UNIVERSITY OF CALIFORNIA
RIVERSIDE

Design Space Exploration of Parameterized Systems using Design of Experiments

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

David Russell Sheldon

December 2011

Dissertation Committee:

Dr. Frank Vahid, Chairperson
Dr. Tony Givargis
Dr. Stefano Lonardi

The Dissertation of David Russell Sheldon is approved:

_____

_____

_____

Committee Chairperson

University of California, Riverside

ABSTRACT OF THE DISSERTATION

Design Space Exploration of Parameterized Systems using Design of Experiments

by

David Russell Sheldon

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2011
Dr. Frank Vahid, Chairperson

Recent trends have led to parameterization of many computing components, such as parameterized processors, caches, FPGAs or networks–on-chip, as well as parameters in design tools such as optimization flags. Tuning parameterized systems to meet design goals like performance, energy, size, or power, has become harder due to the enormous design space created by such parameters and due to the large time required to evaluate each system configuration. Previous design space exploration approaches for parameterized systems have either focused on custom or randomized search heuristics. We map such design space exploration onto a statistical paradigm known as Design of Experiments, a paradigm under development since the 1920s that uses methodical experiment selection and sophisticated analysis to obtain maximum information using a minimum number of experiments. We introduce our DPG (Design-of-experiments Pareto-point Generator) method that performs flexible exploration by allowing the designer to provide information about the number and types of parameters, the approximate time to evaluate a configuration, and the total allowable exploration time.

From that information, DPG automatically determines a custom set of experiments to best explore the design space within the allowable time. Such customized design-of-experiments-based exploration represents the unique contribution of this work. We show that DPG provides competitive results across different domains, without requiring the designer to have a detailed understanding of parameter impacts. We created a web-based DPG tool to support designers from various domains, which accepts information from the designer and generates experiments that the designer conducts (iteratively), and generates data and plots from the analysis, including Pareto-points. The effectiveness of the DoE paradigm for system tuning may have broad applicability for design automation.

Table of Contents -2

Chapter 1 1
1.1 Parameterized caches 2
1.2 Parameterized processors 3
1.2.1 Tunable processors 4
1.2.2 Application specific instruction set processor (ASIP) 5
1.3 Parameterized system on a chip 7
1.4 Parameterized network on a chip 8
1.5 Tunable FPGA fabrics 10
1.6 Design tools 11
1.7 Outline for thesis 12
Chapter 2 13
2.1 Design space exploration for system specific tuning algorithms 13
2.1.1 Cache and memory 13
2.1.2 Processors 14
2.1.3 FPGA 15
2.1.4 System on a chip (SoC) 15
2.2 Single factor analysis (SFA) 17
2.3 Randomized tuning algorithms 18
Chapter 3 20
3.1 Background 20
3.2 Application specific tuning of parameterized systems 26
3.2.1 Traditional CAD approach: 0-1 knapsack 27
3.2.2 Synthesis-in-the-loop approach: impact ordered trees 29

3.3 Experiments 33
Chapter 4. 38
4.1 Single factor analysis 38
4.2 Parameter interdependency graph 42
4.3 Pareto point generation using design of experiments (DoE) 44
4.4 Pareto point generation using DPG 47
4.5 Experiments 52
Chapter 5 61
5.1 Challenges with invalid data points 61
5.2 Handling invalid points 63
5.3 Parameterized systems with invalid points 67
5.3.1 VPR 67
5.3.2 ISE 68
5.4 Results 71
5.4.1 VPR 71
5.4.2 Xilinx ISE 76
Chapter 6 80
6.1 DPG (DoE Pareto-point Generator) 80
6.1.1 2 level parameters – single metric system 82
6.1.2 2 level parameters – multi-metric system 85
6.1.3 Multi-level parameters – single metric system 86
6.1.4 Multi-level parameters – multi-metric system 86
6.2 Results 87
6.2.1 Network on chip (NoC) 87
6.2.2 FPGA tuning 89
6.2.3 System exploration 90
Chapter 7 95
References 97

List of Figures -2

Figure 1: Albonesi's [Albonesi 2002] 4 way set associative cache with selective cache ways. Data ways 1-3 are identical to way 0. The cache controller is able to control which cache ways are active. 3

Figure 2: The architecture of the eMIPS platform. The custom instruction units form a parallel pipeline to allow for dynamic nature of the ISA. [Pittman 2006] 6

Figure 3: The architecture of the PICO framework with the systolic array as a coprocessor. [Schreiber 2000] 8

Figure 4: Shin's configurable communication bus and the configurable parameters. [Shin 2004] 8

Figure 5: Noxim NoC design space with over 60,000 configurations. 9

Figure 6: Single Factor Example for aifir benchmark. Base configuration is in bold. . . 17

Figure 7: Equations for calculating Equivalent LUT value of a configured Microblaze. 22

Figure 8: Equivalent LUT values for hard-core units. 23

Figure 9: Size versus application runtime for all Microblaze configurations executing the aifir EEMBC benchmark, with all Pareto points labeled. An additional labeled point (FPU) is highlighted to show the performance overhead of instantiating an underutilized component, due to lengthening of the clock cycle. 24

Figure 10: Speedups for base (Base Microblaze), full (Full Microblaze), and optimal (Optimal Microblaze) Microblaze configurations. 25

Figure 11: Average pairwise speedup-increment additive inaccuracies for all pairs of benchmarks. 28
Figure 12: Speedup increment, size increment, and their ratio, for each MB component for the aifir benchmark. 29
Figure 13: Impact-ordered tree approach: (a) Application-specific impact-ordered tree for the aifir benchmark, (b) Fixed impact-ordered tree. Note that neither approach actually generates the entire tree – both make a single descent to a leaf node. The thick red line shows a single decent through the tree. 31
Figure 14: Speedup increment, size increment, and their ratio, for each MB component averaged across all 12 benchmarks. 32
Figure 15: Average speedups obtained by the various exploration approaches, for: (a) no size constraint, (b) a fixed size constraint set at 80% of the size of a full MB, (c) a per-application-tailored size constraint of 80% of the size of the optimal MB for that application (as determined in (a)), all on a Virtex-II Pro device. 34
Figure 16: Average speedups for the approaches on a Spartan2 FPGA. 35
Figure 17: Estimated tool run times for increasing number of configurable soft-core processor options. 36
Figure 18: Illustration of results of SFA extended for Pareto points, showing dependency on the base configuration selection: (a) A 3-parameter system where 2 parameters are interdependent and the third is independent. The filled points are the points examined by SFA. The circled points are Pareto points found. (b) A different base configuration yields different Pareto points. 40

Figure 19: SFA with three dependent parameters. If all three parameters are interdependent, then single factor analysis breaks down further. 41

Figure 20: Platune's parameter interdependency graph. The arrows indicate dependencies between the different factors. [Givargis 2002] 42

Figure 21: DoE-based Pareto-point Generator (DPG) 48

Figure 22: Parameter interdependencies found by DPG. The major interdependencies have been overlaid over the original Platune parameter interdependency graph – thicker lines indicate larger dependencies. Parameters not in our simulations have been crossed out. 50

Figure 23: Platune's configurable architecture. 52

Figure 24: Pareto curves for JPEG. SFA was able to find the valid endpoints in this example. DoE IOT only found one of the needed endpoints. DPG was initially able to find good points on either side of the main part of the Pareto curve, and DPG's fill-in process finds the remaining points. 53

Figure 25: Runtime comparison for JPEG with different algorithms. 54

Figure 26: Pareto points from Platune versus DPG (with fill-in) for b1_histogram. 55

Figure 27: Pareto points for Platune vs. DPG (with fill-in) for g3fax. 56

Figure 28: NoC Pareto graphs, (a) shows a single application. Both DPG and SFA find the Pareto curve. (b) has two applications and DPG found a shared set of Pareto points. SFA found only one configuration, which is circled below. 58

Figure 29: DPG results for Microblaze configuration. Four benchmarks, aifir, BaseFP01, canrdr, and engine are shown. These benchmarks are typical of the results seen from the

DPG algorithm. DPG finds the complete set of Pareto points in all but the BaseFP01 example, where it misses the point near 50,000,000. Many data points are extremely close in value, when this occurs we added vertical spacing to additional points which creates the vertical bards seen in the Exhaustive graphs. 59
Figure 30: Percent of the design space that needs to be explored in the analysis phase of DPG. 60
Figure 31: DoE flow to handle invalid configurations. 64
Figure 32: Plackett-Burman screening runs. The top table labeled Normal is the beginning of the standard Plackett-Burman set of runs for 11 parameters. The bottom table shows the reverse table, which we use to help handle invalid points. 65
Figure 33: VPR - Table showing experiments and results for two circuits. INV means that the configuration was invalid. 66
Figure 34: VPR parameters and values used to generate the designs. Two parameters, fast and router_algorithm, only have a low and high setting. Inputs-per-cluster is a special case since the value is dependent on the values of two other parameters, Cluster size (A) and LUT size (B). 68
Figure 35: Parameters used in Xilinx ISE experiments 69
Figure 36: VPR experiment—Determined critical path impacts (in seconds) from a screening test on the dsip circuit for each parameter's low and high settings, as determined by: (a) single-factor analysis, (b) DoE. Note the large differences in the determined impacts between the two techniques. 72

Figure 37: VPR experiment—Determined critical path impacts for dsip as determined by single-factor analysis, this time with a base configuration using low values for all parameters rather than high values. Note the dramatic difference from Figure 36(a); single-factor analysis is very sensitive to the base configuration. 73
Figure 38: VPR experiment—Critical path delays of the tuned FPGA for the 19 circuits. 74
Figure 39: VPR experiment—Determined area impact (measured in terms of the feature size of the transistors) on the dsip circuit for each parameter's low and high settings as determined by: (a) single factor analysis, (b) a DoE screening test. 75
Figure 40: VPR experiment—Tool runtimes for application-specific tuning of an FPGA platform for the 19 circuits. 76
Figure 41: Xilinx ISE experiment—The top 10 parameters from an interrupt handler, b06. 77
Figure 42: Xilinx ISE experiment—Critical paths delays for the circuits. 78
Figure 43: DoE tool flow for architecture tuning. 80
Figure 44: DPG flow using design space exploration. A dashed line marks decision points. The bold stages mark the stages affected by the type of system. 83
Figure 45: Example of the initial screening analysis using DPG Plackett-Burman experiments. 84
Figure 46: NoC results for a synthetic application 1. 88
Figure 47: NoC results for a synthetic application 4. 89
Figure 48: Results from VPR for a single metric design. 90

Figure 49: Results from the g3fax Platune example. 92

Figure 50: Results from the jpeg Platune example. 93

Chapter 1

Introduction

Parameterized systems have become increasingly common during the past decade. For example, cache architectures may have configurable size, associativity, and line size parameters, either in hard-core (physical) form [Albonesi 2002][Scott 1999][Zhang 2004] or in soft-core (synthesizable) form [Altera 2011][Arm 2011][Tensilica 2011]. Soft-core microprocessors, such as cores targeted for field-programmable gate arrays (FPGAs) [Albonesi 2002][Xilinx 2011] or for application-specific integrated circuits, (ASICs) [Arm 2011][Tensilica 2011] may have optional data path units. Data path units consist of units such as floating-point or divider units, memory units or configurable pipeline lengths, and so on. System-on-chip platforms may have tunable parameters relating to processors, memories, and buses [Givargis 2002][Kumar 2004][Mohanty 2002][Palermo 2003][Sekar 2003][Sherwood 2004][Szymanek 2004]. The communication networks on many chips are becoming more complex and tools like Noxim [Fazzino 2008] have been developed to help designers build the custom networks on the chips. Generators of

customized field programmable gate array (FPGA) fabrics which are useful for adding FPGAs onto portions of an ASIC for circuits likely to change, have parameters that can be tuned to a particular circuit or circuits [Ahmed 2001].

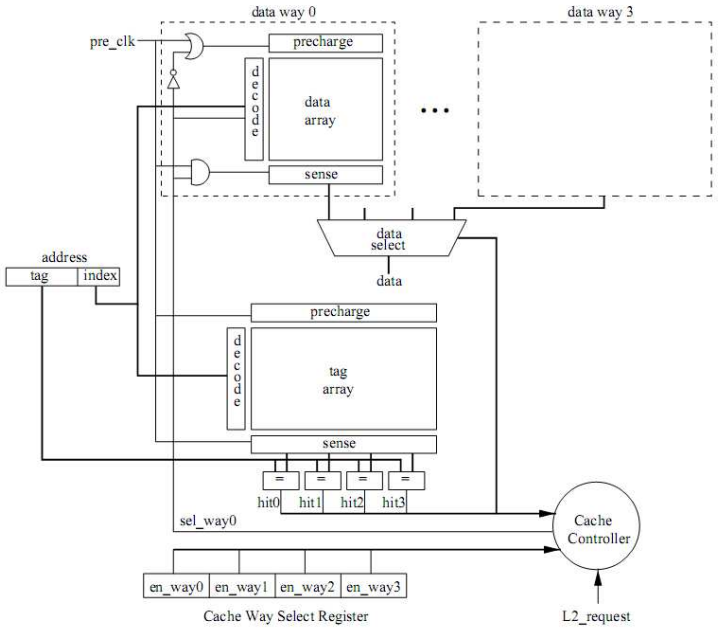
Tuning a parameterized system is a complex problem. The difficulty stems largely from both the very large configuration space, and the long runtime that may be required to evaluate each configuration. A configuration is a particular set of values for all parameters; e.g., a particular configuration of a configurable cache may be a 4 Kbyte, 4-way set-associative, or a 32-byte line size cache. Configuration spaces have exponential size complexity (N parameters with M values yields M^N configurations) with typical values being in the thousands or higher. For some systems the size of the configuration spaces can grow to over $1x10^{24}$ configurations. Evaluating just one configuration may require synthesis and/or simulation, where runtimes may measure in the minutes or hours.

There are many different domains that make use of parameterized systems. The following sections describe various types of parameterized components and systems such as caches, processors, SoC, NoC and custom FPGA fabrics.

1.1 Parameterized caches

Albonesi [Albonesi 2002] proposed a new cache architecture where part of the cache can be disabled when the cache is not under heavy load. By shutting down a portion of the cache the power and energy that the cache uses decreases. The cache will only shutdown at times when the load on the cache is low so the overall impact on the performance of the system is relatively small. Albonesi used a technique called selective cache ways to shutdown a portion of the cache. Selective cache ways are able to shutdown one or more of the ways in the cache to save power. When more cache capacity is required based on

Figure 1: Albonesi's [Albonesi 2002] 4 way set associative cache with selective cache ways. Data ways 1-3 are identical to way 0. The cache controller is able to control which cache ways are active



the workload, the ways are reactivated. Figure 1 shows how the selective way shutdown works. The cache controller is able to disable the clock to each way which effectively removes that way from the cache. When the cache controller enables the clock, the way would then resume normal operation within the cache.

Zhang [Zhang 2004] created a cache with even more configurability. Zhang's cache has three configurable parameters: the total cache size, associativity, and the line size. Zhang's cache is able to change any one of the three parameters during runtime such that cache flushes are kept to a minimum after a change occurs. The increased level of configurability allows for greater control and customization of the cache during runtime which leads to an increase in energy savings.

Scott [Scott 1999] developed the M*Core cache architecture. The M*Core cache is designed to be a unified cache. The M*Core cache is divided into different banks, where each bank is configurable. The banks are then set in one of four configurations:

off, instruction only, data only, or unified. When a bank is set as instruction only, the only lines that will be stored in that bank corresponds to requests from the instruction L1 cache, the same is true for the data only. The unified configuration allows both instructions and data to be stored within the bank. The banks can be turned off to reduce power consumption when the extra space is not needed.

1.2 Parameterized processors

Parameterized processors allow designers the ability to tune the processors to the needed tasks. By tuning a general purpose processor, the designer is able to achieve better performance while still maintaining the benefits of a general purpose processor. There are two main types of parameterized processors; processors that have a limited set of units that can be added or removed, and processors that can have custom units added to the processor.

1.2.1 Tunable processors

Tunable processors are distributed with a set of configurable units that can be added or removed depending on the needs of the designer. The tuning of the processors can be done in hardware by physically adding or removing the units from the processor or by software where units are disabled when not needed. The SimpleScalar tool [Burger 2011] is an example of one such processor.

The Microblaze processor [Xilinx 2011], created by Xilinx, is one example of a tunable processor. The Microblaze core has multiple functional units that the designer can add or remove from the Microblaze core. The functional units that are available within the Microblaze core are a barrel shifter, multiplier, floating point unit, divider, a comparator, and a custom instruction to access special registers. When any of these units

is not present, the Microblaze compiler uses software to replace the functionality of the missing functional units.

Arm [Arm 2011] produces multiple different processors, however, most of the processors have the same core processor and have multiple different preset configurations. The Cortex-M0 is an example of the basic Cortex processor. While the M4 version adds MAC units, a floating point unit, memory protection as well as a variety of other options. The Cortex uses a number of different fixed configurations which still gives designers the freedom to find the features that best match the needs of the applications.

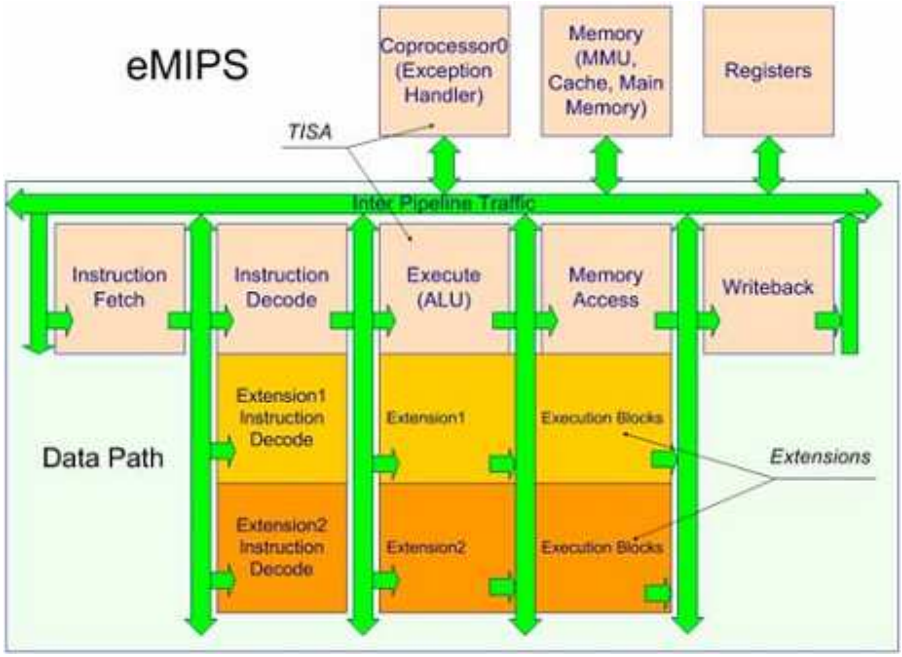
Sekar [Sekar 2003] introduced a dynamic method to modify a parameterized processor. Sekar used a software based technique similar to the approach used by Albonesi, and Zhang in cache configuration. By setting registers, components of the processor can be enabled or disabled. Also, clock generators can be modified to change the clock frequency of the processor.

1.2.2 Application specific instruction set processor (ASIP)

ASIPs are processors where the instruction set has been modified to better meet the needs of the domain of applications that will be run on the processor.

Tensilica [Tensilica 2005] developed a parameterized processor, called the Xtensa, that includes different types of parameters. The Xtensa like the previous tunable processors has many different components that can be enabled to create a tuned processor. The Xtensa can be configured to include functional units like, multipliers, or floating point units. Also, different types of caches or pipeline architectures can be configured. The Xtensa tool chain allows for designers to add a variety of custom

Figure 2: The architecture of the eMIPS platform. The custom instruction units form a parallel pipeline to allow for dynamic nature of the ISA. [Pittman 2006]



instructions to the processor. The custom instructions can be complex multi-cycle instructions. Custom instructions can also have embedded registers or an embedded register file.

Tensilica's Xtenesa processor is a hardcore processor that is designed for an ASIC chip. Altera has created the NIOS soft core processor [Altera 2011], which is designed to be used on an FPGA. The NIOS, like the Xtenesa processor, has a base instruction set and additional instructions can be added to the ISA. The custom instruction slots in the NIOS are, however, more limited in size. The area available for the NIOS custom instructions is determined by the layout of the processor, which provides a limited amount of space to for the custom instructions. While the custom instructions may be multi-cycle there for instance is not sufficient area for a custom register file to be embedded within the custom instruction.

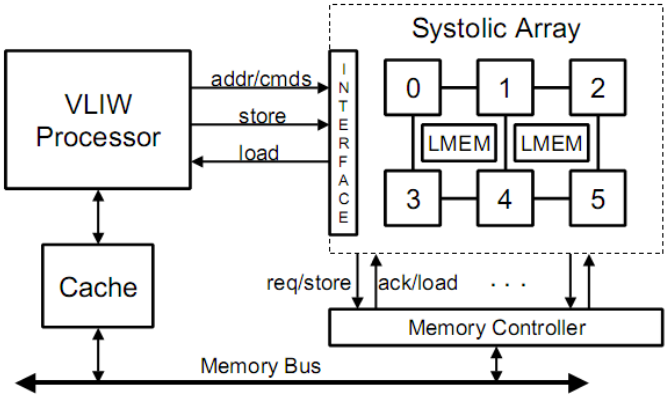
The eMIPS core has a modified datapath that allows for custom instructions to be added to the ISA at runtime. eMIPS uses an FPGA fabric to implement the custom instructions within the datapath. Figure 2 shows the architecture of the eMIPS platform. The dynamic nature of the custom instructions means that the normal pipeline has no knowledge of the custom instructions. Unlike other approaches where the custom instructions are static, the dynamic nature of the instructions means that a new decode block is needed to decode the custom instruction. In the eMIPS architecture, each new instruction is passed to all the decode blocks and the block that understands that instruction will decode the instruction.

1.3 Parameterized systems on a chip (SoC)

Givargis [Givargis 2002] introduced a parameterized SoC called Platune. Platune consists of a processor and two levels of caches. The caches and the busses that connect the caches to each other and main memory are configurable as well as the voltage of the system. There are over a billion different configurations within the Platune SoC system.

Schreiber [Schreiber 2000] developed a processor generator called PICO. PICO generates a custom VLIW processor that has a fabric that is used to implement systolic arrays to aid the processor. PICO uses the application as a base and designs the processor and one or more systolic arrays to accelerate the computations in the processor. Figure 3 shows the high level architecture of the PICO designed chip. The VLIW processor communicates with one or more systolic arrays through a memory interface. Each systolic array has access to the memory bus to increase the memory bandwidth to the systolic arrays.

Figure 3: The architecture of the PICO framework with the systolic array as a coprocessor. [Schreiber 2000]



Shin [Shin 2004] introduced a configurable communication bus for SoC systems, shown in Figure 4. The communication bus consists of two main components; the bus and the memory scheduler. The bus controls the time-slice that is allocated to each bus master and the overall pipeline latency. Some of the parameters of the memory controller include the bandwidth per thread, prefetch limit and change limit. The number of configuration in the configuration space is over 100 billion configurations.

1.4 Parameterized network on a chip (NoC)

Dall'Osso [Dall'Osso 2003] developed a communication framework called xpipes. Xpipes are a scalable and high performance method for connecting components within

Figure 4: Shin's configurable communication bus and the configurable parameters. [Shin 2004]

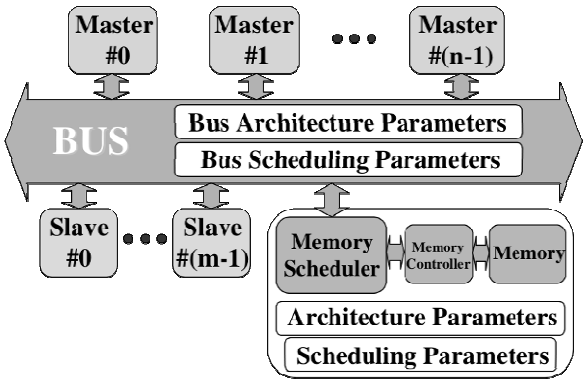
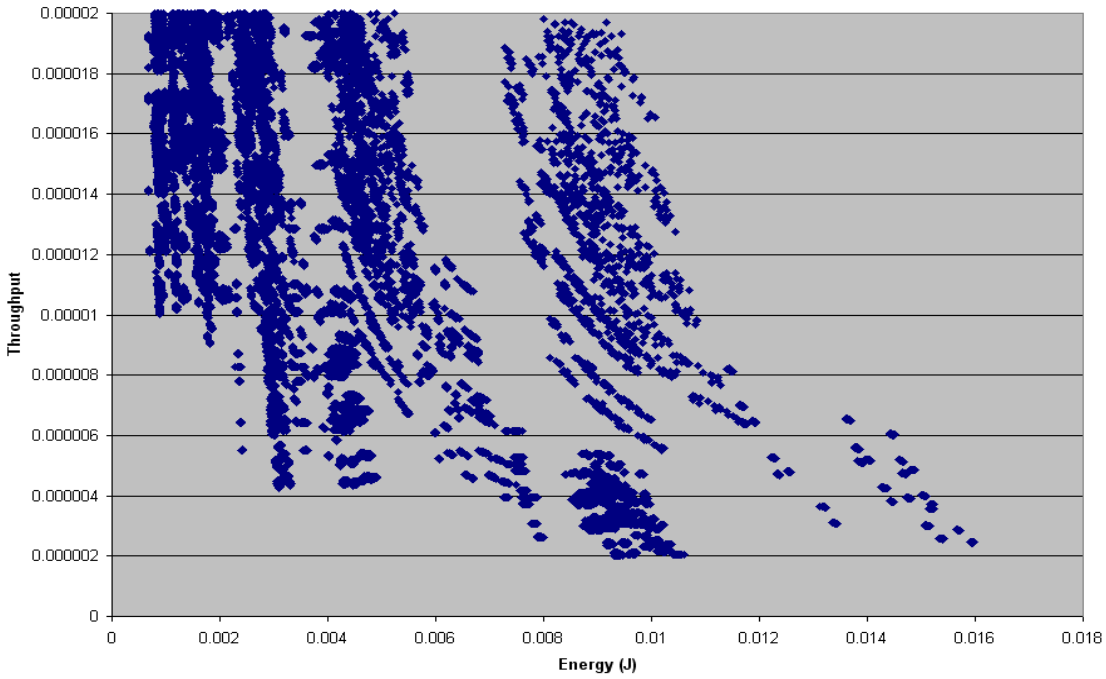


Figure 5: Noxim NoC design space with over 60,000 configurations.



the SoC. The configurability of xpipes allow for both homogeneous and heterogeneous architectures. Xpipes utilize a wormhole switching technique to minimize the area and power usage at the switches. The switches in the xpipes architecture are configurable, where the designer can control parameters such as the number of input and output ports, the channel width, the number of virtual channels, and the size of the buffers.

Fazzino [Fazzino 2008] developed a network on a chip simulator where a designer can model the on chip network. The model includes both the software traffic and the hardware properties such as the routing algorithms and channel widths. The Noxim simulator allows the designer to evaluate a wide range of different NoC configurations to determine the performance and energy usage of the on chip network. The configuration space of the Noxim simulator is shown in Figure 5. The figure shows the design space with the energy on the X-axis and the combined throughput of all the channels within the

NoC on the Y-axis. The NoC design space that we generated from Noxim [Fazzino 2008] consists of over 60,000 configurations.

1.5 Tuned FPGA fabrics

Hauck [Hauck 2006] introduced the Totem FPGA fabric generator. Totem generates a custom FPGA fabric for a domain of applications. Totem aids the designers in mapping the FPGA fabric onto silicon and in the creation of the synthesis tools needed to map applications onto the custom FPGA fabric. An advantage of domain specific fabrics is the fabric can include the type of additional resources that are likely to be needed by the domain. For example, in a fabric designed for signal processing a selection of hardcore units such as multiply accumulator (MAC) or a fast fourier transforms (FFT) could be included within the fabric itself. In addition, the cluster size of the LUTs and the size and length of the routing channels can vary considerably from one domain to another. Hauck found that by using domain specific FPGA fabrics the designer could see a 4.8x improvement in area-delay over traditional FPGA fabrics.

Betz [Betz 2000] created the VPR and the T-VPACK tools that assist designers in creating custom FPGA fabrics. VPR is a tool that allows designers to create and use array based FPGA fabrics, while T-VPACK is a clustering tool to pack the application in to blocks. VPR can create fabrics for a wide range of different FPGA fabrics. VPR allows designers to vary parameters from the size of the LUTS in the fabric including multiple sizes of LUTs within a single fabric, the number and length of the routing channels, as well as specialized hardcore units embedded within the FPGA fabric. The hardcore units could be multipliers, DSP block, or whole processors that are connected to the FPGA

fabric. The level of customization that VPR offers allows designers to generate and evaluate how the applications will run on the custom FPGA fabric.

1.6 Design tools

In addition to the customization of the hardware systems, the software tools that designers use have many parameters that can have an impact on the overall performance of the system. For example, synthesis tools [Xilinx 2011][Altera 2011][Tensilica 2011][Synopsys 2011][Cadence 2011][Mentor 2011] used to map circuits onto ASIC or FPGA platforms go through multiple stages such as synthesis, mapping, placement and routing. Each of these stages consists of NP-complete problems that need to be solved, which is why the tools use heuristics to complete these stages. Each of the stages consists of dozens and in some cases over one hundred parameters to control how the heuristic functions complete the stage. The value of the parameters used to generate the final circuit has a large impact on the performance of the final system.

Similarly to the hardware, compilers used to compile software have a large number of parameters. Compilers such as gcc [gcc 2011] have hundreds of different options to help optimize the code. Some of the gcc default levels that work well for a wide variety of applications are -01 and -03. The default options may not provide a sufficient level of optimization for the system. When additional optimization is needed the designer would then have to tune the optimization settings for the system, using the wide range of optimization options available within the compiler. Other compilers such as LLVM [LLVM 2011] and Intel's ICC [Intel 2011] have a very large number of options that would probably need to be configured to maximize the size and performance of the code.

1.7 Outline for Thesis

In chapter 2 we will discuss the related work in areas of parameterized systems, and algorithms used to navigate the design spaces. Chapter 3 describes an application specific method for design space exploration. The approach uses the characteristics of both the application and the system for the design space exploration. In chapter 4 we will discuss how to use application specific methods to generate the Pareto points for a given system. Chapter 5 discusses how a designer can quickly search a design space even if there are invalid configurations within the design space. Chapter 6 presents an algorithm to guide the designer through the design space exploration process. The algorithm adapts to both the time the designer has and to the properties of the individual system and application. Much of the work in this thesis is described in a series of papers [Sheldon 2006 (a)] [Sheldon 2006 (b)] [Sheldon 2007] [Sheldon 2009].

Chapter 2

Related Work

2.1 Design space exploration for system specific tuning algorithms

A common method to tune a parameterized system is for the designer to create a new system specific algorithm for each system the designer needs to tune. The designer needs to determine the metrics that are relevant and how best to guide the exploration process. System specific exploration methods often produce good results for the system, however, the system specific algorithms are useful for only the intended system. There are many different types of parameterized systems, such as caches and memory, processors, FPGAs, and System on a Chip (SoC), which are discussed in more detail below.

2.1.1 Cache and memory

Researchers have developed system specific tuning algorithms for tuning a parameterized memory hierarchy. For example, Zhang [Zhang 2004] developed a configurable cache, which has three configurable parameters: size, associativity, and line size. Zhang's cache is a full hardware layout such that the cache is fully configurable at runtime. Zhang developed an algorithm that can vary the three parameters based on the current usage of the cache. Gordon-Ross used an M-Core [Scott 1999] [Malik 2000] second level cache

and Zhang's cache for the first level cache for both the instruction and data caches. Gordon-Ross's algorithm was designed to tune both cache levels to decrease the total energy based on the current workload of the system.

Viana [Viana 2003] developed an algorithm that allowed system designers to explore cache configurations while the system is in development. Viana extended the ArchC [Rigo 2004] framework to include the ability to explore the effect of various cache configurations on the system. Expanding on the cache configuration, Szymanek [Szymanek 2004] explored the entire memory hierarchy for a given system. Szymanek generated the Pareto points for the memory hierarchy by modeling the memory hierarchy and the application.

2.1.2 Processors

Processors are another domain where designers tune the system using system specific algorithms. Mishra [Mishra 2001] created a method to explore processor architectures using Architecture Description Language (ADL). Mishra created a set of reusable models so that a single model may be used in multiple ADLs. Sherwood [Sherwood 2004] created Sherpa which quickly explores the processor configurations. Sherwood modeled the various components of the processor. The models contain information on how each variation affected the performance and area of the processor. Sherwood then used an integer linear programming (ILP) solver to explore the design space.

Yiannacouras [Yiannacouras 2005][Yiannacouras 2006] added custom instructions to an application specific instruction set processor (ASIP) to improve the performance. Yiannacouras developed a tool that analyzes the application to determine the set of commonly repeated instructions. The tool then adds the custom instructions to

the instruction set architecture (ISA) and the tool updates the compiler for the new ISA. The compiler can then be used to compile the code for the new ASIP processor. ASIP processors have been explored by many in the field. Most approaches use a compiler aided approach [Atasu 2003] [Cheung 2003] [Clark 2005] [Fin 2001] [Gupta] [Wang 2001]. Bauer [Bauer 2008] modifies the instruction set at runtime.

2.1.3 FPGA

Researchers have developed tools and algorithms to help designers create FPGA fabrics tuned to the application or set of applications that run on a custom FPGA fabric. Hauck [Hauck 2006] developed the Totem tool. Totem used similar techniques to those used in high-level synthesis for ASICs. The FPGA fabrics produced by Totem are up to eleven times better in terms of the area-delay product, showing that a custom FPGA fabric can have a large impact on the final system. Hammerquist [Hammerquist 2008] developed an algorithm to design system specific FPGA fabrics. The algorithm varied parameters such as the lookup tables (LUT) and configurable logic block (CLB) size as well as routing resources like the channel width to create a custom fabric for the application. Hammerquist found that by tuning the system the overall energy usage of the FPGA can drop thirty-five percent on average when compared to a traditional FPGA fabric.

2.1.4 System on a chip (SoC)

System on a chip designs are becoming much more common in recent years which is increasing the need for designers to create tuned SoCs. Platune, developed by Givargis [Givargis 2002], was developed to allows the designer to quickly explore the design space of the memory hierarchy of the SoC. Platune does the exploration by using an interdependency graph that shows which parameters in the system are related to each

other and which can be tuned independently. By dividing the large design space into several smaller problems the total time to explore the design space decreases.

The Metropolis framework [Balarin 2003] was extended by Densmore [Densmore 2006] to help the designer to quickly analyze a large number of design choices, such as hardware / software partitioning, or the tradeoff space between area and frequency for parts of the overall system. Bossuet developed an algorithm that explores both the application and the underlying platform to optimize the overall system. The algorithm is able to analyze the different components of the application to determine the resources needed for each part of the application [Bossuet 2003]. In addition to determining the needs of the application, the algorithm examines many different architectures [Bossuet 2007]. The hardware and software models are used to determine the level of parallelism in the system, the communication structure, and the functional unit sharing. Adding the application to the design space exploration opens up new solutions while increasing the overall size of the design space.

The communication network was examined by Larihi [Larihi 2004] to tune the different components within the communication system. Larihi developed an algorithm that examined the communication structure of the application and explored custom communication networks to improve performance. Larihi showed that by adding more buses and carefully partitioning the communications across the different buses improved the performance of the overall system performance.

Kumar [Kumar 2004][Kumar 2006] presented a different approach to system on a chip exploration. Kumar proposed a system where instead of finding the best configuration at design time, the designer creates a chip with many different

heterogeneous cores. Each core can be tuned for a particular task or general purpose core that vary in performance and energy. Kumar's technique allows the application that is running the ability to switch processors during execution depending on the needs of the application. For example, if the application was doing a lot of input/output (I/O), then the slowest processor that could keep up with I/O devices would save the most power, while heavy computation would benefit from a faster processor and the time savings could provide saving in the total system power.

2.2 Single factor analysis (SFA)

Single factor analysis is another method that is used to tune systems. Single factor analysis is a simple approach that allows a designer to quickly determine the impact of each of the parameters in the system [Montgomery 1986]. Single factor analysis begins with a base configuration. The base configuration is the starting point for all future analysis of the system. The designer then varies each parameter individually. The result from each run is compared to the base configuration. The difference is the effect that the parameter has on the system, as seen in Figure 6. Figure 6 is an example of a single factor analysis experiment. The first row is the base case; all parameters are set to 0 or turned off. Then in each of the following rows one of the parameters is turned on and the effect

Figure 6: Single Factor Example for aifir benchmark. Base configuration is in bold.

BS	MUL	FPU	DIV	MSR	PCMP	Cycles	Effect
0	**0**	**0**	**0**	**0**	**0**	**12,696,265**	
1	0	0	0	0	0	9,905,565	2,790,700
0	1	0	0	0	0	12,696,265	0
0	0	1	0	0	0	10,818,171	1,878,094
0	0	0	1	0	0	12,696,265	0
0	0	0	0	1	0	12,696,265	0
0	0	0	0	0	1	12,696,265	0

on the system is recorded in the cycles column. The effect column shows the difference or the effect of turning on the parameter. The effect is then used by SFA to determine the overall importance of the parameter on the system.

Single factor analysis is not a very robust method for tuning. The base case in single factor analysis biases the results and the designer is left with an inaccurate picture of how the parameters actually affect the overall performance of the system. If the designer already has a good configuration, then single factor might be able to find a slightly better configuration. However, the need for a good configuration to be known before the tuning begins limits the effectiveness of single factor analysis in tuning of systems.

2.3 Randomized tuning algorithms

Randomized algorithms have also been used to aid in the tuning of parameterized systems. One type of randomized algorithm is a genetic algorithm. Genetic algorithms are modeled after how DNA changes from one generation to next. Genetic algorithms use random numbers to determine if and how the DNA sequences will change from one generation to the next. To use genetic algorithms, for the tuning of parameterized systems the configuration is broken down into small parts that represent the genes in the DNA model. Once the designer has created the genome for the system, the genetic algorithm is then run. There are various parameters common to genetic algorithms that impact the overall performance of the genetic algorithm such as the size of each generation and the mutation rate. Another important part of genetic algorithms is the fitness function, which is very similar to an objective function. The fitness function is used to determine which configurations are the best within a single generation.

Genetic algorithms can be adapted to almost any parameterized system. The adaptive nature of genetic algorithms is one reason why genetic algorithms are popular in tuning systems. Palesi [Palesi 2002], Zitzler [Zitzler 2002], Erbas [Erbas 2006] and Ascia [Ascia 2004] all show that genetic algorithms work well even when compared to system specific algorithms.

Slmoka [Slmoka 2004] used a different random approach to tune parameterized systems. Slmoka used a Tabu search algorithm to examine the design space. Tabu search is a version of simulated annealing where the algorithm stores the previous configurations so the algorithm does not need to rerun configurations.

Palermo [Palermo 2003][Palermo 2008] used an algorithm called Random Search Pareto (RSP). RSP is based off of Monte Carlo methods. RSP uses random samples to sample the design space. RSP then uses the known points to generate a new set of random points. RSP is better able to avoid becoming stuck in local minima, which is a problem with most random algorithms.

Randomized algorithms all share one major problem, unpredictable runtimes. Randomized algorithms are usually able to find the useful configurations, however, in order to find the configurations random algorithms need to evaluate many configurations. In most cases the extra configurations that are evaluated do not provide any useful information in the exploration process. The random approach, therefore, is often not a very useful approach since the designer does not know how long the algorithm will take to run or when the algorithm has reached the best configurations.

Chapter 3

Single-factor exploration using an application-specific design space tree

3.1 Background

Presently, FPGA soft-core processor users must manually determine the best core configuration for a software application. Such manual configuration either results in unduly long exploration times due to evaluating too many configurations, or results in a sub-optimal configuration. We consider two approaches, a traditional CAD approach that maps to an abstract problem model and then solves the problem thoroughly while relying on estimations and a synthesis-in-the-loop approach that uses actual synthesis and execution during exploration but searches only a fraction of the solution space. While our work's motivation lies in soft cores for FPGAs, our approaches may apply to ASICs also.

We developed our methodology using a Xilinx Microblaze FPGA soft-core processor [Xilinx 2005], but the methodology would be applicable to other FPGA soft-core frameworks. The Xilinx Microblaze is a 32-bit soft-core processor designed for efficient implementation on Xilinx FPGAs. The Microblaze is a single-issue in-order execution processor. The Microblaze can be configured to instantiate any combination of

the following five components: multiplier, barrel shifter, divider, floating-point unit (FPU), and data cache. The first four components are each "on/off" type, either being instantiated or not instantiated, and only one instance of each component type is needed since the Microblaze is a single issue processor. The data cache, when instantiated, can be 2 Kbyte, 4 Kbyte, or 8 Kbyte, but we only considered 4 Kbytes in this paper for simplicity. Furthermore, the Microblaze supports two cache types, an older basic cache, and a newer better performing "MCH" cache, although only the latter is considered in this paper. Thus we considered 2^5=32 possible Microblaze configurations. When any of the first four components are instantiated, the Microblaze ISA is updated to include a special instruction for the corresponding component (e.g., a multiply instruction), and the Microblaze compiler generates code utilizing that the special instruction. We refer to a Microblaze with none of the five extra components as a base Microblaze.

Instantiating a component increases a Microblaze's size, but may improve an application's performance, depending on the application. We defined the task of customizing a Microblaze for a particular software application as the task of instantiating a particular combination of components, known as a configuration. Customizing allows the designer to meet the design goals, which may involve performance and/or size that are best met for an application running on the customizable Microblaze.

We measured performance as the time to execute an application once from beginning to end (typically an embedded benchmark application loops back to its beginning after the end). The execution time is the number of clock cycles multiplied by the clock period, referred to hereafter as the *application runtime*. We utilized Xilinx ISE and EDK tools to determine the clock period by synthesizing a configured Microblaze

Figure 7: Equations for calculating *Equivalent LUT* value of a configured Microblaze.

$$LUT_{Equivalent} = LUT_{Regular} + LUT_{Equivalent(Mult)} + LUT_{Equivalent(BRAM)}$$

$$LUT_{Equivalent(Mult)} = \#Mult_{Used} / \#Mult_{full\,MB} * LUT_{full\,MB}$$

$$LUT_{Equivalent(BRAM)} = sizeBRAM_{Used} / sizeBRAM_{full\,MB} * LUT_{full\,MB}$$

onto a specific FPGA device. We measured the number of clock cycles by executing an application on a Microblaze mapped to the FPGA device, with the application slightly modified to communicate with a clock-cycle counting circuit. The cycle counting circuit non-intrusively counts clocks cycles while the application executes but does not affect the application's performance.

A basic measure of a soft core's size on an FPGA is the number of utilized lookup tables (*LUTs*)[1]. However, a soft core may also utilize hard-core FPGA resources, such as; hard-core multipliers or block RAMs. To be able to straightforwardly plot and compare sizes of different soft-core configurations, an equivalent LUT value is assigned to hard-core resources. We did so by first measuring the regular LUTs, hard-core multipliers, and block RAM utilized in a full Microblaze. We then combined the individual size metrics into a single size metric representing equivalent LUTs.

Figure 7 presents the equations for calculating equivalent LUTs for a given Microblaze configuration. Assuming each type of resource (LUT, hard-core multiplier, or block RAM) is of equal importance. Figure 8 lists the equivalent LUT values for each hardcore unit. For a given configured Microblaze, the equivalent LUTs, $LUT_{Equivalent}$, is the sum of the regular LUTs, $LUT_{Regular}$, used for logic to support datapath components, the equivalent LUTs for hard-core multipliers, $LUT_{Equivalent(Mult)}$, and the equivalent LUTs

[1] We originally utilized configurable logic blocks (CLBs) as a measure of size, but Microblaze designers at Xilinx informed us that LUTs are a more accurate and useful measure.

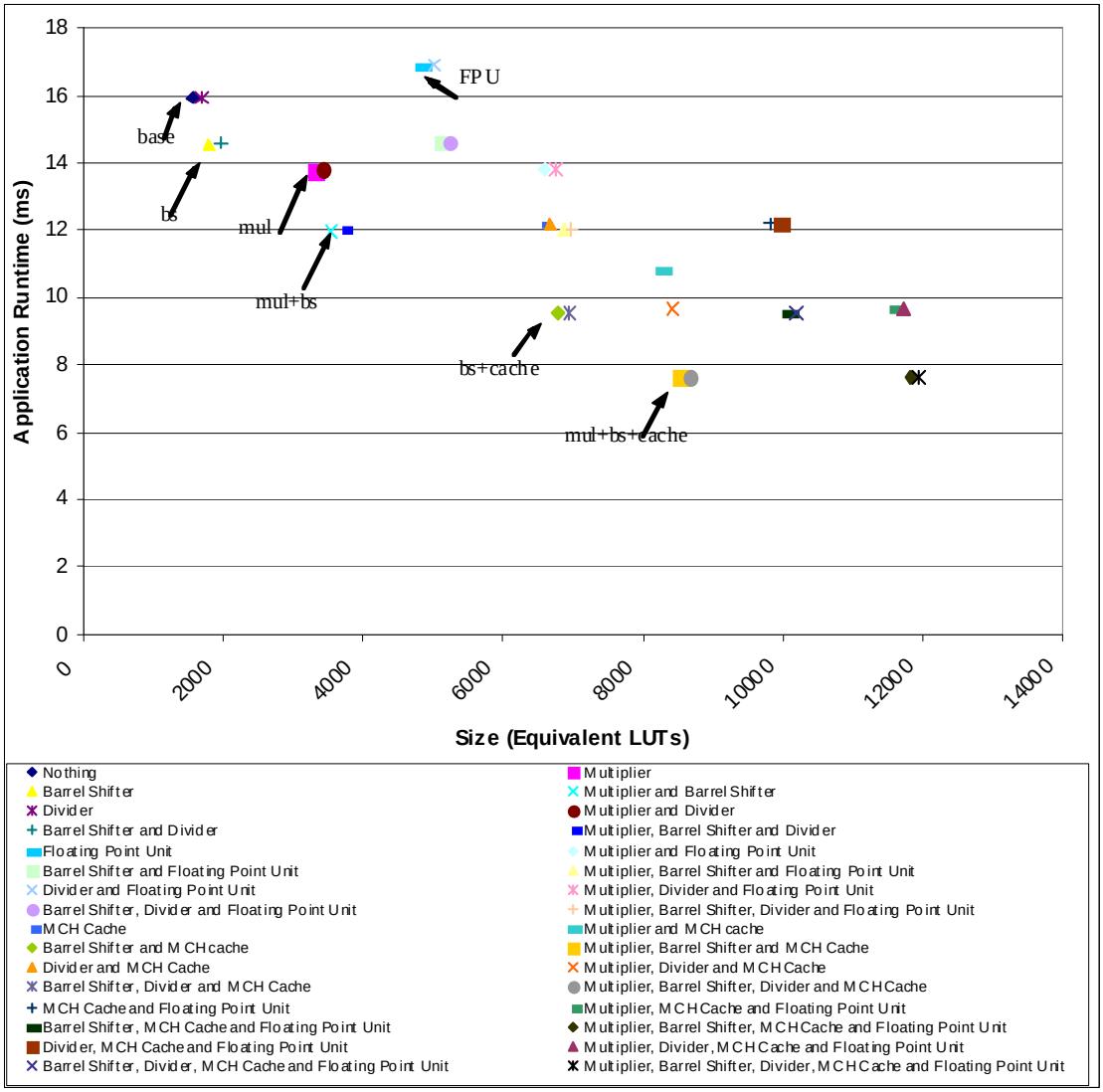
Figure 8: Equivalent LUT values for hard-core units.

Component	Equiv LUT Count
LUT	1
MULT 18x18	569
BRAM	1328

for block RAMs, $LUT_{Equivalent(BRAM)}$. The equivalent LUTs for the utilized multipliers is equal to ratio of multipliers used, $\#Mult_{Used}$, to multipliers in a full Microblaze, $\#Mult_{full\ MB}$, multiplied by the number of regular LUTs in a full Microblaze, $LUT_{full\ MB.}$ Likewise, the equivalent LUTs for the utilized block RAMs is equal to ratio of block RAM used, $sizeBRAM_{Used}$, to block RAM in a full Microblaze, $sizeBRAM_{full\ MB}$, multiplied by the number of regular LUTs in a full Microblaze, $LUT_{full\ MB}$. Of course, a user can weigh regular LUTs, multipliers, or block RAMs more heavily if that resource happens to be more valuable to the developer. We noted that another research group working closely with Altera independently developed a similar equivalent LUT concept for similar size comparison purposes [Yiannacouras 2006] thus lending confidence to the use of the equivalent LUT size metric during soft-core exploration. All LUT data in this paper represents equivalent LUTs. Interestingly, we discovered that our equivalent LUT concept correlates almost perfectly with Xilinx's own equivalent gate concept.

Note that the equivalent LUT concept is essentially a cost function that combines three terms by normalizing and weighing them equally. Our approach is not strictly dependent on the above-described cost function; other functions could be used, including an approach where users specify the relative weights, or where different normalization methods could be used.

Figure 9: Size versus application runtime for all Microblaze configurations executing the *aifir* EEMBC benchmark, with all Pareto points labeled. An additional labeled point *(FPU)* is highlighted to show the performance overhead of instantiating an underutilized component, due to lengthening of the clock cycle.



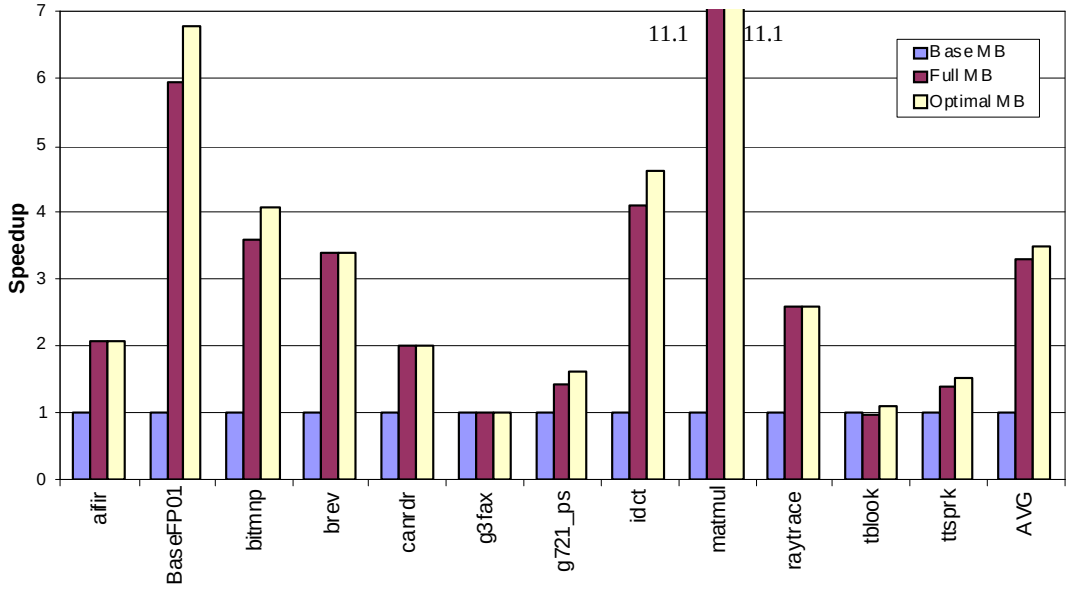
In our experiments, we considered eleven benchmarks selected at random from EEMBC [EEMBC 2006], a benchmark suite intended for embedded systems. We reported data for all of the randomly selected EEMBC benchmarks that we were able to compile and execute on the Xilinx Microblaze. In addition, we considered an internally developed ray tracing application (*raytrace*) that is predominantly a floating-point application.

For each benchmark, we utilized scripts to run our search heuristics, where those scripts automatically performed FPGA synthesis and evaluated the application whenever necessary. The scripts were executed on a computer connected to an FPGA development board (an ML310 board in our case).

Figure 9 demonstrates the benefits of customizing an FPGA soft-core processor for one application. The figure presents the application runtimes for the EEMBC benchmark aifir running on each of the 32 possible Microblaze configurations. Considering only the Pareto-optimal configurations, the Microblaze configurations have a 2X variation in application runtime and a 2X variation in LUTs, clearly demonstrating the benefits of configuring the Microblaze to a particular application and its performance and size constraints.

Figure 10 presents the performance speedups of the performance-optimal configured Microblaze for all 12 benchmarks, as determined by exhaustively examining

Figure 10: Speedups for base *(Base Microblaze)*, full *(Full Microblaze)*, and optimal *(Optimal Microblaze)* Microblaze configurations.



all possible configurations for each application. The optimal Microblaze configuration on average has a 3.5x speedup compared to a base Microblaze and a maximum speedup of 11.1x for the application *matmul*. However, obtaining that data by performing exhaustive exploration for this application required approximately 15 minutes per configuration (with 99% of that time spent on synthesis and with certain configurations requiring more than 15 minutes), resulting in over eleven hours of exploration tool runtime. Even for the relatively small number of configurable options we considered, exhaustively evaluating all possible configurations is quite prohibitive. A core user would need to re-evaluate all configurations anytime significant changes and potentially even small changes were made to the application, a common occurrence in a software design cycle. Furthermore, we expect that the number of configurable options will continue to increase for soft-core processors in the future. As such, if the configurability is doubled from five options to ten options, the execution time for an exhaustive evaluation increases from approximately 11 hours to 11 days.

We sought to develop methods that would execute in approximately 1-2 hours – a tool runtime that we believe FPGA designers would find reasonable during the optimization step of design. By using synthesis which takes on the order of tens of minutes during the exploration process the key feature of our developed heuristics must be that of executing only a few synthesis runs such that total customization time is on the order of 1-2 hours.

3.2 Application specific tuning of parameterized systems

We considered the problem of customizing a Microblaze to minimize a particular application's runtime, with and without a size constraint. Fast tuning of configurable

hardware platforms has been the subject of several recent research efforts. Most efforts assume that hundreds or thousands of configurations can be examined [Abraham 2000][Givargis 2001][Mishra 2001][Mohanty 2002][Sherwood 2004][Szymanek 2004], but the 15 minute synthesis time in the FPGA soft-core problem means that only about 5-15 synthesis runs can be conducted.

3.2.1 Traditional CAD approach: 0-1 knapsack

We first considered developing a traditional CAD approach to tuning soft cores. The approach pre-characterizes the application and processor, then maps the problem to an abstract (and inexact) model, and then uses the model to solve the problem.

The soft-core configuration problem could be approximately cast to a 0-1 knapsack problem, wherein, one seeks to maximize the value of items placed in a knapsack having a weight constraint, with each item having a value and a weight. In the fractional knapsack problem, one can include any fraction of items, while in the 0-1 knapsack problem, the only allowed fractions are 0 or 1, meaning the items are indivisible. We considered each optional MB component as an indivisible item. We assigned a component's value to be the ratio of the speedup increment that occurs when instantiating that component compared to a base MB (e.g., a speedup of 1.4 has an increment of 0.4), over the size increment compared to a base MB. Note that the speedup increment for a component depends on the application, but the size increment is application independent. This cast is approximate, because speedup increments may not always be strictly additive when multiple components are instantiated. For example, component A may have an increment of 0.4 and B of 0.3, but A and B together may only yield an increment of 0.6, not 0.7. Likewise, size increments may not be strictly additive.

Figure 11: Average pairwise speedup-increment additive inaccuracies for all pairs of benchmarks.

Component	Cache	Floating Point	Divider	Multiplier
Barrel Shifter	5.2 %	1.0 %	0.0 %	10.4 %
Multiplier	6.7 %	1.9 %	26.0 %	
Divider	2.9 %	0.0 %		
Floating Point	5.1 %			

Figure 11 presents the inaccuracy of the additive assumption for all pairs of components. The additive assumption holds well (near-zero inaccuracy) for four pairs of components. Adding the multiplier and the barrel shifter speedup increments yields a 10% inaccuracy, since some bit shifts are achievable with a multiplier, and vice versa. Adding multiplier and divider speedup increments yields 26% inaccuracy.

A well-known optimal algorithm for solving the 0-1 knapsack problem first sorts items by their value/weight ratio, and then finds the optimal solutions using a dynamic programming algorithm [Toth 1980]. To execute that algorithm, the speedup increment (value) for each component must first be computed. As the speedup is application dependent, the first six synthesis and executions must be evaluated first: for the base MB, for the MB with a multiplier only, for the MB with a barrel shifter only, with an FPU only, with a divider only and finally with only a cache. Figure 12 shows speedup increments, size increments, and their ratios, for the aifir EEMBC benchmark application.

The dynamic programming algorithm has what is known as a “pseudo-polynomial” runtime complexity of O (n*W), where n is the number of items, and W is the knapsack weight constraint. This algorithm is known to be fast when W is a “small” integer, with a magnitude of perhaps 10,000 – 1,000,000, and of course when n is also small. Fortunately, W is indeed a small integer in the case of our MB configuration

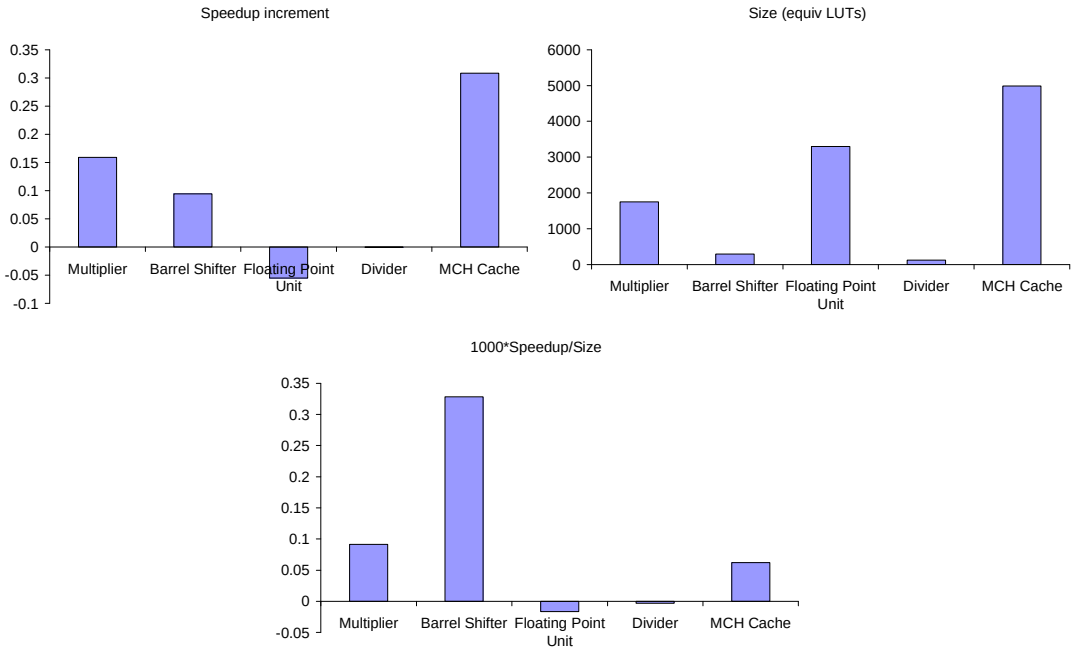
problem (a full MB is only 12,000 equivalent LUTs) and n is also small in our problem (5 instantiatable units).

This approach applies six synthesis/execution runs when initially determining the component speedup and size increments, requiring about an hour, which dominates the approach's runtime. The inputs to the dynamic programming algorithm – *n* (number of soft core parameters) and *W* (number of available LUTs) – can each accommodate large increases before the 0-1 knapsack algorithm runtimes approach a non-negligible time (versus synthesis) of tens of minutes. Even then, we have found that we can "quantize" the knapsacks weights by dividing all weights by 10 to yield a 10x algorithm speedup with almost no degradation in quality of results.

3.2.2 Synthesis-in-the-loop approach: impact-ordered trees

Casting the soft-core configuration problem to 0-1 knapsack yields an approach with

Figure 12: Speedup increment, size increment, and their ratio, for each MB component for the *aifir* benchmark.

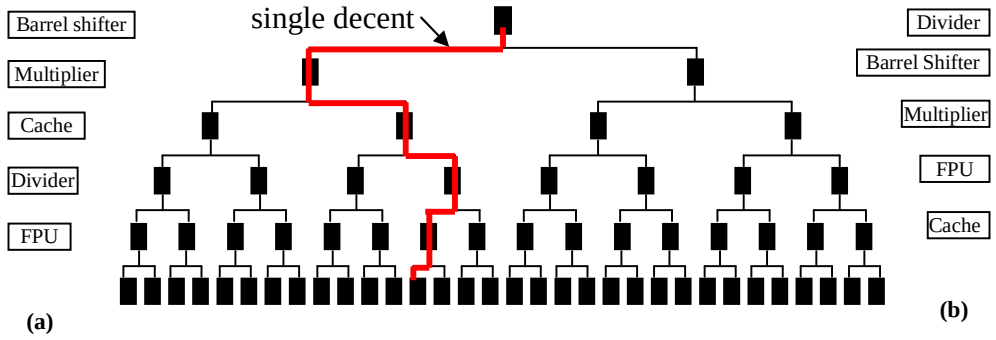


desired tool runtime and near-optimal results. However, the approach makes an assumption that speedup and size increments are additive, which is inaccurate for some pairs of components. As demonstrated in the experiments section later, those inaccuracies can result in sub-optimal solutions. We thus sought to develop an approach that did not rely on the additive speedup increment assumption, but rather used synthesis/execution during exploration (synthesis-in-the-loop) while still executing just a few synthesis/execution runs.

We developed a greedy search method based on an approach proven effective in other parameterized architecture configuration research. The greedy method pre-determines the impact each parameter has on design metrics, and then searches the parameters in sequence, ordered from highest impact to lowest. For example, Zhang [Zhang 2004] used the greedy method for customizing a highly configurable cache, where evaluating each configuration which took many minutes due to lengthy simulations, and found near optimal results. Thus we investigated such an impact-ordered approach.

The first phase of the approach determines the impact of each component. We can define impact simply as the speedup. Through experimentation, we found that a better definition takes the ratio of speedup/size, just as in the knapsack problem. Thus, the first phase of the approach computes speedup increments, size increments, and their ratio; requiring six synthesis and execution runs and resulting in the same data as in Figure 6. This method used is called *single factor analysis*. The second phase considers the components in order of their impact. For the current component, the approach instantiates the component, then synthesizes and executes to determines the application's runtime and

Figure 13: Impact-ordered tree approach: (a) Application-specific impact-ordered tree for the *aifir* benchmark, (b) Fixed impact-ordered tree. Note that *neither approach actually generates the entire tree* – both make a single descent to a leaf node. The thick red line shows a single decent through the tree.



size. If instantiating the component improves runtime and meets size constraints, the component is added; otherwise, it is not. The approach then moves on to the next component.

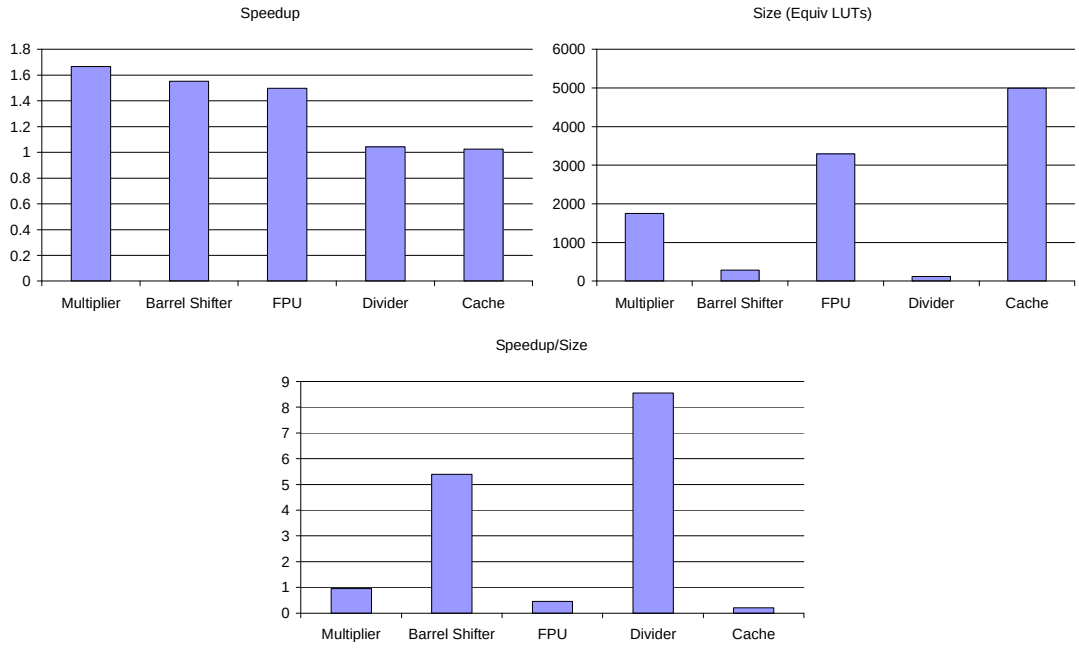
We refer to the above approach as an *application-specific impact-ordered tree approach*. Essentially, if we envision the entire solution space as a tree, as in Figure 13. The approach orders the levels of the tree, and then descends into only one sub-tree at each level, until reaching a single leaf node. The first phase orders the tree's levels, while the second phase makes a single descent. The thick red line shown is an example of how the algorithm descends the tree. This approach requires six synthesis/executions for phase one, and five synthesis/executions for phase two, resulting in 11 total synthesis/executions.

We also investigated a variation of the above approach with the goal of reducing the number of synthesis/execution runs, by pre-determining average component impacts on a suite of typical benchmarks, rather than determining impacts on a per-application basis. The approach essentially moves phase one of the above approach from the tool user to the tool developer, thus cutting out six of the eleven synthesis/execution runs, leaving just five such runs. We refer to this approach as a fixed-order impact-ordered tree

approach, because the impact ordering is fixed. Figure 14 shows the data averaged for all our benchmarks, with the speedup/size data resulting in the impact ordering shown in Figure 13(b).

Each of our algorithms assumes the problem uses an objective function to determine the best soft-core processor configuration given a limited size constraint. Some design scenarios impose no size constraint on the FPGA soft-core processor, instead seeking only the minimum application runtime. In the absence of a size constraint, one might assume minimal application runtimes could be achieved by simply instantiating a full MB. However, this assumption is false, as was illustrated in Figure 10. Figure 10 presented the performance speedup for different MB configurations: a base MB, a full MB, and an MB configured for optimal application runtime (determine by exhaustive search) for the corresponding application compared to the base MB configuration. Notice

Figure 14: Speedup increment, size increment, and their ratio, for each MB component averaged across all 12 benchmarks.



for some applications the full MB is actually slower than the optimal. The reason is because as more components are instantiated, the MB clock period may be lengthened, due in part to longer delays necessary for the increased wire routing within the larger MB. The point labeled FPU in Figure 9 clearly illustrates the impact of longer delay caused by adding an underutilized FPU component.

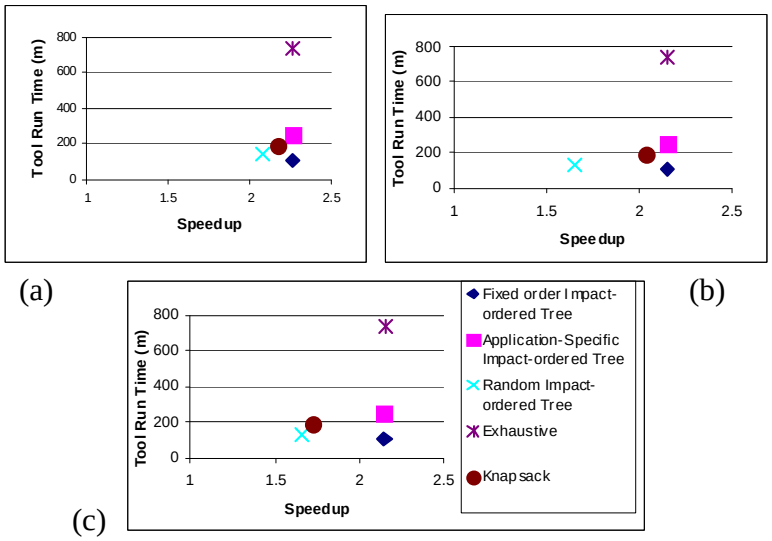
To handle the no size constraint situations, in either the 0-1 knapsack approach or the impact-ordered tree approaches, we simply use a size constraint that is equal to or larger than the size of a full MB.

3.3 Experiments

We implemented the knapsack, application-specific impact-ordered tree, and fixed-order impact-ordered tree approaches as scripts executing with Xilinx Platform Studio synthesis tools, coupled with a Xilinx Virtex-II Pro FPGA development board (ML310), for all 12 embedded benchmark applications. To compare the approaches with optimal results, we implemented an exhaustive search approach that simply performed synthesis/execution for all 32 possible soft-core configurations.

Figure 15(a) presents the average speedups and tool runtimes for each approach for the scenario of unconstrained size. Exhaustive search requires over 700 minutes (11 hours) and finds average speedups of 2.3. The knapsack approach finds near-optimal solutions with a speedup of 2.2. Both impact-ordered tree approaches find the optimal solution. The fixed impact-ordered tree approach has the fastest runtime of 108 minutes. The knapsack approach should actually have roughly the same runtime, as both approaches synthesize about the same number of configurations. One particular configuration examined by the knapsack approach, namely a base MB with a barrel

Figure 15: Average speedups obtained by the various exploration approaches, for: (a) no size constraint, (b) a fixed size constraint set at 80% of the size of a full MB, (c) a per-application-tailored size constraint of 80% of the size of the optimal MB for that application (as determined in (a)), all on a Virtex-II Pro device.

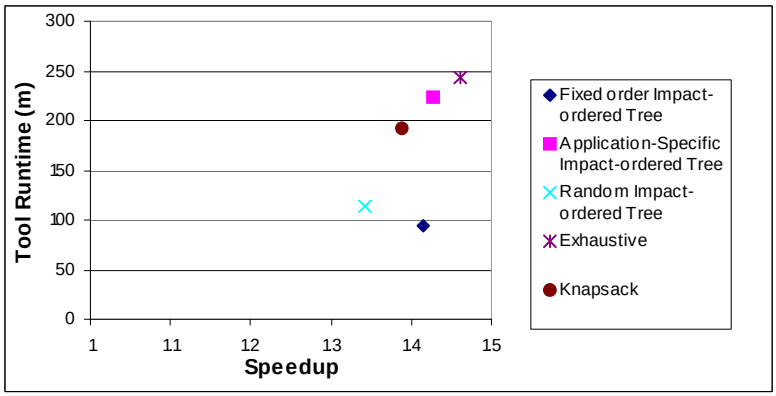


shifter alone, happened to have an unusually long synthesis time. Such anomalous synthesis runtimes are an artifact of the nature of FPGA physical design heuristics. In general, one should assume that the knapsack approach and the fixed impact-ordered tree approach will have equally fast tool runtimes.

One might wonder whether *any* ordering of the tree levels in the fixed impact-ordered tree approach would in fact yield the optimal configuration. Thus, we implemented another heuristic using a random ordering: barrel shifter, cache, FPU, divider, multiplier. Figure 15(a) shows that this random impact-ordered tree approach performs worse; though for the unconstrained size problem this approach is actually somewhat competitive.

Figure 15(b) presents the average speedups and tool runtimes for each approach for a fixed size constraint, chosen to be 80% of the size of a full MB. We also obtained data for a 50% constraint, with similar results (not shown). The plot again shows that the impact-ordered tree approaches find optimal speedups (2.2), the knapsack approach finds

Figure 16: Average speedups for the approaches on a Spartan2 FPGA.

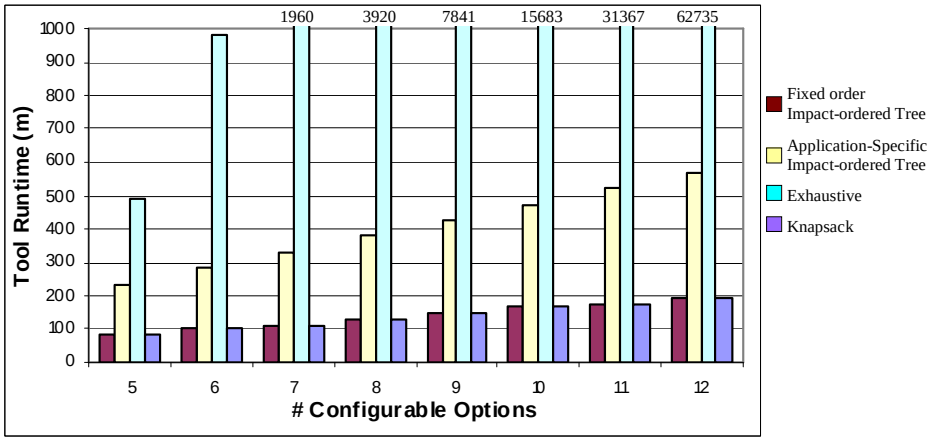


near-optimal solutions (2.0), and the random impact-ordered tree approach is no longer competitive.

We sought to see how each approach would perform in a scenario where the size constraint was tight enough to prohibit use of the best performing MB for a given application. We created a unique size constraint for each application. Figure 15(c) presents the average speedups and tool runtimes for each approach with a tailored size constraint being 80% of the best performing MB for each particular application (as determined through exhaustive search with no size constraint, and choosing the smallest among equally performing configurations). We also obtained data for a 50% constraint, with similar results (not shown). While the fixed and application-specific impact-ordered tree approaches found the optimal, the knapsack heuristic performed very poorly for this size constraint. We found that the reason for the knapsack's poor results is due to the inaccuracy of the additive speedup increment assumption, which caused sub-optimal selection of components.

To further evaluate the effectiveness of the approaches, we re-implemented the entire set of experiments for a Xilinx Spartan2 FPGA. Figure 16 presents the average

Figure 17: Estimated tool run times for increasing number of configurable soft-core processor options.



speedups and tool runtimes for each approach for the case of unconstrained size. Again, the impact-ordered tree approaches were the best performing approaches, but the approaches chose configurations that were slightly below optimal on average. The application-specific approach found the optimal configuration in 11 of 12 cases, with a 20% worsening in performance for only one application. The fixed approach also resulted in a 20% worsening of performance for that same application, along with a 10% worsening for another application, but overall found the optimal configuration in 10 of 12 cases.

From this data, the fixed-order impact-ordered tree approach seems preferable. Of course, one must consider that our fixed-order was determined from the very same 12 benchmarks that we then used to compare the approaches. To examine this issue, we used six randomly selected benchmarks to define the fixed ordering, and then applied the approaches on the other six benchmarks only. The fixed impact-ordered tree approach again found the optimal for the constraint situations in Figure 15(a), (b), and (c), and even found the optimal for the situation in Figure 16. Applying a particular fixed order on a radically different benchmark may yield worse results. Vendors might address that

situation by having different fixed orderings for different application domains (e.g., control, signal processing, etc.), allowing the user to select a domain.

The application-specific impact-ordered tree approach is more robust in the presence of new benchmarks, but at the expense of about twice the tool runtime.

Our formulation considered five components. One can expect the number of soft-core parameters to increase beyond five. Figure 17 shows estimated tool runtimes for five to twelve two-valued parameters. While the approaches are significantly faster than exhaustive methods, the application-specific impact-ordered tree approach's runtime does increase to nearly 10 hours for twelve parameters. In contrast, the fixed-order impact-ordered tree scales well, requiring just less than 3 hours for twelve parameters. Note that the figure only shows runtime and not quality of results.

Chapter 4

Pareto point generation using interdependency graphs

In the previous chapter, we discussed how a tuning methodology can benefit from an application specific approach. In this chapter, we will go into more depth about the strengths and weaknesses of both single factor analysis and design of experiments. We will then discuss an alternative to the application-specific impact-ordered tree. The interdependency graph like the application-specific impact-ordered is built from data generated from the system, but is better able to handle interactions between the various parameters.

4.1 Single Factor Analysis (SFA)

Single Factor Analysis [Peterson 1986] is a common approach in design space exploration. SFA requires the designer to select a base configuration. The base configuration uses a default value for every parameter in the system. SFA changes each parameter, or factor, one at a time. The first parameter is changed to every possible value (or some subset therefore), while all other parameters are held in the default value. After examining all the values for the first parameter, the parameter is set back to the default

value, and the same process is repeated for each remaining parameter. The results from the SFA runs can be used to create a regression model. The model is then used to predict the best configuration for given design goals.

We can extend SFA into an algorithm that generates Pareto points. The new algorithm would first determine the significance of each parameter, using an Impact Ordered Tree (IOT). The significance, or impact, is determined by the maximum change seen for a given parameter. The change is normalized to a value between 0 and 1, using the largest value seen for the metric as 1. Normalization is done for each metric for each parameter. The maximum of the normalized metrics for each parameter is the impact of that parameter. The impact of each parameter is then used to sort the parameters. Starting with the sorted list of parameters, the algorithm takes the two highest-impact parameters and generates all combinations of those two parameters. Assuming every parameter has three possible values, the algorithm generates nine points for this step. All other parameters are held at their default value. By saving the results from previous runs, this step actually only requires four additional runs. The results are then pruned of all non-Pareto points. The resulting set of (intermediate) Pareto points is then used with the third most significant parameter to generate the next set of exhaustive data. The new data is also pruned, leaving only the Pareto points. This process repeats until all parameters have been combined.

Figure 18 shows an example of the results of the SFA approach, showing a simplified design space with only 3 parameters: cache size, cache associativity, and supply voltage. Cache size and cache associativity are interdependent, while supply voltage is independent. Each of the three symbols in Figure 18 represents a different supply voltage. Both the cache size and associativity have two possible values for a total of four configurations. When combining cache parameters and supply voltage, the four configurations move as a group when the supply voltage is varied. In (a), the base configuration is [cache size = 0, cache associativity = 1, supply voltage =1] which finds all but one Pareto point. Using the same environment but changing the base configuration to [0,0,2], three Pareto points are overlooked. SFA can only see the effect of the parameters in relation to the base configuration—a severe deficiency of this approach for Pareto point generation.

Figure 19 shows an example when all 3 of the parameters are interdependent. In

Figure 18: Illustration of results of SFA extended for Pareto points, showing dependency on the base configuration selection: (a) A 3-parameter system where 2 parameters are interdependent and the third is independent. The filled points are the points examined by SFA. The circled points are Pareto points found. (b) A different base configuration yields different Pareto points.

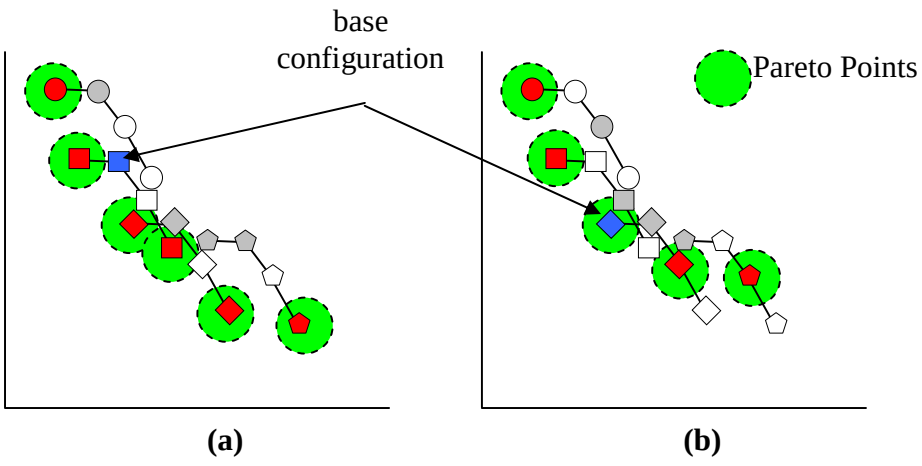
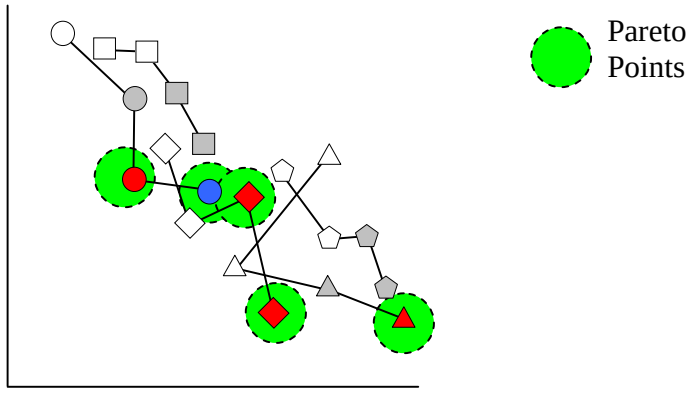


Figure 19: SFA with three dependent parameters. If all three parameters are interdependent, then single factor analysis breaks down further.



this case, SFA misses three Pareto points, for the base configuration shown. The SFA approach can only see points near the base configuration, but interdependencies require more complex explorations. With additional parameters and more interdependencies, the SFA approach fails to find a thorough set of Pareto points.

As shown in the previous examples, SFA is not a very robust approach to finding the Pareto points in a system. One problem is that a base configuration must be chosen. The base configuration is the basis for all other measurements, so by choosing a bad base configuration results in exploration of an inappropriate design space region. Also, SFA performs a very limited search around the base configuration. This means that only the local effect of a parameter will be seen.

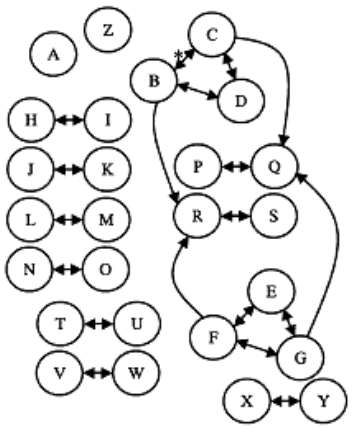
The other major problem with SFA is that the interdependencies between the parameters are not taken into account. SFA assumes that each parameter affects only itself. In most systems this is rarely the case. For example, in a cache, the best associativity for the cache can change as the cache size increases. However, changing the supply voltage of the system will not effect which associativity is best for a given cache size.

4.2 Parameter interdependency graph

Givargis [Givargis 2002] developed a more thorough method for determining the Pareto points of a system by introducing the idea of a parameter interdependency graph. The parameter interdependency graph shows the dependencies between the parameters in the system. Figure 20 shows the graph that was developed by Givargis for the Platune architecture consisting of a processor, two caches, buses, and memory, with each component having numerous parameters. There are many groups of connected parameters, but most of the groups consist of only a few parameters. Platune searches each group exhaustively, finds Pareto points for each group, and combines all points to find the global Pareto points. This approach thus avoids exhaustive search of all parameters, instead only exhaustively searching interdependent parameters, thus greatly reducing the search space while still searching the most important regions of the space.

Figure 20 shows the parameter interdependency graph for the cache size,

Figure 20: Platune's parameter interdependency graph. The arrows indicate dependencies between the different factors. [Givargis 2002]



Node	Core	Parameter		Node	Core	Parameter
A	MIPS	Voltage scale		L	CPU-D$-bus	Data bus width
B	I$	Total size		M		Data bus code
C		Line size		N		Addr bus width
D		Associativity		O		Addr bus code
E	D$	Total size		P	I/D$-Mem-bus	Data bus width
F		Line size		Q		Data bus code
G		Associativity		R		Addr bus width
H	CPU-I$-bus	Data bus width		S		Addr bus code
I		Data bus code		T	Peripheral-bus	Data bus width
J		Addr bus width		U		Data bus code
K		Addr bus code		V		Addr bus width
X	UART	Tx buffer size		W		Addr bus code
Y		Rx buffer size		Z	DCT CODEC	Pixel resolution

* Note that we have used double-arrowed lines in place of two single arrowed-lines for clarity.

associativity and supply voltage example. Each shape in the graph represents a different supply voltage. The supply voltage is independent of the other two variables, so the curve for each shape is the same. The parameter interdependency graph shows that the cache parameters and the supply voltage can be searched independently of each other. The Pareto points for each group can then be combined to generate the Pareto points of the entire system.

Givargis was able to greatly increase the accuracy of the Pareto points versus SFA, while also decreasing the time to generate the Pareto points (searching less than 1% of all configurations. However, the parameter interdependency graph was created manually, requiring that the designer have knowledge of possible interactions between the components' parameters in the system. We observed that the interdependencies between parameters may change depending on which metrics are considered (as will be shown in the experiments section). The interdependency graph should be changed when a metric that was not considered when generating the graph is now being considered. Conversely, if the graph was generated for many metrics and the designer only is considering a small number of those metrics, then many of the edges may not be valid for the given metrics. Generating this graph can be difficult due to the complexity of the system and take an expert a significant amount time even for only a few cores. Integrating multiple parameterized cores, as is commonly done today, compounds the problem. Erring on the side of listing interdependencies slows the search; erring on the side of independence may cause Pareto points to be missed.

Furthermore, Platune's parameter interdependency graph treats all dependencies as equal. Some dependencies are stronger than others. Neglecting the weaker dependencies can speed search with negligible negative impact on results.

4.3 Pareto point generation using design of experiments (DoE)

We propose a new algorithm in order to find the Pareto points. The algorithm combines aspects of the random approaches and the parameter interdependency graph. The algorithm eliminates the need for expert user knowledge, while automatically finding and using the interdependencies between the various parameters in the system.

Design of Experiments (DoE), sometimes called Experimental Design, is a formal systematic method for investigating a process' input factors and their relationship to the process' output. DoE's development began in the 1920s by statistician Sir Ronald Fisher [Peterson 1986] for the purpose of improving farm crop output and has since evolved for use in nearly any form of production (chemical, bio-technical, pharmaceutical) and is even used in company management techniques. The key assumption in DoE is that experiments are costly and thus must be minimized. In DoE, a factor is input process variable that can be controlled by the developer, such as whether a particular chemical is added to a process.

One key aspect of DoE relevant to our purpose is to design a set of experiments that yield provably maximum information about a process' input factors for a given number of experiments. For example, consider a process with three input factors A, B, and C; each with two possible levels (high or low, normally represented as +1 and –1 in DoE, often abbreviated as just + and -). The ideal number of experiments in this case would be 8 (or even more if the process includes random effects) and is known as a *full*

factorial design: --- (A off, B off, C off), --+, -+-, -++, +--, +-+, ++-, and +++. As a trivially-simple illustration of DoE, assume instead that cost constraints allow only four experiments. A poor experimental design would be ---, --+, ++-, and +++, while a better design would be --+, +--, -+-, and +++. The reason is that experiments should be orthogonal to each other in order to provide the maximum amount of information for subsequent analysis.

An experimental design is *orthogonal* if the effects of any factor sum to zero across the effects of the other factors. The poor design had C on (+1) for three experiments and off (-1) for only one experiments (+1+1+1-1=2), whereas the better design had each factor on for exactly two experiments and off for two (+1+1-1-1=0). Orthogonality provides for improved subsequent analysis. It provides for a clear understanding of the limitations of the experiments. For example, the better design was created by enumerating all possible combinations of A and B (--, -+, +-, ++), and then appending C with a value being the product of A and B (so –1*-1=+1, which is why the first experiment was --+). Because C's settings were created in this manner, C's impact is *confounded* with the interaction of A and B. While unavoidable when doing fewer experiments than full factorial, such confounding can be clearly listed in what is known as an aliasing table to inform the experimenter of limitations (and perhaps to enable the experimenter to reorder the factors to avoid confounding factors more likely to interact). Orthogonality is just one of many aspects of the design of good experiments in the DoE framework. Others include randomization, replication, and blocking, which are beyond our scope of discussion.

DoE becomes increasingly challenging, and increasingly beneficial, as the number of allowed experiments differs from the number of possible factor combinations. For example, a process with 8 two-level factors has 256 possible experiments. If only 8 experiments could be run, a naive approach would be to simply run each experiment with exactly one factor high (-------+, ------+-, -----+--, ----+---, ...). However, such experiments tell nothing about the interaction among factors or perhaps factors A and B each has little impact alone, but have a huge impact when combined. A good DoE design would have run experiments run with multiple factors at their high level in each experiment, carefully done to maximize orthogonality.

The other aspect of DoE relevant to this discussion is to analyze a given set of experiments such as to obtain the maximum information about the impact of each factor on the process output and about the interaction among the factors from the given data. Such analysis can be applied to data from any set of experiments but will be of higher confidence if the experiments are well designed (e.g., orthogonal). The key techniques are referred to as ANOVA (analysis of variance), which is used to analyze the effect that each factor has on the final result. Multiple regression methods are also used to determine which factors are statistically significant. Dozens of analysis techniques exist, focusing on different factor models and types of obtained information.

The majority of DoE techniques use two-level factors rather than multi-level factors, due to the powerful statistical methods that two-level factors enable. A factor with a relatively small number of levels exceeding two-level can be mapped to multiple two-level factors in order to benefit from DoE methods, though some methods do directly allow three or more levels for some factors. The most popular DoE approach is known as

fractional factorial design involving experiments representing a fraction (1/2, 1/4, 1/8, etc.) of the full factorial design. Numerous fractional factorial design approaches exist.

Minimal run experiments refer to the situation of running only a small number of experiments relative to the number of possible factor combinations. Numerous techniques have evolved specifically for this purpose, with popular techniques being the Plackett-Burman and the Taguchi techniques [Mishra 2001][Peterson 1986][Scott 1989]. These techniques are used to gather the necessary information while only running a few more runs then there are factors being examined.

In addition to design experiments and analyzing generated data, some DoE techniques predict the best settings of the factors to optimize the output of the given process. Such prediction involves another set of well-developed techniques based on models (mostly linear and continuous) of the input factors.

DoE is a vast statistical discipline comprising a variety of thoroughly studied techniques. This section just lightly touched on some subjects. A thorough understanding requires a textbook of information. While a good understanding is helpful, a key idea of applying DoE to architecture tuning is to *not* have to thoroughly understanding DoE techniques, but rather to apply existing DoE techniques as embodied in well-established commercial DoE toolsets, thus obtaining the benefits of the established discipline. We examined numerous DoE tools and selected DoePRO XL [DOE XL 2006] due to its sufficient coverage of experiments of interest and its integration with Excel spreadsheets.

4.4 Pareto point generation

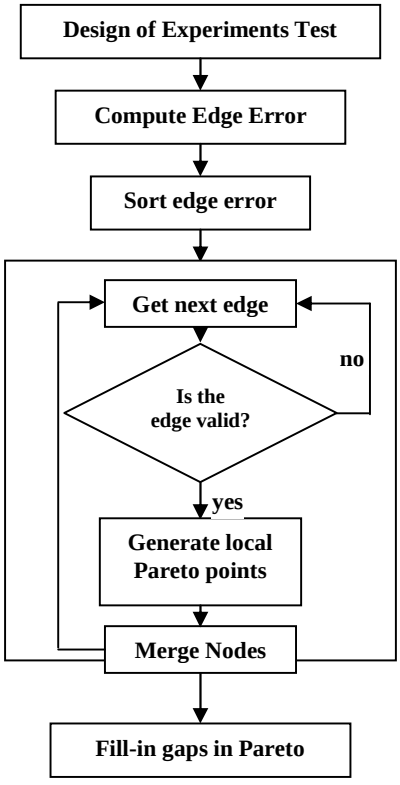
We developed the DoE-based Pareto-point Generation (DPG) algorithm that combines two new techniques. The first technique automatically generates a parameter

interdependency graph, which is a weighted graph whose edges show the dependencies between the parameters. The second technique generates Pareto points from the weighted parameter interdependency graph.

Figure 21 shows an overview of the DPG algorithm. The first step in the DPG algorithm is to evaluate the initial DoE set of experiments. We used the DOE Pro XL [DOE Pro XL 2006] tool to assist in the generation and analysis of the DoE experiments. Each parameter in the system is mapped to either a two or three level DoE parameter. In cases where the parameter has more than three values, DPG chooses the largest and smallest and a midpoint as the three parameters. For the best results, the two level parameters should be used only for parameters that have exactly 2 levels.

After mapping of the parameters, the DoE tools are used to generate a set of

Figure 21: DoE-based Pareto-point Generator (DPG)

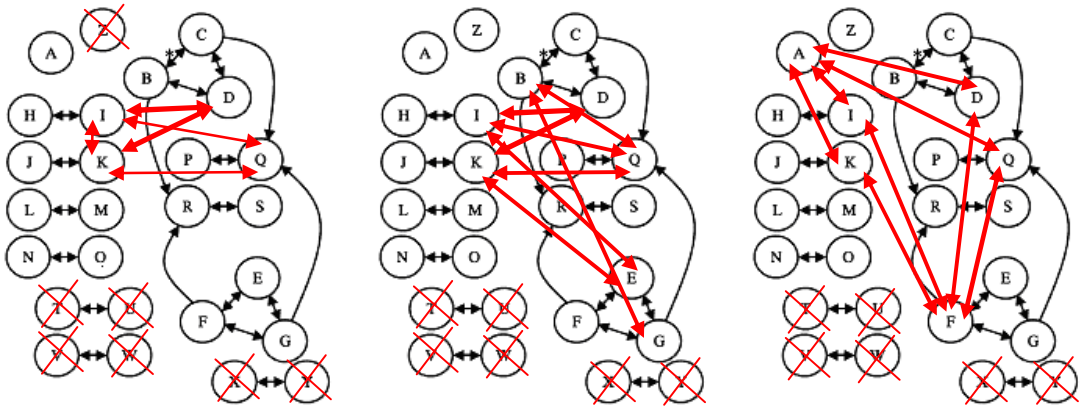


experiments. DPG uses a Plackett-Burman (PB) [Peterson 1986] set of experiments. Using a PB set of experiments provides the accuracy needed while only growing linearly as the parameter count grows. The number of experiments needed for a PB experiment grows linearly with the number of factors. The PB test will gives DPG the average value of each of the parameters in the system, as well as the impact of each parameter.

DPG begins by estimating the values for the unknown configuration for each pair of parameters and for each metric. The system is evaluated for each estimated configuration to determine the accuracy of the estimate. The difference between the actual and estimated values is then used to compute the edge error for each pair of parameters. An edge error of zero means the estimates were accurate. The edge error grows as the two parameters are more dependent on each other. The edge error is the weighted parameter in the interdependency graph. Figure 22 shows the primary edges from the parameter interdependency graph found by DPG overlaid on Platune's original graph. In Platune's original parameter interdependency graph, the buses that connect the processor to the caches and then to memory are all considered to be independent of each other.

DPG found that the bus network and the cache parameters are sometimes interdependent. The interaction between the caches and buses varies over the benchmarks, as shown in Figure 22. In Figure 22 B, C, and D represent the instruction cache. The data cache's parameters are E, F, and G. Parameters I, K, and Q represent the coding method for each of the three buses in the system. In the jpeg application, only the instruction cache is interdependent with the buses, while the buses have some interdependencies between each other. For b1_histogram, the caches and bus network are

Figure 22: Parameter interdependencies found by DPG. The major interdependencies have been overlaid over the original Platune parameter interdependency graph – thicker lines indicate larger dependencies. Parameters not in our simulations have been crossed out.



(a) (b) (c)

all highly interconnected. In g3fax, the supply voltage is interdependent with the cache and bus network. For g3fax, the bus network has no strong interdependencies between the buses.

DPG is able to create *application specific parameter interdependency graphs* for each metric, using both the application and the platform, which will generate a more accurate interdependency graph. The interdependency graphs for each metric are then normalized and combined to create a single error value for each pair of parameters. This normalization step allows DPG to handle an arbitrary number of metrics. To continue to the ultimate goal of generating the Pareto points, the edges are sorted based on the error between the actual and estimated values.

DPG starts with the pair of parameters that has the highest edge value. The subset of parameter values that was chosen earlier is then used to generate exhaustive data for that pair of configurations, a maximum of nine configurations. The set of points is then pruned to contain only the local Pareto points. The edge is then removed from the graph and the two nodes are merged into one. The generated Pareto points now become the

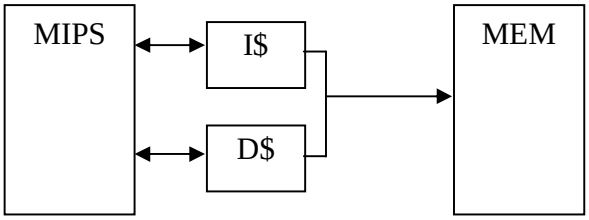
valid values for the combined node. This process repeats until all nodes have been merged into a single node. Edges are only used when two different nodes are connected by that edge. For example, in Figure 22(a), parameter D and I will merge first into one node. Then D and K will merge, but this new node consists of D, I and K. When the edge that connects I and K is visited, the two nodes are already part of the same node, so the edge is ignored.

If the system has parameters with more that three values, a partial Pareto graph will be generated. This partial Pareto graph will show the extremes of the system and give the designer the general shape of the Pareto curve. The designer then can see what part of the Pareto curve is the most interesting and focus on that subsection only. We extended DPG to fill in the missing region, by using the points on either end of the region of interest. DPG fills in the region by seeing which parameters vary between the points on either edge of the region. DPG then determines which parameters are held constant on either side of the region and locks those parameters. The remaining parameters are then varied based on the ranges seen at the edges of the region. By using the parameter interdependence graph, a local search of the region is done. This search can either add some of the intermediate values or run a complete search, depending on time and resources available. This step allows DPG to search a much larger design space while focusing initially on a much smaller and manageable space.

Figure 23: Platune's configurable architecture.

i$			d$			m-i$			
size	linesize	assoc	size	linesize	assoc	width	code	addr width	addr code
256	16	1	12	8	1	32	Bi	32	Bi
2048	2		2048	16	2		Gr		Gr
8192	8		8192	32	4		In		In

m-d$				$-m				
width	code	addr width	addr code	width	code	addr width	addr code	Supply Voltage
32	Bi	32	Bi	32	Bi	32	Bi	1
					Gr			2.8
					In			4.1



4.5 Experiments

To test the DPG algorithm, we used three different platforms, the Platune SoC simulator, Noxim Network on Chip simulator, and the Microblaze platform. Platune SoC simulator will be examined first. Figure 23 shows three values used in our tuning for each parameter and the constants used for the remaining parameters, along with the basic block diagram of the system. Platune has 19 parameters that the developer can modify, ranging from the properties of caches, memory buses, and voltage scaling. These 19 parameters can be combined to form approximately $6x10^{12}$ configurations. For this work, we used a subset of the parameters that limited the design space to 1.3 million configurations.

We randomly chose six benchmarks from the EEMBC [EEMBC 2006] and Powerstone benchmark suites. We compared to SFA and for thoroughness, we also considered another DoE approach, in which SFA rather than DoE (using Plackett-

Burman experiments) was used to find the impact of each parameter, followed by the Pareto point generation approach for SFA – we refer to this approach as DoE IOT. Figure 24 shows the results for the JPEG benchmark. SFA was able to find similar points as was our DPG algorithm for the reduced three level set of configurations having a total of 729 configurations.

The DoE IOT approach also achieved similar results on average, but on JPEG was unable to find any point near the top of the region shown in Figure 24. However, as previously shown, SFA can perform badly depending on the base configuration. Three different base configurations were used and significant variation was found in the results. As the design space of the system increases, the negative effects of SFA will increase.

Figure 24: Pareto curves for JPEG. SFA was able to find the valid endpoints in this example. DoE IOT only found one of the needed endpoints. DPG was initially able to find good points on either side of the main part of the Pareto curve, and DPG's fill-in process finds the remaining points.

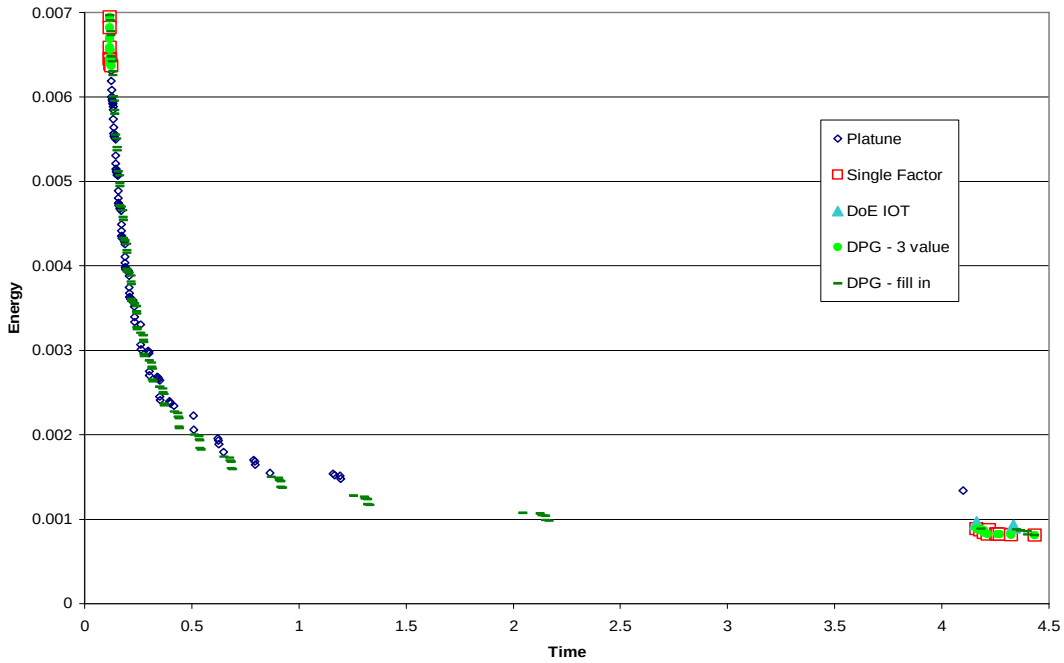
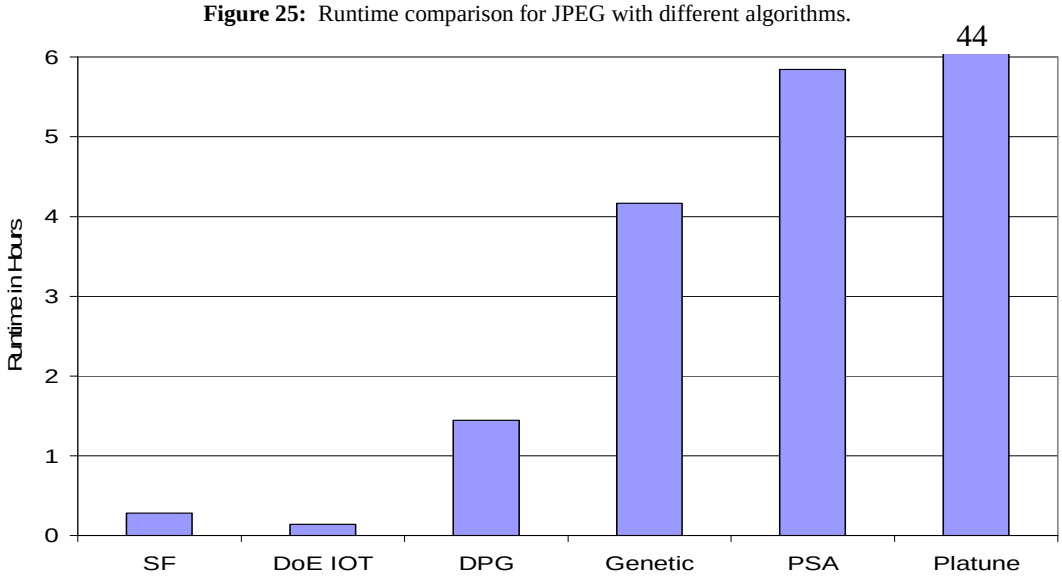


Figure 25: Runtime comparison for JPEG with different algorithms.



DoE IOT achieved similar results to SFA, but in less time, as shown in Figure 25.

The DPG algorithm was always able to find points that were at least as good as SFA. Figure 24 shows the filled-in portion of the Pareto curve, as well as two end points used to fill in the region. We compared the randomized algorithms [Ascia 2005] and our work from chapter 3 since both used Platune for similar design spaces, and both used JPEG as a benchmark. Our DPG techniques evaluate three to four times fewer configurations. The randomized approached used 1500 and 2000 runs respectively, while DPG required 576 runs to complete. Such reduction intuitively makes sense – randomized approaches intentionally search large numbers of configurations and then narrows in on good configurations, while DPG carefully pre-specifies which configurations to consider such that the design space is thoroughly searched in a statistically rigorous manner. Platune's parameter interdependence graph approach required over 10,000 runs. In our experiments, involving relatively small benchmarks, DPG executed for about one hour while the randomized approaches would have executed

for four to six hours (based on the number of configurations examined as reported in previous papers).

While DPG is able to achieve a 3-4x application speedup over genetic and simulated annealing approaches, DPG also produced *better* results than randomized approaches and for Platune'. DPG was able to generate a better Pareto curve than Platune's parameter interdependency graph. Platune covered only 75% of the Pareto points found by DPG for the JPEG example. A Pareto point is considered covered if Platune found a point within 5% from DPG's Pareto point. By comparing minimum Euclidian distance between the two Pareto curves yields an average error of 12%. Previous randomized approaches were reported to find fewer Pareto points than the Platune approach (typically 90%-95% coverage), and thus would have even less coverage of DPG's points.

Figure 26 compares Platune's parameter interdependency graph with DPG for the

Figure 26: Pareto points from Platune versus DPG (with fill-in) for b1_histogram.

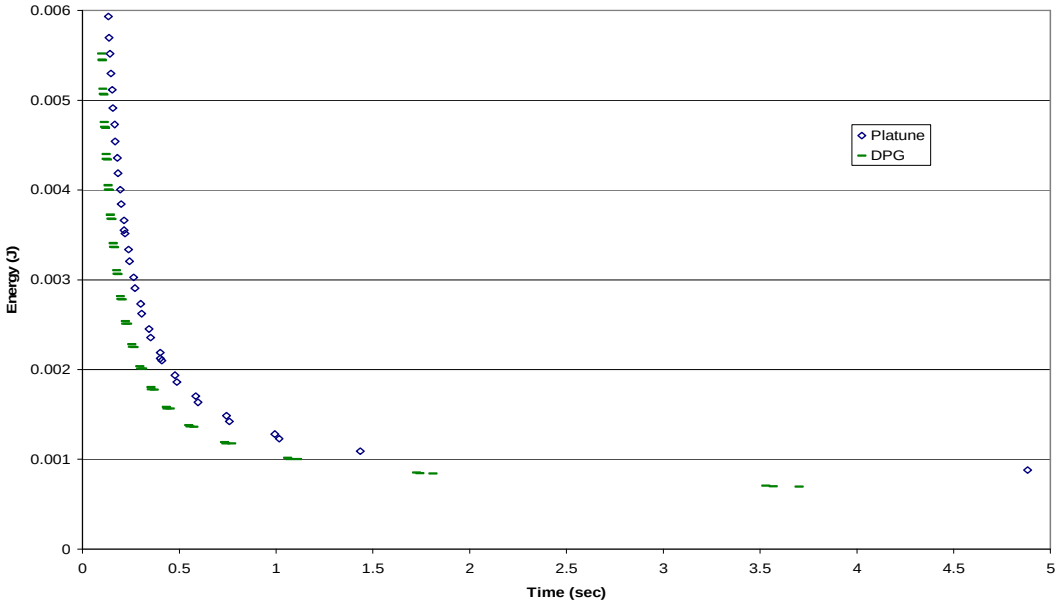
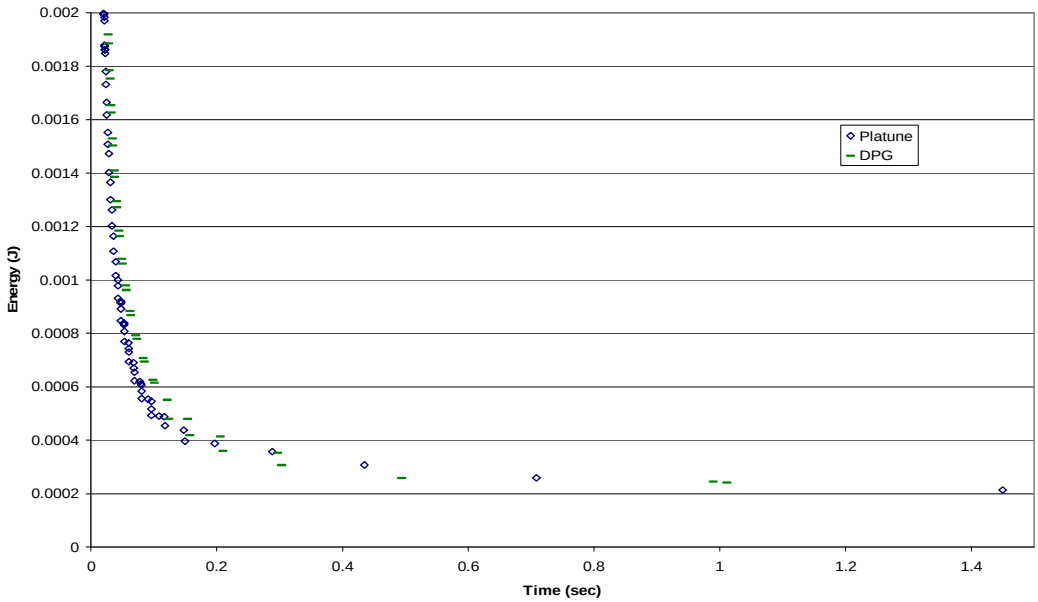


Figure 27: Pareto points for Platune vs. DPG (with fill-in) for g3fax.



b1_histogram benchmark. As seen in the figure, DPG generates better Pareto points over the entire region. The average distance between the two curves is 26%. DPG needed 270 runs compared to the 1800 needed by Platune, for a speedup of over 6x in terms of tool runtime.

Figure 27 examines the g3fax benchmark. Platune requires 3500 runs to generate the Pareto curve, while DPG requires 750 runs. The average distance between the two curves is 14%. DPG consistently finds more points on the horizontal tail of the curve. Platune's parameter interdependency graph has large gaps in this portion of the graph, as seen in the three graphs. DPG is able in all our tests to generate points within this region.

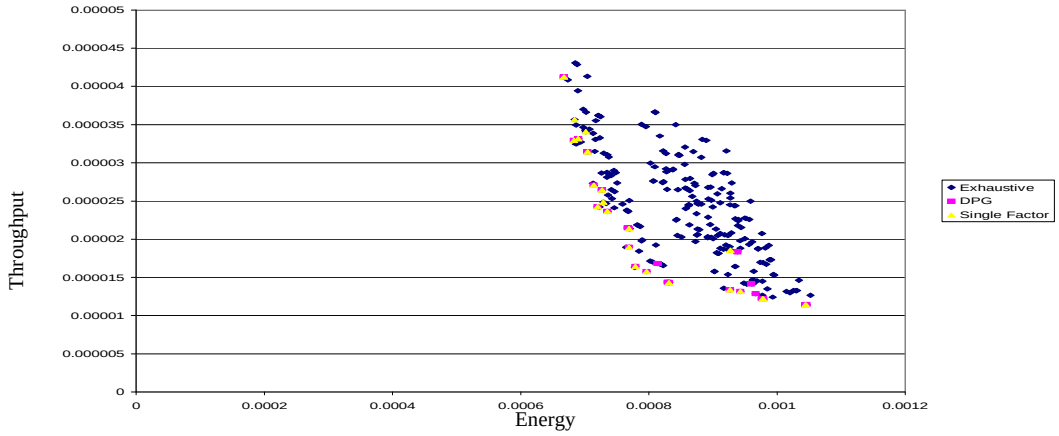
DPG is able to quickly find the location of the Pareto curve. The designer is then able to use this location to determine what region will best suit the designer's needs. DPG then can focus only on that region to complete the Pareto curve. The designer finds the Pareto points of interest without needing to find all Pareto points.

We also experimented with Noxim, a Network on Chip simulator. We evaluated ten of Noxim's parameters. The parameters were divided into two groups, those that represent the hardware on the system and those that represent the software running on the platform. From the possible applications, we randomly selected a series of benchmarks with one, two, or seven applications in each benchmark. When combining multiple applications to create a new benchmark, equal weight was given to each application. We then ran the DPG algorithm and SFA to determine the Pareto points.

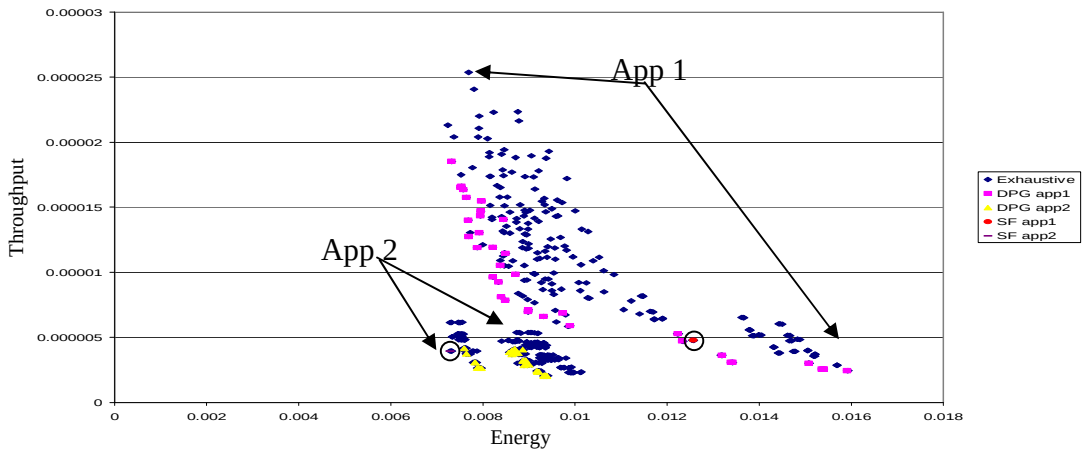
Figure 28 shows the final results from two of the seven benchmarks that we ran on Noxim. Part (a) shows a single algorithm, where both SFA and DPG performed well. All the other single application benchmarks that we examined showed similar results. Part (b) shows a two-application benchmark. In the multi-application benchmarks, the metrics for each application were combined to form the new metrics. However, if both applications need to be run, the designer may need to examine not just the combined performance but how each benchmark individually performs on the platform. Part (b) shows the individual performance of each application. The graph has two clusters of points. Application 1 is the larger curve, and Application 2 is the small cluster of points below Application 1. DPG was able to find a good set of Pareto curves for the combined benchmark as well as the individual applications. By using synthesis which takes on the order of tens of minutes during the exploration process the key feature of our developed The point SFA found is the Pareto point for both applications, however, a single point does not allow the designer many design options.

We also tested the DPG algorithm on the Microblaze platform. For these tests we used Xilinx Platform Studio and the ML310 development board. We examined 64 configurations for 12 benchmarks from the EEMBC and powerstone benchmark [EEBMC] suites. 64 configurations were examined; the parameters examined are the 6 datapath components that can be added to the Microblaze: barrel shifter, floating-point unit, multiplier, divider, MSR, and PCMP instructions. Figure 29 shows the results of the DPG algorithm for four of the benchmarks. DPG was able to find almost all of the Pareto

Figure 28: NoC Pareto graphs, (a) shows a single application. Both DPG and SFA find the Pareto curve. (b) has two applications and DPG found a shared set of Pareto points. SFA found only one configuration, which is circled below.

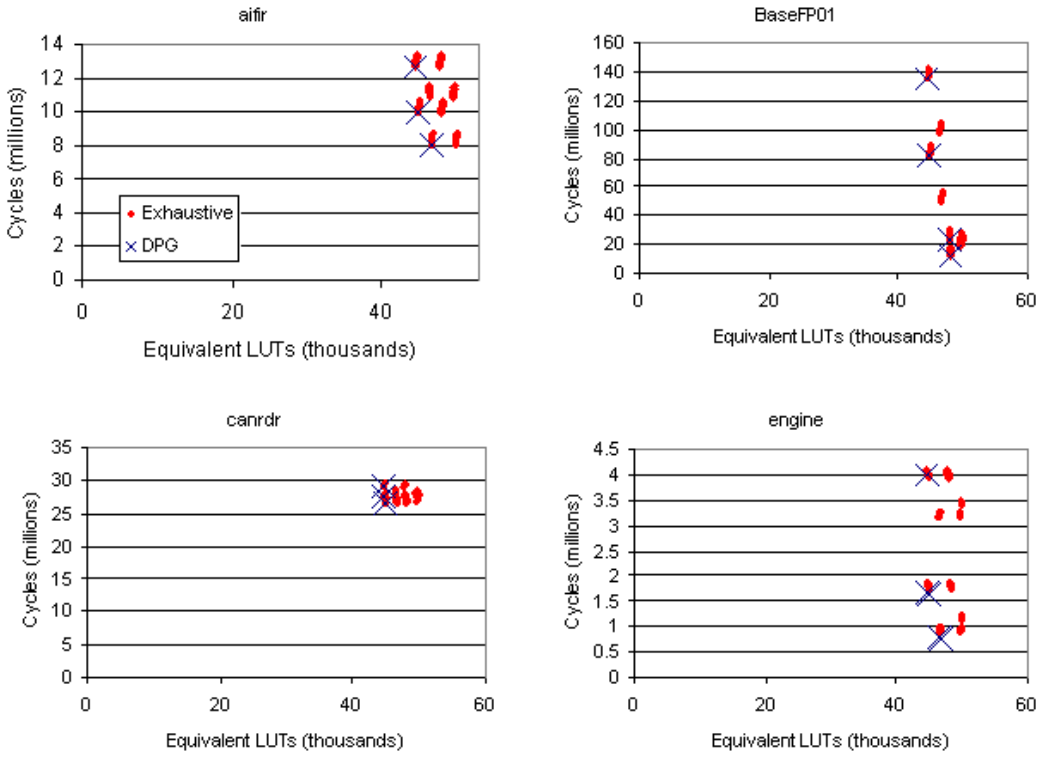


(a)



(b)

Figure 29: DPG results for Microblaze configuration. Four benchmarks, aifir, BaseFP01, canrdr, and engine are shown. These benchmarks are typical of the results seen from the DPG algorithm. DPG finds the complete set of Pareto points in all but the BaseFP01 example, where it misses the point near 50,000,000. Many data points are extremely close in value, when this occurs we added vertical spacing to additional points which creates the vertical bards seen in the Exhaustive graphs.



points, when compared to the exhaustive data set. DPG was unable to find all the Pareto points in the BaseFP01 benchmark. The point around 50 million equivalent LUTs was not found. DPG examined on average 38 configurations, for the examples shown in Figure 29.

The aifir, BaseFP01, canrdr, and engine benchmarks required 38, 44, 38, 38 configurations respectively to generate the Pareto points. The average runtime was 10 hours to complete the exploration, while exhaustive took 16.5 hours to complete. DPG requires some initial configurations for analysis of the platform and application. This analysis step consists of the Design of Experiments Test Runs and Compute Edge Error

Figure 30: Percent of the design space that needs to be explored in the analysis phase of DPG.

Number of parameters	Analysis phase	Total design space	Percent of design space
6	34	64	53.13%
10	67	1,024	6.54%
15	136	32,768	0.42%
20	234	1,048,576	0.02%

stages as shown in Figure 21. As the number of parameters increases, the total design space will increase exponentially, while the analysis phase increases linearly.

Figure 30 shows the growth of both the analysis phase and the design space, as the number of parameters increases. As the number of parameters increases, the percent of the overall design space that needs to be searched decreases quickly. We wanted to show the exhaustive data as a point of comparison, so we limited the total number of parameters. With multiprocessor systems, the benefit of DPG in the Pareto search will become much more pronounced in reduced tool runtimes.

Chapter 5

DoE exploration in the presence of invalid data points

Many configurable systems with a high level configurability have some invalid configurations. An invalid configuration is a configuration where the application is unable to execute using the configuration. Design of experiments assumes that every configuration is valid in the design space. In design spaces where some configurations are invalid modifications are needed in the DoE approach. In this chapter, we will present a method to handle invalid configurations and to predict invalid configurations and avoid the invalid configurations.

5.1 Challenges with invalid data points

FPGAs are an example of a system that has invalid configuration with the design space. FPGAs are used in a variety of cases from general logic to application specific acceleration. To explain the challenges we will use FPGAs as an example.

Off-the-shelf FPGAs support a wide variety of circuit applications. Occasionally, FPGAs are incorporated into an integrated circuit (IC) as a core, to implement a circuit application likely to be revised after the IC is manufactured, accommodating changes in

published standards, bug fixes, or upgrades. The core FPGA supports reprogrammability while obtaining better performance than a microprocessor. Incorporating the core FPGA into the IC rather than using an off-the-shelf FPGA chip can reduce parts, board size, power, and cost. Major FPGA vendors offer core versions of their architectures, though not broadly advertising this feature.

For core FPGAs, tuning the FPGA architecture to the application can reduce size or improve performance compared to the general architecture of off-the-shelf FPGAs. For example, a low-complexity application may only need 4-input LUTs (lookup tables), while a more complex application might benefit from 6-input LUTs. Similarly involves application-specific processors versus general-purpose processors [CHUNHO et al. 2000; CONG et al. 2004]. Application-specific FPGAs were proposed before [HAMMERQUIST 2008][LYSECKY 2008] and are available commercially.

A problem with some configurable systems is that not all applications can be mapped onto all possible configurations of a platform; for example, dense connectivity in a particular application may cause place-and-route on an FPGA to fail for a configuration lacking sufficient routing resources. Such failure results in *invalid points* in the design space. Such points can derail convergence of design space exploration algorithms.

We mapped the core FPGA tuning problem to the DoE paradigm to see what effects the architecture and tool parameters have on application performance and FPGA size. The DoE tool flow for such tuning appears in Figure 31. The DoE tool generates an initial set of “screening” experiments based on allowable exploration time as specified by the designer. The user runs the experiments and provides the resulting performance/size data back to the DoE tool. The tool analyzes the data for parameter impacts. The DoE

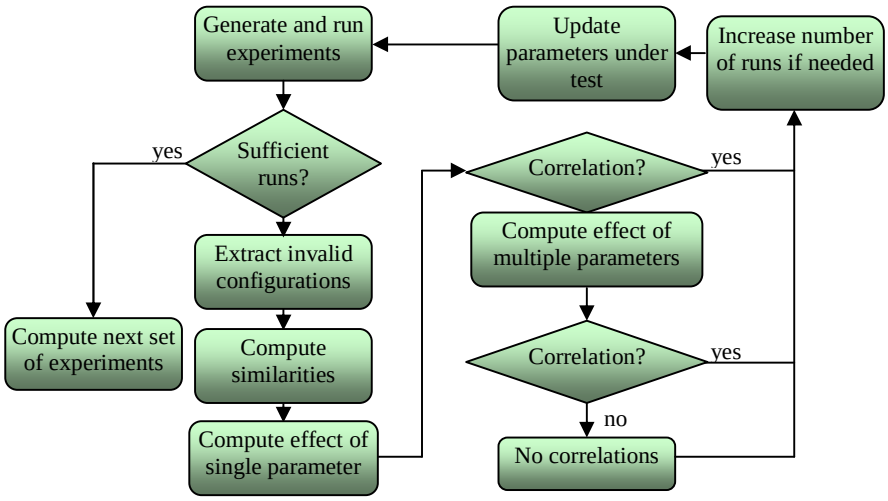
tool uses the information about each parameter to determine the overall impact of the parameter on the design. Then, in this work, the tool predicts the best configuration directly from the impact of the parmeters, ordering parameters by positive impact magnitude and greedily selecting the high or low value which yielded the positive impact. Alternatively, the prediction heuristic could involve running more experiments.

5.2 Handling invalid points

The non-continuous nature of the IC and FPGA domains yields invalid points in the design space. Two types of invalid points exist. The first is a structural invalid point. Structural invalid points are points where the configuration defines an invalid system. For example, an FPGA with 48 inputs but containing only one LUT of 4 inputs is invalid; a single 4-input LUT cannot handle 48 FPGA inputs. The second type of invalid point is a runtime invalid point. A runtime invalid point occurs when the configuration defined is valid but the CAD tools cannot map the application onto the configuration. For example, a configuration may have 20 routing channels, but a tool may not be able to find a routing using less than 30 channels. We assume that the DoE tool does not know why a configuration fails. Rather, the DoE tool merely is informed that the point is invalid. The tool then tries to determine if the invalid points follow a recognizable pattern.

Figure 31 shows the process the tool uses to analyze the invalid points. The tool first determines if the results are usable or if a retest is needed. If the number of valid points is greater than the number of parameters plus one, the tool proceeds to the next stage. However, when the threshold is exceeded, the tool analyzes the invalid points by checking if a parameter's values strongly correlate with the invalid points. To determine if a parameter has a correlation to the invalid data points the tool examines the ratio of

Figure 31: DoE flow to handle invalid configurations.



invalid to valid data points for each parameter. The ratio is developed by counting the number of invalid points for each level of the parameter. The ratios are computed for all parameters. Parameters that have no correlation to the invalid configurations typically have a ratio of one to one, meaning that the number of invalid points is equal for both the high and low levels for the parameter. If the number of invalid configurations is at least 20% more than the number of valid configurations, the tool assumes that the parameter correlates with the invalid points. A correlating parameter is assumed to be responsible for the invalid points. The tool then generates new experiments where the parameter is set to its “good” value which does not cause the invalid points. When no single correlating parameter is found, the tool looks for pairs of parameters that correlate, using an n-squared algorithm. The same method is used when looking for pairs of parameters as when finding a single parameter which correlates to the invalid points.

In many cases, not all invalid points will be explained with the above correlations. When there are unexplained points left, the tool will increase the number of experiments. If the number of unexplained points is less than half of the total runs, then the tool will

Figure 32: Plackett-Burman screening runs. The top table labeled Normal is the beginning of the standard Plackett-Burman set of runs for 11 parameters. The bottom table shows the reverse table, which we use to help handle invalid points.

Normal	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
run 1	1	1	-1	1	1	1	-1	-1	-1	1	-1
run 2	1	-1	1	1	1	-1	-1	-1	1	-1	1
run 3	-1	1	1	1	-1	-1	-1	1	-1	1	1
run 4	1	1	1	-1	-1	-1	1	-1	1	1	-1
…											
run 11	-1	1	1	-1	1	1	1	-1	-1	-1	1
run 12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Reverse	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
run1	-1	1	-1	-1	-1	1	1	1	-1	1	1
run2	1	-1	-1	-1	1	1	1	-1	1	1	-1
run3	-1	-1	-1	1	1	1	-1	1	1	-1	1
run4	-1	-1	1	1	1	-1	1	1	-1	-1	-1
…											
run 11	1	-1	1	-1	-1	-1	1	1	1	-1	1
run 12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

double the number of runs. If the number of unexplained points is greater than half of the total runs, the runs are quadrupled.

DPG uses two methods to increase the number of experiments. The first is to reverse the array. Figure 32 shows an example of how this is done. The top table is the normal method for generating a Plackett-Burman table. To generate the reversed table, the defining run is reversed, from which the remaining runs are generated via left shifts, as before. The last run on the reversed table is the same as the last run in the original table (all -1s), so can be omitted when the two tables are combined.

The second method to increase the number of experiments is to invert the table. The tool uses this method after the above reversal method. The values for each parameter are inverted, i.e., all 1s are changed to -1s, and all -1s are changed to 1s.

When the two methods are combined, the number of experiments is quadrupled. The increased number of experiments allows for a greater number of failed experiments while increasing the final quality of the results. Figure 33 illustrates a screening test for

Figure 33: VPR - Table showing experiments and results for two circuits. INV means that the configuration was invalid

	0	1	2	3	4	5	6	7	8	9	10	dsip	clma
1	1	1	-1	1	1	1	1	1	-1	1	-1		INV
2	1	-1	-1	1	1	1	1	-1	1	-1	1		
3	-1	-1	1	1	1	1	-1	1	-1	1	-1		
4	-1	1	1	1	1	-1	1	-1	1	-1	-1		
5	1	1	1	1	-1	1	-1	1	-1	-1	-1	INV	INV
6	1	1	1	-1	1	-1	1	-1	-1	-1	-1		
7	1	1	-1	1	-1	1	-1	-1	-1	-1	1		
8	1	-1	1	-1	1	-1	-1	-1	-1	1	1	INV	INV
9	-1	1	-1	1	-1	-1	-1	-1	1	1	-1		
10	1	-1	1	-1	-1	-1	-1	1	1	-1	1		INV
11	-1	1	-1	-1	-1	-1	1	1	-1	1	1	INV	INV
12	1	-1	-1	-1	-1	1	1	-1	1	1	-1	INV	INV
13	-1	-1	-1	-1	1	1	-1	1	1	-1	-1	INV	
14	-1	-1	-1	1	1	-1	1	1	-1	-1	1		INV
15	-1	-1	1	1	-1	1	1	-1	-1	1	1		INV
16	-1	1	1	-1	1	1	-1	-1	1	1	1		INV
17	1	1	-1	1	1	-1	-1	1	1	1	1		
18	1	-1	1	1	-1	-1	1	1	1	1	-1		INV
19	-1	1	1	-1	-1	1	1	1	1	-1	1		INV
20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	INV	INV

two benchmark circuits, dsip and clma, on an FPGA with 11 parameters (to be introduced shortly). INV means the tool obtained an invalid result for that benchmark with that configuration of parameters.

The dsip circuit's 20 runs resulted in six invalid points; the 14 valid points are greater than the minimum of 12 needed for basic analysis. The tool will recommend to the user that, time permitting, a new set of 20 experiments be run, but the tool can proceed with the given runs.

The clma circuit's 20 runs resulted in 12 invalid points. The eight valid points are insufficient for a meaningful analysis. The tool seeks a correlating parameter and finds that parameter 4 has the strongest correlation—eight of the ten experiments with parameter 4 at -1 are invalid. The tool sets parameter 4 to 1 and generates a new set of

experiments. Since parameter 4 does not fully explain all invalid points, the tool will double the total number of runs for the new set of experiments.

5.3 Parameterized systems with invalid points

5.3.1 VPR

This section describes our experimental setup for application-specific FPGA tuning with the Virtual Place and Route (VPR) framework using DoE methods. Our objective is to find the best FPGA configuration in terms of critical path or area for each application. Invalid points present a challenge, as previously discussed.

We examined 11 parameters within the VPR framework's flow. The eleven parameters span the three parts of the framework's flow, known as T-VPack, VPR, and the architecture file. Six of the parameters affect the VPR tool and the remaining five affect the FPGA architecture. T-VPack uses three of the architecture parameters to cluster the logic into larger blocks. The VPR documentation lists expected tradeoffs for many of the parameters.

Figure 34 shows each parameter and which part of the flow uses the parameter, and shows each parameter's possible values. A brief description of the parameters follows; for further information consult the VPR User Manual [BETZ 2000]. Below, P means Parameter, and # means number:

- P 0 – Fast place and route
- P 1 – # of iterations per simulated annealing step
- P 2 – # of routing channels available.
- P 3 – Algorithm used to route the circuit
- P 4 – Starting temperature of simulated annealing
- P 5 – Algorithm used to place circuit.
- P 6 – # of inputs per cluster within the FPGA
- P 7 – # of LUTs per cluster
- P 8 – # of I/O ports per row and column of the FPGA
- P 9 – Size of each LUT in the FPGA
- P 10 – Type of switch matrix used in the FPGA

Figure 34: VPR parameters and values used to generate the designs. Two parameters, fast and router_algorithm, only have a low and high setting. Inputs-per-cluster is a special case since the value is dependent on the values of two other parameters, Cluster size (A) and LUT size (B)

					Settings		
#	Parameter	vpr	vpack	arch	Low	Middle	High
0	Fast	X			disable		enable
1	Inner number	X			1	10	20
2	Route channel width	X			50	100	150
3	Router algorithm	X			breadth first		timing driven
4	Initial temperature	X			50	100	150
5	Placement algorithm	X			bounding box	net timing driven	path timing driven
6	Inputs per cluster		X	X	A	(A*B)/2	A*B
7	Cluster size		X	X	4	6	8
8	Pads			X	4	6	8
9	LUT size		X	X	4	5	6
10	Switch type			X	subset	wilton	universal

5.3.2 ISE

We created an experimental setup for application-specific tuning of the Xilinx ISE [XILINX 2011] synthesis process using DoE methods. Our objective is to find a parameter configuration for each application leading to the best critical path. A challenge is that not all configurations are valid for each application being examined, as discussed earlier. ISE has a great number of parameters that can affect the critical path of the final circuit. In this study, we examined 81 of those parameters. The parameters were picked as if the user had no understanding of the parameters. Part of our goal is to show that even with “useless” parameters, the tool can still determine the meaningful parameters and ignore the useless parameters. For example, the “Generate Asynchronous Delay Report” parameter will not have any effect on the final circuit since the parameter only generates documentation for the designer. However, designers may not fully understand

Figure 35: Parameters used in Xilinx ISE experiments

ID	Parameter name	Stage	-1	1
1	Allow Logic Optimization Across Hierarchy	Map	FALSE	TRUE
2	CLB Pack Factor Percentage	Map	33	100
3	Combinatorial Logic Optimization	Map	FALSE	TRUE
4	Extra Effort	Map	None	Continue on Impossible
5	Generate Detailed MAP Report	Map	FALSE	TRUE
6	Ignore User Timing Constraints	Map	FALSE	TRUE
7	Map Effort Level	Map	Standard	High
8	Map Slice Logic into Unused Block RAMs	Map	FALSE	TRUE
9	Optimization Strategy (Cover Mode)	Map	Area	Speed
10	Pack I/O Registers/Latches into IOBs	Map	Off	For Inputs and Outputs
11	Perform Timing-Driven Packing and Placement	Map	FALSE	TRUE
12	Power Reduction	Map	FALSE	TRUE
13	Power Reduction	Map	FALSE	TRUE
14	Register Duplication	Map	Off	On
15	Starting Placer Cost Table (1-100)	Map	1	67
16	Trim Unconnected Signals	Map	FALSE	TRUE
17	Use RLOC Constraints	Map	No	Yes
18	Extra Effort (Highest PAR level only)	Place & Route	None	Continue on Impossible
19	Generate Asynchronous Delay Report	Place & Route	FALSE	TRUE
20	Generate Clock Region Report	Place & Route	FALSE	TRUE
21	Generate Post-Place & Route Power Report	Place & Route	FALSE	TRUE
22	Generate Post-Place & Route Simulation Model	Place & Route	FALSE	TRUE
23	Ignore User Timing Constraints	Place & Route	FALSE	TRUE
24	Place & Route Effort Level (Overall)	Place & Route	Standard	High
25	Placer Effort Level (Overrides Overall Level)	Place & Route	None	High
26	Power Reduction	Place & Route	FALSE	TRUE
27	Power Reduction	Place & Route	FALSE	TRUE
28	Router Effort Level (Overrides Overall Level)	Place & Route	None	High
29	Starting Placer Cost Table (1-100)	Place & Route	1	67
30	Use Bonded I/Os	Place & Route	FALSE	TRUE
31	Add I/O Buffers	Synthesize - XST	FALSE	TRUE
32	Asynchronous To Synchronous	Synthesize - XST	FALSE	TRUE
33	Automatic BRAM Packing	Synthesize - XST	FALSE	TRUE
34	BRAM Utilization Ratio	Synthesize - XST	33	100
35	Case Implementation Style	Synthesize - XST	None	Full-Parallel
36	Case	Synthesize - XST	Maintain	Upper
37	Cross Clock Analysis	Synthesize - XST	FALSE	TRUE
38	Decoder Extraction	Synthesize - XST	FALSE	TRUE
39	Equivalent Register Removal	Synthesize - XST	FALSE	TRUE

the parameters that can be configured and the tool must be able to handle equally well parameters that have a large impact and parameters that have little to no effect.

The 81 parameters are divided into four general groups: translate, synthesize-XST, map, and place and route. The four groups represent the four major stages in the

Figure 36 (cont.): Parameters used in Xilinx ISE experiments

ID	Parameter name	Stage	-1	1
40	FSM Encoding Algorithm	Synthesize - XST	Auto	One-Hot
41	FSM Style	Synthesize - XST	LUT	BRAM
42	Generate RTL Schematic	Synthesize - XST	No	Yes
43	Global Optimization Goal	Synthesize - XST	All Clock Nets	Maximum Delay
44	Keep Hierarchy	Synthesize - XST	No	Yes
45	Logical Shifter Extraction	Synthesize - XST	FALSE	TRUE
46	Max Fanout	Synthesize - XST	500	100000
47	Move First Flip-Flop Stage	Synthesize - XST	FALSE	TRUE
48	Move Last Flip-Flop Stage	Synthesize - XST	FALSE	TRUE
49	Multiplier Style	Synthesize - XST	Auto	LUT
50	Mux Extraction	Synthesize - XST	Yes	Force
51	Mux Style	Synthesize - XST	Auto	MUXCY
52	Netlist Hierarchy	Synthesize - XST	As Optimized	Rebuilt
53	Optimization Effort	Synthesize - XST	Normal	High
54	Optimization Goal	Synthesize - XST	Area	Speed
55	Optimize Instantiated Primitives	Synthesize - XST	FALSE	TRUE
56	Pack I/O Registers into IOBs	Synthesize - XST	No	Yes
57	Priority Encoder Extraction	Synthesize - XST	Yes	Force
58	RAM Extraction	Synthesize - XST	FALSE	TRUE
59	RAM Style	Synthesize - XST	Distributed	Block
60	Read Cores	Synthesize - XST	FALSE	TRUE
61	Register Balancing	Synthesize - XST	No	Yes
62	Register Duplication	Synthesize - XST	FALSE	TRUE
63	Resource Sharing	Synthesize - XST	FALSE	TRUE
64	ROM Extraction	Synthesize - XST	FALSE	TRUE
65	ROM Style	Synthesize - XST	Distributed	Block
66	Safe Implementation	Synthesize - XST	No	Yes
67	Shift Register Extraction	Synthesize - XST	FALSE	TRUE
68	Slice Packing	Synthesize - XST	FALSE	TRUE
69	Slice Utilization Ratio	Synthesize - XST	33	100
70	Use Clock Enable	Synthesize - XST	No	Yes
71	Use Synchronous Reset	Synthesize - XST	No	Yes
72	Use Synchronous set	Synthesize - XST	No	Yes
73	Use Synthesis Constraints File	Synthesize - XST	FALSE	TRUE
74	Verilog 2001	Synthesize - XST	FALSE	TRUE
75	Write Timing Constraints	Synthesize - XST	FALSE	TRUE
76	XOR Collapsing	Synthesize - XST	FALSE	TRUE
77	Allow Unexpanded Blocks	Translate	FALSE	TRUE
78	Allow Unmatched LOC Constraints	Translate	FALSE	TRUE
79	Create I/O Pads from Ports	Translate	FALSE	TRUE
80	Netlist Translation Type	Translate	Timestamp	Off
81	Use LOC Constraints	Translate	FALSE	TRUE

synthesis process. Translate has 5 parameters, Synthesize-XST has 46, Map has 17, and Place and Route has 13. The full list of the parameters is found in Figure 35. The table shows the parameter ID, the parameter name, which stage of the synthesis process the

parameter affects, and the low (-1) and high (1) values used in the experiments, for all 81 parameters. For an explanation of each of the parameters, see the Xilinx documentation [XILINX 2011].

5.4 Results

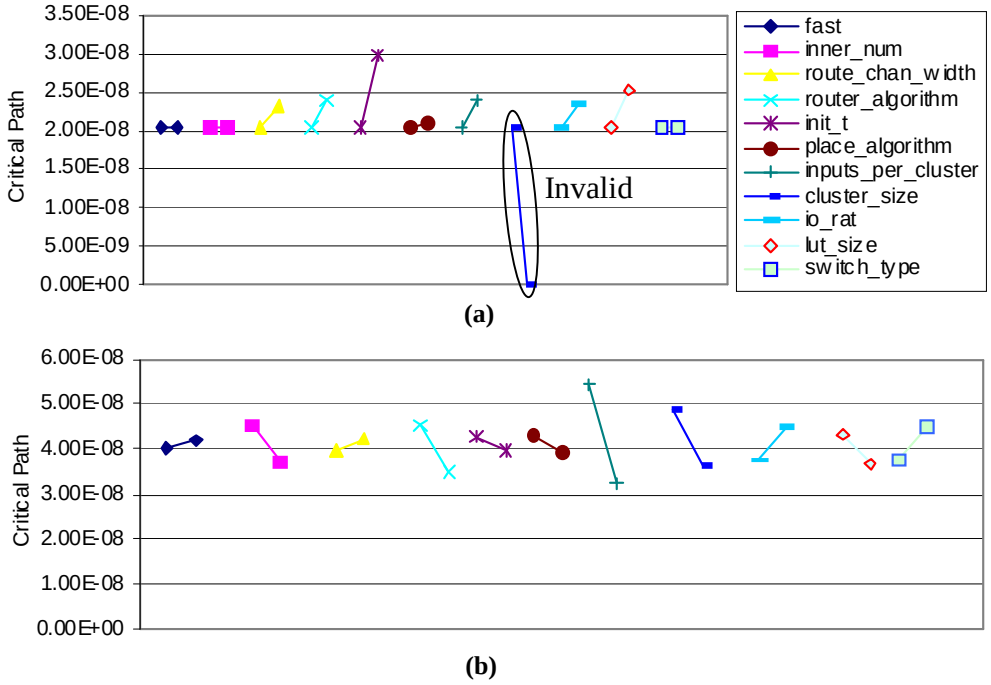
5.4.1 VPR

We used our DoE technique to perform parameter screening on 19 benchmark circuits distributed with VPR. VPR has a total of 20 benchmarks but one of the benchmarks failed on all evaluations, so we have not included it. We also ran single-factor analysis for comparison, using it to generate parameter impacts, and then using the same best-configuration prediction heuristic as used in our DoE approach.

Figure 36 (a) shows the screening results of single-factor analysis for the critical path of the dsip benchmark circuit. Single-factor analysis requires 12 experiments, one for the base configuration and one for each of the eleven parameters. The cluster size parameter (circled) yields an invalid point so its impact can't be determined. The parameter having the largest effect is init_t, the initial temperature for simulated annealing. The lower temperature setting appears to produce a better critical path than the higher setting. That finding is counterintuitive. Normally a higher temperature yields better results at the cost of longer runtime. Single-factor analysis' determination is likely a fluke of the base configuration and not general across other configurations. Note that single-factor analysis determines several other parameters to have little or no effect.

Figure 36 (b) shows results of our DoE screening for the same dsip circuit. DoE determines that all parameters have some effect. DoE determines inputs per cluster to have the largest effect on the critical path. The finding is intuitive. Increasing the number

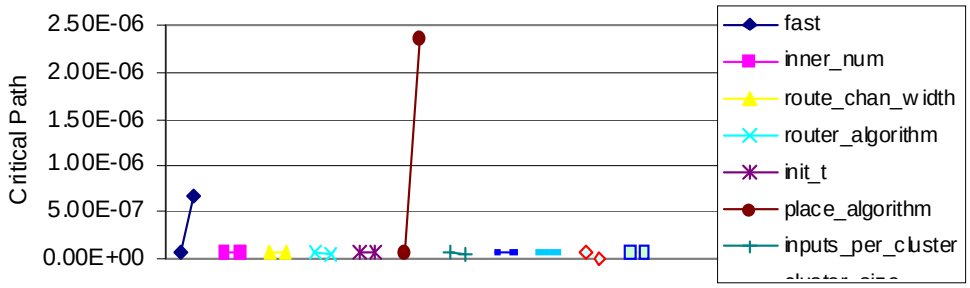
Figure 36: VPR experiment—Determined critical path impacts (in seconds) from a screening test on the dsip circuit for each parameter's low and high settings, as determined by: (a) single-factor analysis, (b) DoE. Note the large differences in the determined impacts between the two techniques.



of inputs per cluster allows for more logic to be placed within each cluster, thus reducing critical path length. The parameter with the second largest effect is the cluster size. This finding is also intuitive. A larger cluster size allows more logic to be placed within a single cluster, reducing critical path length. In contrast to single-factor analysis, DoE's determination of init_t's effect is intuitive, with the higher setting yielding a shorter critical path.

Using the data from the single factor and the DoE screening tests, the tool predicts the best configurations. The new configurations were then run to determine performance and area. The single-factor approach yielded a critical path of 7.10e-8 seconds, while the DoE approach yielded 2.56e-8—a 2.7x speedup, though requiring 20% longer to run the analysis.

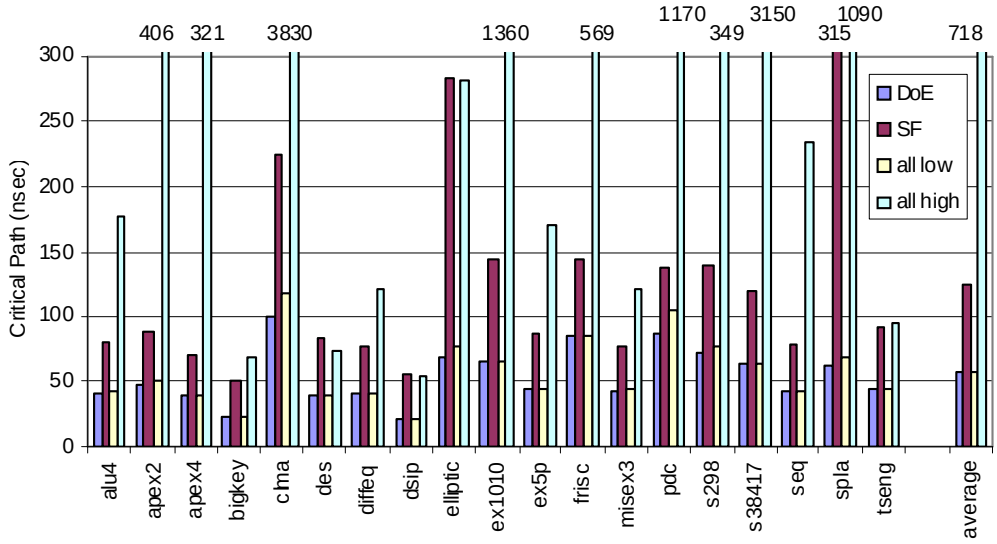
Figure 37: VPR experiment—Determined critical path impacts for dsip as determined by single-factor analysis, this time with a base configuration using low values for all parameters rather than high values. Note the dramatic difference from Figure 36(a); single-factor analysis is very sensitive to the base configuration.



Furthermore, DoE does not require selection of a base configuration, in contrast to single-factor analysis. The earlier single-factor analysis base case had all parameters set to their high values. Figure 37 shows determined critical path impacts by single-factor analysis if instead the base case used the low values for all parameters. The configuration yielded by single-factor analysis yields a critical path of 2.29e-8 seconds in 12 runs. That path is shorter than obtained with the previous base configuration, though DoE still yields a 13% shorter path. Note that the all-low base configuration may not be best for a different tool/platform. In fact, the best base configuration for single-factor analysis could have some parameters high and others low. Finding the best base configuration is an exploration problem itself. Single-factor analysis can be improved by using multiple base configurations with a wider variety of parameter settings; such improvements are precisely the intent of DoE.

We ran the DoE approach on all 19 circuits. 10 of the 19 circuits had sufficient valid points for the analysis. Nine circuits required additional experiments: apex2, clma, elliptic, ex1010, frisc, pdc, s38417, seq, and spla. For each, an additional set of 40 experiments was sufficient to provide the needed data. For none of those 9 benchmarks

Figure 38: VPR experiment—Critical path delays of the tuned FPGA for the 19 circuits.



was a single parameter responsible for all invalid points. The DoE tool, therefore, could not assume that a new set of 20 would be free of invalid points, so the tool doubled the number of experiments from 20 to 40 for the second round. The DoE tool thus required a minimum of 20 experiments to determine the impact of each parameter and a maximum of 60, with an average of 38 experiments per benchmark.

Interestingly, the experiments showed that three parameters had the highest likelihood of causing invalid points: the initial temperature for simulated annealing, the placement algorithm, and the inner number, tending to yield invalid points for their low values.

Figure 38 shows the results from the tuning process for all 19 circuits, for DoE, single-factor, and two static configuration (all low, and all high) approaches. The results shown for single factor analysis use an all-high base configuration. The DoE approach yielded an average 1.3x improvement in the critical path speed versus single-factor analysis, as much as 2.2x for bigkey. The all-low static configuration performed on average almost as well as the DoE approach. The success of the all-low static

configuration is by chance in this experiment. If the designer modified the meaning of the low and high parameters from Figure 34, the results would change. The DoE approach does not rely on such extremes. The extremes in some frameworks are not necessarily good or even valid choices.

We also experimented with optimizing area rather than critical path. VPR provides a slightly optimistic area number when buffers are used, which is what we used here. Figure 39(a) shows determined impacts by single-factor using the all-high base configuration. Figure 39(b) shows DoE-determined impacts. For the b06 circuit when the final configurations were created, the areas of the two circuits were the same.

We ran area optimization on all 19 circuits. DoE outperformed single-factor analysis (all-high base configuration) by 50%, whereas, single-factor analysis was better by 40% for the all-low base configuration.

Figure 39: VPR experiment—Determined area impact (measured in terms of the feature size of the transistors) on the dsip circuit for each parameter's low and high settings as determined by: (a) single factor analysis, (b) a DoE screening test.

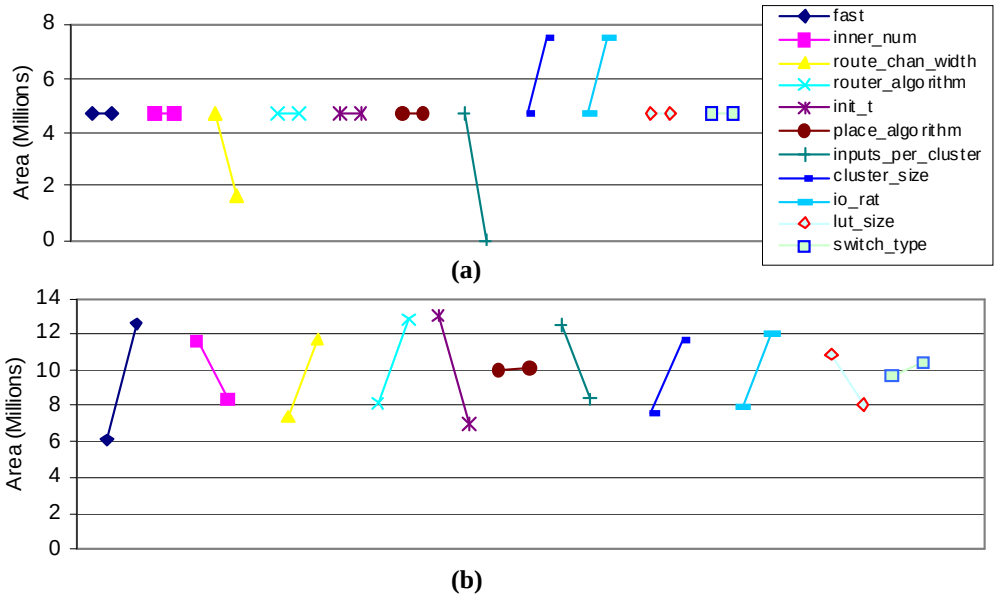


Figure 40: VPR experiment—Tool runtimes for application-specific tuning of an FPGA platform for the 19 circuits.

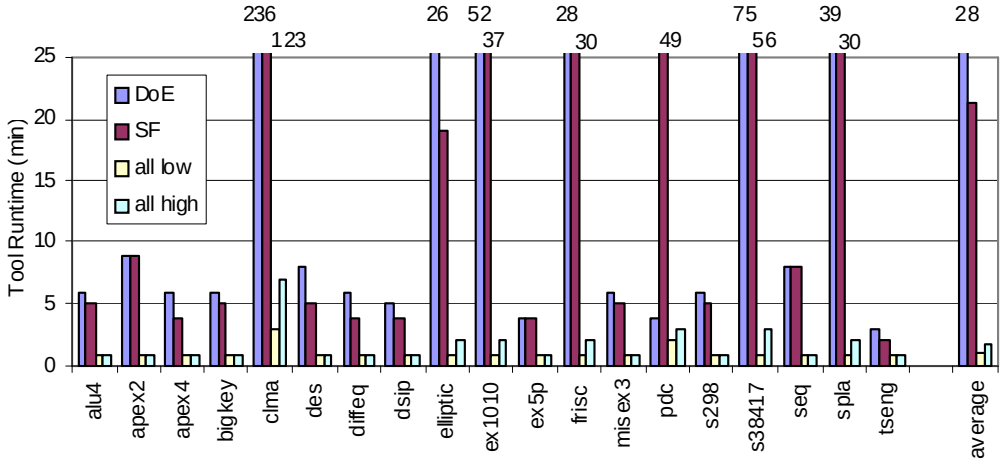


Figure 40 shows the tool runtimes to generate the results. The static configurations required only one run each. On average, our DoE method ran 30% longer than single-factor analysis.

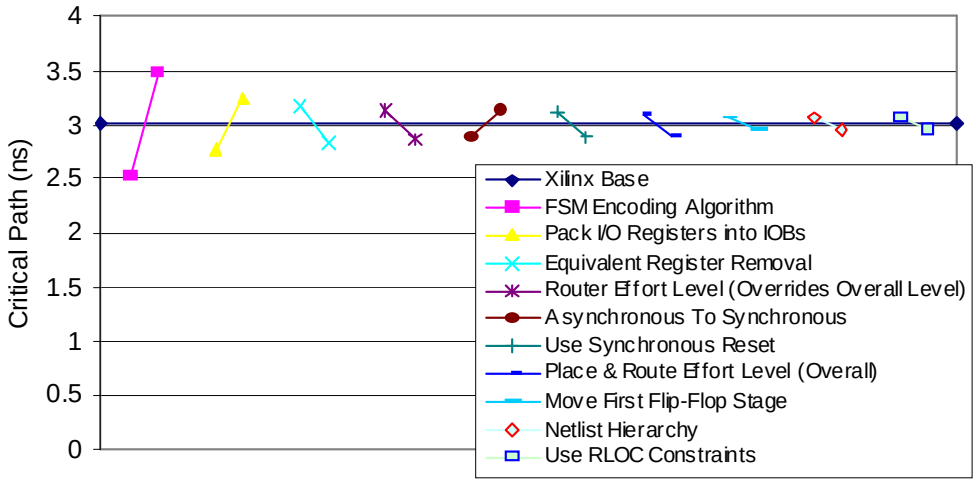
While this paper deals with application-specific FPGA tuning, the same techniques can be used for domain-specific tuning of FPGAs. By running the tests on a suite of applications and combining the results for a given metric, a designer can tune the FPGA to the average case needed by the given domain.

5.4.2 Xilinx ISE

We also applied our techniques to synthesize the circuits for a fixed commercial FPGA, using Xilinx ISE. Xilinx ISE is a synthesis IDE (integrated development environment) used to generate circuits for FPGAs. We used the ITC'99 benchmarks [CORNO et al. 2000]. We selected only the first 11 benchmarks due to time constraints. The benchmarks included simple finite state machines, interrupt handlers, and simple encryption algorithms.

The DoE tool first runs a series of 84 experiments to try to determine the effect of each parameter. For the b06 benchmark, half of the first set of runs resulted in invalid

Figure 41: Xilinx ISE experiment—The top 10 parameters from an interrupt handler, b06.



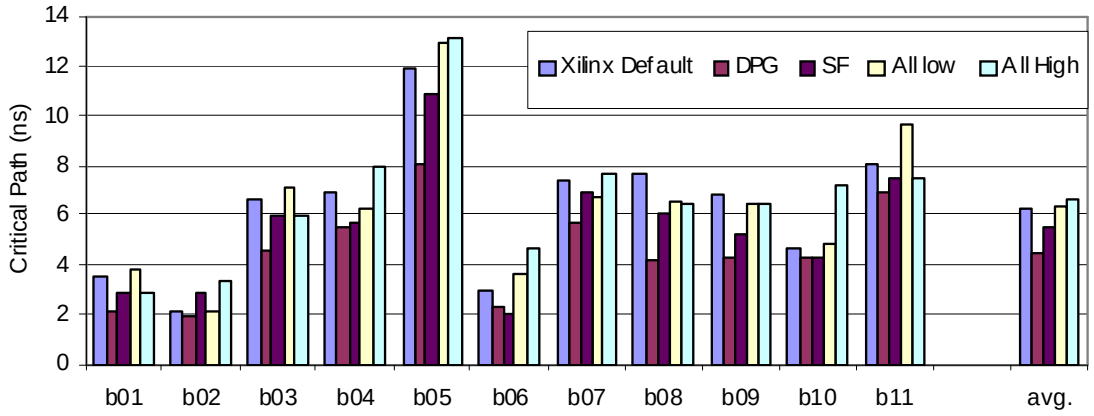
points. Invalid points in the context of Xilinx ISE mean that ISE did not produce a working circuit. The tool found that one parameter correlated perfectly with the invalid points— the "Add I/O Buffers" parameter. When the buffers are disabled, the benchmark failed. After the tool set the value of the I/O Buffer parameter to always include the buffers, the second round of 84 tests was run. The second set of tests completed without any invalid points. After the completion of the 84 runs, the tool examined the results of the 125 valid runs and determined the effect of each parameter on the circuit. All the benchmarks behaved in the same manner. The "Add I/O Buffers" parameter must be enabled for any of the benchmarks to route successfully.

Figure 41 shows the impact of the top 10 parameters on b06, an interrupt handler. The graph shows that the FSM encoding algorithm is the most important parameter, almost by a factor of 2x. After the first parameter, the impact of the parameters drops off, producing a long tail of low impact parameters. Additionally, the graph shows the critical path of the Xilinx default configuration.

Figure 42 shows the comparison between the base Xilinx configuration, our DoE approach, single factor, and two static configurations (all low and all high). The base Xilinx configuration needs to evaluate one configuration, while the DoE approach required 169 runs. The Single Factor approach with a high base case required 82 runs to complete. Single Factor with a low base case only produced one valid run, when the I/O buffers were enabled, which was not sufficient to determine a good configuration. The DoE approach took 3-5 hours to complete for each benchmark. The graph shows that the DoE approach outperformed the other options in all but one case. In one case, b06, single factor was best. The DoE approach on average produced circuits 40% faster than the Xilinx default. In the case of b08, DoE outperformed the Xilinx default by 85%. DoE outperformed single factor on average by 22%.

After examining the impact of the parameters over all the benchmarks, we found the importance of the parameters varies greatly over these benchmarks. For example, in b06 the most important parameter was the FSM encoding algorithm. However, averaged over all the benchmarks, the encoding algorithm ranked in the bottom third in terms of overall importance.

Figure 42: Xilinx ISE experiment—Critical paths delays for the circuits.



Four parameters on average were in the top 10 across all the benchmarks: router effort level, place and route effort level, asynchronous to synchronous, and use synchronous reset. Router effort level and place and route effort level are always in the top 10 across all the benchmarks. The other two are sometimes part of the tail for a few benchmarks. DPG previously predicted that the “Generate Asynchronous Delay Report” parameter should have no impact on the final system and these results were confirmed.

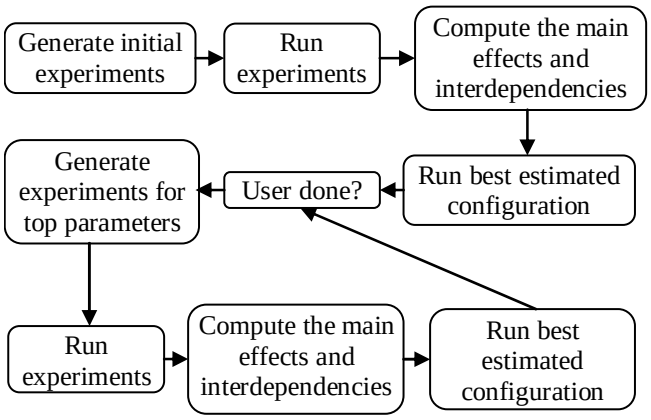
Chapter 6

DPG: Flexible design space exploration for parameterized systems

6.1 DPG (DoE Pareto-point Generator)

We mapped the design space exploration problem to the DoE paradigm, which we call the DoE Pareto-point Generator. The DPG flow for such tuning appears in Figure 43. We developed a methodology that will guide the designer through the search space exploration process. We created a web frontend to the DPG flow, which will allow others to use the DPG flow. DPG works on the assumption that the designer does not always

Figure 43: DoE tool flow for architecture tuning.



understand the effect or interdependencies of the parameters within the system. DPG therefore asks the designer to input only basic information about the parameters in the system. DPG asks the designer to specify what outputs the designer wishes to use to evaluate the system, or the metrics of the system. DPG asks for the number of parameters and the number of levels each parameter has. DPG asks the designer the number of metrics and if there are any weights that the designer wishes to apply. The information DPG requests would be easily available to any designer trying to configure a system. DPG also requests the estimated time for each experiment and the amount of time the designer has available to run the experiments. DPG uses the times to estimate the maximum number of experiments that can be evaluated. DPG allows the designer to specify the total time for exploration unlike most randomized techniques where the overall runtime in unpredictable. DPG's novelty lies in modifying the flow of the DoE-based design space exploration. Previous methods have created a method to run the design space exploration but commonly use either random approaches or a fixed methodology. DPG tailors the search to both the system and the time restrictions of the designer.

DPG uses the parameter, metric and time restrictions to determine the first step in the tuning process. There are two primary sources for variation in the execution of the flow; the number of levels each parameter has and the number of metrics. DPG can handle systems that have two levels for all parameters, or a mixture of parameters with a variety of levels.

DPG has two major phases. The first is a characterization phase, where DPG looks at the parameters and tries to characterize the system under investigation. In the

second stage, DPG navigates the design space. DPG uses up to half the total number of experiments in the analysis phase and the remaining experiments are used to generate the Pareto points.

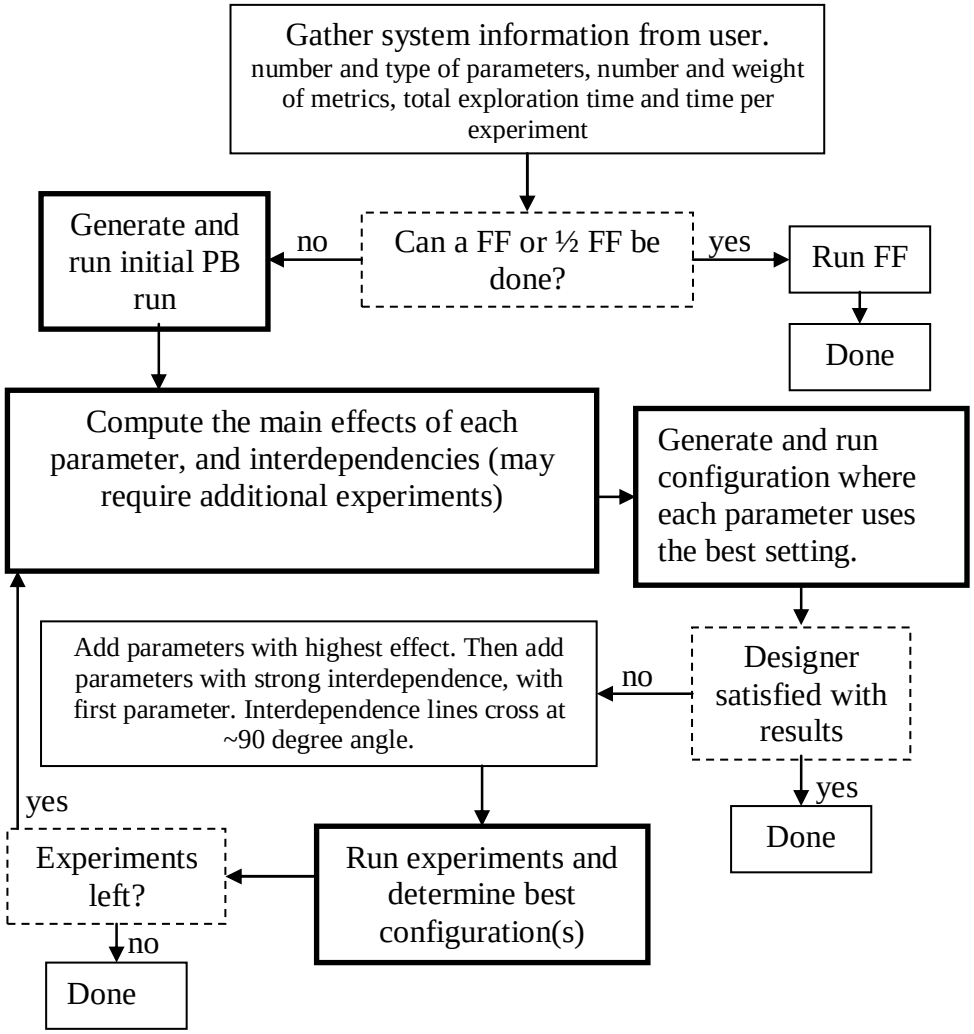
The rest of this section describes the four different paths that our DPG methodology can take depending on the inputs and outputs of the system. Each step of the methodology takes into account the number of remaining experiments which modifies the number of experiments DPG generates in the next step.

6.1.1 2 level parameters – single metric system

A 2 level design is a design where every parameter in the system has only two levels. The two levels are defined as high and low in this work. For unordered parameters, the choice of high and low is up to the designer. As seen in Figure 44, the first step is to determine if an exhaustive or near exhaustive set of tests is possible, given the number of experiments that the designer has available. In many cases, exhaustive runs will not be possible. DPG uses a set of Plackett-Burman experiments as a first step to begin the characterization of the system. The Plackett-Burman experiments are a small set of experiments which grow linearly with respect to the number of parameters. Unlike exhaustive search which grows exponentially, the linear growth of the Plackett-Burman experiments, allow DPG to work on systems with a high number of parameters.

DPG then uses the information gathered from the initial set of experiments to determine the relative importance of the parameters in the system. Figure 45 shows the results of the initial screening tests, for the dsip benchmark using the Platune system. The first parameter has a very small effect compared to the other parameters. Parameters 7

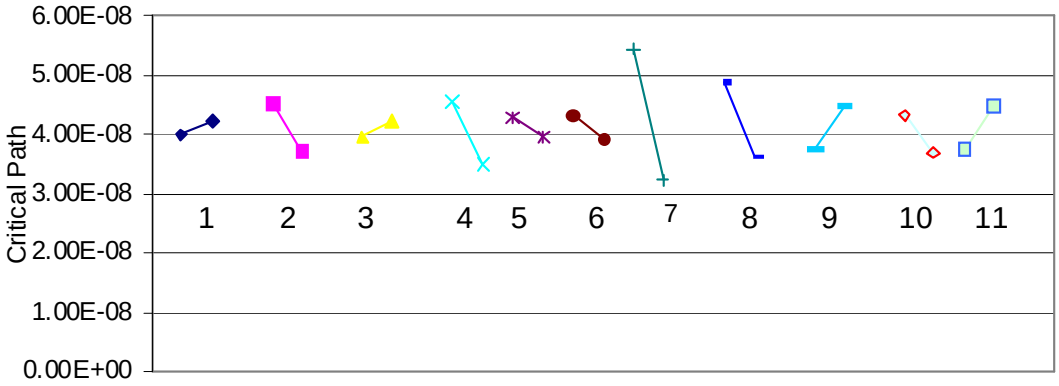
Figure 44: DPG flow using design space exploration. A dashed line marks decision points. The bold stages mark the stages affected by the type of system.



and 8 appear to be most important to the overall system. Parameters 2, 4, 9, 10 and 11 are important but less so than parameters 7 and 8.

DPG then uses the information about the relative importance of the parameters to determine how to proceed in finding the best configuration. DPG will use the most important parameters first in the search for Pareto points. After this first set of experiments, DPG can generate a guess as to the best configuration for the system. DPG uses a simple approach to generate the best configuration. The designer can then run that

Figure 45: Example of the initial screening analysis using DPG Plackett-Burman experiments.



configuration and see if that configuration or any previously seen configurations meet the designer's requirements.

In many cases, the first set of experiments will not produce a design that meets the designer's requirements. DPG will then determine the interdepdencies between the parameters if there are experiments left in the characterization phase. DPG will then create a set of experiments that will build the interdependencies between the important parameters, using the remaining experiments in the characterization phase. Using the above example DPG will look at the interdependencies between parameters 7 and 8. DPG will include the interdependency tests for as many of the parameters as possible given the number of experiments. The interdependency information tells DPG which parameters are interdependent. The interdependencies for the parameters are computed and ranked from the most interdependent to the least. DPG computes the slope for each parameter. If the slopes are equal then the parameters are not interdependent. When the lines are not parallel then the angle that the lines meet determines the degree of interdependence within the system.

After the characterization phase, DPG moves into the second phase where DPG guides the designer through the search space. As shown in Figure 44, DPG builds the first set of experiments by first adding the parameter that has the highest impact on the overall system to the list of parameters that to be examined. DPG then adds any parameters that have a high degree of interactions to any of the parameters currently on the list. For this phase, a high degree of interactions means if the lines cross at an angle of 60 degrees or more. If there are additional experiments left, DPG will then add the next highest parameter that is not already on the list, along with any interdependent parameters. DPG will repeat this process until either there are no more experiments left. The designer can stop the processes at any point if the designer is satisfied with the current results. The goal of the second phase is to find the points in the design space that are on the Pareto frontier. The Pareto frontier is the optimal set of Pareto points [Peterson 1986].

6.1.2 2 level parameters – multi-metric system

When multiple metrics are used is very similar to the single metric version. The main difference occurs when DPG is deciding which configurations are the best. In a multi-metric system, DPG generates the Pareto-points rather than a single point. In the characterization phase, DPG will generate an interdependency graph for the major parameters for each of the metrics. The parameters will affect the metrics in different ways and have different interdependency graphs. With the interdependency graphs for each metric, the graphs are merged together. The merger happens by taking the maximum value from the graphs for each edge. By using the maximum value in the interdependency graph, the graph shows the interdependencies that may have the greatest impact on the system.

The second phase will guide the designer to find the Pareto points for the system rather than the maximum or minimum of the system. In many multi-metric systems, an objective function is used to find the best configuration. However, with an objective function, the designer needs to know the exact relationship between the metrics in order to create the correct objective function. If the designer wants to use an objective function, the single metric flow is available. Pareto points generated by DPG have the advantage that the designer can see the range of each of the metrics and gain a better understanding of how the metrics affect each other. DPG uses the combined interdependency graph to generate the first set of experiments of the parameters that have the greatest effect on the system.

6.1.3 Multi-level parameters – single metric system

In designs that have parameters with more than two levels require changes to how DPG generates the experiments. The initial screening is modified to include a value in the middle of the two endpoints. The additional midpoint helps DPG to understand the shape of the curve for each parameter. DPG will then focus the rest of the characterization phase on looking at the interdependencies of the parameters, as was done in systems with only 2 level parameters. For designs that are a combination of multi-level and 2 level parameters, DPG will not add in the midpoint for the 2 level parameters.

The initial experiment in the second phase is unchanged. After the first set of Pareto points are generated, DPG will continue to generate experiments, until DPG no longer finds any improvements. Once the Pareto-points have stabilized, then DPG will use the remaining experiments and allow all the levels of the parameters to be used.

6.1.4 Multi-level parameters – multi-metric system

In systems that have both multiple levels and multiple metrics, combinations of the two flows are used. The multiple metric flow only affects the analysis of the points generated, while the multi-level flow only affects the generation of the experiments. These changes do not interfere with each other so DPG will combine them.

6.2 Results

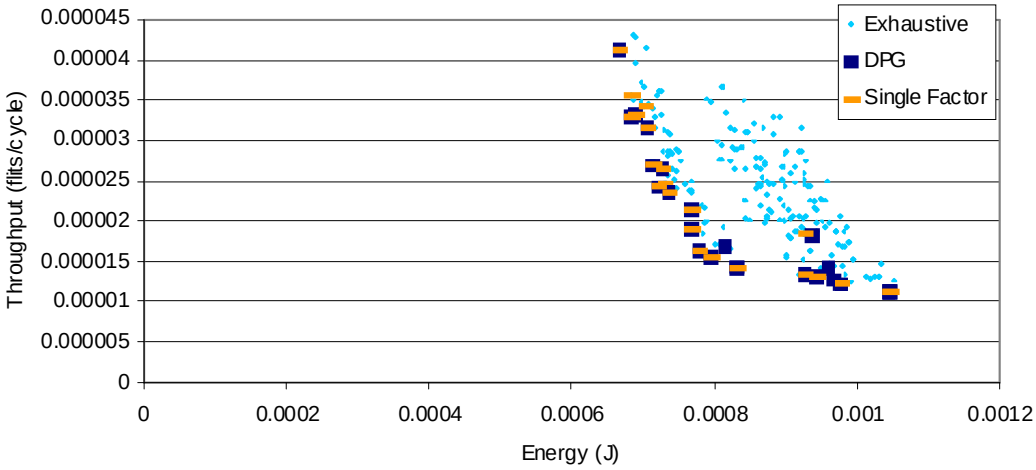
We examined several different types of applications to see how the DPG approach would work in a variety of different system types. We examined three platforms that use the various paths through the DPG flow.

For some systems, we were able to obtain exhaustive search results. Exhaustive search usually take much longer than the designer wishes to spend tuning the system. We obtained such results to evaluate how well DPG was finding the true Pareto points of the design space. The exhaustive data for the systems in this work took weeks to generate.

6.2.1 Network on chip (NoC)

We experimented with Noxim, a Network on Chip simulator [Fazzino 2008]. We evaluated ten of Noxim's parameters. We divided the parameters into two groups, those that represent the hardware on the NoC system and those that represent the software running on the system. From the possible applications, we randomly selected a series of benchmarks with one, two, or seven applications in each benchmark. We combined multiple applications to create new benchmarks. We then ran the DPG algorithm and Single Factor (SF) analysis to determine the Pareto points.

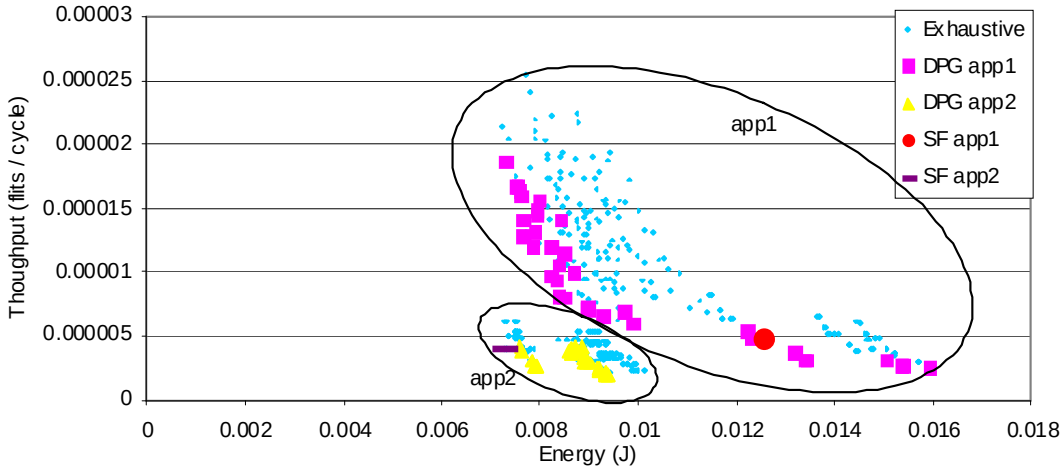
Figure 46: NoC results for a synthetic application 1.



DPG using the multi-metric flow previously discussed generated the Pareto points for the design space. We ran these tests using three different design times. Figure 46 shows the results of DPG as well as the SF results. The graph shows the exhaustive design space for the application. DPG was able to find the Pareto curve for the application using only 50 experiments. The 50 experiments required less than one hour to execute. In this example, the SF method was able to produce almost the same results. We also examined how DPG would perform given two hours and three hours to explore the design space. In this example, DPG generated the same set of Pareto-points. For this example, DPG was able to find all the Pareto-points. When we allowed DPG to run for three hours, DPG had sufficient time to run a half factorial design.

Another scenario we examined was a combined application where two applications are being run but the designer does not know which application will be run more often in the field, so both must work equally well. Since the designer needs both applications to run equally well, both applications must be tuned together. Figure 47 shows the results when the design points are split up so the designer can see the results

Figure 47: NoC results for a synthetic application 4.

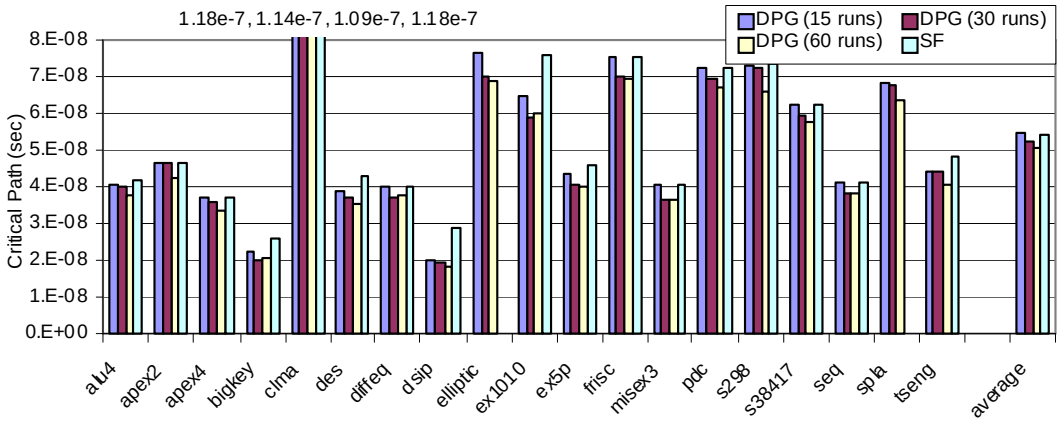


for each application. In this example, DPG was again able to find the same set of Pareto points, in contrast to SF which finds only one point. The point that SF found is on the Pareto curve but a single point does not give the designer options. Again, the full exhaustive data is shown for reference. In this case, the two applications are combined to form a new benchmark, Application 4. The new benchmark contains two applications, Application 2 is the smaller cluster of points below the Application 1 space. For Application 4, we gave DPG two hours and three hours to perform the exploration and in each case, DPG found the same set of Pareto-points. The three hour test was able to use a half factorial to determine the Pareto-points.

6.2.2 FPGA tuning

We examined VPR which is a routing tool for FPGAs. We used eleven VPR parameters to customize the FPGA architecture to the applications. Some of the parameters used are the size of each LUT, number of LUTs per cluster, the type of switch matrix, and the routing algorithm. We used DPG to minimize the critical path for the circuits. In this

Figure 48: Results from VPR for a single metric design.



example, we are using two levels for each parameter and are optimizing the designs for one metric.

We ran DPG three different times with different maximum runtimes for the exploration. Figure 48 shows the results across a range of VPR benchmarks. The three DPG bars show the final critical path generated by DPG for the given number of experiments. Each bar shows the critical path of the circuit after the exploration. The graph shows that DPG was able to perform well across the entire range of benchmarks. By adding in the extra experiments, the 60 experiment version was able to find a critical path that was 7% better than the 15 experiment version, while requiring an increase in the total exploration time from 30 minutes to 6 hours on average. The single factor results are presented for comparison in Figure 48.

6.2.3 System exploration

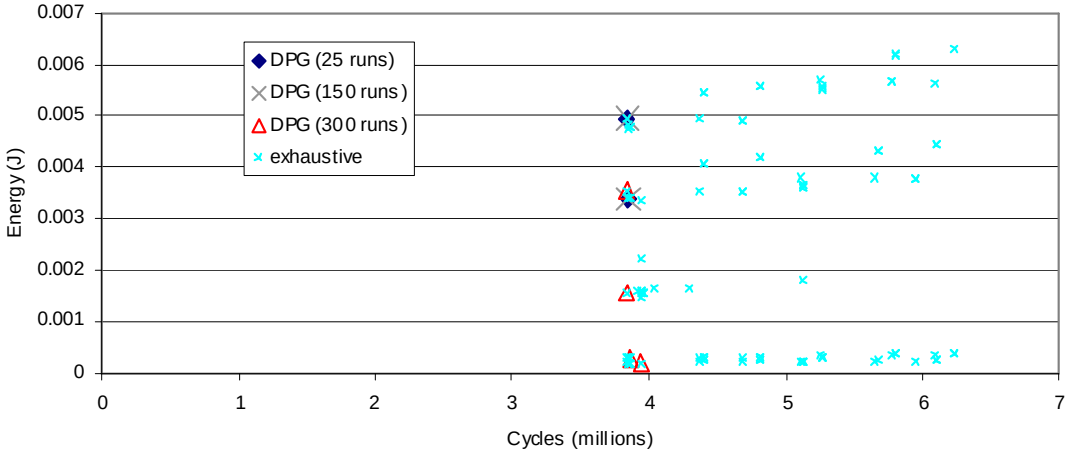
Givargis developed Platune [Givargis 2004] for tuning a configurable system-on-a-chip (SoC) platform. Platune allows the designer to tune the instruction and data caches as well as the buses that connect the processor to the main memory. Due to the complex interactions among the parameters, Givargis deemed a fast equation-based estimation

approach for evaluating a particular configuration to be too inaccurate. Platune instead simulates an application running on a configured SoC to evaluate a configuration. Simulation of one configuration can take seconds to minutes. Due to the enormous configuration space of 4.48 x1012 possible SoC configurations, simulation of all possible configurations for a given application could take decades. Platune thus heuristically prunes the configuration space based on assumptions of independence among parameter groups, allowing the designer to explore less than 0.1% of the configuration space, reducing exploration times to hours in general. Palesi [Palesi 2002] reduced exploration time further using genetic search heuristics, though still requiring tens of minutes to hours. Platune achieves 14x energy reductions by tuning a SoC to a given application, compared to running on a base configuration of the SoC.

All of the above times were for small applications; large applications may require hours per configuration simulation, resulting in weeks or months for exploration.

Platune has nineteen separate parameters that can be configured. Each parameter has anywhere from three to twenty-four levels. The parameters range from cache sizes, cache to memory bus widths, and even voltage levels. Due to the large number of parameters, we need to tune this system in stages. Based on the time required per configuration evaluation and the time available for tuning, we decided to first design a test with twenty-four experiments to determine what factors have a significant impact on the final energy consumption of the system. For the first stage, the parameters are mapped to two level DoE by using the largest and smallest value of each parameter. Using a two level Plackett-Burman experiment, we assign the low level to be smallest valid value for each factor and the high level to be the highest value for each factor. The

Figure 49: Results from the g3fax Platune example.

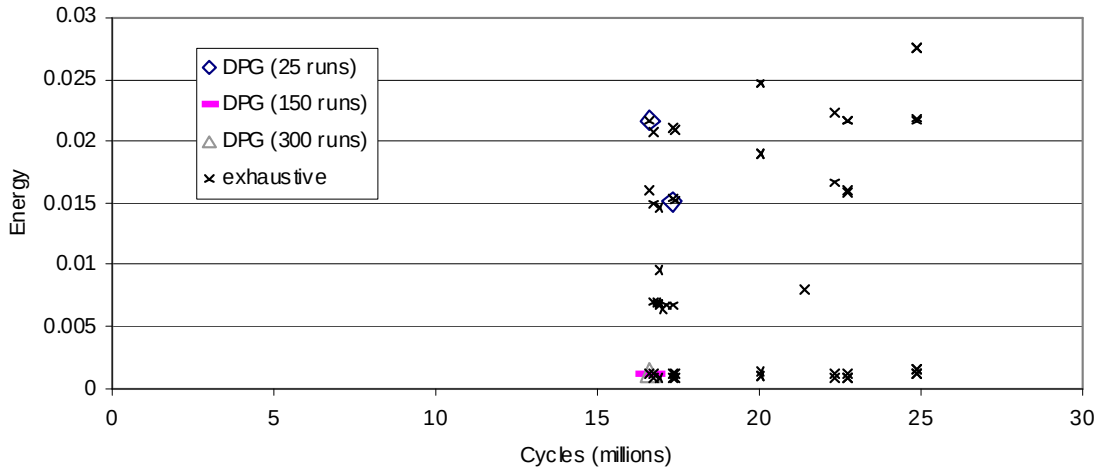


twenty-four experiments yield a set of four to seven factors that have a significant impact on the energy consumption of the entire system. Since we are using the largest and smallest values, if the change between these is not significant, we assume that the parameter does not have a significant effect on the energy of the system.

To determine the best configuration, we need to further examine the significant factors. We therefore ran a second experiment with only the parameters that are found to be significant in the first experiment, with all other factors set to the value that was determined by the DoE tool to be the best for the non-significant values. The second experiment uses a three level experiment using twenty-seven experiments for all the benchmarks. The three level experiments allow better determinations of how each factor or group of factors affects the overall energy of the system. The third level allows DPG to see the design space for each parameter not as a line but as a curve for better estimation of the effect of the parameter.

DPG was used to examine the Platune System Configuration tool [Givargis 2004], where we examined nine parameters each with three levels, and two metrics cycle count

Figure 50: Results from the jpeg Platune example.



plus the energy of the system. We show two representative benchmarks g3fax and jpeg, in three different scenarios. The three scenarios use 25, 150 and 300 experiments, which represent a total design time of 10 minutes, 35 minutes, and 70 minutes. Figure 49 shows the results from three different runs of DPG using different runtimes for the g3fax benchmark. In Figure 49 we see that the small and medium experiment produced the same Pareto points. The large 300 run experiment was able to find a set of Pareto points near the Pareto frontier. The full data set is shown in the graph for reference. In this example, the DPG did not find all the points on the Pareto frontier. However, in the large set DPG found a point that had the same cycle count and was within 0.001 J on the energy axis.

Figure 50 shows the results from the jpeg example. In this case, we see the small set did not perform well but the medium and large experiments found the same point DPG found only one point was needed for the Pareto curve, since the point was better than all the other points in both dimensions. As with the previous example, DPG did not

find the optimal point but was within 0.02 Joules of the minimum energy and the same in terms of the cycles needed.

Chapter 7

Contributions

This work developed a flexible framework for the tuning of parameterized systems, which may include invalid configurations without the designer have any knowledge of how the parameters affect the system and the framework adapts to the designer's requirements as well as the time available for the tuning.

The DPG tool provides the designer guidance for design space exploration. The designer tells the tool the parameters and metrics that are of interest and the tool designs the experiments to determine which parameters are important to the overall system, and how the parameters relate to each of the metrics. The tool takes the time constraints of the designer when designing the needed experiments. After the tool determines the properties of the parameters, DPG begins to calculate the Pareto points for the system. Once the Pareto frontier has been found, DPG will begin to fill in the missing points of the frontier using the additional levels for the parameters with more than three levels.

DPG is able to provide results that are comparable with application specific methods with fewer configurations evaluated. Using the Platune framework, DPG was able to find the Pareto points in 1.5 hours while genetic and Pareto simulated annealing

took about 4 and 6 hours respectively. The Platune method took 44 hours. The different approaches produced similar results.

Some aspects of design space exploration still need improvement. As the number of parameters in a given system grow beyond one hundred the Plackett-Burman runs can not handle that large number of parameters. A modification will be required, probably using a hierarchical approach to divide the system parameters initially to determine the most important parameters. Another area where further study is needed is for parameters that have many levels. The current method uses three points for most of the analysis, which is effective when the parameter is either linear or quadratic. Higher order functions could cause a problem as the tool would not understand the real shape of the curve.

References

[1] Al-Badi, R.; Al-Riyami, M.; Alzeidi, N.; , "A parameterized NoC simulator using OMNet++," *Ultra Modern Telecommunications & Workshops, 2009. ICUMT '09. International Conference on* , vol., no., pp.1-7, 12-14 Oct. 2009.

[2] Altera Corp. http://www.altera.com, 2011.

[3] Abraham, S., B. RauAU. Efficient design space exploration in PICO. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES). 2000.

[4] Ahmed, E. The Effect of Logic Block Granularity on Deep-submicron FPGA performance and density. Thesis, University of Toronto. 2001.

[5] Albonesi, D.H. Selective cache ways: on demand cache resource allocation. Journal of Instruction Level Parallelism. 2002.

[6] ALTERA Corp. Nios II Processors. http://www.altera.com/products/ip/processors/nios2/ni2-index.html, 2005.

[7] ARM. http://www.arm.com. 2011.

[8] Ascia, G., V. Catania, M. Palesi. A GA-Based Desgin Space Exploration Framework for Parameterized System-On-A_chip Platforms. IEEE Transactions on Evolutionary Computation. Vol. 8 No 4. 2004.

[9] Atasu K., L. Pozzi, P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. Design Automation Confrence (DAC). 2003.

[10] Bauer, L. M. Shafique, J. Henkel. Run-time Instruction Set Selection in a Transmutabl Embedded Processor. Design Automation Confrence (DAC). 2008.

[11] Betz, V. VPR and T-VPACK User's Manual (Version 4.30). 2000.

[12] Bossuet, L., Gogniat, G., Bossuet, L., Gogniat G., Phillippe J. Communication Costs Driven Design Space Exploration for Reconfigurable Architectures. The

International Conference on Field Programmable Logic and Applications (FPL). Lisbon, Portugal. 2003.

[13] Bossuet, L., Gogniat, G., Bossuet, L., Phillippe J. Communication-Oriented Design Space Exploration for Reconfigurable Architectures. EURASIP Journal on Embedded Systems. 2007.

[14] Burger, D., T, Austin. The SimpleScalar Tool Set, Version 2.0 www.simplescalar.com, 2011.

[15] Cheung, N., J. Henkel, S. Parameqwaran. Rapid Configuration & Instruction Selection for an ASIP: A Case Study. Proceedings of the Design, Automation and Test in Europe Confrence and Exhibition (DATE) 2003.

[16] Clark, N., J. Blome, M. Chu, S. Mahlke, S. Biles, K. Flautner. An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. Proceedings of the 32nd annual international sysmposium on Computer Architecture (ISCA) 2005.

[17] Cong, J., Fan Y., Han G., Zhang Z. Application-Specific Instruction Generation for Configurable Processor Architectures. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). 2004.

[18] Corno, F., Sonza Reorda M., Squillero G., RT-Level ITC 99 Benchmarks and First ATPG Results. IEEE Design and Test of Computers, July, 2000.

[19] Dall'Osso, M., Biccari, G.; Giovannini, L.; Bertozzi, D.; Benini, L.; , "Xpipes: a latency insensitive parameterized network-on-chip architecture for multiprocessor SoCs," *Computer Design, 2003. Proceedings. 21st International Conference on* , vol., no., pp. 536- 539, 13-15 Oct. 2003.

[20] DOE Pro XL, 2007. DOE Pro FAQ. http://sigmazone.com/doepro_faqs.htm.

[21] Erbas, C., S. Cerav-Erbas, A. Pimentel. Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-on-Chip Design. IEEE trasactions on Evolutionary Computation. 2006.

[22] Fazzino, F., M. Palesi, D. Patti. Noxim: Network-on-Chip Simulator, http:/noxim.sourceforge.net. 2008.

[23] Fin, A. F. Fummi, G. Perbellini. Soft-Cores Generation by Instruction Set Analysis. International symposium on System synthesis. 2001.

[24] Givargis, T., F. Vahid. Platune: A Tuning Framework for System-on-a-Chip Platforms. IEEE Transactions on Computer Aided Design, Vol. 21, No. 11, pp. 1317-1327. 2002.

[25] Gordon-Ross, A., F. Vahid. A Self-Tuning Configurable Cache. Design Automation Conference (DAC). 2007.

[26] Gupta, T. R. Ko, R. Barua. Compiler-directed Customization of ASIP Cores. Proceddgings of the tenth international symposium on Hardware/Software codesign. (CODES). 2002.

[27] Hammerquist, M., Lysecky, R. Design Space Exploration for Application specific FPGAs in system-on-a-chip designs. IEEE International SOC Conference. 2008.

[28] Hauck, S., Compton K., Eguro K., Holland M., Phillips S., AND Sharma A. Totem: Domain-Specific Reconfigurable Logic. IEEE Transactions on VLSI Systems. 2006.

[29] Huang, W. Karthik Sankaranarayanan, Kevin Skadron, Robert J. Ribando, Mircea R. Stan, "Accurate, Pre-RTL Temperature-Aware Design Using a Parameterized, Geometric Thermal Model," IEEE Transactions on Computers, pp. 1277-1288, September, 2008.

[30] Ipek, E., Mckee S., Supinski B., Schulz M., AND Cauana R. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2006.

[31] Joshua, Y., D. Lilja, D. Hawkins. "A Statistically Rigorous Approach for Improving Simulation Methodology," International Symposium on High-Performance Computer Architecture. 2003.

[32] Kathail, V.; Aditya, S.; Schreiber, R.; Ramakrishna Rau, B.; Cronquist, D.C.; Sivaraman, M.; , "PICO: automatically designing custom computers," Computer. vol.35, no.9, pp. 39- 47, Sep 2002.

[33] Kumar R., D. Tullsen, P. Ranganathan, N. Jouppi, K. Farkas. Kumar. R., D. Tullsen, P. Ranganathan, N. Jouppi, K. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In 31st International Symposium on Computer Architecture, ISCA-31. 2004.

[34] Kumar, R., D. Tullsen, N. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. International Conference on Parallel Architectures and Compilation Techniques, PACT, Seattle. 2006.

[35] Kumar, S. A. Jantsch, J. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, A. Hemani. A Network on Chip Architecture and Design Methodology. ISVLSI. 2002.

[36] Larihi, K., A. Raghunathan, S. Dey. Design Space Exploration for Optimizing on-chip communication architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). 2004.

[37] Lee, C., Kin J., Pottonjak M., AND Mangione-Smith W.H., Media Architecture: General Purpose vs. Multiple Application-Specific Programmable Processor. Design Automation Conference (DAC). 1998.

[38] Malik A., B. Moyer, D. Cermak. The MCore M340 Unified Cache Architecture. International Confrence on Computer Design: VLSI in Computers & Processors (ICCD) 2000.

[39] McLean, R., V. Anderson. Applied Factorial and Fractional Designs. Marcel Dekker, Inc. New York, New York. 1984.

[40] Mishra, P., Dutt N., AND Nicolau A. Functional Abstraction driven Design Space Exploration Heterogeneous Programmable Architectures. International Symosium on System Synthesis (ISSS). 2001.

[41] Mohanty, S., Prasanna, V. K., Neema, S., and Davis, J. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. Joint Conference on Languages, Compilers and Tools For Embedded Systems, 2002.

[42] Moyer, B., Tune Multicore Hardware for Software. Xcell Journal, Issue 58, pp 55-57. 2006.

[43] Padalia, K., Fung R., Bourgeault M., AND Egier A., Rose J. Automatic Transistor and Physical Design of FPGA Tiles From An Architectural Specification. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). Monterey. 2003

[44] Palermo, G., C. Silvano, S. Valsecchi, V. Zaccaria. A System-Level Methodology for Fast Multi-Objective Design Space Exploration. Great Lakes Symposium on VLSI (GLVLSI). 2003.

[45] Palermo, G.; Silvano, C.; Zaccaria, V. 2008 "An efficient design space exploration methodology for multiprocessor SoC architectures based on response surface methods," Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on , vol., no., pp.150-157, 21-24

[46] Palesi, M., T. Givargis. Multi-objective Design Space Exploration Using Genetic Algorithms. Hardware Software CoDesign (CODES). 2002.

[47] Petersen, R., Design and Analysis of Experiments. Mercel Dekker Inc. New York, New York. 1985.

[48] Pittman, R. N., Lynch, N. L., Forin, A. eMIPS, A Dynamically Extensible Processor, MSR-TR-2006-143, Microsoft Research, WA, October 2006.

[49] Rigo, S., G. Araujo, M. Bartholomeu, R. Azevedo. ArchC: A SystemC-Based Architecutre Description Language. In proceedings of the 16th Symposium on

Computer Architecture and High Performance Computing (SBAC'04). Foz do Iguacu - Brazil, October 2004.

[50] Robert Schreiber, Shail Aditya, B. Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors* (ASAP '00). IEEE Computer Society. 2000.

[51] Sacks, J., W. Welch, T. Mitchell, H. Wynn. Desogm and Analysis of Computer Experiments. Statistical Science, Vol. 4, No. 4. 1989.

[52] Scott, J., L.H. Lee, A. Chin, A. Arends, W. Moyer. Designing the M*CORE M3 CPU architecture. In Proceedings of International Conference on Computer Design (ICCD). 1999.

[53] Sekar K., K. Lahiri, S. Dey. Dynamic Platform Management for Configurable Platform-Based System-on-Chips. Intl. Conf. on Computer-Aided Design (ICCAD). 2003.

[54] Sheldon, D., Vahid F., Making Good Points: Application-Specific Pareto-Point Generation for Design Space Exploration using Statistical Methods. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). 2009.

[55] Sheldon, D., Vahid F., S. LONARDI. Soft-Core Processor Customization Using the Design of Experiments Paradigm. IEEE/ACM Design Automation and Test in Europe (DATE). 2007.

[56] Sheldon, D., R. Kumar, R. Lysecky, F. Vahid, D. Tullsen. Application-Specific Customization of Paramaterized FPGA Soft-Core Processors. Intl. Conf. on Computer-Aided Design (ICCAD). 2006 (a).

[57] Sheldon, D., R. Kumar, F. Vahid, D.M. Tullsen, R. Lysecky Conjoining Soft-Core FPGA Processors IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Nov. 2006 (b).

[58] Sherwood, T., M. Oskin, B. Calder. Balancing design options with Sherpa. 2004. International Conference on Compilers, Architecture, and Synthesis For Embedded Systems (CASES). 2004.

[59] Shin, C. Y. Kim, E. Chung, K. Choi, J. Kong, and S. Eo. 2004. Fast Exploration of Parameterized Bus Architecture for Communication-Centric SoC Design. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1* (DATE '04).

[60] Silvano, C., W. Fornaciari, G. Palermo, V. Zaccaria, F, Castro, M. Martinez, S. Bocchio, R. Zafalon, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, M. Wouters, C. Kavka, L. Onesti, A. Turco, U. Bondik, G. Marianik, H. Posadas, E.

Villar, C. Wu, F. Dongrui, Z. Hao, and T. Shibin. 2010. MULTICUBE: Multi-objective Design Space Exploration of Multi-core Architectures. In Proceedings of the 2010 IEEE Annual Symposium on VLSI (ISVLSI '10). IEEE Computer Society, Washington, DC, USA, 488-493.

[61] Szymanek, R. F. Catthoor, and K. Kuchcinski. Time-Energy Design Space Exploration for Multi-Layer Memory Architectures. Design, Automation and Test in Europe (DATE), 2004.

[62] Tensilica, Inc. The XPRES Compiler: Triple-Threat Solution to Code Performance Challenges. http://www.tensilica.com/ pdf/XPRES-Triple-Threat_Solution.pdf. 2005.

[63] Viana, P., E. Barros, S. Rigo, R. Azevedo, G. Araujo. Exploring memory hierarchy with ArchC. Symp. on Computer Architecture and High-Performance Computing. 2003.

[64] Wang, A., E. Killian, D. Maydan, C. Rowen. Hardware/Software Instruction Set Configurability for System-on-Chip Processors. Design Automation Confrence (DAC). 2001.

[65] Xilinx. www.xilinx.com.2011.

[66] Yiannacouras, P., J. Rose, J. Steffan. The Microarchitecture of FPGA-based soft processors International Conference on Compilers, Architecture, and Synthesis For Embedded Systems (CASES). 2005.

[67] Yiannacouras, P., J. Steffan, J. Rose. Application-Specific Customization of Soft Processor Microarchitecture. FPGA. 2006.

[68] Zhang, C., F. Vahid, R. Lysecky. A Self-Tuning Cache Architecture for Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS), Vol. 3, No. 2. 2004.

[69] Zitzler, E., L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Study and Strength Pareto Approach. IEEE Transactions on Evolutionary Computation. 1999.

[70] Zuckowshi, P., Reynolds C., Grupp R., Davis S., Cremen B., and Troxel B. A Hybrid ASIC and FPGA Architecture. International Conference on Computer Aided Design (ICCAD). 2002.