., and Srikanthan, T., 2009. Rapid design exploration framework for application-aware customization of soft core processors. In Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '09), pages 539-542, Dec. 2009.

[20] Rajotte, S., Gil, D.C., and Langlois, J.M.P., 2011. Combining ISA extensions and subsetting for improved ASIP performance and cost. In Proceedings of the International Symposium on Circuits and Systems (ISCAS '11), pages 653-656, May 2011.

[21] Sheldon, D., Kumar, R., Lysecky, R., Vahid, F., and Tullsen, D., 2006. Application-Specific Customization of Parameterized FPGA Soft-Core Processors. . In Proceedings

of the International Conference on Computer-Aided Design (ICCAD '06), pages 261-268, Nov. 2006.

[22] Sheldon, D., Vahid, F., and Lonardi. S., 2007. Soft-core Processor Customization using the Design of Experiments Paradigm. In Proceedings of the International Conference on Design, Automation & Test in Europe (DATE '07), pages 1-6, 2007.

[23] Shendi, R. (2015). CUSTOMIZATION OF A SOFT-CORE CPU ON AN FPGA. Master's Thesis. University of Manchester.

[24] SIGMAZONE CORPORATION. 2016. DOE PRO. http://www.sigmazone.com/doepro.htm

[25] Vakili, S., Langlois, J. and Bois, G. 2013. Customised soft processor design: a compromise between architecture description languages and parameterisable processors. IET Computers & Digital Techniques 7, pages 122-131, 2013.

[26] Vakili, S., Langlois, J. and Bois, G. 2016. Accuracy-aware processor customisation for fixed-point arithmetic. IET Computers & Digital Techniques 10, pages 1-11, Jan. 2016.

[27] XILINX CORPORATION. 2010. Virtex-4 family overview. http://www.xilinx.com/support/documentation/data sheets/ds112.pdf

[28] XILINX CORPORATION. 2011. Virtex-II pro and Virtex-II Pro X platform FPGAs data sheet. http://www.xilinx.com/support/documentation/data sheets/ds083.pdf

[29] XILINX CORPORATION. 2016. MicroBlaze Soft Processor Core. http://www.xilinx.com/products/design-tools/microblaze.html

[30] Yiannacouras, P., Steffan, J. and Rose, J. 2007. Exploration and Customization of FPGA-Based Soft Processors. In Transactions on Computer-Aided Design of Integrated Circuits and Systems (ICCAS '06), pages 266-277, Feb. 2007

[31] Yuichiroh Tanaka, Shimpei Sato, and Kenji Kise. 2014. The Ultrasmall soft processor. SIGARCH Comput. Archit. News 41, 5 (June 2014), 95-100.

[32] Zhiduo Liu, Aaron Severance, Satnam Singh, and Guy G.F. Lemieux. 2012. Accelerator compiler for the VENICE vector processor. In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '12)