POLITECNICO DI MILANO

Scuola di Ingegneria dell'Informazione

Master of Science in

Computer Engineering

DESIGN SPACE EXPLORATION METHODOLOGY FOR COMPILER PARAMETERS IN VLIW PROCESSORS

Supervisor: Prof. Cristina Silvano

Co-Advisor: Prof. Vittorio Zaccaria

Co-Advisor: Dr. Sotiris Xydis

M.Sc Thesis of:

Amir Hossein Ashouri

10327573

Academic Year: 2011- 2012

DESIGN SPACE EXPLORATION METHODOLOGY FOR COMPILER PARAMETERS IN VLIW PROCESSORS

A Thesis
Submitted to the Department of
Computer Engineering
at Politecnico di Milano
in Partial Fulfillment of
the Requirements
for the Degree of
Master of Science in Computer Engineering

Supervisor: Cristina Silvano, Ph.D.
Co-Adviser: Vittorio Zaccaria, Ph.D.
Co-Adviser: Sotirios Xydis, Ph.D.

Amir Hossein Ashouri
DECEMBER 2012

ABSTRACT IN ENGLISH

Embedded systems can be considered as specialized computing systems which can be used for multi-purpose application varying from mobile-phone to military and home-automation devices. Although the functionalities of these devices are differed, the computational structure and design is tightly connected with the platform and programmability in which they rely on. Consequently, by introducing the VLSI technology, designing complex systems-on-chip (SoC) platform and related Network-on-Chip (NoC) has to be finely tuned.

The target is a multi-objective optimization problem: to maximize the performance of the platform and minimize the power consumption or other non-functional metrics. During this design phase, Design Space Exploration (DSE) plays a major role to benefit the designer, to prune the large design space and support the designer during the analysis phase.

The research thesis targets the exploration of compiler options parameters, in order to automatically explore the design space and analyze the compiler-architecture co-design in VLIW processor by applying random design of experiment algorithm. The thesis tackles the aforementioned problem by proposing an automatic methodology based on a tool-chain including the MOST tool(Multi-Objective System Tuner), a Ubuntu wrapper and two open-source compilers; namely, LLVM and VEX. The proposed tool-chain enables the designer to automatically explore, optimize and analyze the options by using several standard benchmarks for both high-end embedded and signal processing applications.

The analysis could be used as a tool-chain for benchmarking the compiler options and expanded to architectural options in the near future. The optimization phase could be done as a further step of the research to generalize the explored trends in the results' analysis.

In this dissertation, the thesis is supported by a large set of experimental results relying on solid sets of statistical analysis which clearly shows the characteristics and the effects of each transformation. We targeted benchmarking with MOST software, VEX and LLVM simulator to provide solid experimental setup. In addition, the Appendix provided a complete hand-manual for designers in order to use as a multiple-purpose reference.

Keywords: Compiler Options, Design Space Exploration, VLIW processors, Compiler Optimizations, DoE, Tool chain Benchmarking

ABSTRACT IN ITALIAN

I sistemi embedded possono essere considerati come sistemi di calcolo specializzati che possono essere usati per applicazioni multi-purpose che possono spaziare da telefoni cellulari fino ad applicazioni militari o di domotica. Sebbene le funzionalità di questi dispositivi siano diverse, la struttura di calcolo e il relativo progetto è strettamente collegato con la piattaforma e il paradigma di programmazione utilizzato. Di conseguenza, introducendo la tecnologia VLSI, il progetto di piattaforme complesse di tipo System-on-Chip (SoC) e della relativa rete di interconnessione on-chip (Network-on-Chip) deve essere dettagliatamente raffinato.

L'obiettivo è massimizzare le prestazioni della piattaforma e minimizzare la potenza dissipata e altre metriche non funzionali del sistema. In tale fase di progetto, l'esplorazione dello spazio di progetto (Design Space Exploration) gioco un ruolo fondamentale per filtrare automaticamente i punti dello spazio di progetto e supportare il progettista nella fase di analisi.

La presente tesi di ricerca ha come obiettivo principale l'eplorazione dei parametri del compilatore, in modo da esplorare automaticamente lo spazio di progetto e analizzare in modo congiunto i paramteri del compilatore e architetturali nei processori VLIW applicando tecniche casuali per il progetto degli esperimenti (Design of Experiment).

La tesi affronta il problema proponendo una metodologia automatica basata su una tool-chain che include il tool MOST (Multi-Objective System Tune), un wrapper Ubunti e due compilatori open-source: LLVM e VEX. La tool-chain proposta consente al progettista di esplorare automaticamente, di ottimizzare e di analizzare le opzioni dello spazio di progetto usando diversi benchmark standard per applicazioni high-end embedded e di elaborazione dei segnali.

La metodologia di analisi proposta può essere usata come tool-chain di benchmarking per valutare i parametri del compilatore e come sviluppo futuro per valutare i paramteri architetturali. La fase di ottimizzazione può essere eseguita come sviluppo futuro del progetto di ricerca per generalizzare gli andamenti evidenziati nell'analisi dei risultati sperimentali.

Nel presente lavoro di tesi, l'approccio proposto è supportato da un ampio insieme di risultati sperimentatli che si basano su un insieme solido di analisi statistiche che evidenziano chiaramente le carattersitiche e gli effetti di ogni trasformazione applicata. L'analisi presenta risultati ottenuti utilizzando la metodologia proposta basata sui tool MOST, VEX e LLM che forniscono un solido ambiente di sperimentazione. Inoltre, nell'Appendice sono raccolti tutti i risultati sperimentali ottenuti nella presente tesi da utilizzare come rifermento per analisi successive.

Parole chiave: Opzioni del compilatore, Esplorazione dello Spazio di Progetto, Processori VLIW, Ottimizzazioni del Compilatore, progettazione degli esperimenti, Tool-chian Benchmarking.

ACKNOWLEDGMENTS

This Master of Science thesis has been carried out at the Department of Electronics and Computer at Politecnico di Milano University. The work has been performed within System Architecture Engineering Group of professors Cristina Silvano and Vittorio Zaccaria who introduced the topic and provided limitless support during the course of the project.

The team work was truly fun and challenging at the same time. I learned a lot and met engineers who shared their knowledge and experience which I am very grateful and would like to thank.
First I would very much like to thank my supervisors at Politecnico di Milano, Cristina Silvano and Vittorio Zaccaria whom always had the answer to all my questions and guided me to the right way. Their constant encouragement and support throughout the project made it possible for me to complete the work.

I would also like to thank Dr. Sotirios Xydis, Post-Doc researcher in the System Architecture Engineering Group; it was a pleasure having his advices and excellent experiences in the field.

Last but not least, I would like to appreciate lifetime support of my perfect family; Mother, Father and younger brother whom always been backbone for me during the hard-times and good-times. Thank you for keep giving me the positive energy to carry-on and thanks for urging me to choose this path for my life.

Finally, I would like to thank everyone in Politecnico Di Milano University circle, from my colleagues, secretaries to the professors, whom got involved in such a way to let this checkpoint of my life happens.

Thank you all,

Amir-Hossein Ashouri

Contents

Table of Figures

List of tables

Chapter 1

Introduction

Increase in speed at which processor are clocked have led to higher performance benefits - applications now run faster; it is now possible to run realistic graphics, interactive games and simulators. This is primarily because of improvements in semiconductor technology in terms of both speed and technology. These processors seek out independent operations/instructions in a sequential program and execute them in parallel to expose what is commonly called instruction level parallelism (ILP). On one hand we could have a processor with large and complex control path and relatively small data path while on the other hand we could have a processor with vice versa configurations. The VLIW processors use the latter approach; making it easy for parallelism and simpler control systems [1].

It is often very difficult to find a single modeling approach or analysis tool which is capable of fulfilling all the challenges of systems-on-chip design. There is a certain need for tuning the chip in order to have the best outcome. Configurable simulation models are used to accurately tune the on-chip architectures and to satisfy the requirements of the target application in terms of performance versus intensity trade-off, battery lifetime and area.

The performance indicators (such as power consumption, delay, area, etc.) are impacted considerably by altering the parameters. The design space exploration (DSE) is an optimization phase which aims at tuning the configurable system parameters to find the best trade-off in terms of the selected figures of merit. The DSE generally consists of a multi objective optimization (MOO) problem and pruning a large design space of parameters. In addition, DSE can be used in the compiler level, tuning the compiler-options in order to exploit the best possible trade-off and even mix those with the architectural parameters such as Cache size, word size, etc.

The overall goal of the DSE phase is to find the optimal parameterized configurations of either architectures and/or applications in order to minimize the number of executing simulations during the exploration phase. So far, several heuristic techniques have been proposed to address this problem; however, they were not efficient enough for identifying the Pareto front of feasible solutions in a reasonable amount of time. That is exactly the main objective and contribution of the dissertation which is going to be elaborated in the following section.

1-1 Dissertation Contribution

The aim of this thesis is to define an efficient tool-chain to explore and analyze the design space formed by the compiler option parameters for ILP processors.

The main contribution presented in this thesis consists of the definition of a multi-objective benchmarking, analysis methodology for compiler options in VLIW processors.

Our study will show a clear way, how to calculate performance and do analysis on these compiler options which is definitely necessary for many purposes such as graphic AGP cards, embedded systems, etc. Within this dissertation, we focus on VLIW (Very Long Instruction Word) processors, which are suitable for low-power embedded high-end computers.

In order to introduce the methodology, first it starts by explaining the status-quo and the background work already presented about DSE and compiler options. Consequently, the tool-chain details will be introduced. In addition, the final methodology and test-bed which has designed to test the performance of these compiler options will be clarified. Furthermore, the experimental results will be introduced, Followed by conclusion and future works and the complete hand-manual appendix.

This dissertation focused on exploration of research field not yet well faced with as a methodology analysis, it describes the performance metrics of the most common compiler options introduced by LLVM in several standard and useful benchmarks.

In order to exploit the best benefits of VLIW processors, there is certainty for tuning the configuration tree based on design space exploration. Therefore, understanding the performance and the pros and cons of each compiler option could play an important role in the era of computational lower-orders tasks.

The methodology proposed, in Chapter 3, has main target to provide best and complete information regarding the compiler options and their benchmarking. Given the increasing complexity of multi-processor system on-chip architectures, a wide range of architecture parameters must be explored at design time to find the best trade-off in terms of multiple competing objectives (such as energy, delay, bandwidth). Therefore, the design space of the target architectures is huge because it should consider all possible combinations of each parameter. The experimental tool in which we used, MOST: Multi-Objective System Tuner [2], under proprietary of Politecnico Di Milano, helps driving the designer towards near-optimal solutions to the architectural exploration problems.

1-2 Dissertation Organization

The structure of this dissertation is as follows; first, the state-of-art and background of the topic is going to be illustrated in Chapter 2. In Chapter 3, the selected compiler option is going to be bolded, then the two open-sourced compilers LLVM [3] and VEX [4] are going to be introduced. In Chapter 4, by introducing the tool-chain and MOST, the methodology is going to be illustrated. Finally in Chapter 5, the experimental results will be shown and will have the conclusion and future works on Chapter 5 and 6. At the end of this dissertation, there will be an Appendix representing the overall results in classified mode.

Chapter 2

Main Background

To better understand the work and methodology, some theoretical points regarding the topic of the dissertation reviewed. In 2-1 Theoretical Background, the main background of the topic such as VLIW architecture, Design Space Exploration (DSE), Performance Models, etc are going to be represented at a glance. Afterwards, in section 2-2 State of Art, recent works regarding the performance evaluation of the compiler options are referred.

2-1 Background

2-1-1 ILP architecture

Instruction level parallelism (ILP) is a family of processors and compiler design techniques that speed-up execution by causing individual machine operations, such as memory load and stores, integer addition and floating point multiplications, to execute in parallel. [5] The operations in which they involve are the normal RISK-style operations, and the program is performing a single program written with a sequential processor in mind. The intrinsic of this technique could lead to improvements in speed, but unlike the traditional multiprocessor parallelism, this action is totally transparent to the users. The prominent example of ILP usage could be found in VLIW [6] architecture and superscalar systems.

The end result of ILP is that multiple operations are simultaneously in execution, either due to the result of having been issued simultaneously in the issue phase or because of having a greater time for completing the execution phase than issuing the successive operation. The classification of ILP could be as following [5]:

- **Sequential Architectures** (without the necessity of conveying any explicit information regarding parallelism. i.e. superscalar processors)
- **Dependence Architectures** (By indicating the dependencies which exist between the operations. i.e. Data flow processors)
- **Independence architectures** (In this architecture, the program provide information as which operation are independent from one another. A good example could be VLIW processors.)

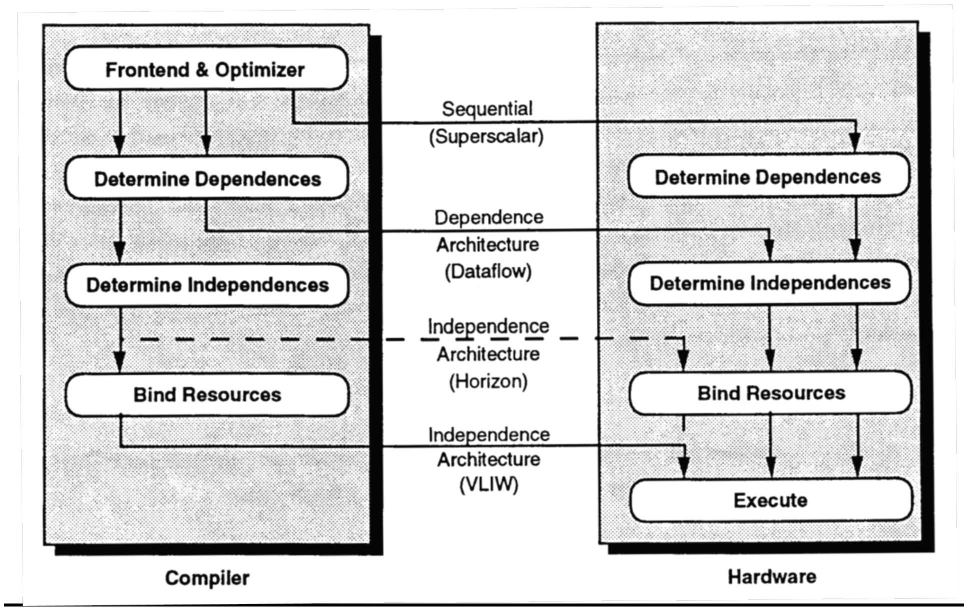


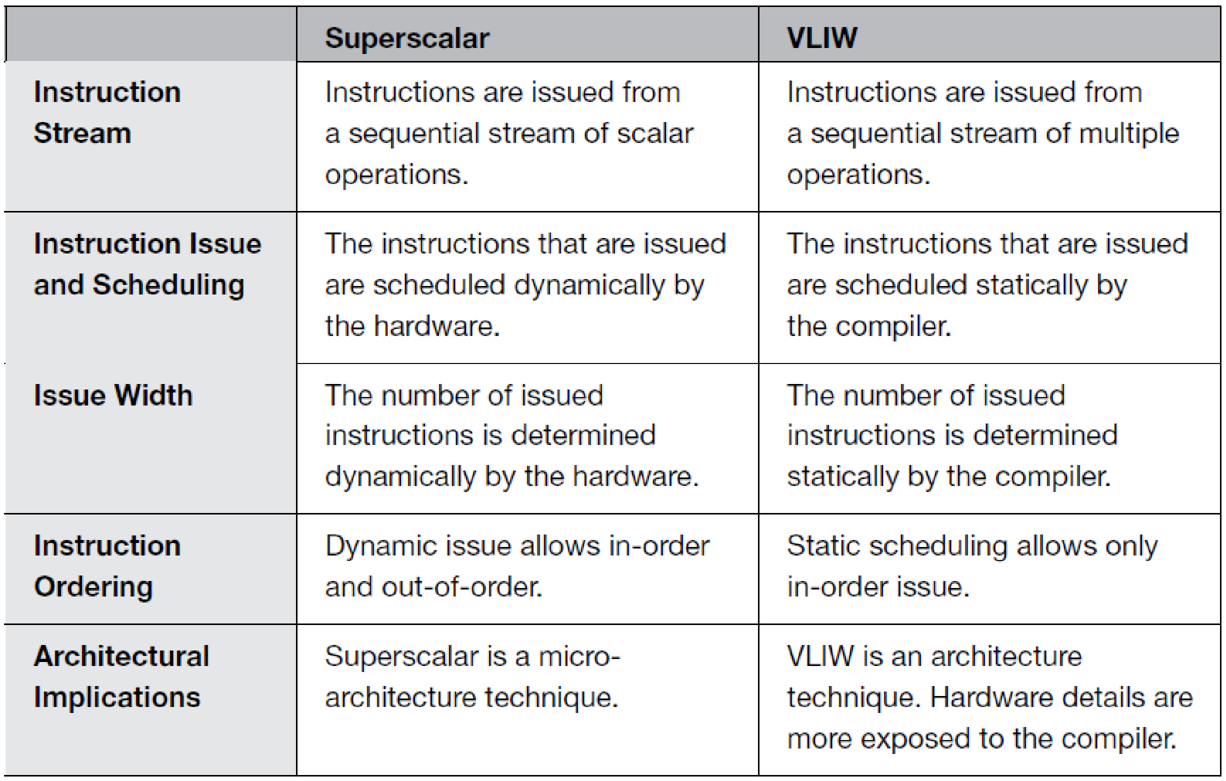
Figure 1- ILP architecutre classifications [7]

2-1-2 VLIW Processor Architecture

Since introducing ILP in 80's, there were lots of systems taking advantage of it. VLIW (Very Long Instruction Word) was more like a design philosophy for a long time. A succinct statement of VLIW philosophy could be "Expose instruction-level parallelism in the architecture" [7] . But it could apply to many levels of the system, including compiler, instruction-set architecture, etc. In addition, parallelism should be revised as it could refer just to run independent task separately rather than in sequentially. We have to take into account lots of interconnection between VLIW and superscalar, VLIW and Compilers, etc.

Recent high performance processors have depended on Instruction Level Parallelism (ILP) to achieve high execution speed. ILP processors achieve their high performance by causing multiple operations to execute in parallel, using a combination of compiler and hardware techniques. Very Long Instruction Word (VLIW) is one particular style of processor design that tries to achieve high levels of instruction level parallelism by executing long instruction words composed of multiple operations. [8]

As an example of differences between superscalar and VLIW, could be the scheduling process; in which superscalar does in via hardware but VLIW have compiler rearrange the code to be executed without changing the hardware. In some processor, there is a special control hardware that examines the operation as it comes from instruction streams. A principal of VLIW has been said as "don't wastes silicon, avoid hardware that computes anything other than the intended computation on the critical path of every instruction" [7]



	Superscalar	**VLIW**
Instruction Stream	Instructions are issued from a sequential stream of scalar operations.	Instructions are issued from a sequential stream of multiple operations.
Instruction Issue and Scheduling	The instructions that are issued are scheduled dynamically by the hardware.	The instructions that are issued are scheduled statically by the compiler.
Issue Width	The number of issued instructions is determined dynamically by the hardware.	The number of issued instructions is determined statically by the compiler.
Instruction Ordering	Dynamic issue allows in-order and out-of-order.	Static scheduling allows only in-order issue.
Architectural Implications	Superscalar is a micro-architecture technique.	VLIW is an architecture technique. Hardware details are more exposed to the compiler.

Figure 2- VLIW and Superscalar Differences [7]

2-1-3 Design Space Exploration

By introducing the VLSI [9] technology, designing complex systems-on-chip (SoC) platform parameters and the network infrastructure on the chip (NoC) of these devices has to be finely tuned. The target is to maximize the performance of the platform and minimize the non-functional costs of the system like Power Consumption, etc. Mapping programs onto configurable architectures is a difficult problem. The set of design choices from which a designer must perform trade-offs in enormous. The designer must detect and exploit characteristics in the sequential application to manage the data movement within the program, determine the data movement in the memory subsystem, and assign system resources to program components to maximize system performance. The large number of degrees of freedom creates a complex design space [10]. This is where Design Space Exploration (DSE) plays the main role to benefit the designer, to prune the large amount of unnecessary design space and actuate the multi-objective problem for the best trade-offs.

Figure 3- (Design Space Exploration General Flow) shows the flow of applying design space exploration. In general, we are interested in finding the solution on each architecture we applied the method. However, quite often it happens that we won't reach the exact and complete solution. There are possibilities in which we reach the succinct point via some algorithms i.e. Simulated Annealing [11], Design of experiment (DOE) [12], etc.

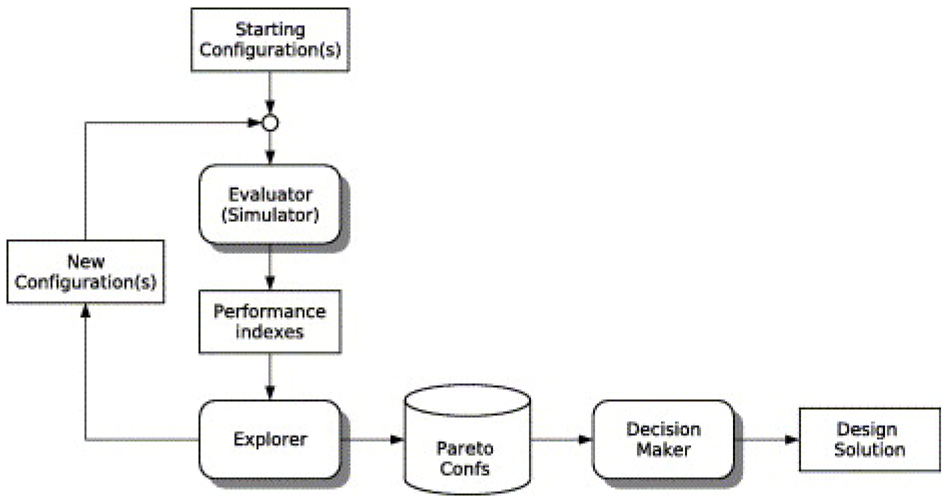


Figure 3- Design Space Exploration General Flow (Courtesy of sciencedirect.com)

2-1-4 Compiler Options

Using more optimized compilers have been always a goal in computer science, however, reaching this goal has its own tolerance and trade-off. Occasionally it happens to sacrifice the code size for better performance or portability versus code size. Consequently, there should be a precaution when using these options otherwise it ends up heavier and less-usable.

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results [13]. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you expect from the source code.

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

The compiler performs optimization based on the knowledge it has of the program. Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them. Not all optimizations are controlled directly by a flag.

Most optimizations are only enabled if an -O level is set on the command line. Otherwise they are disabled, even if individual optimization flags are specified. Generally, there are some levels of optimizations defined in which it could be specified the level and the routine of optimization. The main classifications of GNU [14] C family compilers' optimizations are as following:

- **-O1**

  Optimizing compilation takes somewhat more time, and a lot more memory for a large function. With -O, the compiler tries to reduce code size and execution time, without performing any optimization that takes a great deal of compilation time.

- **-O2**

  Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to clean -O, this option increases both compilation time and the performance of the generated code.

- **-O3**

  Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on some of the other optimization flags like “inline” and “loop_unswitch”. The complete list of Compiler parameters involves with LLVM is being described completely in the following section.

- **-O0**

  Reduce compilation time and make debugging produce the expected results (the default option)

Still there are lots of more optimization flags to be mentioned, but in main stream, the role of using these flags depends on the compiler architecture and its behaviors.

In this dissertation, 15 compiler parameters which aggregated to the popular LLVM capabilities of compiler flags have selected to be used for our analysis on the benchmarks. These are taken from and listed in Table 1-(List of compiler transformations in LLVM) [15]:

Compiler Transformation	**Full Unabbreviated Name**	**Description**
Constprop	Constant Propagation	It looks for instructions involving only constant operands and replaces them with a constant value instead of an instruction.
Dce	Dead Code Elimination	Dead code elimination is similar to dead instruction elimination, but it rechecks instructions that were used by removed instructions to see if they are newly dead
Inline	Function Integration/Inlining	Bottom-up inlining of functions into callees.
Instcombine	Combine Redundant Instruction	Combine instructions to form fewer, simple instructions. This pass does not modify the CFG This pass is where algebraic simplification happens.
Licm	Loop Invariant Code Motion	Attempting to remove as much code from the body of a loop as possible. It does this by either hoisting code into the pre-header block, or by sinking code to the exit blocks if it is safe.
Loop_reduce	Loop Strength Reduction	This pass performs a strength reduction on array references inside loops that have as one or more of their components the loop induction variable.
Loop_rotate	Rotates Loops	A simple loop rotation transformation.
Loop_unroll	Unroll Loops	This pass implements a simple loop unroller.
Loop_unswitch	Unswitch Loops	This pass transforms loops that contain branches on loop-invariant conditions to have multiple loops
Mem2reg	Promote Memory To Register	This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses.
Memcpyopt	Memcpy Optimizations	This pass performs various transformations related to eliminating memcpy calls, or transforming sets of stores into memset's.
Reassociate	Reassociate Expressions	This pass reassociates commutative expressions in an order that is designed to promote better constant propagation
Scalarrepl	Scalar Replacement of Aggregates (DT)	This transform breaks up *alloca* [16] instructions of aggregate type (structure or array) into individual alloca instructions for each member if possible.
Sccp	Sparse Conditional Constant Propagation	Assumes values are constant, Basic Blocks are dead unless proven otherwise, Proves values to be constant, and replaces them with constants and Proves conditional branches to be unconditional.
Simplifycfg	Simplify the CFG	Performs dead code elimination and basic block merging.

Table 1-List of compiler transformations in LLVM

2-1-5 Performance Model and Floating Point

Similar to every other science, the whole attempts should lead to a better performance and lower functional cost. Therefore, there have been lots of different models for performance evaluations regarding the design space and all the matters. Regardless of what model we choose, there is a possibility of misleading us to the fine goal, justifying the right result and mapping them to the experimental one could be the hardest task of each researcher.

Stochastic analytical models [17] and statistical performance models [18] can predict program performance on multiprocessors accurately; however, it is rarely to suggest an insight on how to improve these measurements either for compilers, programs or computers.

In the Section 2-2 (State of Art), some of the recent performance models are going to be introduced, but meanwhile an important model in which the dissertation has been illustrated.

For a given kernel, we can find a point on the X-axis based on its operational intensity. If we draw a vertical line through that point, the performance of the kernel on that computer must lie somewhere along that line.

The horizontal and diagonal lines give this bound model its name. The Roofline [19] sets an upper bound on performance of a kernel depending on its operational intensity. If we think of operational intensity as a column that hits the roof, either it hits the flat part of the roof, which means performance is compute bound, or it hits the slanted part of the roof, which means performance is ultimately memory bound.

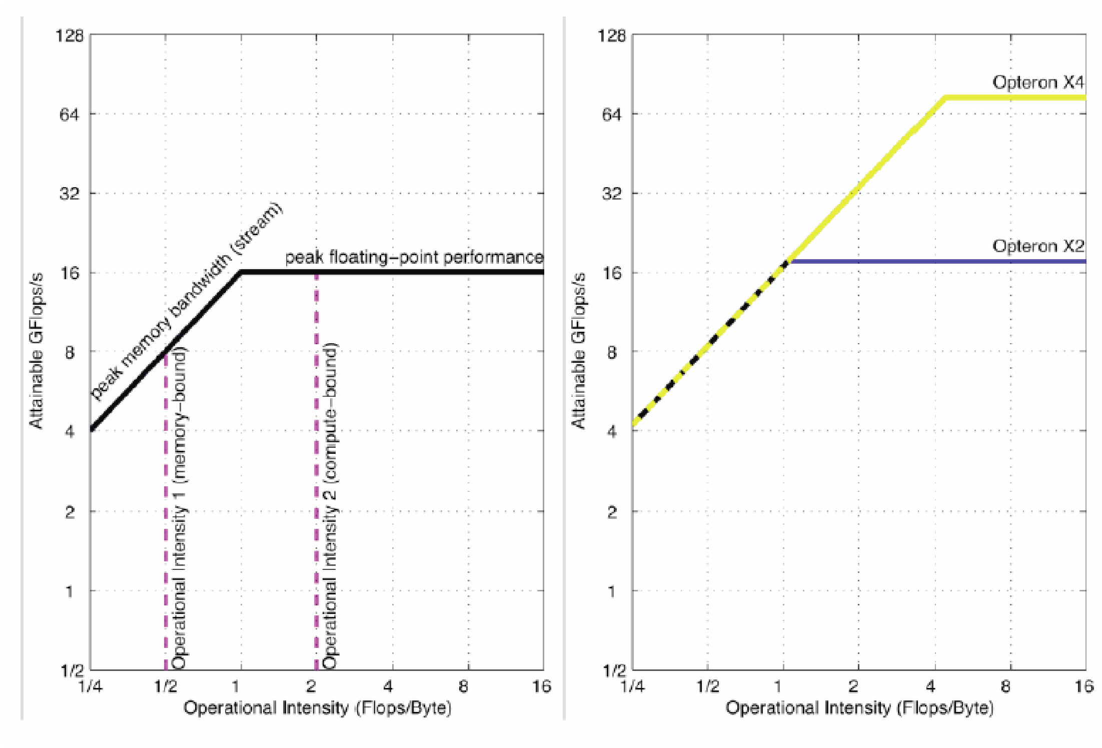


Figure 4-Roofline Model [19]

Consequently, we have to make sure the way we traverse in these areas will be on the verge of higher levels of performance versus intensity; otherwise we hit the roof either in the straight or slanted line and end up being compute and memory bound.

2-2 State of Art

In the field of Design Space Exploration for compilers in VLIW processors, there have been some quality works done recently which in this section are going to be illustrated. However, none of those are exactly applied to the very current topic of this dissertation. This section is presented combinatorial like as for the most important works could be viewed chronologically.

2-2-1 Design Space Exploration for Compiler Options

David. Fischer et al. [20] in their co-exploration work tried to characterize the design space of both compiler frontend (intermediate code optimization) and backend (architecture-specific code generation) that is used in order to do Architecture/Compiler Co-Exploration for the search of optimal architecture/compiler combinations. Their results have been published as a framework entitled, “BUILDABONG”.

A. Halambi et al. in their 2001 work [21], namely “Expression”, designed and introduced a language supporting architectural design space exploration for embedded Systems-on-Chip (SoC) which was capable of automatic generation of a retargetable compiler/simulator toolkit. As a key feature of their work, it could be explicitly being specified for the memory subsystem, therefore some new ways of memory organization and hierarchies were possible. Meanwhile the work wasn’t being self-adaptive for architectural-based compiler flow for each architecture it had the need of specifying the dependencies.

B. So et al. [22] described an automated approach to hardware design space exploration, through collaboration between parallelizing compiler technology and high-level synthesis tool. Their algorithm was to be said to have a quicker search space exploration and could derived a closely matched to best performance model.

M. O’Boyle et al. [23] defined an iterative optimization using machine learning which it uses predictive modeling from the domain of machine learning to automatically focus search on those areas likely to give greatest performance. This approach was independent of search algorithm, search space or compiler infrastructure and scales gracefully with the compiler optimization space size

O. Mencer et al. [24] defined a stream compiler (ASC) which allows users to express and reason about the design space, extract parallelism at each level and quickly evaluate different design choices. They have tested their work with benchmarks like wavelet compression and Kasumi encryption and had optimization in latency and memory usage on both.

C. Dubach et al. [25] went for another solution on the DSE tree. They used machine learning techniques to rapidly explore and predict the design space since it costs a lot of time to explore the tree for each application. This architecture-centric approach used prior knowledge from off-line training and applies it across benchmarks which allowed the model to predict the performance of any new program across the entire micro-architecture configuration space with just 32 further simulations.

2-2-2 Design Space Exploration in VLIW Processors

In the recent years, there have also some works done with the new coming open-sourced compiler for VLIW architecture, namely VEX [4]. One of the benefits of using this compiler is to have degree of freedom in changing the architecture based-on the needs and have the detailed compilation log. It supports 32 bits compilation for native C language with the standard of -C89 and -C99 [26]. As a matter of fact, by introducing the pre-defined scenarios, VEX compiler is capable if evaluating good architectural parameters i.e. total cycles, cache usages, etc.

P. K. Saraswat et al. [27] used simulated annealing for finding the best custom VLIW architecture for GSM decoder application using mentioned VEX compiler. The suitability and the efficiency of the simulated annealing-based Design Space Exploration Algorithm is evaluated and compared against the exhaustive exploration of the complete design space.

In addition, there has been a digital signal processing application done with VEX for a custom VLIW architecture. D. Saptono et al. [28] presented a design space exploration experience for an embedded VLIW processor that allows finding out the best architecture for given application. The proposed method has been implemented and tested using an image processing chain for direct photo printer. The results show a considerable improvement in hardware cost and performance, after identifying the best architecture, they applied a technique to optimize the code in VEX system that uses "inlining" function in order to reduce execution time.

M. Kumar et al. [29] have verified SIM-A Simulator with VLIW based Vex Simulator. Their work discussed the working and configurationally issues involve in Vex Simulator. They have compared the results obtained from VEX and SIM-A simulator in various levels and claimed some inconsistency between those.

Taking into considerations all these appreciated efforts, in the following section, the proposed work is going to be presented.

Chapter 3

Proposed Methodology

The ongoing advances in computer architectures and processors have been led to create a necessity of walking on the right trend in order to comply with the wave. Therefore, applying design space exploration in a right manner plays a vital role in that matter. Therefore, the main contribution presented in this direction is based on the definition of a multi-objective benchmarking, analysis methodology for compiler options in VLIW processors.

As explained in 2-1-3 Design Space Exploration, the variety of parameters both in architectural and compiler side, have made the DSE a huge complex tree to traverse. There is the need to apply further optimizations algorithms to prune the unpromising branches and leafs in-order approach toward the succinct optimal solution. The leaf nodes are the configurations, reaching these points is not as easy as it sounds like, even with the best supercomputers so-far, it takes a lot to calculate the space tree.

3-1 Problem Description

Optimization problems are very common in many design phases of each engineering phases. Nevertheless, understanding the current situation, analyzing the trend and try to find a solution could be pre-phases toward the latter matter.

When we face compiler and architectural options in design space for VLIW processors, we are accounting thousands of parameters in a giant complex tree to traverse. As an example, provided with 15 compiler optimization options, each there are possibilities either to “take” or “exclude”, in addition there are 18 more architectural levels in which there could be a range to taking. Provided with the constraint of taking the integer numbers in between those ranges, we are going to end up having the Table 2- (Our Problem Design Space Exploration_ Example):

No.	Parameters	Possible Values (Integer Range)	Final Outcome
1	Compiler Optimization Parameters	2^{15}	32768
2	lg2CacheSize	[11,30]	22
3	lg2Sets	[0,3]	4
4	lg2LineSize	[5,9]	5
5	lg2ICacheSize	[11.30]	22
6	lg2ICacheSets	[0,3]	4
7	lg2ICacheLines	[5,9]	5
8	CoreCkFreq	[300,500] step=50	5
9	BusCkFreq	[200,400] step=50	5
10	NumCaches	[1,2]	2
11	NumClusters	[1,4]	4
12	IssueWidth	[1,16]	16
13	NumAlus	[1,16]	16
14	NumMuls	[1,4]	4
15	MemLoad	[1,8]	8
16	MemStore	[1,8]	8
17	Memory	[1,8]	8
18	PFT	[0,8]	9

Table 2-Our Problem Design Space Exploration_ Example

The so far mentioned design simply has $5.9868 * 10^{18}$ space size to be explored to each benchmark. Therefore, not applying the right method, definitely leads us to suboptimal leafs. In addition, when we are dealing with these multiple parameters, there is a necessity of using DOE (for design of experiment) in order to sampling the tree. For instance, when it is said, expanding the “inline” compiler parameter, the designer has to take into account the possible manners for each and every design when the inline has been chosen or not (excluded). That is 2 multiply the type of compiler options (which is here 15) added to the exploration problem. Taking into accounts the 15 compiler option each having two phases, it will be

$$2 \times 2 \times \underbrace{\ldots}_{\text{No. of compiler Options}} \times 2 = 2^{15}$$

In this dissertation, the main focus were on the compiler options parameters, therefore the architectural parameters have been assumed as fixed with the values reported in Table 3:

No.	Parameters	Values (Integer Range)
2	lg2CacheSize	16
3	lg2Sets	2
4	lg2LineSize	5
5	lg2ICacheSize	16
6	lg2ICacheSets	2
7	lg2ICacheLines	6
8	CoreCkFreq	500
9	BusCkFreq	300
10	NumCaches	1
11	NumClusters	2
12	IssueWidth	8
13	NumAlus	8
14	NumMuls	2
15	MemLoad	4
16	MemStore	4
17	Memory	4
18	PFT	4

Table 3-Our Design Space Exploration Fixed Arch Parameters

Many different DoEs have been studied for design space exploration; some of them are as follows [12]:

1- **Full Factorial**: experiment all the factors included in the experiment.
2- **Fractional Factorial**: runs a fractioned factored randomly based on the predefined heuristics.
3- **Screening Factorial**: more extreme way of factorial.
4- **Response Surface**: is an off-line optimization, two factors studied usually.
5- **EVOP** : online evolutionary experiments
6- **Mixture** : Based on the context it will add the constraints

Given the large size of our design space, in this dissertation, Fractional factorial which has the randomized selection of experiments has been used. For instance, by running 500 times for each and every compiler options, the system has a good estimation of the whole design space. The algorithm will sample the space equal to the *N* defined in the script, then by using the *Random Effects* option in the scripts, the system divide the sample nodes (here is 500) to two 250 and allocate them for each of the phases (here is two: exclude or include) the interested compiler parameter which to be explored. The other points are being chosen randomly.

3-2 Designed Model

As it abstracted in the section " 1-1-1 Analyzing Compiler parameters ", the opposed methodology of benchmarking the design space exploration for compiler options in VLIW processor was consisted of a built tool-chain (a generic-wrapper), *MOST* [2] (for Multi-objective system tuner), two open-sources C compilers, namely, *LLVM* [3] and *VEX* [4] and some sets of standard benchmarks inside the HP-VEX, namely, ***GSM*** [30] and some benchmarks of *Chstone series* [31], namely, ***Jpeg***, ***Aes***, ***Adpsm*** and ***Blowfish***. The very first benchmark was used for mostly focus on the intensity which is caused to system and the latter's one were mostly used in order to see the high level gate filled up with embedded applications of multimedia.

In this chapter, the detail of the methodology is going to be illustrated. Wherever it is needed for further mathematical backgrounds, there would be a section with that title. The high-level schema of the proposed tool-chain is as following:

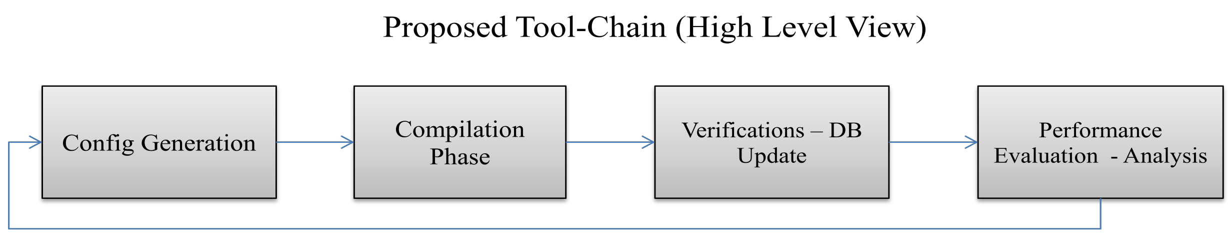

Proposed Tool-Chain (High Level View)
Config Generation
Compilation Phase
Verifications – DB Update
Performance Evaluation - Analysis

Figure 5- High-level View of Proposed Tool-Chain

3-2-1 MOST Generic Wrapper (MGW)

This *Perl*, *Bash* wrapper gets to manage the whole system in order to feed MOST based on the defined settings, i.e. design space exploration settings for compiler and architectural, iterations inputting the benchmarks, etc, and subsequently get the output results and import it to the database of MOST, initiating the next run for that matter. It has a randomized function which randomly generates the input points MOST needs for running the benchmark. As it mentioned in the Section 3-1 (Problem Description), the *DoE* methodology in which it has been used was randomized factorial, therefore in order to avoid the gigantic design space tree caused by the parameters calculated in the Section 3-1 (Problem Description), there should have been a generator for these points at the beginning.

MOST GENERIC WRAPPER [32] (MGW) is a Perl wrapper designed to simplify the integration of tools for the design space exploration (DSE) phase by using MOST. It hides most of the integration details in term of MOST XML input/output files (except for the XML Design Space description file) providing to the designer a simpler way to integrate its problem in MOST. The execution config file includes 3 main sections:

- **Input files declaration**: This section is used to let the MGW what are the input parameters and where to include the values in those files.
- **Output files declaration:** This section is used to let the MGW what are the output files where to read the metrics and how to read the values. The section is composed by several lines, one for each metric declared in the XML design space definition file.
- **Execution script:** It should include all the commands needed for the generation of the output files (including the metrics).

A simple example of initiating the MGW is shown below:

Input File Declaration	Output File- Coordinates	Execution script

[...]
Core numeber = 4
ICache size = 2048
DCache size = 4096
Bus size = 64
[...]
	type;hitRate[%];Accesses; power [mw];
	icache; 97.9; 10401; 145;
	dcache; 83.1; 8300; 132;
	L2cache; 76.3; 3219; 347;
		#!/bin/sh
		set -e
		echo "requests 438 " > output.txt
		echo "accesses
		@__MOST_GENERIC_WRAPPER__param1__@
		" >> output.txt

Table 4- MGW sections's Example

A simple schematic view of the system is drawn as in Figure 6 - (Proposed Tool-Chain Schematic):

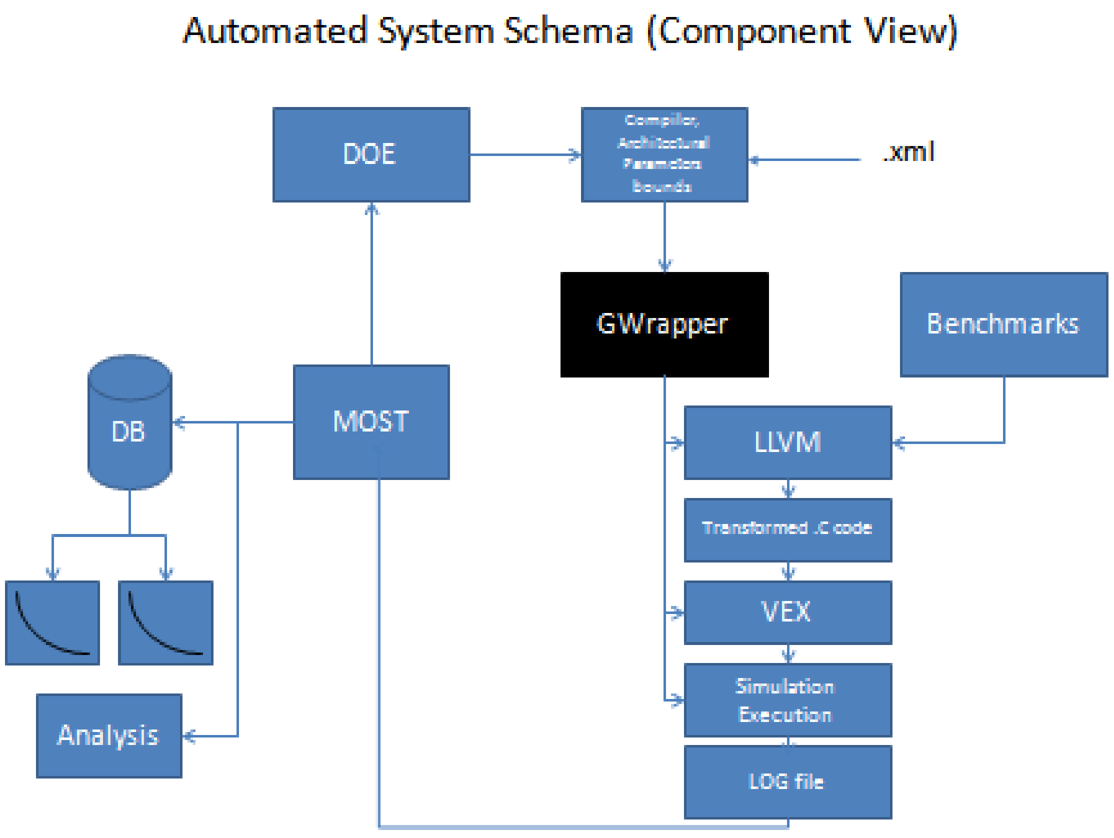


Figure 6-Proposed Tool-Chain Schematic

The proposed methodology has been defined and designed for multiple-benchmarks and only inputs the benchmark and settings for the faster and cleaner explorations. In other word, as it will be shown in the Experimental Results, it is able to input multiple benchmarks from high level synthesis to high performance and explore, analyze and synthesize the system.

3-2-2 Multi-Objective System Tuner (MOST)

MOST is a tool for architectural and compiler design space exploration [2] [33]. It is an interactive program that lets the designer explore a design space of configurations for a particular architecture for which an executable model or driver exists. It can be also extended by introducing new optimization algorithms such as Monte Carlo optimization, sensitivity based optimization, etc. For instance, Taguchi design of experiments [34].

The overall goal of this framework aims at providing a methodology and a re-targetable tool to drive the designer towards near-optimal solutions to the architectural exploration problem, with the given multiple constraints. The final product of the framework is a Pareto curve of configurations within the design evaluation space of the given architecture. To meet this goal it has been implemented a skeleton for an extendible and easy to use framework for multi-objective exploration.

The strength of MOST is that drivers and optimization algorithms can be dynamically linked within MOST at run-time, without the need of recompiling the entire code base. This is supported by well defined interfaces between the driver and the optimization algorithms versus the kernel of MOST. The proposed DSE framework is flexible and modular in terms of: target architecture, system-level models and simulator, optimization algorithms and system-level metrics.

3-2-2-1 MOST Structure

The Overall structure of MOST can divide its modules into three different categories: [33]

1. **<u>MOST internal modules</u>**: They are represented in blue in figure 2. Those modules are internal to the MOST structure. They are composed by the MOST Kernels, the MOST shell, the MOST internal database management and the design of experiments and optimization modules. In the following, each module is described more in detail:
    a. **<u>The MOST Kernel</u>** engine represents the core of the design space exploration tool. It orchestrates the optimization process by invoking the constituent and inter-changeable blocks of the framework.
    b. **<u>The MOST shell</u>** is the command line interpreter. By using this shell (or equivalent *batch* scripts) it is possible to specify the optimization problem and the related exploration strategy. This particular interface is suitable for remote execution of design space exploration on server farms. The **MOST** interpreted language gives now the possibility to define complex objective functions.
    c. **<u>The MOST Internal Database Manager</u>** is used to store all the results coming from simulations. Moreover, it is used for combining metrics values (as estimated by the simulator) into objective functions, to train analytical models (RSM) and to generate output reports of the exploration process.
    d. **<u>The design of experiments and optimization modules</u>** are the basic components for building the exploration strategies. The internal organization of the software has been factored in order to provide standard and common APIs for the various modules associated with the fundamental functionalities of MOST. The standard API consists of a corresponding dynamic linkable object interface which can be used to develop new models, aside from the existing ones.
2. **<u>MOST External Modules:</u>** Those modules are within the MOST packages but are composed of external executables that will be called through the MOST interfaces. In particular, they are represented by the response surface models.
    a. **<u>The response surface models (RSM)</u>** are used for building analytical models of the target system response. A similar standard data interchange format (as previously done for DoE and optimizers) is used for supporting the introduction of response surface models in **MOST**.

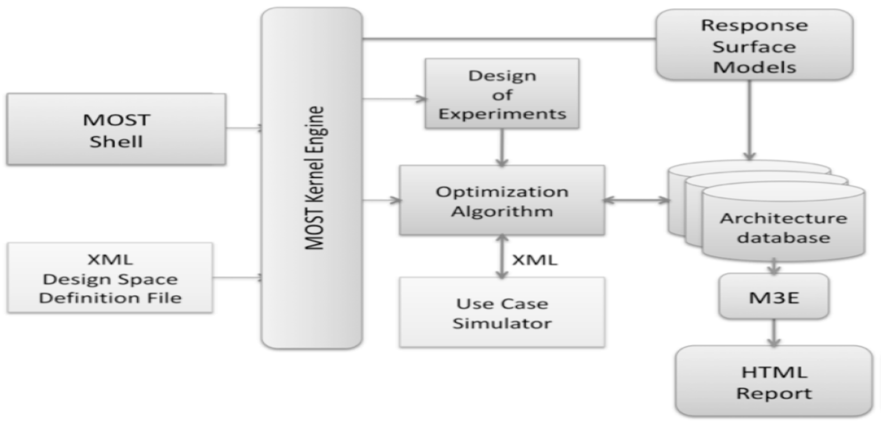


Figure 7- MOST Schematic (Courtesy of Dr. Vittorio Zaccaria, Politecnico Di Milano)

As mentioned in the Section 3-1 (Problem Description), the DoE used in this dissertation was based on Random factors which generated a set of random designed points. In addition, the optimization algorithm used here was *parallel DoE (PDoE)* [12] which was based on the possibility of performing concurrent evaluation of the different design points. Consequently, in these experimental analyses, for each benchmark compiler option, the number of exploration was 500. It would enough points for the system to use for DoE and Optimizer to generates the effects and metrics beside the Pareto points (if exists).

3-2-3 LLVM

LLVM is a collection of modular and reusable compiler and tool-chain techniques. LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of different subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "UIUC" BSD-Style license [3].

The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs. Therefore, for this dissertation research it was chosen as the C code optimizer which transformed the native C codes of the benchmarks to *transformed.c* and let the second compiler in chain (HP-VEX) used it as the feed.

As an overview, some of the LLVM features could be as following [35]:

- Front-ends for C, C++, Objective-C, Fortran, etc
- A stable implementation of the LLVM instruction set, which serves as both the online and offline code representation, together with assembly (ASCII) and byte-code (binary) readers and writers, and a verifier.
- A powerful pass-management system that automatically sequences passes (including analysis, transformation, and code-generation passes) based on their dependences, and pipelines them for efficiency
- A wide range of global scalar optimizations
- An easily re-targetable code generator
- APIs and debugging tools to simplify rapid development of LLVM components
- A test framework with a number of benchmark codes and applications
- 64bits C code transformer

3-2-4 HP-VEX

VEX ("***V****LIW* ***Ex****ample*" [4]) is a compilation-simulation system that targets a wide class of VLIW processor architectures, and enables compiling, simulating, analyzing and evaluating C programs for them.

VEX system include three main components [36]:

1. **The VEX Instruction Set Architecture**. VEX defines a 32-bit clustered VLIW ISA that is scalable and customizable to individual application domains. Scalability includes the ability to change the number of clusters, execution units, registers and latencies; customizability enables users to define *special-purpose* instructions in structured way.
2. **The VEX C Compiler**. It is a robust, ISO/C89 compiler that uses *Trace Scheduling* [37] as its global scheduling engine. A very flexible table-like machine model determines the target architecture. For VEX, we selectively expose some of the parameters to allow architecture exploration by changing the number of clusters, execution units, issue width and operation latencies, without having to recompile the compiler.

3. **The VEX Simulation System**. The VEX simulator is an architecture-level (functional) simulator that uses *compiled simulator* technology to achieve a speed of many equivalent `MIPS'. A simple built-in cache simulator (level-1 cache only), and an API that enables other plug-ins used for modeling the memory system.

VEX has the capability of writing output log files based on the architectural parameters; i.e. No. of cycles, No. of stalls, etc. This is the base of mathematical calculations and metrics for MOST databases.

Got to be mentioned there is a problem aroused by using VEX after LLVM since LLVM compiler feed the VEX with 64 bits of compiled, transformed code. In some of the benchmarks, lots of efforts have been issued to fix and make those in-chain output-inputs compatible to each other.

3-3 Benchmarks

As mentioned in the Section 3-2 (Designed Model), there is variety of benchmarks that have been used to expand the usability of the proposed methodology in this dissertation. The higher level embedded applications like JPEG to more complex ones like GSM. The selected set of benchmarks is composed of:

1. GSM
2. AES
3. ADPCM
4. JPEG
5. BLOWFISH

There is a necessity of explanation here about some of the differences of "*Intensity*" the parameter in the next chapter results which are the difference between the target applications. In another word, those *ChStone benchmark applications* [31] are high level synthesis field, therefore the input data is not so large in-order to be able to simulate at the gate level. For these applications, in this dissertation, the impact of compiler transformation on performance is more interested rather than intensity itself.

3-4 Analysis Types

Taking into account the multi-objective facet and complexity of the problem, in this dissertation, for each benchmark explored, there have been several strong statistical analyses performed in order to support the evaluation process. All have been done by powerful open-sourced statistical software *R* [38]. The types of analysis are:

- ANOVA
- Kruskal-Wallis
- Principal Component Analysis (PCA)
- Correlation Plots
- Box-Plots, Scatter Plots, Matrix Plots
- Densities

For each benchmark, the type analyses mentioned in section Experimental Results of the Chapter - 4 have been elaborated. In the following section, the definitions of these analyses are being illustrated.

3-4-1 ANOVA Analysis

One of the best tests for evaluating the obtained results in the normal parametric distributions could be *ANOVA* [39] (for ANalysis OF VAriances).

ANOVA is a collection of statistical models, and their associated procedures, in which the observed variance in a particular variable is partitioned into components attributable to different sources of variation. In its simplest form, ANOVA provides a statistical test of whether or not the means of several groups are all equal, and therefore generalizes *t-test* [40] to more than two groups. T-test gets a significant acceptance value as (α), and then decides to accept or reject the model if the acceptance is lower or higher than the calculated value. ANOVA is a particular form of statistical hypothesis testing heavily used in the analysis of experimental data. A statistical hypothesis test is a method of making decisions using data. A test result (calculated from the null hypothesis and the sample) is called statistically significant if it is deemed unlikely to have occurred, *assuming the truth of the null hypothesis*. A statistically significant result (when a probability (p-value) is less than a threshold (significance level)) justifies the rejection of the null hypothesis. The computer method calculates the probability (p-value) of a value of F greater than or equal to the observed value (Pr). The null hypothesis is rejected if this probability is less than or equal to the significance level (α). The two methods produce the same result. In this dissertation, the significance level (α) is equal to 5%, therefore, for accepting a model (Pr) should be greater than F.

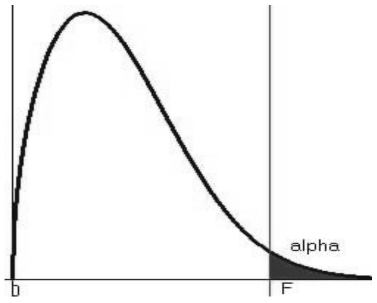
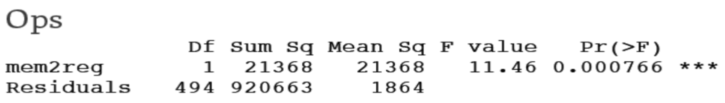


Figure 8- ANOVA

As an example, for *GSM* benchmark and the *mem2reg* compiler option, we have this ANOVA result for performance value:



```
Ops
             Df Sum Sq Mean Sq F value   Pr(>F)
mem2reg       1  21368   21368   11.46 0.000766 ***
Residuals   494 920663    1864
```

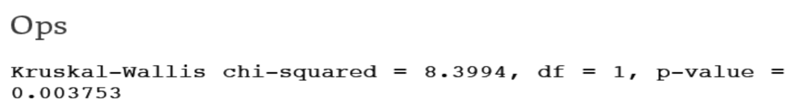
Figure 9- ANOVA Example

As it may be seen, the (Pr) is greater than (F) and the value is lower than 5%, so the test will be accepted and it is possible the declare existence of significant impact of mem2reg on the performance metric on the model.

3-4-2 Kruskal-Wallis

Unlike ANOVA, this analysis test is for non-parametric data. *Kruskal-Wallis* [41] compares between the medians of two or more samples to determine if the samples have come from different populations. Firstly, it has to be checked if the data are independent from each other and the distribution do not have to be normal and the variance do not have to be equal. The more important thing is that the individuals must have equal chance of being selected.

As an example, just like the last method, the acceptance test has to be based on the significance level (α) which is supposed to be 5% in this dissertation. By having:



Ops

```
Kruskal-Wallis chi-squared = 8.3994, df = 1, p-value =
0.003753
```

Figure 10- Kruskal_Wallis_Example

3-4-3 Correlation Analysis

In this dissertation a couple of different correlation analyses have been made in order to better elaborate the experimental results.

First, the *Correlation Matrix,* which is similar to the Covariance Matrix of the standardized random variables [42] is going to be illustrated. In this matrix, maximum correlation in the same way of the parameter is going to be shown by (+1) and vice-versa in the opposite way will be (-1). In between those points, the correlation will be distributed and of course on the main diagonal of the matrix the value will be zero as of NO correlation for each same couple.

Second, by varying the parameters with the metrics, the deltas for each parameters will be reached. Therefore, this type of correlation matrix could be used in order to illustrate the impact of the other parameters on both the metrics and the other parameters

Third, the Correlation matrix of PCAs, is just like the normal correlation matrix with this different in which the main parameters for making correlation to will be the principal components of the metrics. For instance, for GSM benchmark and the mem2reg compiler parameter, the experimental result led to have the following correlation plot:

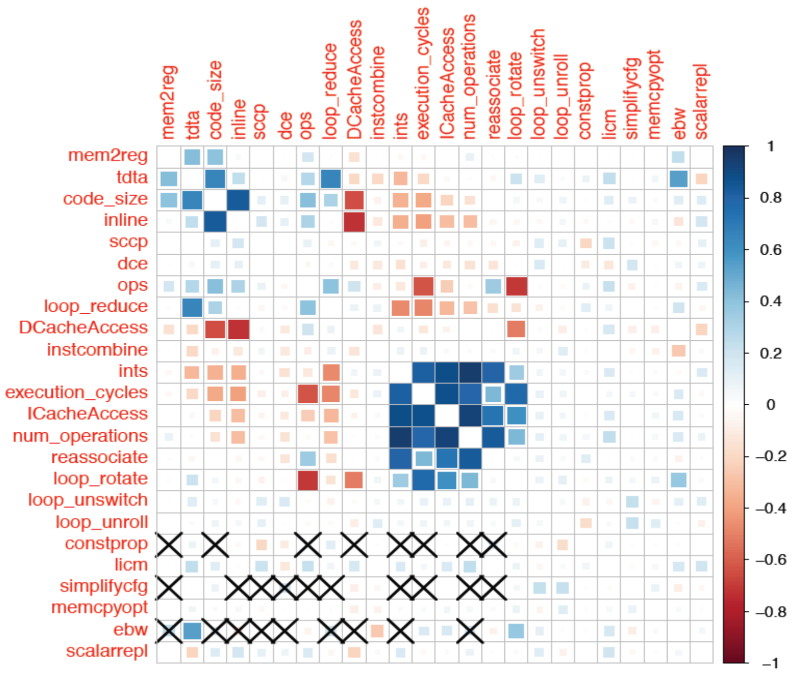


Figure 11- Correlation Matrix_GSM_Inline Example

As it may be seen, Performance (ops) seems positively correlated with mem2reg, code_size, inline, loop_reduce, reassociate while it is negatively correlated with loop_rotate.

3-4-4 Principal Component Analysis

Principal Component Analysis (PCA) is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. In addition, the number of principal components is less than or equal to the original values, or in another word, is less than or equal to the number of *eigenvalues* of the matrix [43]. PCA is a way of identifying the patterns in data and expressing the data in such a way as to highlight their similarities and differences.

In the metrics of the analysis, it is been tried to focus on finding and analyzing the most influential patterns regarding the performance and intensity in the experimental results, therefore using PCA could be a good tool in order to define new levels for the analysis. As an example, again for GSM_Inline parameter, the PCA plot is defined as Figure 12- (PCA for GSM_Inline):

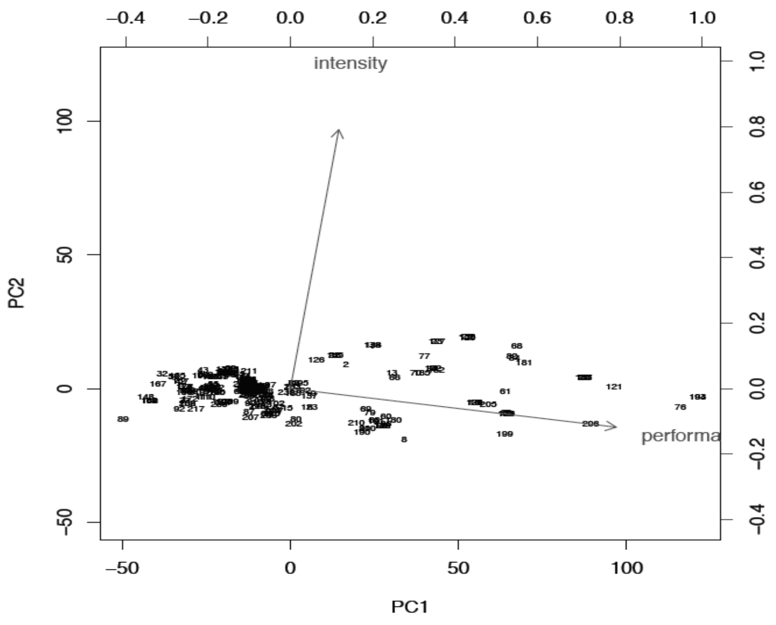


Figure 12- PCA for GSM_Inline

Figure 12- (PCA for GSM_Inline) shows strong impact of performance component (around 98%) to the model, and it is the main or the first principal component of the model.

Chapter 4

Experimental Results

In this chapter of dissertation, the achieved experimental results are going to be illustrated. The procedure of elaboration will be benchmark by benchmark, and then some of the selected diagrams of each are going to be introduced. Finally, the whole experimental of each benchmark will be classified via a table.

All the selected benchmarks have been evaluated and explored with the following scenario:

1- Fixed architectural parameters with the value mentioned in the section
2- 500 iterations under *RandomDoE* algorithm for each compiler parameter designed and executed by *MOST* [33]
3- Optimized with the *Parallel Doe* and being transformed.
4- Being measured regarding the metrics of the roof-line model, the basic metrics have been generated by *VEX*, then calculated for each iteration by the roof-line model equations [19]
5- Further analysis has been done with open-source software *R* [38] which the selected of them is going to be illustrated for each benchmark. The Analysis are :

   a. *"ANOVA"* test, defined in Section 3-4-1 (ANOVA Analysis)
   b. *"KRUSKAL"* test, defined in Section 3-4-2 (Kruskal-Wallis)
   c. *"Box Plots"* of Intensity, Performance for enabling/excluding each compiler parameter
   d. *"Correlation Analysis"*, defined in the Section 3-4-3 (Correlation Analysis)
   e. *"Scatter Plots"* of the effects obtained by varying each compiler parameter
   f. *"Principal Component Analysis"*, defined in Section 3-4-4 (Principal Component)
   g. *"Average Increment of Performance and Intensity"* for each compiler parameter option
   h. *Densities* regarding the performance and intensity and activating the specified compiler parameter and the second chosen parameter.

4-1 Motivation

There are several facets to be taken into considerations when we deal with design space in VLIW processors. First, as it was mentioned in the Section 2-1-5 (Performance Model and Floating Point), the roof-line model defines the limits in which it won't be possible to surpass this line. To certify the theory, as it has been illustrated in Figure 13, the *GSM* benchmark has been explored 4000 times with total random architectural options and the dce random effect.

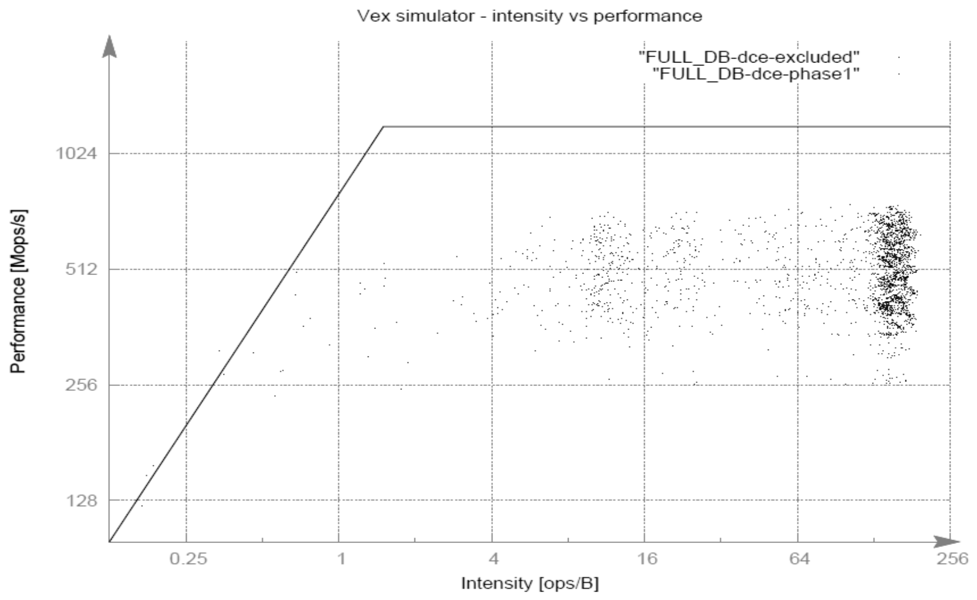


Figure 13 - GSM_dce_4000 iteration_ Roofline Certification

Second, since there are quite a lot of parameters involved in the problem, even with analyzing the effect of activating each transformation, it won't be easy classification of the results. In Figure 14-(GSM_Mem2reg Effect), by exploring 500 times *GSM* with mem2reg effect and filtering the configuration point both before and after activations with their metrics (Intensity and Performance), the effectual arrows have been drawn. As it could be observed, still lots of parameters have been involved affecting the trends and behaviors of the system.

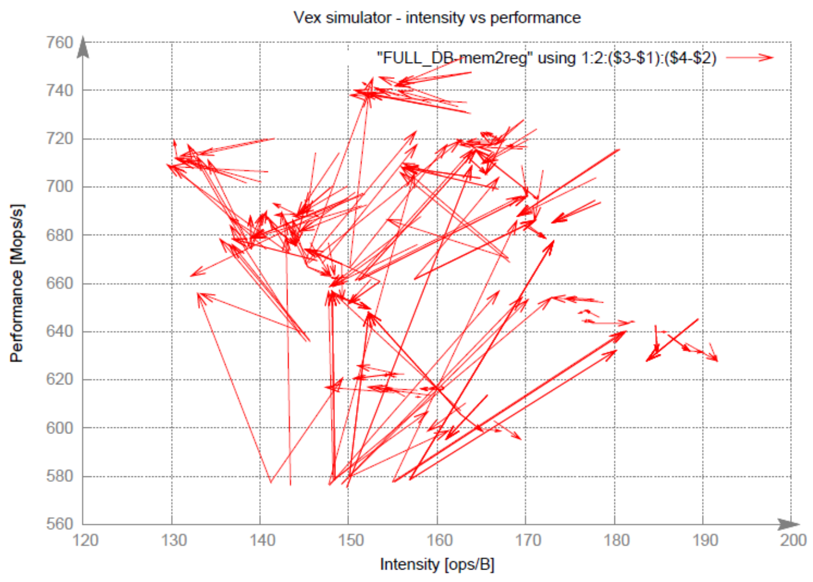


Figure 14-GSM_Mem2reg Effect

A meaningful visualization about the effect of varying the compiler option, the traversing under the roof-line could be vital since being either memory bound or computation bound could be resulted in refraining the further progression of the system resources.

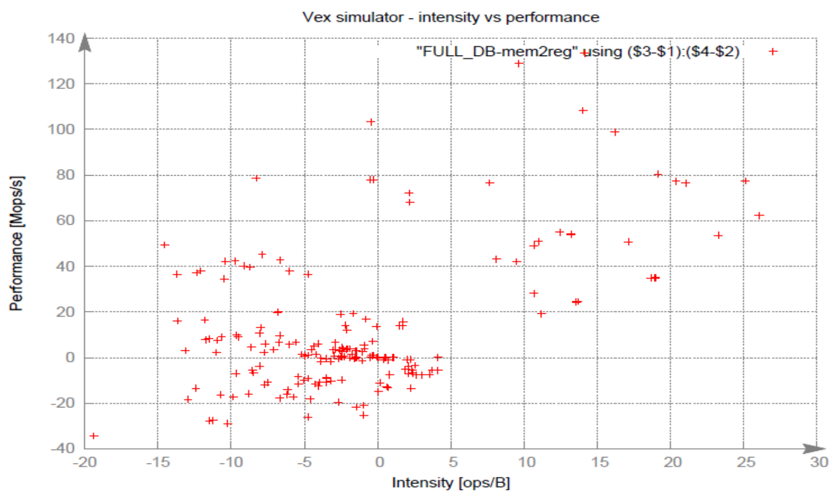


Figure 15-GSM_Mem2reg Effect_2

Figure 15- (GSM_Mem2reg Effect_2) shows the exact effects of Figure 14, provided with the points have been transformed to the relative origin point of O (0, 0) of the Cartesian. If we split the diagram into fourth, it is going to be seen that the majority of the points are located in the section fourth (minus intensity, minus performance). This will be base of starting the analysis (PCA, etc) which is going to be illustrated in the following.

4-2 Benchmark No.1 - GSM

It is one of the high intensity benchmarks available for testing the compiler performance at a high and low level; it has an Encoder/Decoder which is capable of sending and receiving the signals [30]. The *GSM* benchmark could place a good and reliable load into the system which is definitely needed to exemplify the use of the designed methodology.

In this section of dissertation, some of the most important results achieved by running the proposed methodology are going to be presented for GSM. Since the results and figures are pretty high and varied for all the 15 compiler options (named in section 2-1-4 Compiler Option), for some of the selected parameters the experimental results are going to be illustrated and the end of the section the whole table will be shown. In order to be complete, the whole results have been put in the section Appendix. For GSM, the parameters chosen were ***Inline***, ***Loop_Unroll*** and ***Mem2reg*** because those were good representative of all the space combination depicted in the Table 5- (GSM_ANOVA). More in detail we have that:

- *Inline* passes both tests for Intensity and performance,
- *loop_unroll* failed the both,
- *mem2reg* have only passed the impact on Performance.

Therefore, selecting these three could be a good representation of the whole sets of transformations available.

4-2-1 ANOVA

ANOVA Analysis _ **Inline**

Intensity

```
             Df Sum Sq Mean Sq F value   Pr(>F)
inline        1  12847   12847   66.81 2.52e-15 ***
Residuals   496  95380     192
---
```

Ops

```
              Df Sum Sq Mean Sq F value   Pr(>F)
inline         1  41832   41832    22.8 2.37e-06 ***
Residuals    496 910193    1835
```

ANOVA Analysis _ **Loop_Unroll**

Intensity

```
              Df Sum Sq Mean Sq F value Pr(>F)
loop_unroll    1      0    0.08       0  0.984
Residuals    492 100129  203.51
```

Ops

```
              Df Sum Sq Mean Sq F value Pr(>F)
loop_unroll    1      2     1.7   0.001  0.976
Residuals    492 913735  1857.2
```

ANOVA Analysis _ **Mem2reg**

Intensity

```
              Df Sum Sq Mean Sq F value Pr(>F)
mem2reg        1     70   69.91    0.35  0.555
Residuals    494  98740  199.88
```

Ops

```
              Df Sum Sq Mean Sq F value   Pr(>F)
mem2reg        1  21368   21368   11.46 0.000766 ***
Residuals    494 920663    1864
```

Table 5- GSM_ANOVA

Provided with the Table 5, it could be seen that:

- **Inline**: a significant impact on **Intensity** is being observed based on the ANOVA test.
- **Loop_Unroll:** No significant changes observed
- **Mem2reg:** a significant impact on performance (**Ops**) could be observed.

4-2-2 Kruskal-Wallis

Kruskall-Wallis_**Inline**
Intensity
`Kruskal-Wallis chi-squared = 55.0613, df = 1, p-value = 1.168e-13`
Ops
`Kruskal-Wallis chi-squared = 23.3781, df = 1, p-value = 1.331e-06`

Kruskall-Wallis_**Loop_unroll**
Intensity
`Kruskal-Wallis chi-squared = 0.0022, df = 1, p-value = 0.9625`
Ops
`Kruskal-Wallis chi-squared = 0.007, df = 1, p-value = 0.9332`

Kruskall-Wallis_**Mem2reg**
Intensity
`Kruskal-Wallis chi-squared = 0.1376, df = 1, p-value = 0.7107`
Ops
`Kruskal-Wallis chi-squared = 8.3994, df = 1, p-value = 0.003753`

Table 6- GSM_Kruskal

Provided with the Table 6, it could be seen that:

- **Inline**: a significant impact on **Intensity** is being observed based on the kruskal test.
- **Loop_Unroll:** No significant changes observed
- **Mem2reg:** a significant impact on performance (**Ops**) could be observed.

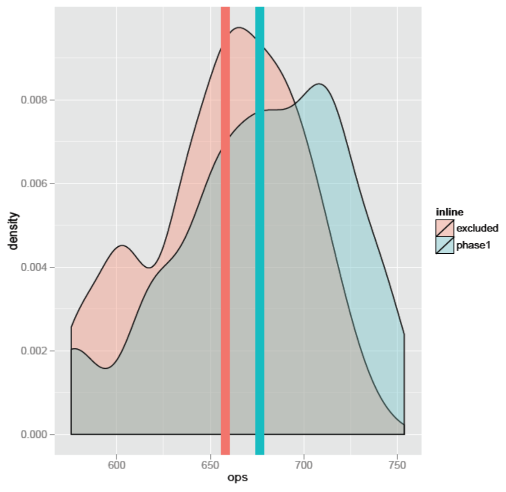
4-2-3 Distributions

In this section, presented on each page, there will be the densities of the transformations both in plot and box view. ANOVA, Kruskal-Wallis analyses can be certify the median lines of the figures.

Inline

Performance

Intensity



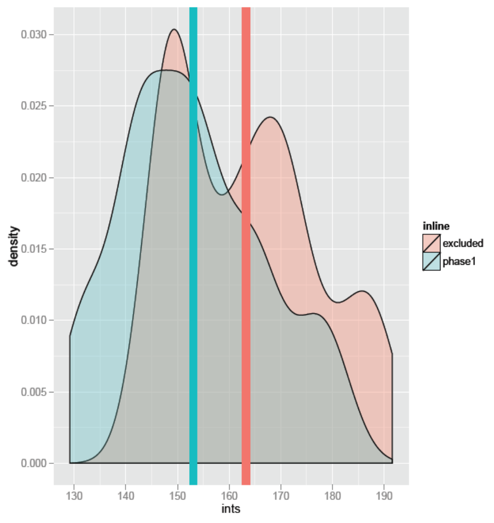
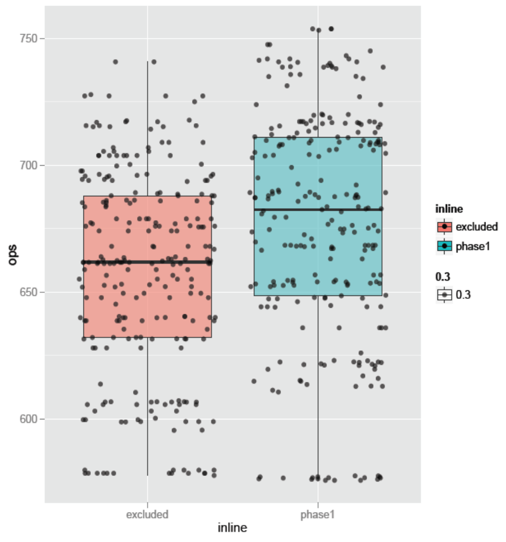


Figure 16-GSM_Inline_Distribution

Box-Plots

Performance

Intensity



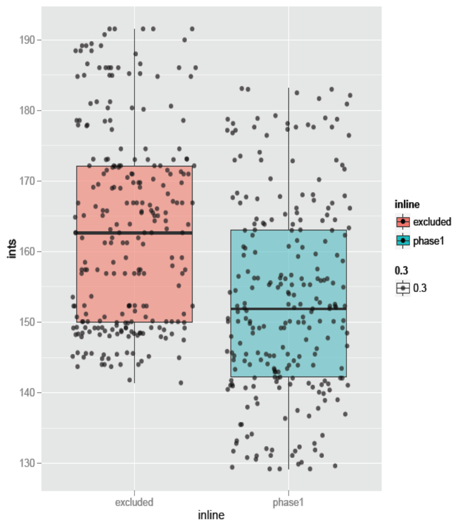
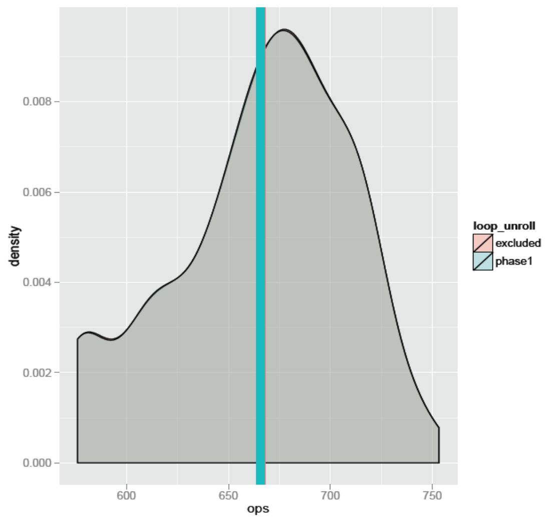


Figure 17-GSM_Inline_BoxPlot

It can be observed from the Figure 17, there are significant impacts on the median of Performance and Intensity by activating the Inline transformation. This statement could be certifies by ANOVA as well.

Loop_Unroll

Performance



Intensity

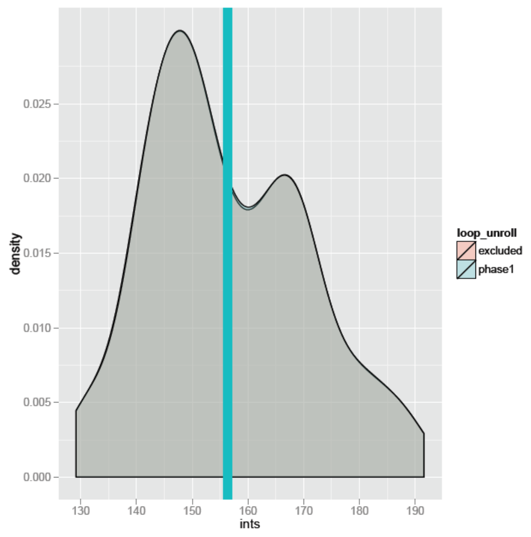
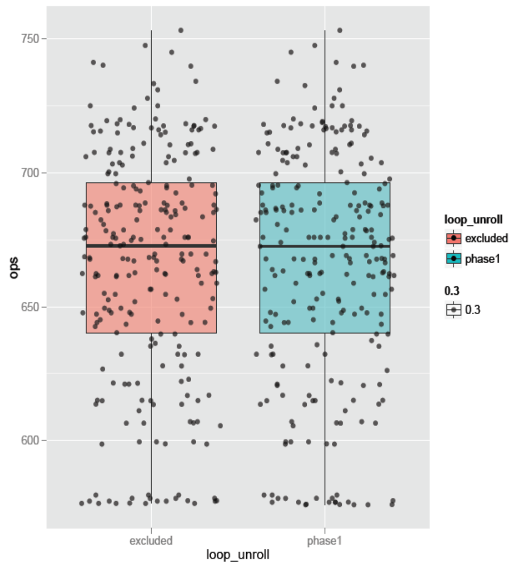


Figure 18-GSM_Loop_unroll_Dist

Performance **Intensity**



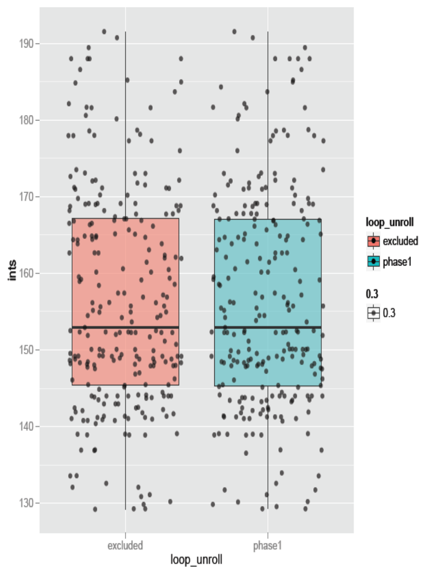
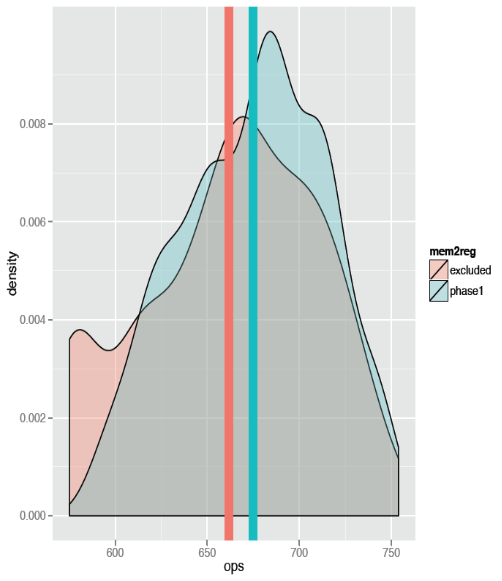


Figure 19-GSM_Loop_unroll_Box

As we could guess by ANOVA (refer to 4-4-8 GSM Conclusion), there are no significant change in the medians of loop_unroll. The Figure 19 certifies this hypothesis. Here as well, it could be observed that the medians are the same, so no significant impact on metrics.

Mem2reg

Performance Intensity



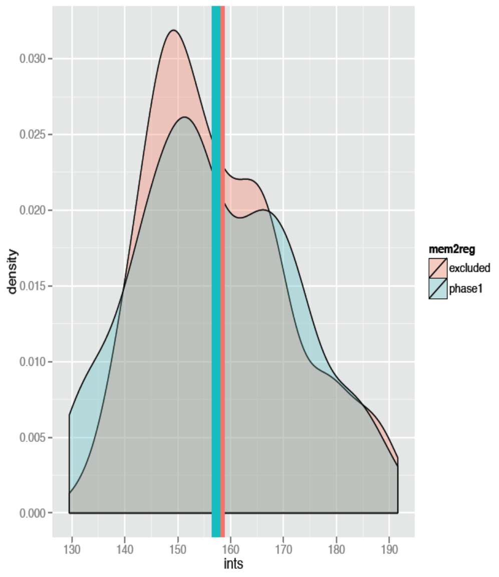
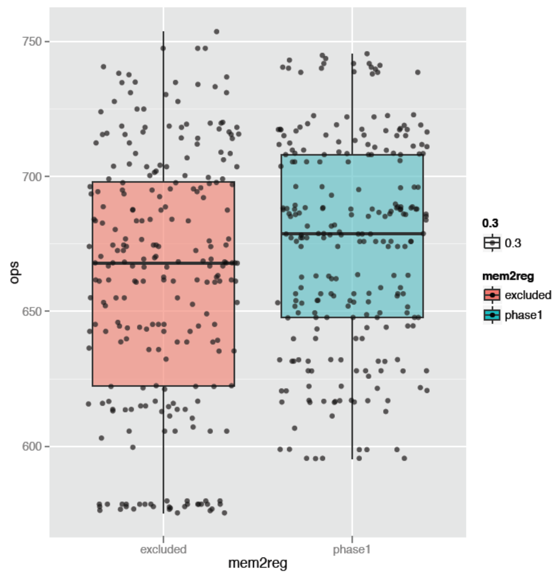


Figure 20-GSM_Mem2reg_Dist

Performance Intensity



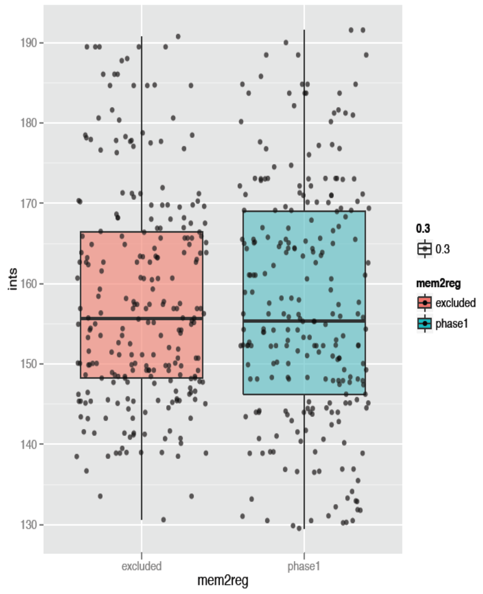


Figure 21-GSM_Mem2reg_Box

Illustrated by Figure 21, mem2reg transformation has significant impact on Performance metrics.

4-2-4 Scatter Plots

By drawing intensity and performance in a same figure, it can be possible to have a plot which shows the variety of data and experimental points scattered in the figure as in Figure 21:

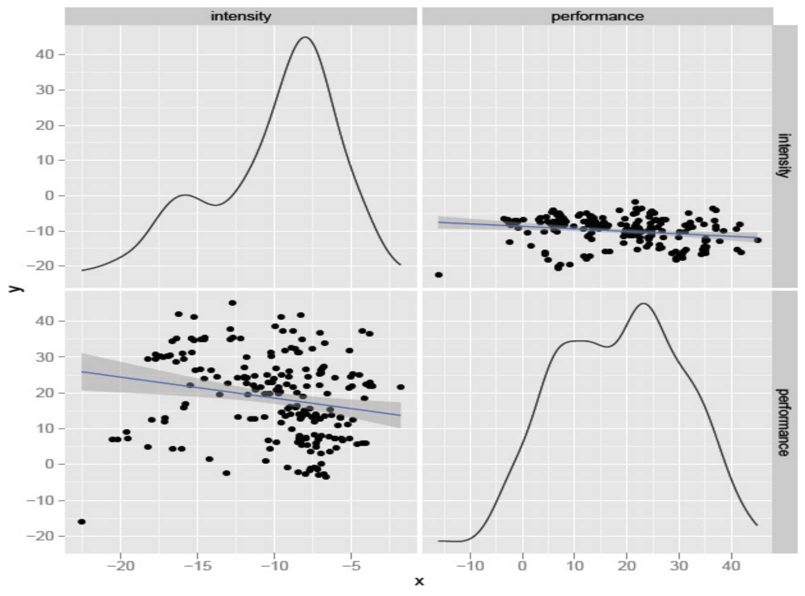


Figure 21-GSM_Inline_ScatterPlot

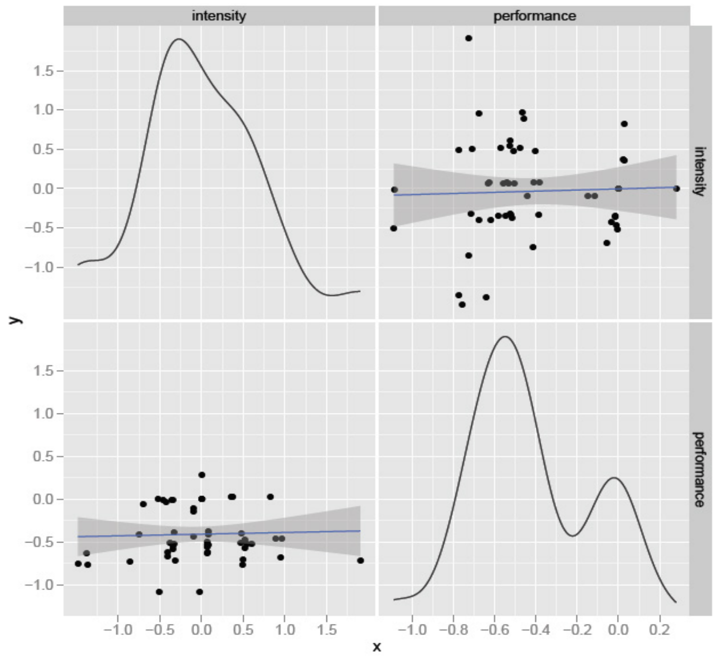


Figure 22-GSM_ScatterPlot_Loop_unroll

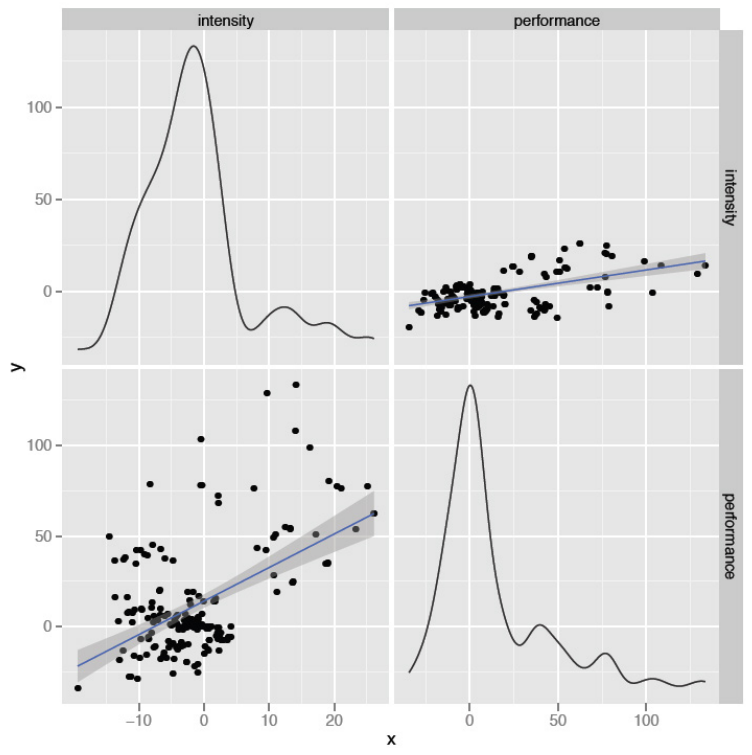


Figure 23--GSM_mem2reg_Scatterplot

Figures 21-23 are a good representation of the deltas while seeing the both metrics together. Distributions could be seen easily and the trends (if any) could be extracted.

4-2-5 Principal Component Analysis

As explained in the section 3-4-4 Principal Component, using this analysis will re-coordinate the way we look at the figures in such a way that the more important components based on the highest variety are categorized as the first and second components. Therefore, the figure can be analyzed by the better knowledge of knowing the main affected factor.

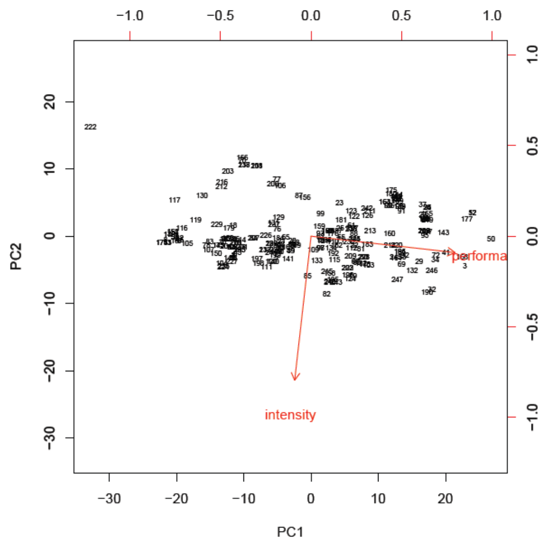


Figure 24-GSM_Inline_PCA

As it can be observed by the figure the principal component is performance. The second component is the Intensity.

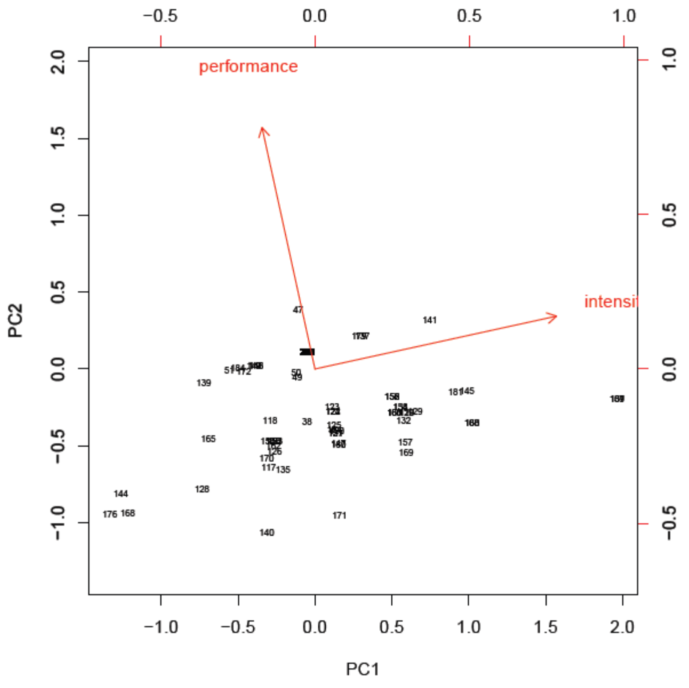


Figure 25-GSM_Loop_unroll_PCA

With a low slope, the first principal component is related to intensity here and slopped performance is the second key.

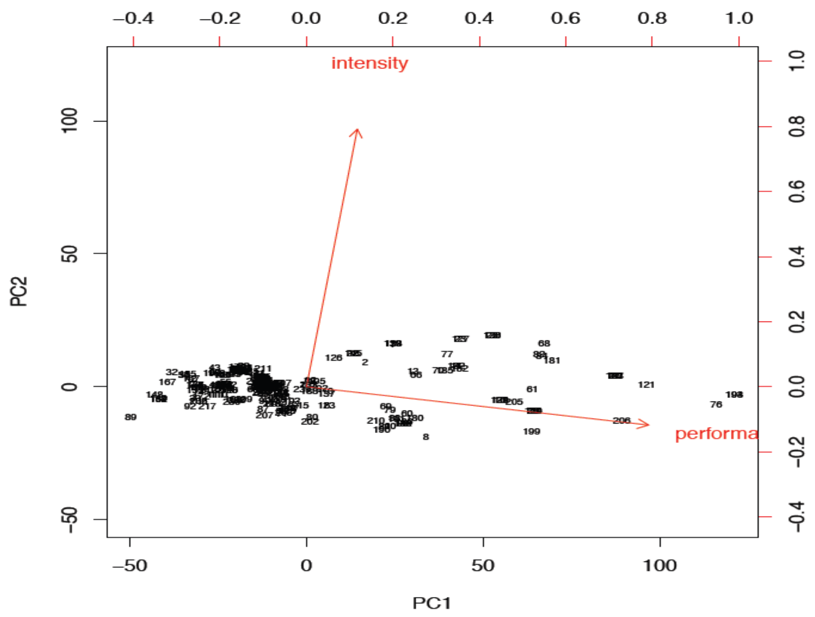


Figure 26-GSM_Mem2reg_PCA

As it can be observed by the figure the principal component is performance. The second component is the Intensity.

4-2-6 Correlation Analysis

In this section of experimental result, three types of correlation are going to be presented.

1. **Correlation on raw data**: Simply by having the output data and the metrics, there is a possibility of calculating the correlation between each two component of the performance and compiler parameters

2. **Correlation on deltas**: As it was depicted in Figure 12-GSM_Mem2reg Effect and Figure 13-GSM_Mem2reg Effect_2, by filtering the specified compiler parameter and their metrics (Performance and Intensity), there will be derived four points which was the result of exclusion and inclusion of that compiler parameter with the results. This kind of correlation is calculated based on these deltas of the points.

3. **Correlation of the Principal Component**: After defining the PCA of the exploration, it is also possible to do the correlation with respect to the first and second principal component.

1- Correlation on raw data

Inline

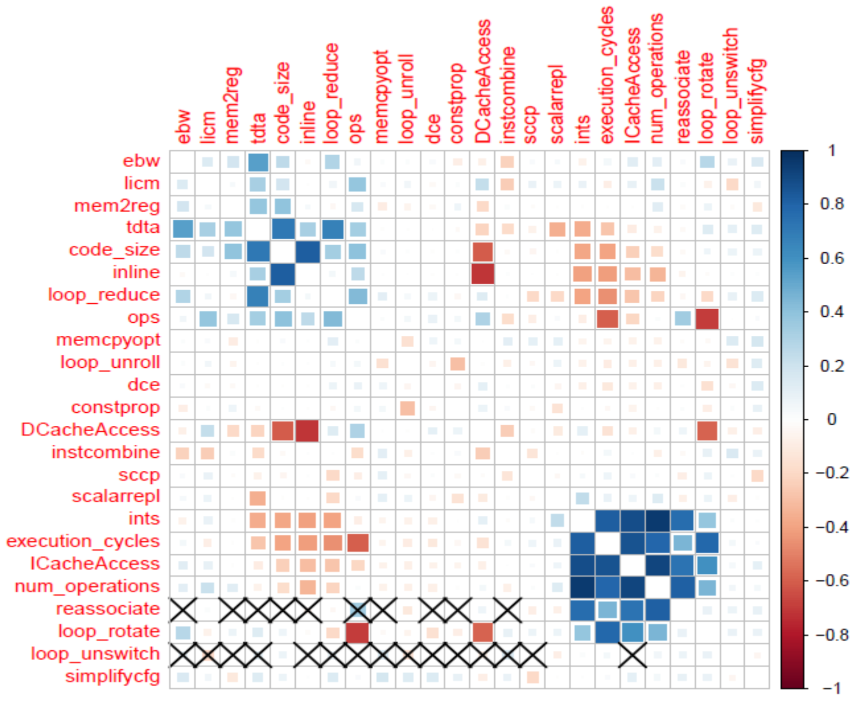


Figure 27-GSM_Inline_Corr_raw_data

As it could be seen from the result:

- **Performance (opt);** seems positively correlated with **loop_reduce**, **inline**, **mem2reg**, **reassociate**, **memcpyopt**. **Licm** while it is negatively correlated with **loop-rotate** and **instcombine.**
- **Intensity (ints);** positively correlated with **loop-rotate**, **reassiciate** and **scalarrepl** and negatively with **loop-reduce** and **inline**
- Small negative correlation between Intensity and performance in the table

Loop_Unroll

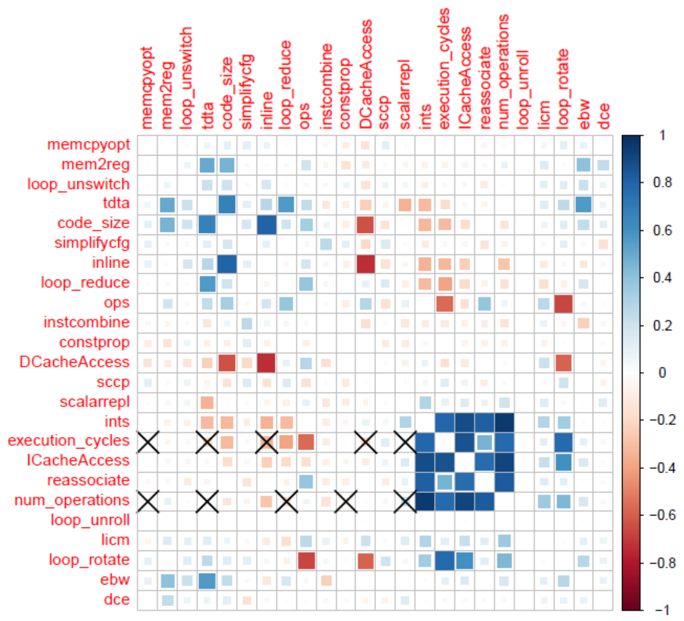


Figure 28- GSM_Loop_unroll_Corr_raw

- **Performance (opt);** seems <u>positively</u> correlated with **loop_reduce**, **inline**, **mem2reg**, **reassociate**. **Licm** while it is <u>negatively</u> correlated with **loop-rotate**
- **Intensity (ints);** <u>positively</u> correlated with **loop-rotate**, **reassociate** and **scalarrepl** and <u>negatively</u> with **loop-reduce** and **inline**
- <u>THERE IS NO</u> correlation between Intensity and performance

Mem2reg

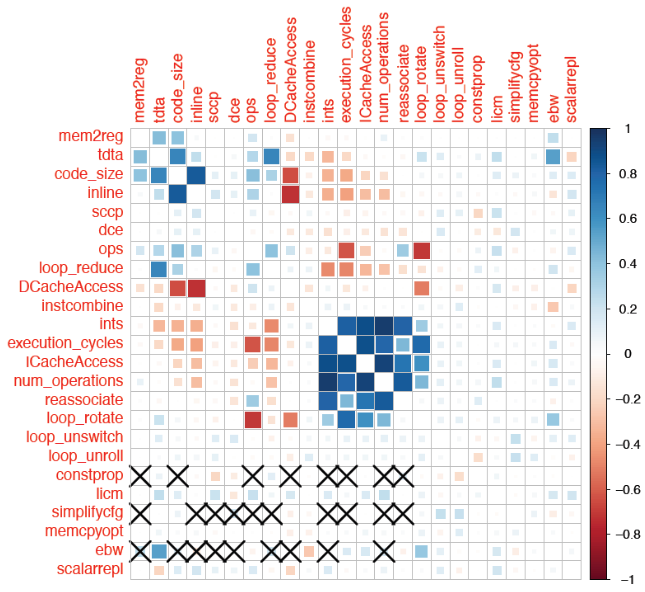


Figure 29-GSM_Mem2reg_Corr_Raw

2- Correlation on Deltas

Inline

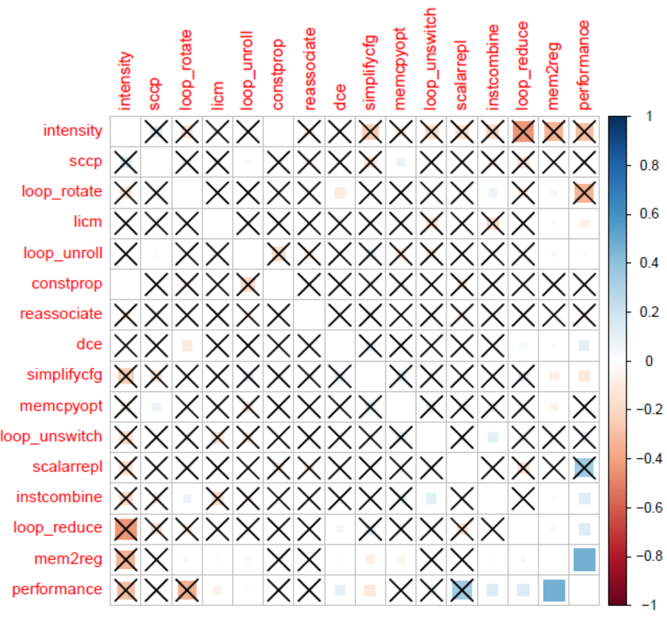


Figure 30-GSM_Inline_Corr_Deltas

It can be observed in the figure that:

- No significant impact on Intensity
- Performance is modified positively by both the activation of **inline** and **mem2reg** and **loop_reduce, instcombine** and **dce.** It can negatively modified by activation of **inline** and **simplifycfg** and **licm**

Loop_Unroll

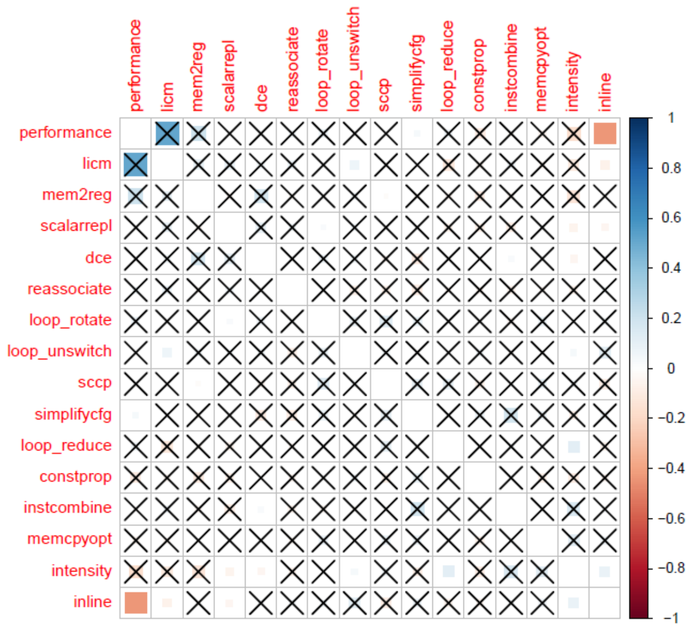


Figure 31-GSM_Loop_Unroll_Corr_Delta

- Intensity can be decreased by activating **loop_unroll** and **inline** and **loop_reduce.** and negatively by activating **loop_unroll** and **scalarrepl** and **dce.**
- Performance is modified negatively by both the activation of **loop_unroll** and **inline.** Also positively with **loop_unroll** and **simplifycfg**

Mem2reg

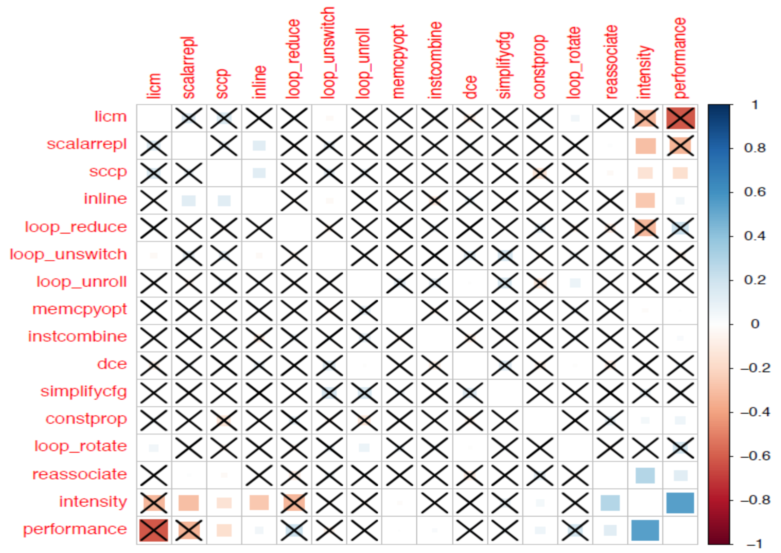


Figure 32-GSM_Mem2reg_Corr_Delta

3- Correlation of the PCA

Inline

Since it was shown on Figure 22-GSM_Inline_PCA, the PCA for the GSM_Inline were depicted. Based on these data, the correlation between the data and the components can be shown as Figure 33:

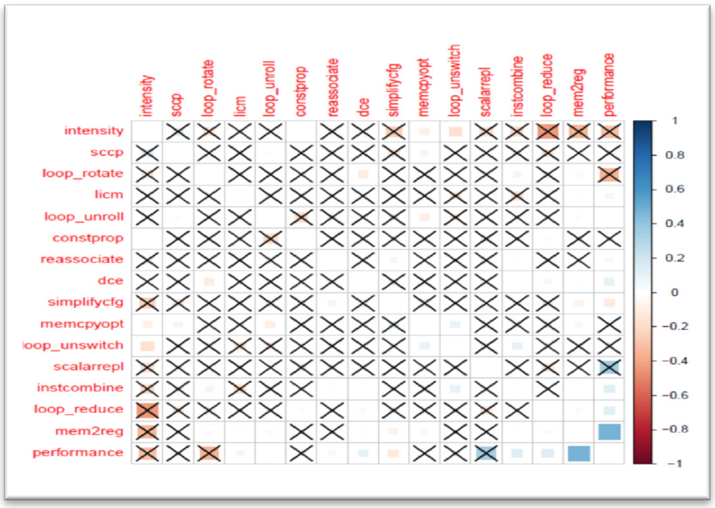


Figure 33-GSM_Inline_Corr_PCA

For the PCA correlation, it can be observed that the transformation mem2reg has positive correlation with the first principal component and simplifycfg has negative correlation with performance.

Loop_Unroll

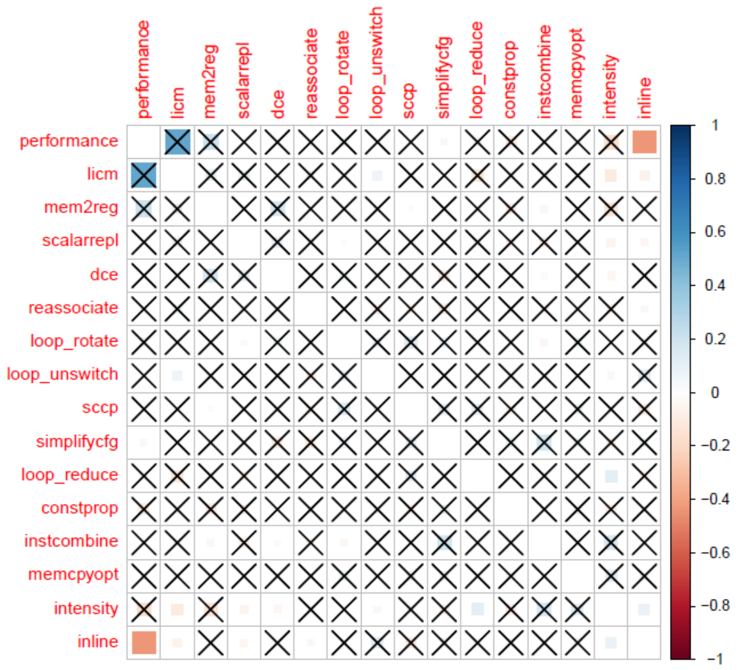


Figure 34-GSM_Loop_Unroll_Corr_PCA

Mem2reg

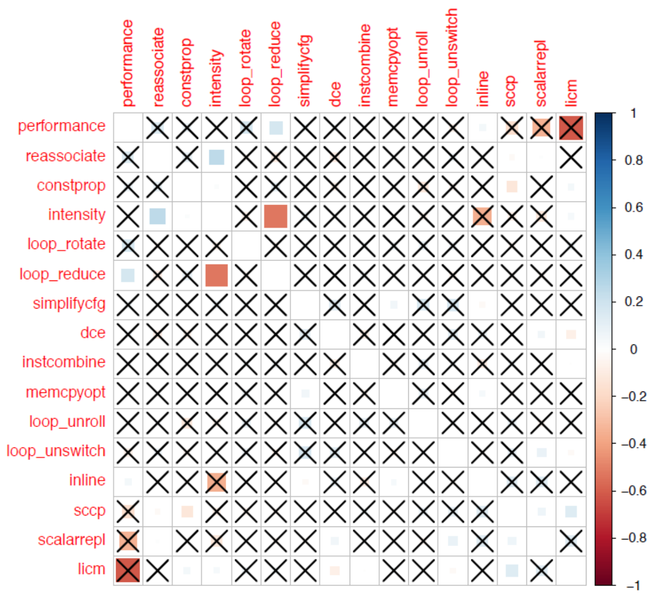


Figure 35-GSM_Mem2reg_Coo_PCA

Regarding the above correlation plot, it can be said that,

- **Loop_reduce** seems positively impacting the first component (represented by Performance). The second component (Intensity) is negatively impacted by **loop_reduceb** and positively impacted by **reassociate.**

4-2-7 Matrix Plot

After calculation of the metrics, another way of presenting the information could be by matrix-plot. In this plot "tdta" stands for *Total Data Across Bus*, "ebw" stands for *Effective bandwidth* and "int" and "ops" are representatives of *Intensity* and *Performance*.

Inline

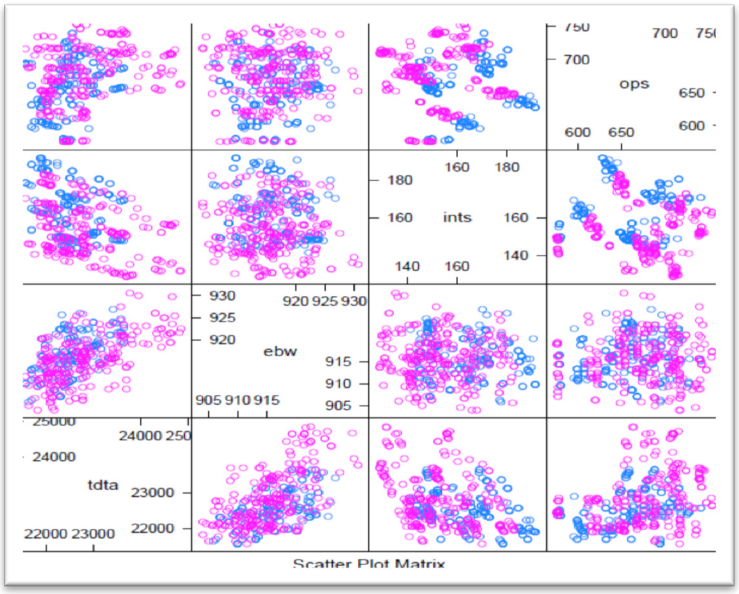


Figure 36-GSM_Inline_MatrixPlot

Loop_Unroll

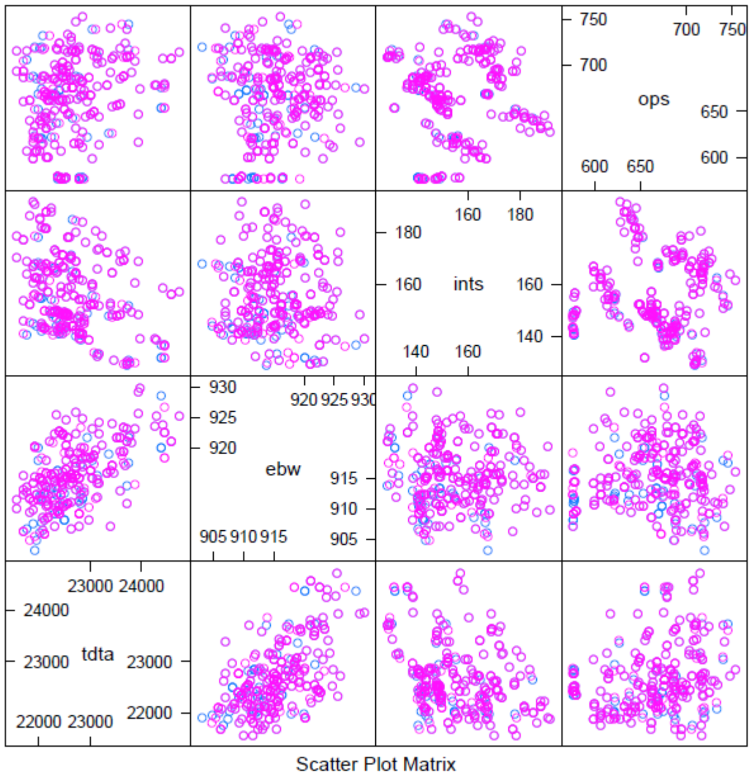


Figure 37-GSM_Loop_Unroll_Matrixplot

As it could be guessed, since the first principal component has high dependency with intensity, so the majority of the points have been indicated by pink, which refers to the intensity.

For the other two figures, the issue is vice-versa, thus there are enormous blue points in the system have been observed.

Mem2reg

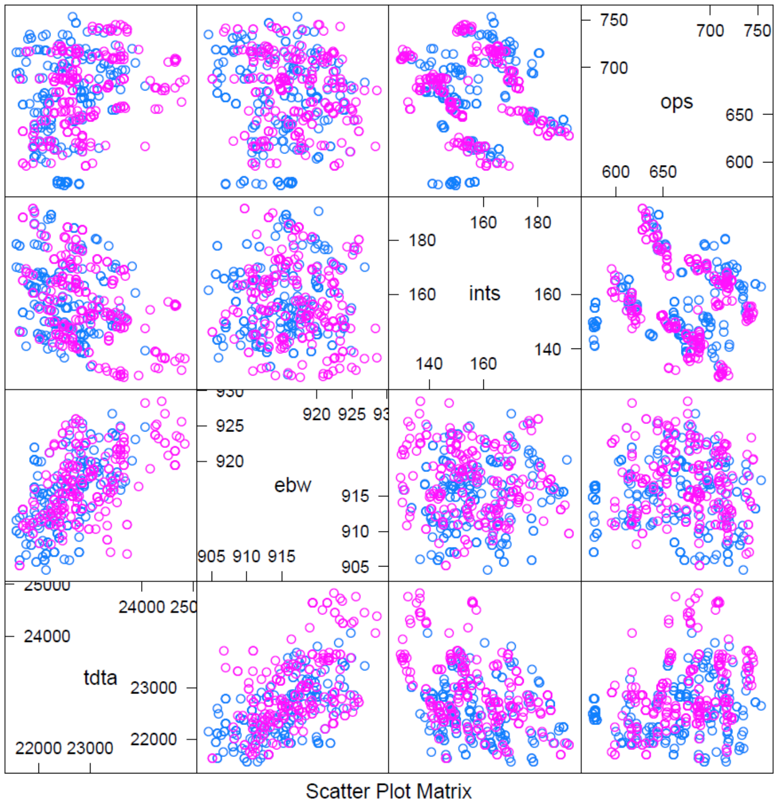


Figure 38-GSM_Mem2reg_MatrixPlot

Densities

When the benchmark has been explored with respect to the specified compiler parameter, there will also a possibility to see the effects of adding the second parameter (include/exclude) with respect of having the first parameter activated already, i.e. in this scenario now there is "*Inline*" option activated already for exploration, we can see the effect of having a second parameter meanwhile.

In the Figure 39, *Inline* parameter have been already activated, for the *intensity* metric we are interested in seeing the effect of activating “scalarrepl” as well. Therefore for both case of including and excluding the parameter, the following figure is drawn:

Inline

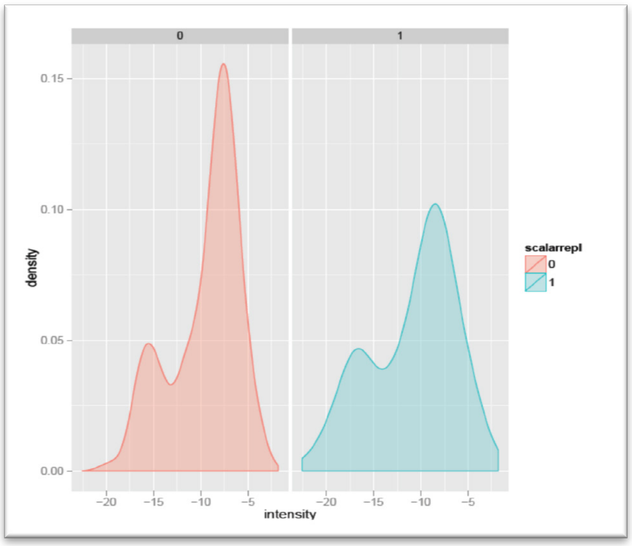


Figure 39-GSM_Inline_Densities

Loop_unroll

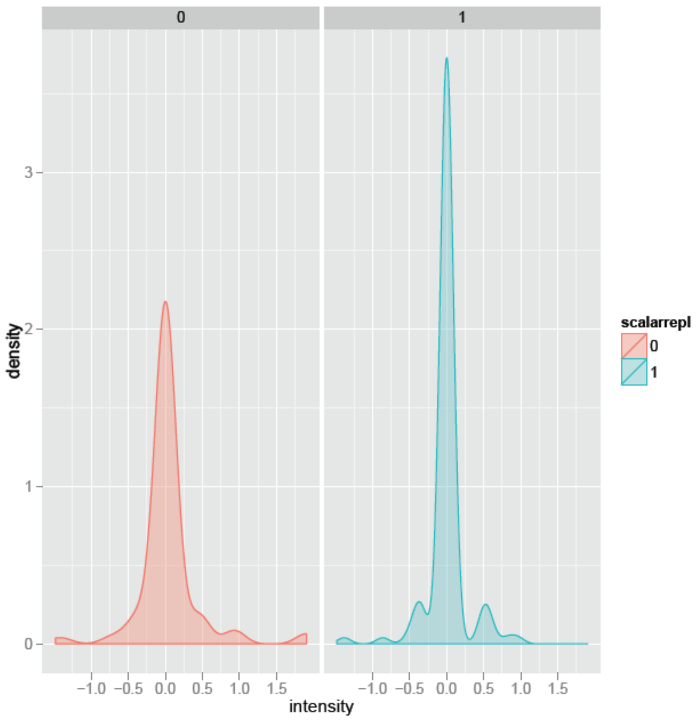


Figure 40-GSM_Loop_Unroll_Densities

Mem2reg

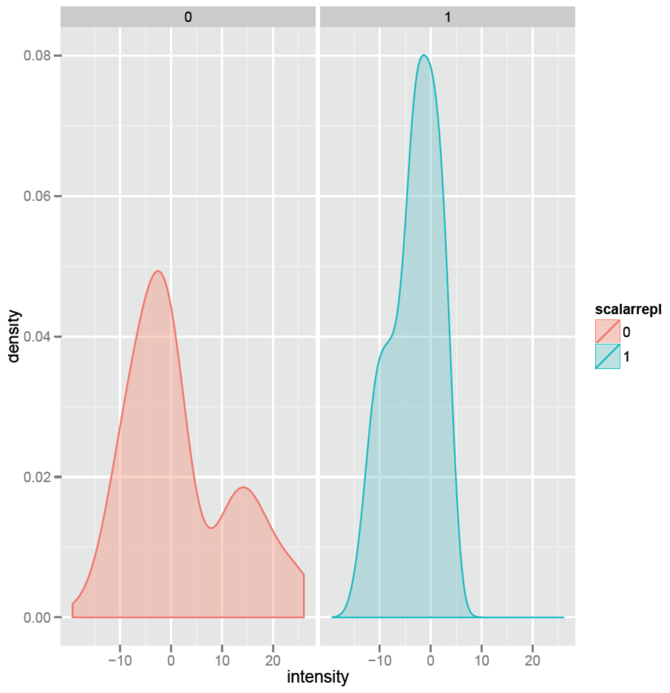


Figure 41-GSM_Mem2reg_Densities

4-4-8 GSM Conclusion

The results which have been illustrated, was for the compiler parameter "*Inline*". Since the compiler parameters explored in this dissertation were 15, for being abstract regarding the results publications and figures in this text, the author assumed it suffice to present only one parameter out of those 15. For the sake of completeness, at the end of each benchmark there will be a conclusion section which presents all the complete data in a quantitative table.

In the following page, the classification of results for *GSM* is being illustrated.

	ANOVA (<5%)		KRUSKAL (<5%)		CORR		Corr on delta (5%)		CORR (PCA) 5%		DATA summmary	
Parameters	INT	OPS	INT	OPS	INT	OPS	INT	OPS	INT	OPS	AVG increment of INT	Average increment of OPS
constprop	0.937	0.997	0.9249	0.9804	Mean: 159.2 Min:129.3 Max:191.6	Mean: 671.3 Min:576.2 Max:754.8	+ scalarrepl (.005)	- loop_unswitch(.06)	PC2= .25 + mem2reg(.09) - inline(.8)	PC1= .75 - instcombine(.19) - licm(.07)	0.1	-0.02
dce	0.986	0.996	0.9851	0.9858	Mean: 158.7 Min:129.3 Max:191.6	Mean: 669.6 Min:575.3 Max:751.3	-constprop	+ loop_rotate - licm	PC2= .12 + instcombine (.13) - licm (.12)	PC1= .88 No accepted result regading sig-level 5%	-0.02	-0.02
inline	2.52E-15	2.37E-06	1.17E-13	1.33E-06	Mean: 158.2 Min:129.2 Max:191.6	Mean: 667.3 Min:575.7 Max:753.7	no accepted results for 5% sig-level	+ mem2reg (.54) + loop_reduce(.01) + dce (.08) - simplifycfg (.11) - licm(.08)	PC2= .09 + simplifycfg (.28)	PC1=.91 - loop_reduce (.14)	-10.15	18.33
instcombine	0.612	0.145	0.4239	0.03213	Mean: 158.3 Min:129.2 Max:191.6	Mean: 670.0 Min:575.3 Max:753.7	+ performance(.56) - licm(.34)	+ intensity(.39) - licm(.3)	PC2= .13 No accepted result regading sig-level 5%	PC1=.86 ' + inline (.31)	0.69	-5.17
licm	0.0494	2.01E-05	0.0364	2.09E-05	Mean: 157.3 Min:129.3 Max:191.6	Mean: 670.0 Min:575.3 Max:747.6	+ loop_unswitch	no accepted results for 5% sig-level	PC2=.05 + loop_reduce (.52) - inline (.16)	PC1=.95 No accepted result regading sig-level 5%	2.55	15.54
loop_reduce	1.10E-11	1.20E-16	1.71E-09	1.15E-15	Mean: 157.5 Min:129.2 Max:191.6	Mean: 669.6 Min:575.3 Max:751.6	- mem2reg (.42) - scalarrepl(.38) - reassociate (.15)	no accepted results for 5% sig-level	PC2= .05 No accepted result regading sig-level 5%	PC1= .95 + mem2reg (.36) - instcombine (.15)	-8.54	31.3
loop_rotate	4.73E-13	1.20E-16	1.19E-11	2.20E-16	Mean: 157.5 Min:129.2 Max:191.6	Mean: 678.8 Min:575.3 Max:753.7	+ mem2reg - simplifycfg	no accepted results for 5% sig-level	PC2= .05 + loop_reduce (.49) - mem2reg (.12)	PC1= .95 + mem2reg (.26)	8.92	-52.14
loop_unroll	0.984	0.976	9.63E-01	0.9332	Mean: 156.4 Min:129.2 Max:191.6	Mean: 665.8 Min:576.1 Max:753.2	+ inline	- inline (.43)	PC2= .31 - inline (.41)	PC1= .68 - licm (.17)	0.02	-0.11
loop_unswitch	0.825	0.896	0.8139	0.9337	Mean: 158.3 Min:129.3 Max:191.6	Mean: 668.0 Min:575.5 Max:754.8	no accepted results for 5% sig-level	+ instcombine + mem2reg	PC2= .10 + inline (.24) - mem2reg (.26)	PC1= .90 + loop_reduce (.17)	-0.28	0.51
mem2reg	0.555	0.000766	0.7107	0.003753	Mean: 157.7 Min:129.5 Max:191.6	Mean: 668.3 Min:575.3 Max:753.7	+ performance (.54) + reassociate (.28) - scalarrepl(.26) - inline(.14) - sccp(.03)	+ intensity(.54) - sccp(.16)	PC2= .05 + scalarrepl (.11)	PC1= .95 No accepted result regading sig-level 5%	-0.75	13.12
memcpyopt	1	1	1	1	Mean: 157.0 Min:130.2 Max:188.5	Mean: 677.5 Min:576.1 Max:754.8	no accepted results for 5% sig-level	no accepted results for 5% sig-level	Zero variance ??	Zero Variance ??	0	0
reassociate	1.20E-16	1.07E-11	2.20E-16	5.21E-11	Mean: 157.4 Min:129.2 Max:191.6	Mean: 667.5 Min:575.3 Max:753.7	+ performance (.96) + scalarrepl(.42) - instcombine (.11) - dce (.09) - loop_reduce (.08)	+ intensity (.96) + scalarrepl(.42) - instcombine (.11) - dce (.09) - loop_reduce (.08)	PC2= .01 + loop_rotate (.38)	PC1= .99 No accepted result regading sig-level 5%	19.88	26.11
scalarrepl	0.00033	0.0255	0.0001392	0.1392	Mean: 157.3 Min:129.4 Max:190.8	Mean: 669.8 Min:575.3 Max:753.2	+ performance (.82) + loop_rotate(.24) + loop_unroll (.12) - loop_reduce (.32) - mem2reg (.15)	+ intensity (.82) + loop_rotate(.19) + loop_unroll (.14)	PC2= .04 + reassociate (.13)	PC1= .96 + inline (.12)	4.52	8.46
sccp	0.893	0.91	0.8956	0.9124	Mean: 157.1 Min:129.2 Max:191.6	Mean: 668.3 Min:576.2 Max:751.3	no accepted results for 5% sig-level	+ instcombine (.35) + scalarepl(.08) - inline (.38) - simplifycfg (.19) - dce (.08)	PC2= .21 No accepted result regading sig-level 5%	PC1= .79 - simplifycfg(.19) - licm (.14) - mem2reg (.13)	0.17	-0.46
simpifycfg	0.491	0.905	0.5474	0.8483	Mean: 156.9 Min:129.2 Max:191.6	Mean: 672.5 Min:575.3 Max:747.5	+ performance (.84)	+ intensity (.84) + loop_unswitch(.27) - inline (.06) - instcombine (.05) - scalarrepl (.05)	PC2= .04 + loop_unswitch (.07)	PC1= .96 - inline (.13)	-0.88	-0.47

4-3 Benchmark No.2 *AES*

As it was mentioned in the section 3-3 Benchmarks, the explored benchmarks from No.2 to No.5 have been used from the *CHStone* benchmark package [31], and are some quality ones in order to see the impact of compiler parameters to performance but rather to I/O and intensity. These benchmarks are from high level synthesis field, so the input data is not so large in order to be simulated in the gate level.

Therefore, unlike *GSM* (refer to section 4-2 GSM Results), just the figures with meaningful results have been mentioned here. The overall focus was mostly dedicated to watch the Performance altering by using compiler parameters and draw a possible sketch of explaining why and how. Keep in mind that, no one could generalize rules easily out of 4-5 benchmarks what so ever, but the trend of altering the metrics might deliver a meaningful pattern in order to draw attention to.

4-3-1 ANOVA

ANOVA Analysis _ **Loop_Reduce**

Intensity

```
            Df Sum Sq Mean Sq F value   Pr(>F)
loop_reduce   1  0.196 0.19629   24.64 9.52e-07 ***
Residuals   496  3.951 0.00797
```

Ops

```
            Df Sum Sq Mean Sq F value Pr(>F)
loop_reduce   1  94018   94018   123.2 <2e-16 ***
Residuals   496 378399     763
```

ANOVA Analysis _ **Inline**

Intensity

```
             Df Sum Sq Mean Sq F value   Pr(>F)
inline        1  0.433  0.4334   55.53 4.14e-13 ***
Residuals   494  3.855  0.0078
```

Ops

```
            Df Sum Sq Mean Sq F value Pr(>F)
inline        1    657   657.1   0.731  0.393
Residuals   494 444251   899.3
```

ANOVA Analysis _ **Mem2reg**

Intensity

```
            Df Sum Sq Mean Sq F value  Pr(>F)
mem2reg       1  0.139 0.13880   15.72 8.4e-05 ***
Residuals   498  4.396 0.00883
```

Ops

```
             Df Sum Sq Mean Sq F value Pr(>F)
mem2reg       1 142418  142418   218.4 <2e-16 ***
Residuals   498 324714     652
```

Table 7-AES_ANOVA

Regarding the above table, provided with 5% of acceptance rate, it can be said:

- **Intensity**: all three have been passed and shows a significant impact on the intensity while using these benchmarks.
- **Performance (Ops)**: a significant impact could be seen on **Loop_reduce** and **mem2reg** while inline was left non-impacted.

4-3-2 Kruskal-Wallis

Kruskal Analysis _ **Inline**

Intensity

```
Kruskal-Wallis chi-squared = 31.7114, df = 1, p-value = 1.789e-08
```

Ops

```
Kruskal-Wallis chi-squared = 1.2454, df = 1, p-value = 0.2644
```

Kruskal Analysis _ **Loop_reduce**

Intensity

```
Kruskal-Wallis chi-squared = 13.8727, df = 1, p-value = 0.0001956
```

Ops

```
Kruskal-Wallis chi-squared = 66.6889, df = 1, p-value = 3.179e-16
```

Kruskal Analysis _ **Mem2reg**

Intensity

```
Kruskal-Wallis chi-squared = 10.9974, df = 1, p-value = 0.0009124
```

Ops

```
Kruskal-Wallis chi-squared = 124.5624, df = 1, p-value < 2.2e-16
```

Table 8-AES_Kruskal_Wallis

Defined by Table 8, it could be observed that:

- **Intensity**: all three have been passed and shows a significant impact on the intensity while using these benchmarks.
- **Performance (Ops)**: a significant impact could be seen on **Loop_reduce** and **mem2reg** while inline was left non-impacted.

4-3-3 Distributions

Inline

Performance

Intensity

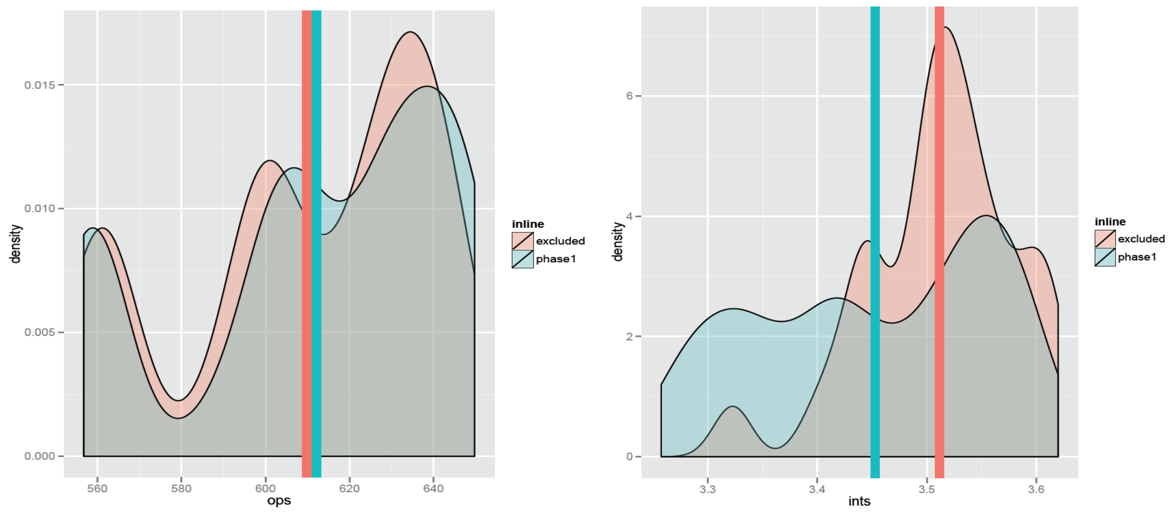


Figure 42-AES_Distributions

Performance

Intensity

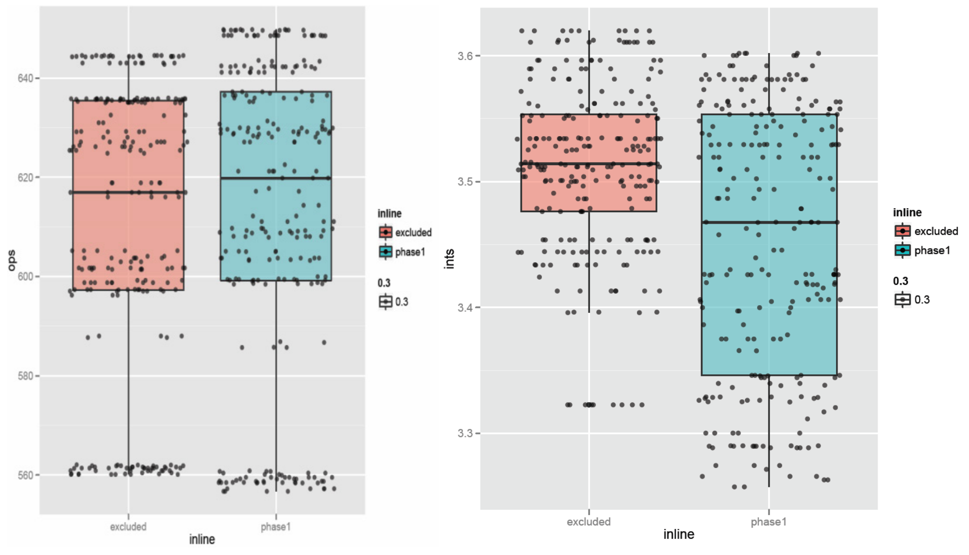
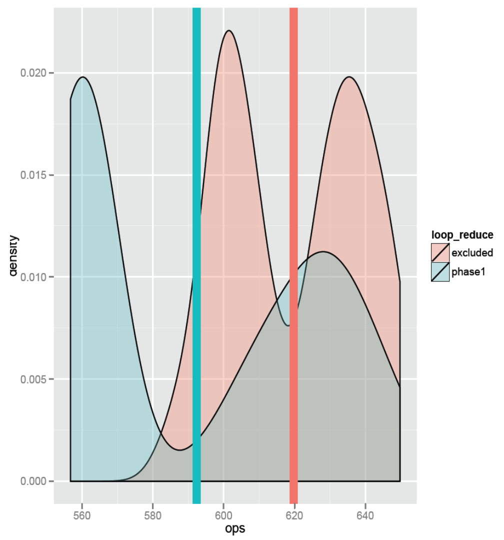


Figure 43-AES_Boxplot

As it was suggested on ANOVA test as well, a significant impact on the intensity metrics could be observed by inline transformation. In Figure 43 - (AES_Boxplot) as well, there is the box-plot of Inline. The medians could be seen impacted.

Loop_reduce

Performance Intensity



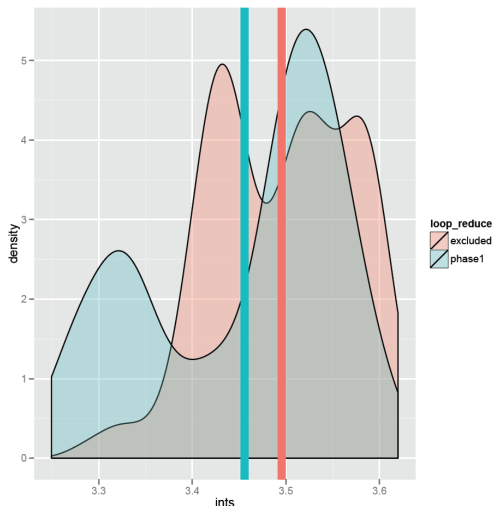
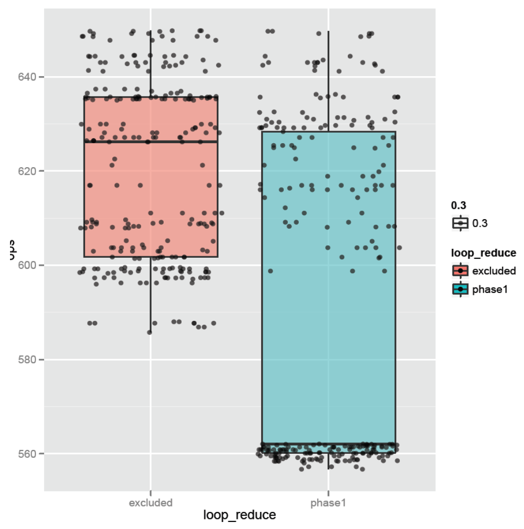


Figure 44-AES_Loop_reduce

Performance Intensity



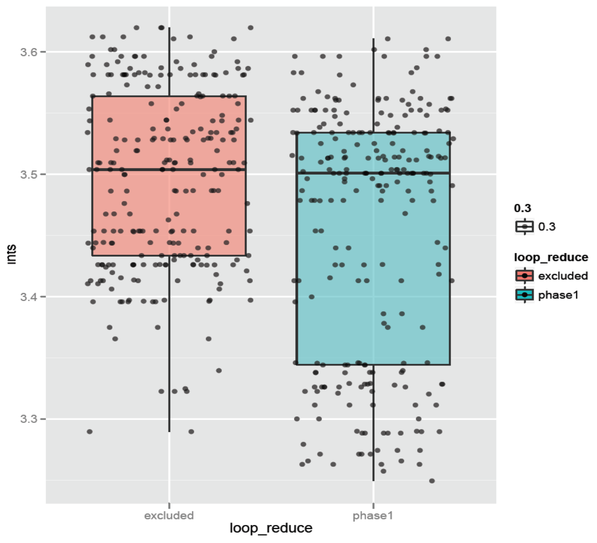
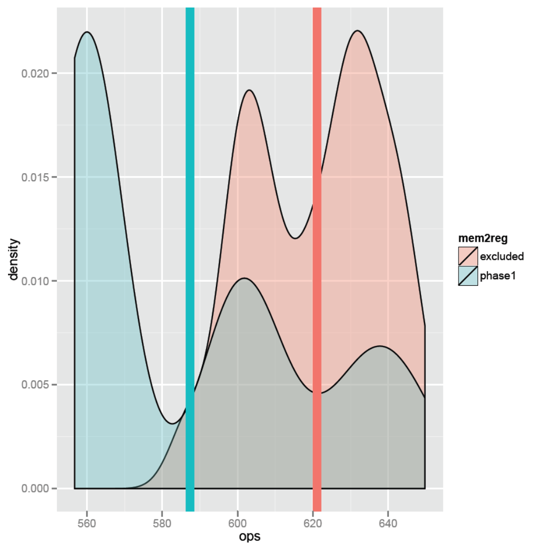


Figure 45-AES_Loop_reduce_Box

As it was suggested on ANOVA test as well, a significant impact on the both metrics could be observed by loop_reduce transformation in both Figure 44- (AES_Loop_reduce) and Figure 45- (AES_Loop_reduce_Box)

Distribution "Mem2reg"

Performance



Intensity

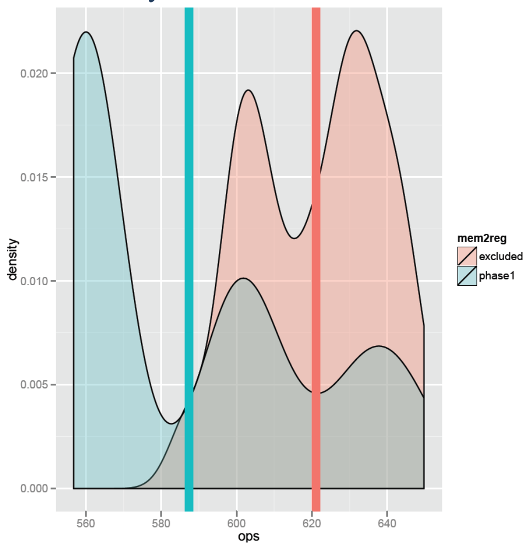
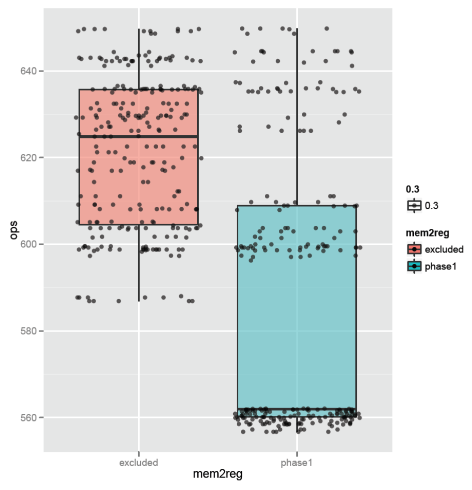


Figure 46-AES_Mem2reg_Distributions

Box-Plots "Mem2reg"

Performance



Intensity

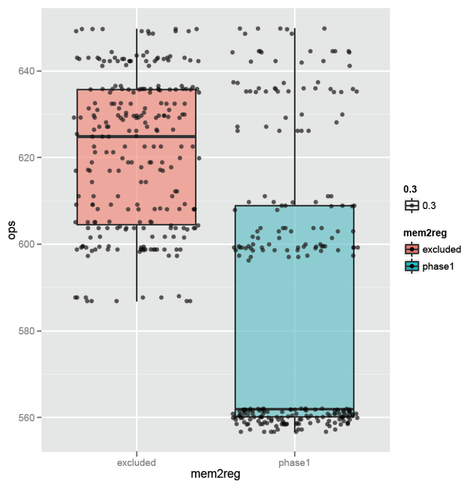


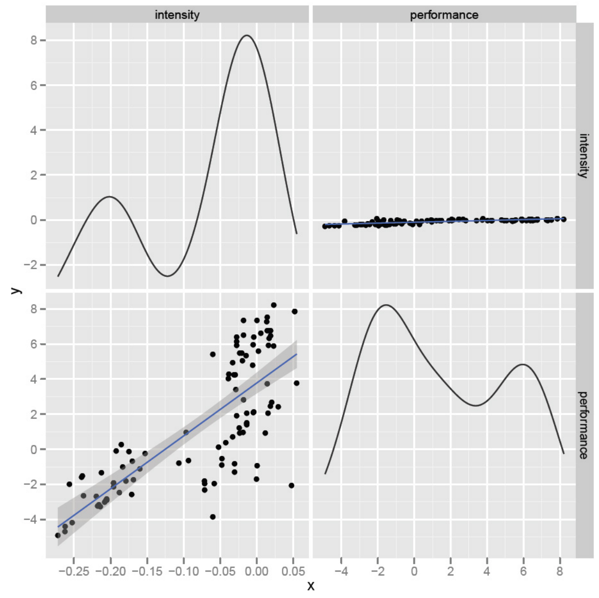
Figure 47-AES_Mem2reg_Box

Relying on ANOVA test on Table 7- (AES_ANOVA), a significant impact on the Performance metrics could be observed by mem2reg transformation

4-3-4 PCA

Provided with the pre-knowledge defined at the beginning of the benchmark, it was expected that the performance could be the first principal as the high level synthesis field mostly focus on the optimizing the performance, not the intensity at the gate level. The figures are as bellow:

Inline



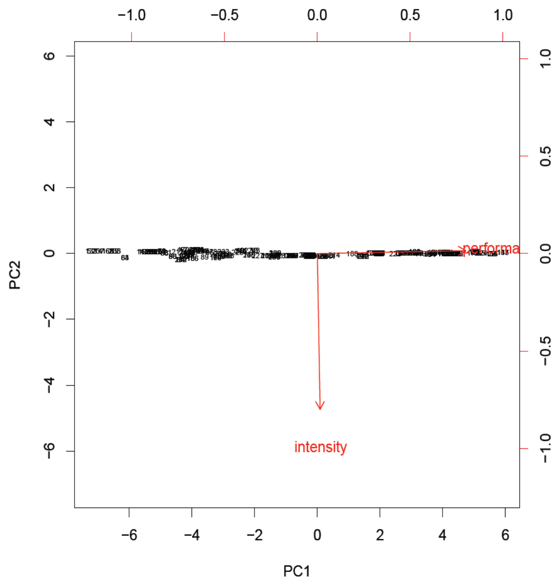
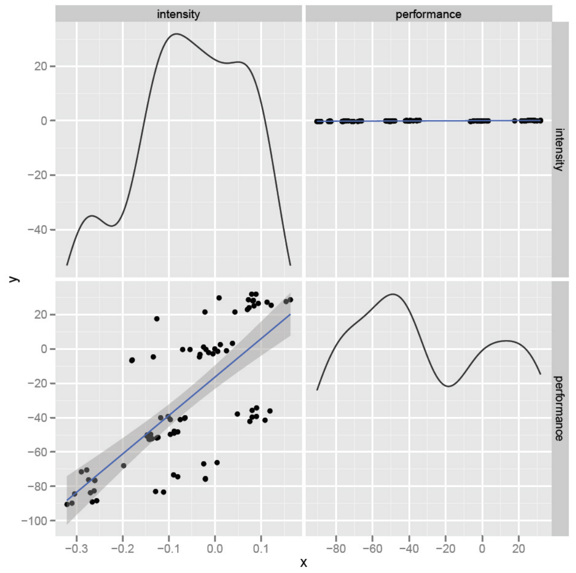


Figure 48-AES_Inline_PCA

Loop_reduce



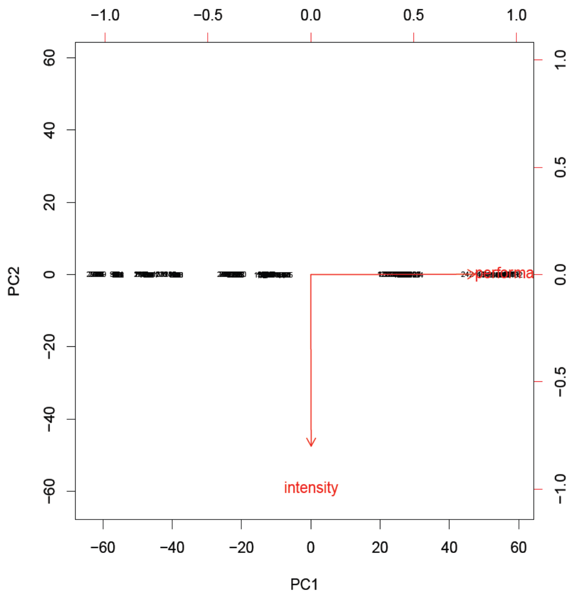


Figure 49-AES_Loop_reduce_PCA

Mem2reg

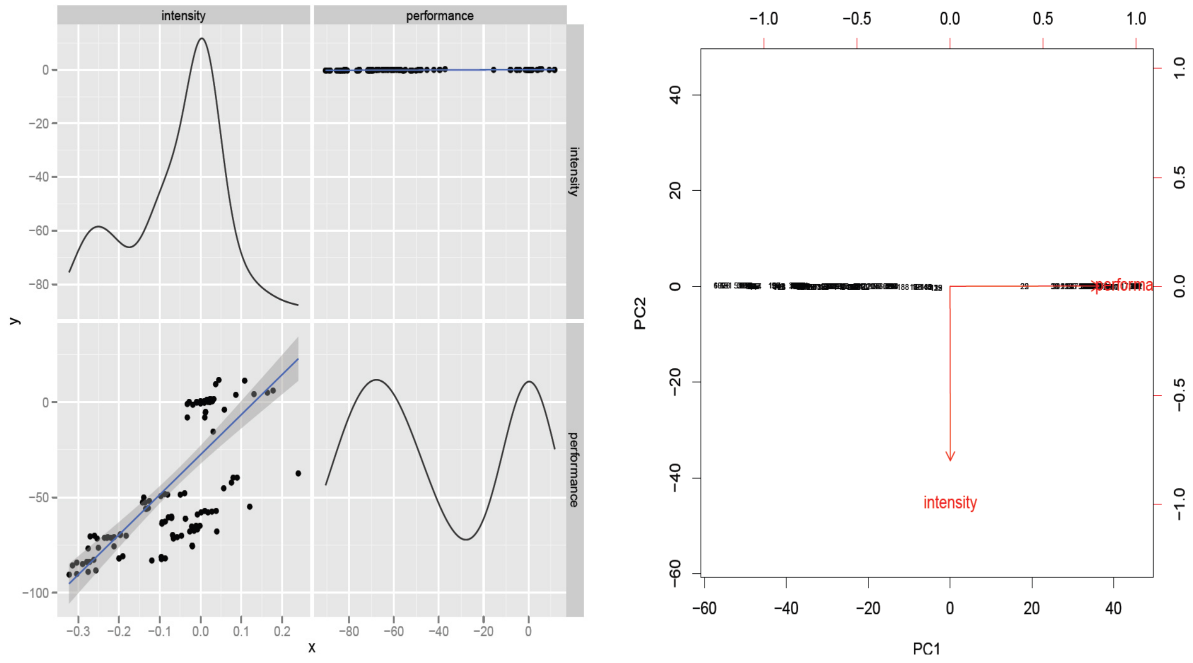


Figure 50-AES_Mem2reg_PCA

Provided with the results above, it clarifies out previous hypothesis regarding the low intensity benchmarks, all three have the Performance as the first principal component and intensity as the second with more or less the same degree between the first PCA and the second.

4-3-5 Densities

In this section the densities of the so-far explained parameters are going to be illustrated while the second parameter, namely, **scalarrepl**, is activated as well.

Inline

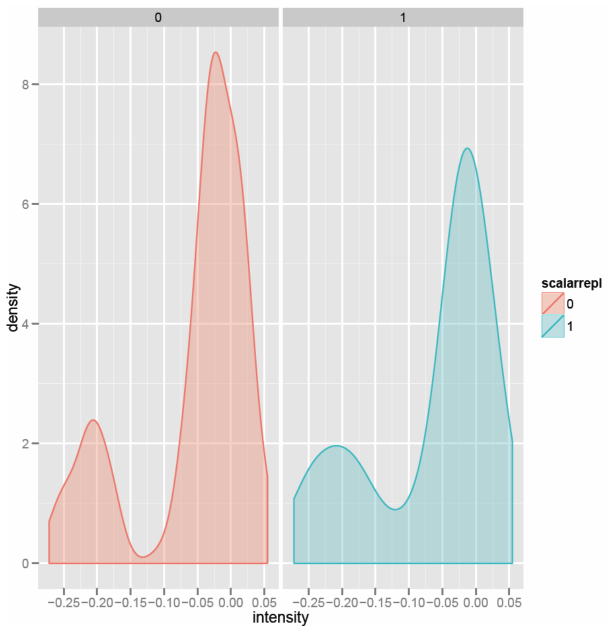


Figure 51-AES_Inlie_Densities

As it could be observed, by activating the second parameter (scalarepl) the performance of the whole compilation system will be reduced.

Loop_Reduce

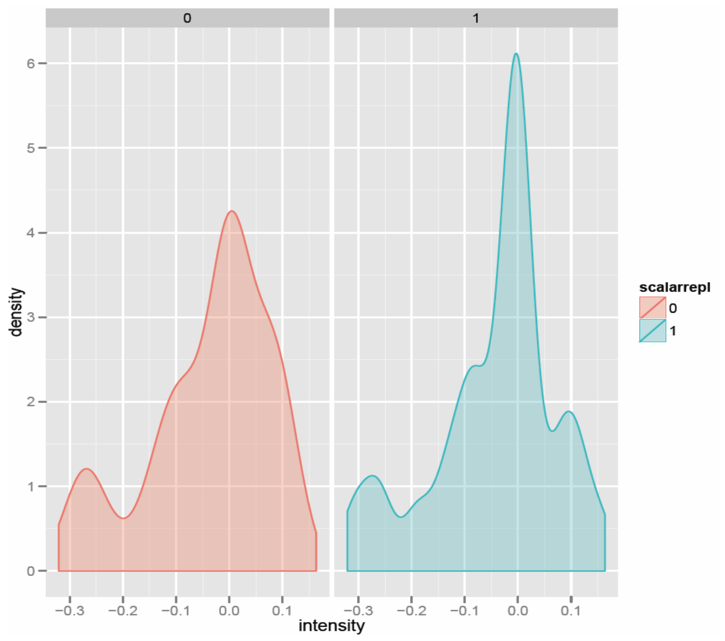


Figure 52-AES_Loop_reduce_Densities

Mem2reg

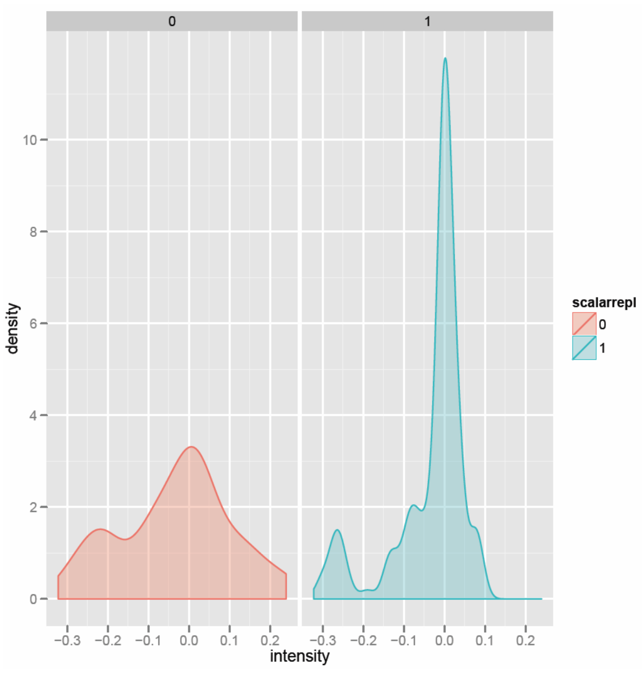


Figure 53-AES_Mem2reg_Densities

4-3-6 AES Synthesis Conclusion

Just like the other benchmark, in this section the whole synthesis table will be illustrated for the reference. The intensity quantitative will be as expected low comparing with *GSM*, but the effect of activating the optimization parameters could be observed on each and every compiler parameters.

In each section the values have been calculated and reported. The passed parameters in the ANOVA and Kruskal-Wallis test have been marked with green box in order to be distinguished.

	ANOVA (<5%)		KRUSKAL (<5%)		CORR		Corr on delta (5%)		CORR (PCA) 5%		DATA summmary	
Parameters	INT	OPS	INT	OPS	INT	OPS	INT	OPS	INT	OPS	AVG increment of OPS	Average increment of INT
constprop	1	1	1	1	1st Qu.:0 Median :0 Mean :0 3rd Qu.:0 Max. :0 Min. :0	Min. :0 1st Qu.:0 Median :0 Mean :0 3rd Qu.:0 Max. :0	NA	NA	NA	PC1 NA	0.1	-0.02
dce	0.976	0.997	0.9701	0.9987	Min. :0.0000000 1st Qu.:0.0000000 Median :0.0000000 Mean :0.0002475 3rd Qu.:0.0000000 Max. :0.0611400	Min. :0.00000 1st Qu.:0.00000 Median :0.00000 Mean :0.01109 3rd Qu.:0.00000 Max. :2.73900	NA on 5%	NA on 5%	NA on 5%	PC1 NA	3	0.02
inline	4.14E-13	3.93E-01	1.79E-08	2.64E-01	Min. :-0.27196 1st Qu.:-0.07695 Median :-0.02740 Mean :-0.05912 3rd Qu.: 0.00035 Max. : 0.05463	Min. :-4.905 1st Qu.:-1.043 Median : 2.111 Mean : 2.302 3rd Qu.: 5.486 Max. : 8.220	-loop_rotate +simplyfycfg +performance	-licm -loop_rotate +intensity +loop_rotate	+simplifycfg +performance	PC1 +intensity -loop_rotate -licm	2	-0.08
instcombine	0.0717	0.129	0.07315	0.01573	Min. :-0.23282 1st Qu.:-0.01834 Median : 0.01037 Mean : 0.01435 3rd Qu.: 0.06256	Min. :-10.285 1st Qu.: -0.063 Median : 1.651 Mean : 4.138 3rd Qu.: 7.989	-licm +performance	-LICM -loop_reduce +intensity	+performance -loop_rotate	PC1 +intensity -licm	0.69	-5.17
licm	0.768	7.26E-02	0.5413	5.57E-04	Min. :-0.196980 1st Qu.:-0.028790 Median :-0.006050 Mean :-0.002445	Min. :-19.295 1st Qu.:-11.066 Median : -8.641 Mean : -5.020	+loop_rotate +performance	-inline +intensity	+loop_rotate +performance	PC1 -inline +intensity	24.5	-0.01
loop_reduce	9.52E-07	<2e-16	1.96E-04	3.18E-16	Min. :-0.32081 1st Qu.:-0.09698 Median : 0.00000 Mean :-0.03971 3rd Qu.: 0.01170	Min. :-90.43 1st Qu.:-66.06 Median :-34.41 Mean :-27.48 3rd Qu.: 0.00	-inline	NA on 5%	-inline	PC1 NA on 5%	60.5	0.05
loop_rotate	<2e-16	3.18E-16	< 2.2e-16	< 2.2e-16	Min. :-0.23934 1st Qu.:-0.14654 Median :-0.07540 Mean :-0.07274	Min. :-43.110 1st Qu.:-38.345 Median :-28.870 Mean :-22.049	NA on 5%	NA on 5%	NA on 5%	PC1 NA on 5%	-17.5	-0.07
loop_unroll	0.979	0.999	9.66E-01	0.9965	Min. :0.000000 1st Qu.:0.000000 Median :0.000000 Mean :0.000188 3rd Qu.:0.000000 Max. :0.046820	Min. :-1.112000 1st Qu.: 0.000000 Median : 0.000000 Mean :-0.004466 3rd Qu.: 0.000000 Max. : 0.000000	NA on 5%	NA on 5%	NA on 5%	PC1 NA on 5%	0.02	-0.03
loop_unswitch	0.907	0.992	0.8921	0.9515	Min. :-0.0702900 1st Qu.: 0.0000000 Median : 0.0000000 Mean : 0.0009852 3rd Qu.: 0.0091600 Max. : [illegible]	Min. :-1.87400 1st Qu.:-0.20675 Median : 0.00000 Mean :-0.02866 3rd Qu.: 0.16925 Max. : [illegible]	NA on 5%	-loop_rotate -simplifycfg	NA on 5%	PC1 -loop_rotate -simplifycfg	0.001	0.02
mem2reg	8.40E-05	<2e-16	0.0009124	2.20E-16	Min. :-0.322500 1st Qu.:-0.088130 Median :-0.000125 Mean :-0.033323 3rd Qu.: 0.023410 Max. : 0.239420	Min. :-90.47400 1st Qu.:-67.77700 Median :-39.71900 Mean :-33.75415 3rd Qu.: 0.00375 Max. : 11.73500	-inline	'instcombine	-inline	PC1 -instcombine	-32.3	-0.03
memcpyopt	1	1	1	1	Min. :0 1st Qu.:0 Median :0 Mean :0 3rd Qu.:0 Max. :0	Min. :0 1st Qu.:0 Median :0 Mean :0 3rd Qu.:0 Max. :0	NA	NA	NA	PC1 NA	0	0
reassociate	9.85E-01	9.87E-01	2.20E-16	9.29E-01	Min. :-0.0100800 1st Qu.: 0.0000000 Median : 0.0000000 Mean :-0.0001611 3rd Qu.: 0.0000000 Max. : [illegible]	Min. :-1.01000 1st Qu.: 0.00000 Median : 0.00000 Mean :-0.04526 3rd Qu.: 0.00000 Max. : [illegible]	+instcombine +performance	-loop_rotate -instcombine +intensity	+instcombine +performance	PC1 -loop_rorate -instcombine +intensity	-1.5	-0.01
scalarrepl	0.146	0.855	0.2368	0.7468	Min. :-0.20737 1st Qu.: 0.00000 Median : 0.00000 Mean : 0.01298 3rd Qu.: 0.02166 Max. : 0.19692	Min. :-15.40700 1st Qu.: -0.03975 Median : 0.00000 Mean : -0.52311 3rd Qu.: 0.00000 Max. : 14.91700	NA on 5%	NA on 5%	NA on 5%	PC1 NA on 5%	-3.1	0.04
sccp	0.985	0.999	0.9645	0.9967	Min. :-0.0468200 1st Qu.: 0.0000000 Median : 0.0000000 Mean :-0.0001463 3rd Qu.: 0.0000000 Max. : 0.0106900	Min. :-0.32900 1st Qu.: 0.00000 Median : 0.00000 Mean : 0.00317 3rd Qu.: 0.00000 Max. : 1.11200	NA	NA	NA on 5%	PC1 NA on 5%	0	0
simpifycfg	0.205	0.898	0.4234	0.6828	Min. :-0.04990 1st Qu.: 0.00000 Median : 0.00000 Mean : 0.01117 3rd Qu.: 0.04052 Max. : 0.06376	Min. :-1.6020 1st Qu.: 0.0000 Median : 0.0000 Mean : 0.3735 3rd Qu.: 1.1490 Max. : 2.1560	NA on 5%	NA on 5%	-licm -sccp	PC1 NA on 5%	3	0.02

Chapter 5

Conclusions

Based on the experimental results mentioned in the previous chapter, in this chapter of the thesis the conclusions and final evaluations of the results will be illustrated. Finally, the next chapter will describe future evaluation of the thesis work.

5-1 Targeted Problem

The main contribution of this dissertation was focused on explore, evaluate and analyze the compiler options parameters in VLIW processor. As showed in Chapters 3 and 4, the methodologies and tool-chain were designed, implemented and exploited. Design space exploration was used in order to benefit the designer, to prune the large amount of unnecessary design space and actuate the multi-objective problem for the better best trade-offs .

5-2 Approach Review

As it was depicted in Figure 5-Tool-chain Schematic, the designed methodology is able to explore multi-benchmark system starting from high level synthesis to high performance applications. *MOST* (refer to 3-

2-2 Multi-Objective System Tuner (MOST)) is able to set the type of DoE and the sampling mode which is needed in order to explore the benchmarks. Using two powerful open-sourced compilers, namely, *LLVM* and *VEX* (refer to 3-2-3 LLVM and 3-2-4 HP-VEX), resulted in transforming the source codes using the interested optimization parameters. Consequently we evaluated the performance of the compilation and calculate the needed metrics in order to be fit in the performance model, namely, *Roofline* (refer to 2-1-5 Performance Model and Floating Point).

Figures have been drawn by open-source statistical software *R* in Linux for synthesizing. Using hundreds of results for five explored benchmarks, there could be common explanations in order to derive a trend of activities regarding the mentioned compiler parameters which is going to be elaborated in the following section.

5-3 Analysis Result Conclusion

5-3-1 per Benchmarks

No. 1 – GSM

In this dissertation (refer to the section 4-1 Benchmark No. 1- GSM) three out of fifteen compiler parameters have been illustrated by figures and explanations. For the complete review of the benchmark please refer to the section 4-4-8 GSM Conclusion. Regarding the depicted figures it can be observed that:

Looking at the benchmark results, having acceptance value α set equal to 5%,

1. **For ANOVA:**
    - **a.** Inline, Licm, Loop_reduce, Loop_rotate, reassociate and scalarrepl have passed the ANOVA test for intensity metrics
    - **b.** Inline, instcombine, licm, loop_reduce, loop_rotate, mem2reg, reassociate and scalarrepl have passed the ANOVA performance metric test.

2. **For Kruskal-Wallis:**

   a. Inline, Licm, Loop_reduce, Loop_rotate, reassociate and scalarrepl have passed the Kruskal test for intensity metrics
   b. Inline, instcombine, licm, loop_reduce, loop_rotate, mem2reg and reassociate have passed the Kruskal performance metric test

3. The **maximum intensity** observed in those 15 compiler parameters was 191.6 (flops/byte) which belongs to constprop, dce, inline, instcombine, licm, loop_reduce, loop_rotate, loop_unroll, loop_unswitch, mem2reg, reassociate, sccp, simplifycfg.

4. The **maximum performance** value observed for this metrics in those 15 compiler parameters was 754.8 (Gflops/s) which belongs to constprop, loop_unswitch and memcpyopt.

5. Performance and intensity metrics have been observed impacted by each other in the same direction for compiler parameters instcombine, mem2reg, reassociate, scalarrepl and simplifycfg

6. Performance metrics have been observed as the most impressing component in **Principal Component Analysis** for all the 15 compiler parameters with 99% as the highest value for reassociate. In addition, the highest proportion value of valiance for intensity was seen as 31% for loop_unroll.

7. Regarding the **average increment**, reassociate has 19.88 and inline has -10.15 as the highest decrement one for intensity, in addition, for performance, the highest observed was 26.11 for reassociate and -52.14 for loop_rotate.

No.2 AES

Looking at the benchmark results, having acceptance value α set equal to 5%,

1. **For ANOVA:**
    a. Only Inline, Loop_reduce and Loop_rotate have passed the ANOVA test for intensity metrics
    b. Only licm, loop_reduce, loop_rotate and mem2reg have passed the ANOVA performance metric test.

2. **For Kruskal-Wallis:**
    a. Inline, Loop_reduce, Loop_rotate, mem2reg and reassociate have passed the Kruskal test for intensity metrics
    b. instcombine, licm, loop_reduce, loop_rotate, mem2reg and reassociate have passed the Kruskal performance metric test

3. The **maximum intensity** observed in those 15 compiler parameters was 0.19692 (flops/byte) which belongs to scalarrepl. (as it was expected the intensity in these benchmark suits are low since they are high level synthesis application and the effect of performance is more interested in exploring these application rather than intensity)

4. The **maximum performance** value observed for this metrics in those 15 compiler parameters was 31.87 (Gflops/s) which belongs to loop_reduce.

5. Performance and intensity metrics have been observed impacted by each other in the same direction for compiler parameters inline, instcombine, licm and reassociate

6. Performance metrics have been observed as the most impressing component in **Principal Component Analysis** for all the 15 compiler parameters with near 100% as the highest value for reassociate. In addition, the highest proportion value of valiance for intensity was seen as 0.260 % for loop_unswitch.

7. Regarding the **average increment**, loop_reduce has 60 value and mem2reg has -32.3 as the highest decrement one for performance metric, in addition, for intensity, the highest observed was 0.05 for loop_reduce and -5.17 for instcombine.

No.3 – No.5 Benchmarks

For the sake of synthesis in this dissertation, the results and synthetic conclusions for the remaining benchmarks have been moved to the appendix chapter at the end.

5-3-2 Cross Benchmarks

Extracting the trends in each and every science could be a difficult and complicate task which needs to be taken into account hundreds of factors such as induction rules, enough samples, risk and error evaluation, etc.

In this dissertation, the main goal was designing and implementing a methodology for setting benchmarks and performance evaluation of compiler options in VLIW processor, therefore, the generalization has to be taken care in a future defined work which will be mentioned in the following chapter.

- As first hypothesis, it could be observed that all the transformations of the *AES*, have the Performance by far as their principal component.
- In the *GSM* benchmark, the latter result is the same with little mixture of intensity to the PCA, as the benchmark have put a large load on the system in the gate.

In both explored benchmarks; only loop_reduce and loop_rotate have had significant impact on both metrics (Intensity and Performance), while:

- Inline, licm, mem2reg and reassociate have at least two metrics impacted in both two benchmarks.
- Instcombine and scalarrepl have only one metric impacted.

5-3-3 ANOVA Cross-Benchmark

Using *ANOVA* and *Kruskal-Wallis* Analyses defined in Sections 4-2-1 ANOVA and 4-2-2 Kruskal-Wallis, hereby there is going to be the cross-benchmark review of the experimental results:

	GSM	AES	ADPCM	JPEG	Blowfish
Constprop					
Dce					
Inline	✓		✓	✓	
Instcombine	✓		✓	✓	✓
Licm	✓	✓	✓	✓	✓
Loop_reduce	✓	✓	✓	✓	✓
Loop_rotate	✓	✓	✓	✓	
Loop_unroll					
Loop_unswitch					
Mem2reg	✓	✓	✓	✓	✓
Memcpyopt					
Reassociate	✓				
Scalarrepl	✓				✓
Sccp					
simplifycfg					

Table 9-ANOVA_Cross-benchmak_Performance

The acceptance rate of (α) variable has been set to 5% as it has been defined in the Section 3-4-1 ANOVA Analysis, therefore, the transformation which have pass this threshold acceptance rate have been marked with a tick checkmark sign (✓). This shows the Performance metric (Ops) has had the significant impact on the medians of the transformation in that specific benchmark.

Observing Table 9- (ANOVA_Cross-benchmak_Performance), it could be seen that four transformations, namely, *licm*, *loop_reduce* and *mem2reg* have the same trend on all the explored benchmarks. Relying on their own intrinsic behaviors, these transformations could impact the performance in the proposed methodology.

5-3-4 Kruskal-Wallis Cross-Benchmark

The overall cross-benchmark view of the Kruskal-Wallis analysis have been mentioned in the Table 10- (Kruskal-Wallis_Cross-benchmark_Performance):

	GSM	AES	ADPCM	JPEG	Blowfish
Constprop					
Dce					
Inline	✓		✓	✓	
Instcombine	✓	✓	✓	✓	✓
Licm	✓	✓		✓	✓
Loop_reduce	✓	✓	✓	✓	✓
Loop_rotate	✓	✓	✓	✓	
Loop_unroll					
Loop_unswitch					
Mem2reg	✓	✓	✓	✓	✓
Memcpyopt					
Reassociate	✓	✓			
Scalarrepl					✓
Sccp					
simplifycfg					

Table 10- Kruskal-Wallis_Cross-benchmark_Performance

As it could be observed in the Table 10- (Kruskal-Wallis_Cross-benchmark_Performance), in this analysis, three transformations, namely, *instcombine*, *loop_reduce* and *mem2reg* have passed all benchmark test regarding impacts on performance metric.

5-3-5 Parameters Effect

Similar to what we have done with the correlation matrix on deltas defined in 3-4-3 (Correlation Analysis), in order to have useful cross-benchmark high-level view between the parameter interactions, a interaction table could be calculated with transformation parameters on the sides, therefore it will be diagonal, and number of positive-negative interaction between parameters and metric (Performance) in each transformation per benchmark could be add up to sketch a disk bubble. So the quantity of transformations multiply number of benchmarks could estimate the maximum number of interactions. The more the number of interaction is the higher the diameter of the bubble. In this case, the researcher could have a conclusive high level view to extract information out of the explorations. This analysis will show the effect of activation of the second transformation parameter on performance metric with respect to have the main transformation being activated already.

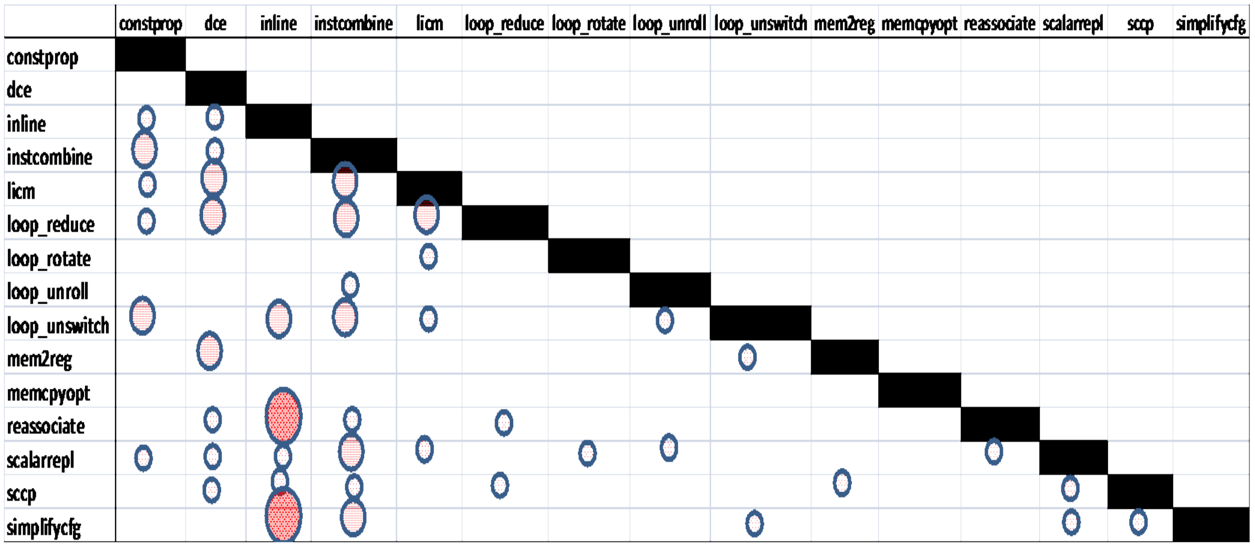


Figure 54- Transformations Bubble Effects

In the Figure 54- (Transformations Bubble Effects), four levels of effects have been illustrated:

1- No effects: no signs
2- Degree of effects equal to 1 : the white fill small ovals
3- Degree of effects equal to 2: medium size cross patterns ovals
4- Degree of effects equal to 3: large red filled ovals

It could be observed that having *reassociate* activated already, by adding *inline* transformation, we could expect to impact the performance. This phenomenon is also true for *simplifycfg* and *inline*.

Chapter 6

Future Works

As it mentioned on Chapter 5, the main objective of this dissertation was focus on using DSE for compiler parameters in VLIW processors. Consequently, the benchmarks used in order to be explored were mostly elaborated on seeing the effects of using these options in the issue. Due to the complexity and size of the topic, there are some future ideas that could be taken care of as following.

6-1 Combining Architectural Parameters

In Table 2-Our Problem Design Space Exploration_ Example, the range of these architectural parameters have been mentioned already. Combining the so-far topic with architectural parameters will add complexity and bigger orders of explorations to the problem; therefore, it could be an interesting future work which needs to be elaborated in near future.

Indeed, architectural parameters involved the infrastructures and hardware machines to the problem which could be really interesting for the industry and enterprise partners in order to be researched on. Choosing the best suit of architectural configurations

6-2 Extended Benchmarks

Since multiple benchmark usage was one of the key features of the designed methodology in this dissertation, it could be used with so many great and more sophisticated benchmarks i.e. high performance video applications, Encoder/Decoder applications, etc.

By the date of writing this dissertation, the efforts of embedding a new benchmark, namely *H264 Decoder* [44], have been started for a while. Hopefully finishes exploring soon to have better reasoning about the phenomena of impacting metrics.

6-3 Further Algorithms of Optimizations

There are bunch of other interesting problems still on the course of research which actuate the need of extending the current work for future. *Phase Ordering in compilers Optimization*, which has been an interesting target for researchers. A single sequence of optimization phases is highly unlikely to produce optimal code for every application (or even each function within an application) on a given machine. The problem of ordering optimization phases can be more severe when generating code for embedded applications. [45]

Bibliography

[1] S. Balakrishnan, "Very Long Instruction Word Processor," *Ressonance,* pp. 61-68, December 2001.

[2] V. Zaccaria, C. Silvano and G. Palermo, "MOST: Multi-Objective System Tuner - DSE for system architects," in *DATE*, Grenoble, France, 2011.

[3] LLVM Co., "LLVM," [Online]. Available: www.llvm.org.

[4] HP Co., "HP-VEX," [Online]. Available: www.vliw.org/vex/.

[5] B. R. Rau and J. A. Fisher, Instruction-Level Parallelism: history, overview, and perspective, Springer, 1993.

[6] VLIW Organization, "LLVM," [Online]. Available: http://www.vliw.org.

[7] J. A. Fisher, P. Faraboschi and C. Young, in *Embedded Computing; A VLIW Approach to Architecture, Compilers and Tools*, Morgan Kaufmann, 2005, pp. 57-63.

[8] B. Mathew, "Very Large Instruction Word Architectures (VLIW Processors and Trace Scheduling)," *Computer Engineering Handbook, CRC Press LLC,* 2001.

[9] R. Sherief, "Lectures on Design and Implementation of VLSI Systems," Brown University.

[10] H. E. Ziegler, in *Compiler-directed Design Space Exploration for Pipelined FPGA application*, University of southern california, 2006, pp. 1-10.

[11] "http://mathworld.wolfram.com/SimulatedAnnealing.html".

[12] "DoE Types," [Online]. Available: http://www.qualitytrainingportal.com/resources/doe/doe_types.htm.

[13] GNU, "GNU GCC Manual," 2012. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options.

[14] GNU Organization, [Online]. Available: www.gnu.org.

[15] LLVM Organization, "LLVM Transformations," http://llvm.org/docs/Passes.html.

[16] L. O.-. C. Lattner, "LLVM Language Reference Manual," 2012. [Online]. Available:

http://llvm.org/docs/LangRef.html#i_alloca.

[17] Briggs, M. Dubois and F. A., "Performance of Synchronized Iterative Processes in Multiprocessor Systems," in *IEEE Trans. on Software Eng.*, 1982.

[18] E. Boyd, W. Azeem, H. Lee, T. Shih, S. Hung and E. Davidson, "A Hierarchical Approach to Modeling and Improving the Performance of Scientific Applications on the KSR1," in *Int'l Conf. on parallel Processing*, 1994.

[19] S. Williams, A. Waterman and a. D. Patterson, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures," in *ACM Communication*, 2008.

[20] D. Fischer, J. Teich, R. Weper and U. Kastens, "Design space characterization for architecture/compiler co-exploration," in *CASES '01 Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, 2001.

[21] A. Halambi, P. Grun, V. Ganesh, A. Khare and N. Dutt, "EXPRESSION: a language for architecture exploration through compiler/simulator retargetability," in *conference on Design, automation and test in Europe*, NY, 1999.

[22] B. So, M. Hall and P. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *ACM SIGPLAN*, 2002.

[23] M. O'Boyle, Agakov and Felix, "Using machine learning to focus iterative optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[24] O. Mencer, D. Pearce and L. Howes, "Design space exploration with A Stream Compiler," in *Field-Programmable Technology (FPT)*, 2003.

[25] C. Dubach, T. Jones and M. O'Boyle, "Microarchitectural Design Space Exploration Using an Architecture-Centric Approach," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[26] ISO Organization, "C99 C standards," ISO/ITEC.

[27] P. Saraswat, E. Zamsha and A. Jankovic, "A fast and Efficient Simulated Annealing based Design Space Exploration for a Custom VLIW Architecture for GSM Decoder and Optimizations using VEX compiler," in *Relation 10.1.91*, 2008.

[28] Saptono, Brost and Yang, "Design Space Exploration for a custom VLIW Architecture: Direct photo printer hardware setting using VEX Compiler," in *Signal Image Technology*, 2008.

[29] M. K. Jain and G. K. Ranka, "VLIW BASED VEX TOOL AND VALIDATION OF SIM-A WITH VEX," *Journal of Global Research in Computer Science,* vol. 2, no. 9, 2011.

[30] European Standard (Telecommunications series), "http://WEBAPP.ETSI.ORG/exchangefolder/en_300724v080001p0.zip," Technical Committee Special Mobile Group (SMG), 1999.

[31] E.R.T.L Corporation, "http://www.ertl.jp/chstone/," Japan, E.R.T.L.

[32] S. Xydis and G. Palermo, "Most Generic Wrapper (MGW)," Politecnico Di Milano, Dei Department, System Architecture group, Milan, 2012.

[33] V. Zaccaria, "MOST (Multi-Objective System Tuner) Overview," Politecnico Di Milano, Department of Computer and Electric Eng., Milan, 2001-2011.

[34] R. Roy, Design of experiments using the Taguchi approach: 16 steps to product and process improvement, Wiley, 2001.

[35] http://llvm.org/Features.html, "LLVM- Features," LLVM Organization.

[36] HP-VEX, "Vex Systems," HP, 2012.

[37] J. Fisher, "Traced Scheduling, A Technique for Global Microcode Compaction," *Computers, IEEE Transactions on Computers,* Vols. C-30, no. 7, pp. 478-490, 1981.

[38] R-Project Organization, "R Statistical Software," [Online]. Available: www.r-project.org.

[39] D. MCFATTER, "Computational Formulas for ANOVA," Louisiana, USA.

[40] J. P. Key, Oklahoma State University, [Online]. Available: http://www.okstate.edu/ag/agedcm4h/academic/aged5980a/5980/newpage26.htm.

[41] T. Gaten, "Kruskal-Wallis non-parametric ANOVA," University of Leicester, [Online]. Available: http://www.le.ac.uk/bl/gat/virtualfc/Stats/kruskal.html.

[42] R. Rebonato, "The most general methodology to create a valid correlation matrix," Quantitative Research Centre of the NatWest Group, 1999.

[43] L. I. Smith, "A tutorial on Principal Components Analysis," University of Otago New Zealand, 2002.

[44] K. Sühring, "H.264/AVC Reference Software," [Online]. Available: https://ipbt.hhi.fraunhofer.de/. [Accessed 2012].

[45] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley and J. Davidson, "Finding effective optimization phase sequences," in *ACM SIGPLAN Notices*, 2003.

[46] P. V. a. K. S. C. Nilanjan Banerjee, "A Power and Performance Model for Network-on-Chip Architectures," in *Design, Automation and Test in Europe Conference and Exhibition, 2004.*

Proceedings, 2004.

[47] A. B. Kahng, B. Li, L.-S. Peh and K. Samadi, "ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Belgium ©2009, 2009.

[48] Y. Jin, N. Satish, K. Ravindran and K. Keutzer, "An Automated Exploration Framework for FPGA-based," in *Proceeding CODES+ISSS '05 Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, NY, USA, 2005.

[49] M. Palesi and T. Givargis, "Multi-objective design space exploration using genetic algorithms," in *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, 2002.

[50] J. KEINERT, M. STREUBUHR and T. SCHLICHTER, "SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Transactions on Design Automation of Electronic Systems (TODAES) TODAES,* vol. 14, no. 1, 2009.

[51] B. So, M. W. Hall and P. C. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *PLDI '02 Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[52] "DoE Definition," [Online]. Available: http://www.itl.nist.gov/div898/handbook/pmd/section3/pmd31.htm.

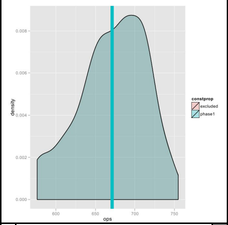
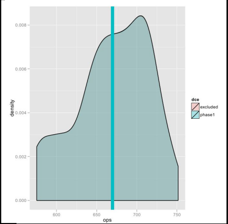
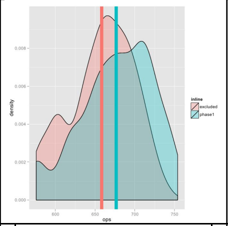
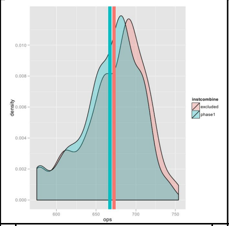
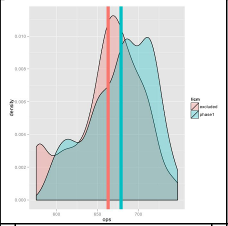
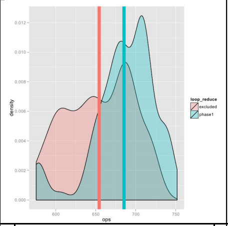
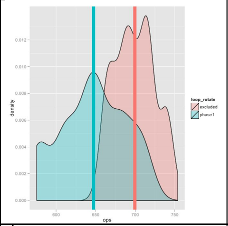
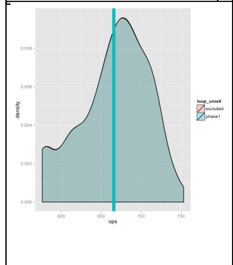
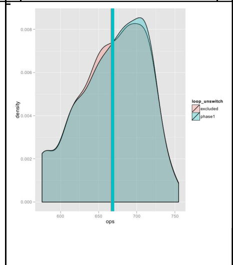
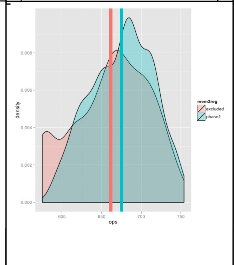
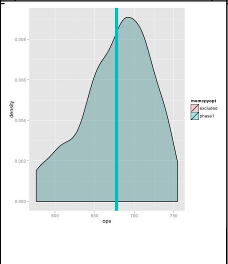
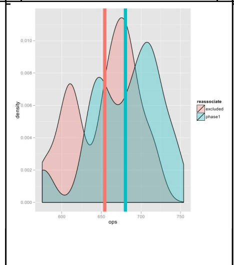
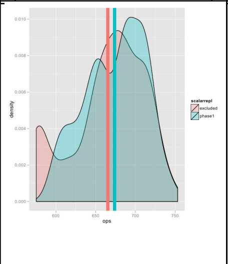
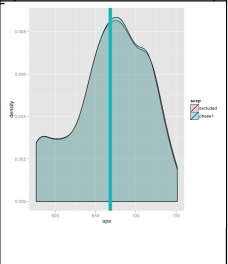
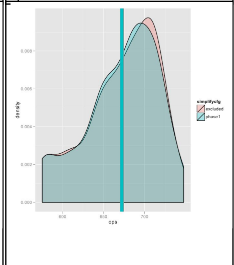
Appendix

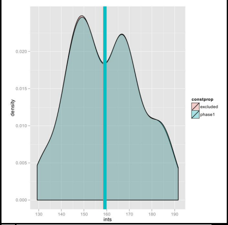
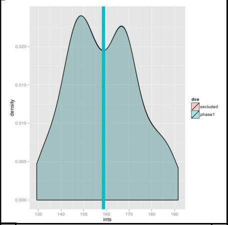
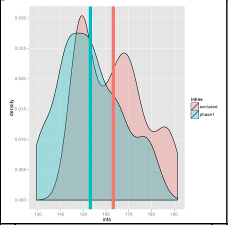
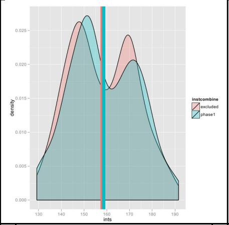
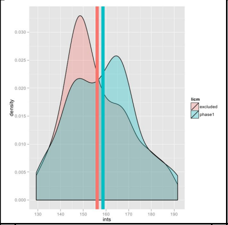
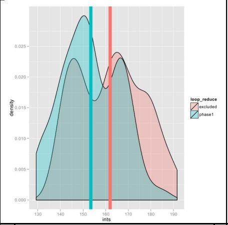
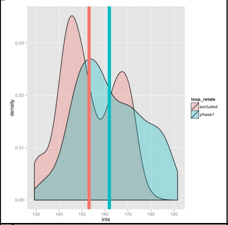
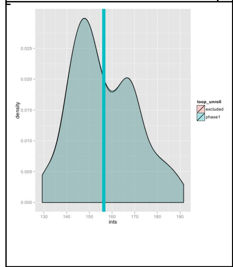
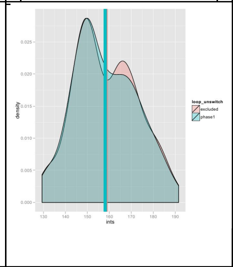
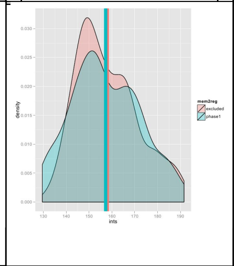
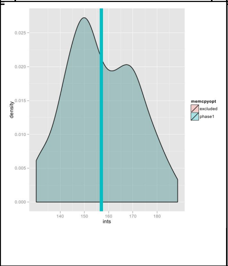
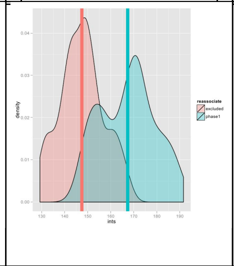
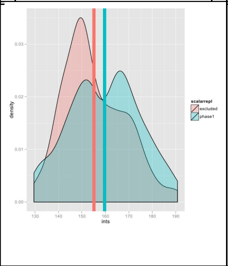
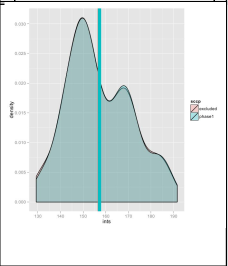
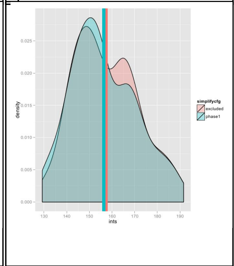
For the sake of abstractness in this dissertation, only two out of the 5 explored benchmarks have been mentioned during the content (refer to Experimental Results). One high intensity *GSM* and one out of the CHStone benchmark suits, namely *AES*.

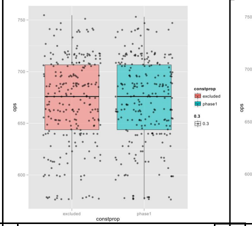
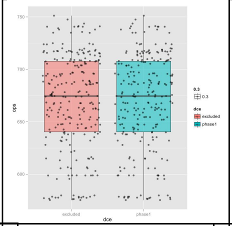
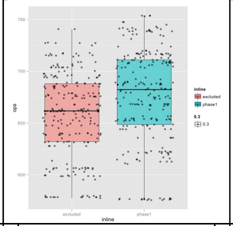
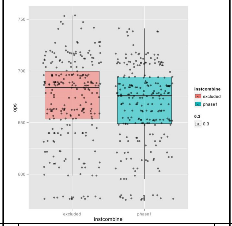
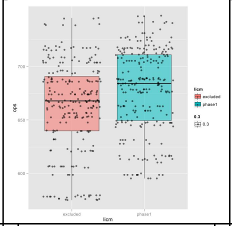
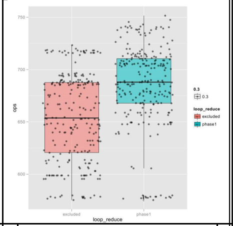
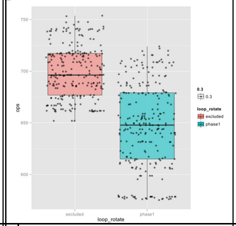
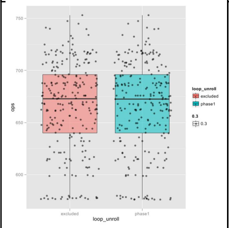
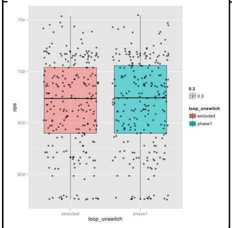
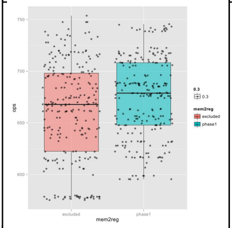
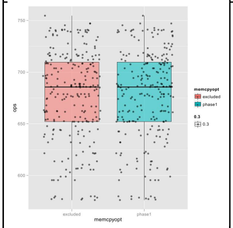
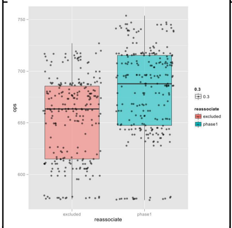
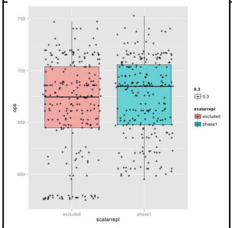
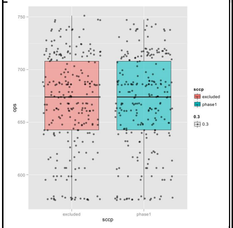
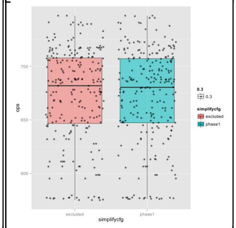
In this section all the results are going to be classified based on the benchmark-transformation, in this case the reader could get a clear idea of what have we done in this dissertation to analyze the compiler options for VLIW processors.

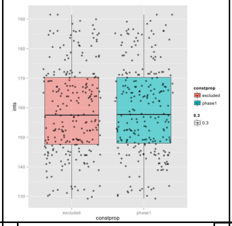
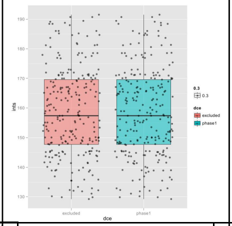
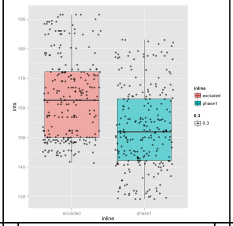
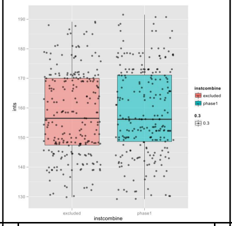
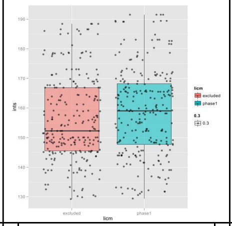
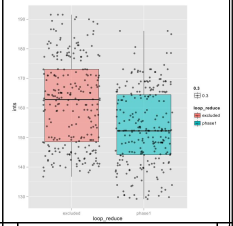
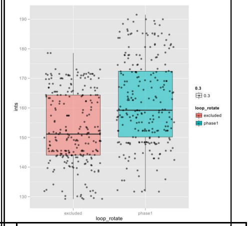
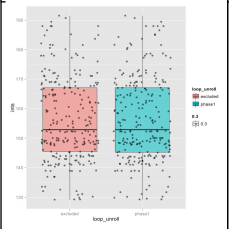
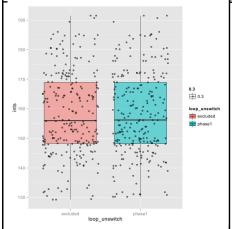
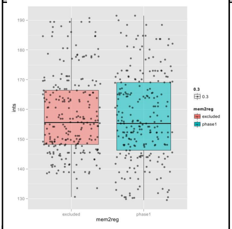
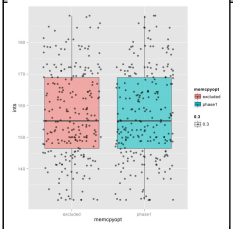
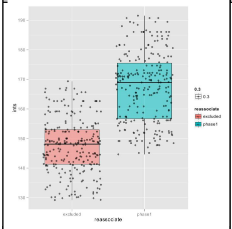
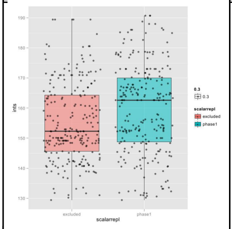
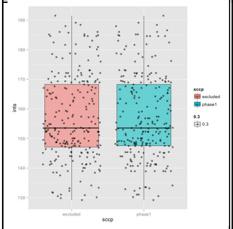
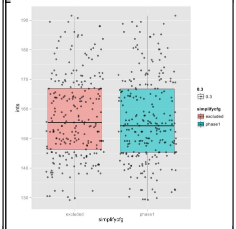
The trend of this section will be as following:

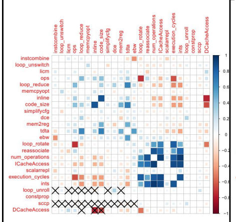
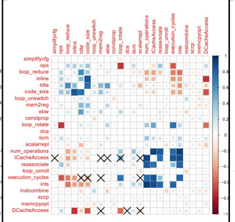
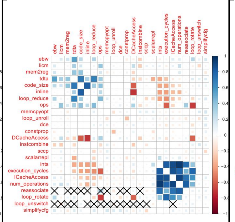
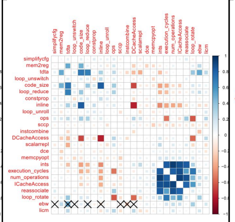
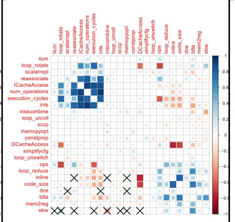
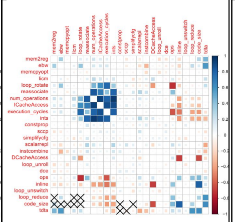
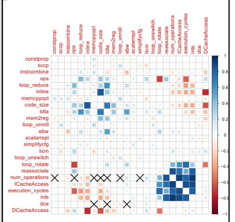
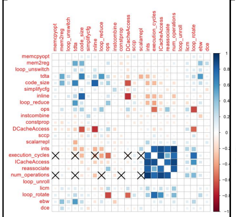
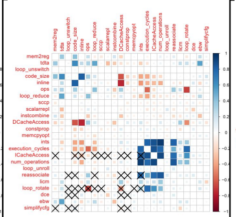
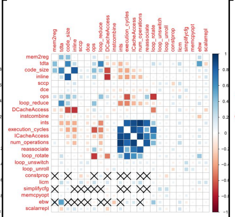
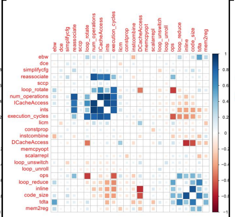
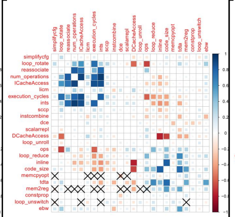
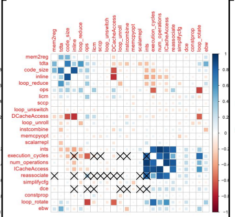
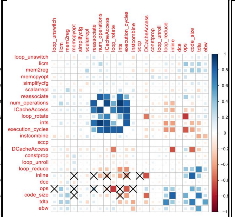
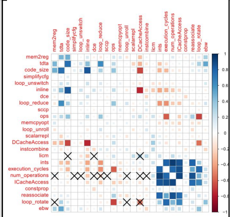
- Benchmark Name
  - Distributions
  - Box-Plots
  - Correlations
    - Raw Data
    - On Deltas
  - Scatter-Plot
  - Principal Component Analysis

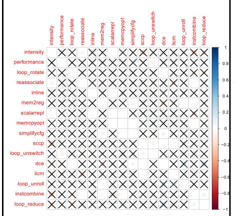
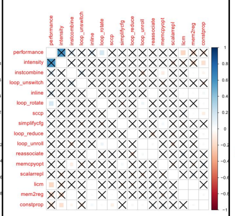
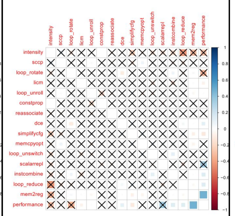
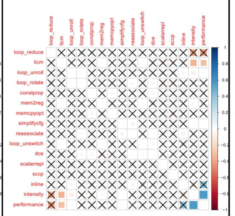
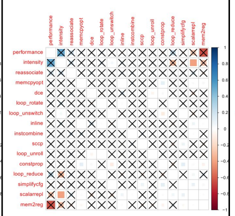
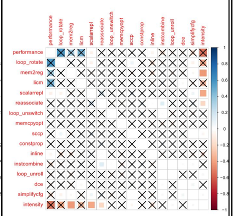
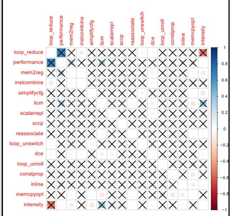
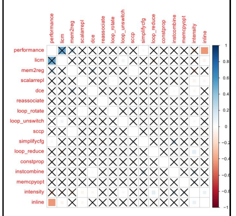
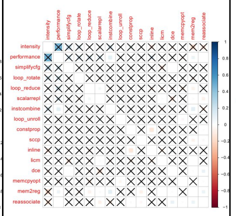
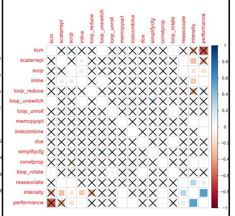
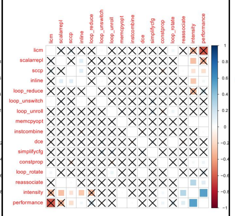
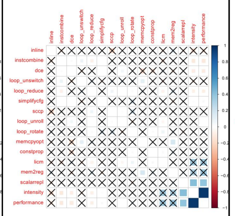
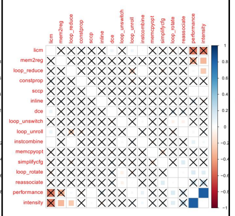
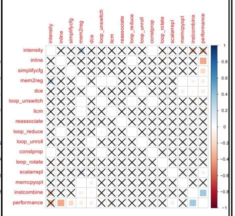
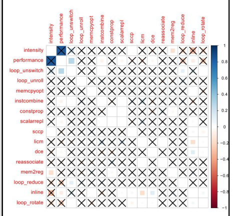
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
GSM Distributions Performance							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

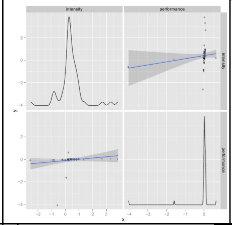
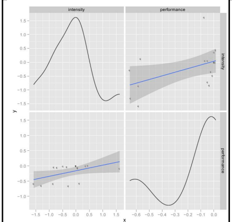
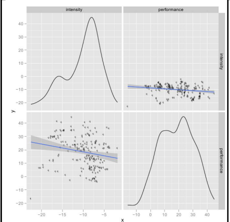
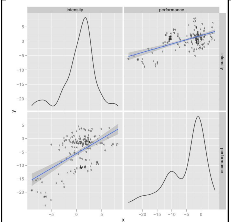
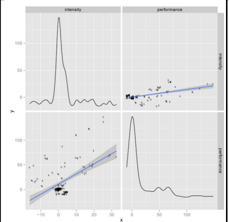
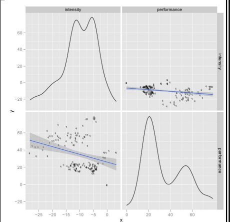
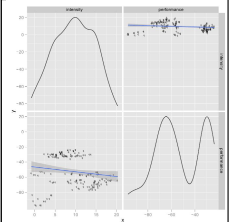
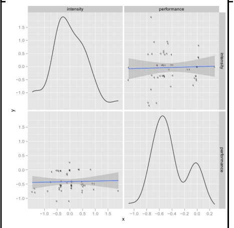
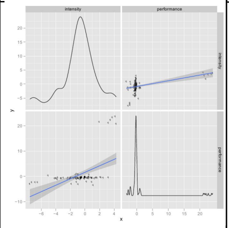
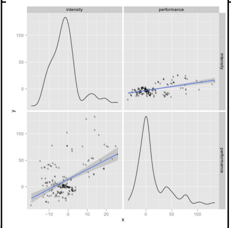
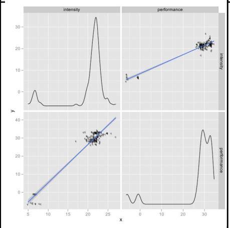
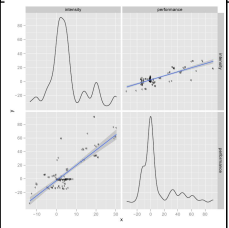
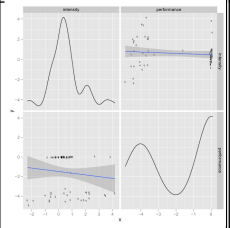
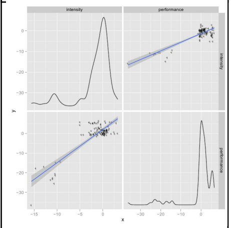
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
GSM Distributions Intensity							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

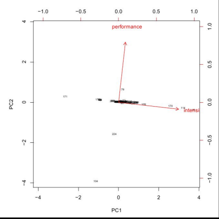
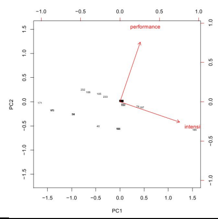
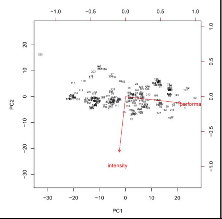
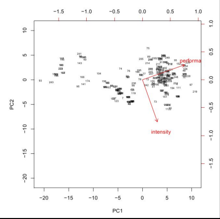
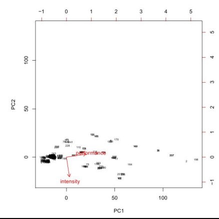
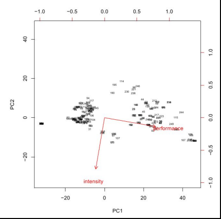
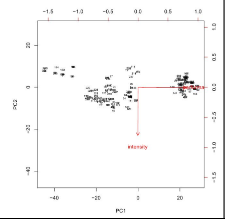
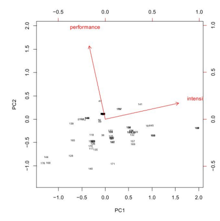
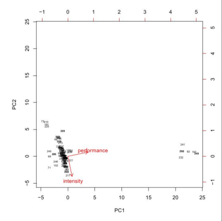
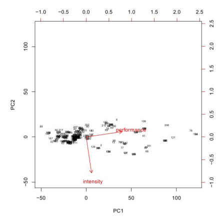
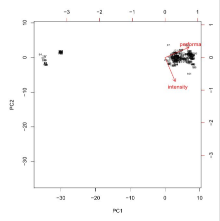
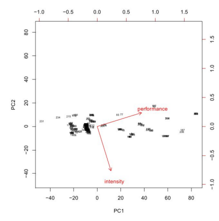
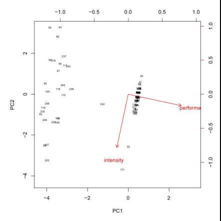
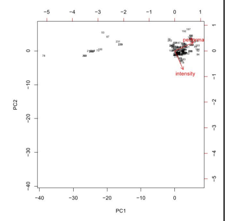
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
GSM **Box-plot** **Performance**							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

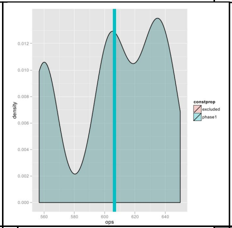
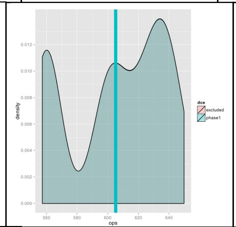
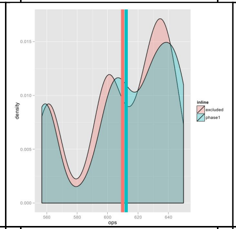
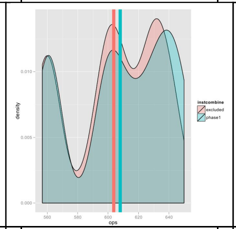
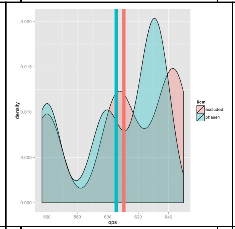
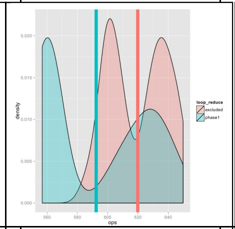
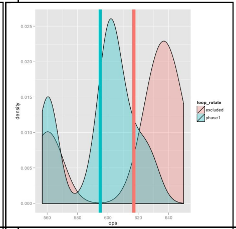
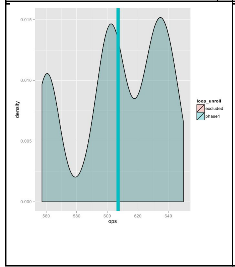
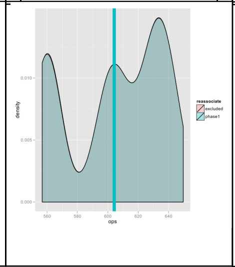
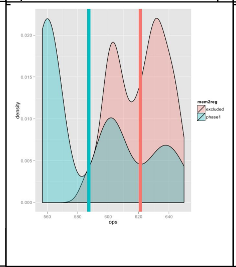
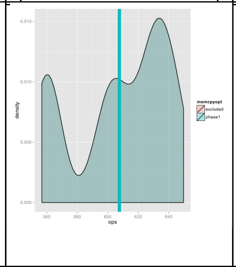
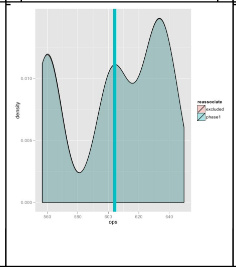
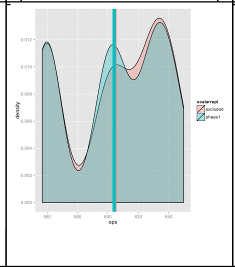
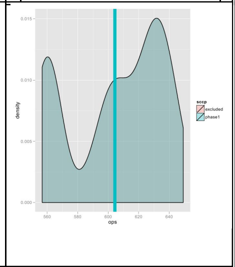
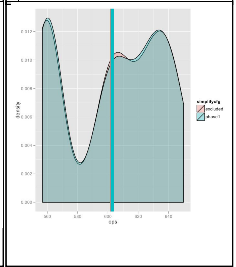
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
GSM **Box-plot** **Intensity**							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

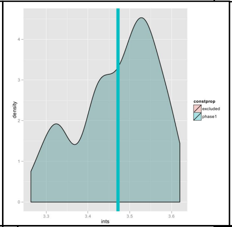
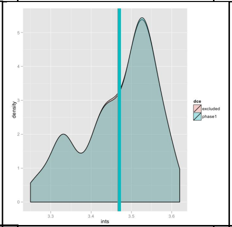
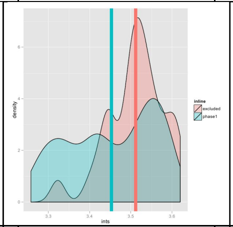
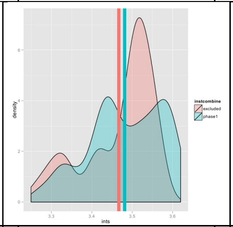
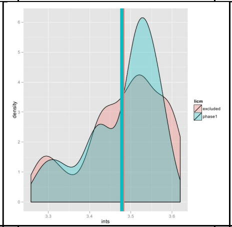
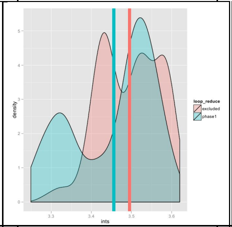
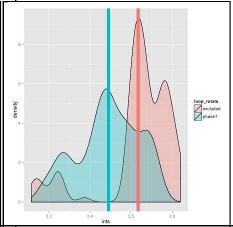
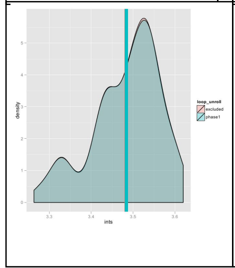
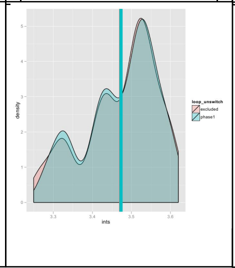
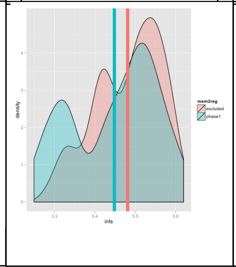
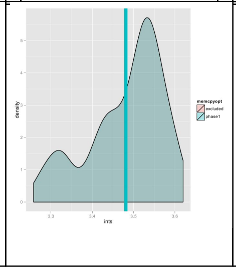
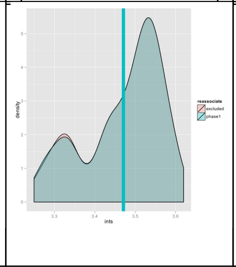
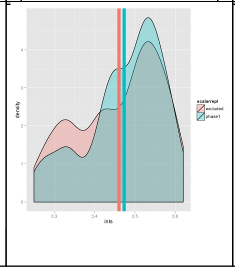
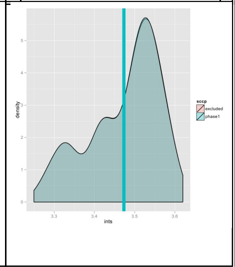
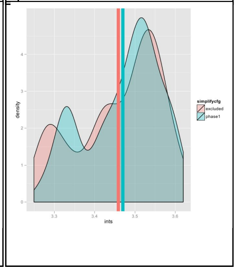
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
GSM Correlation on raw data							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

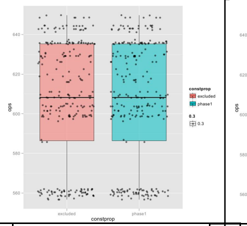
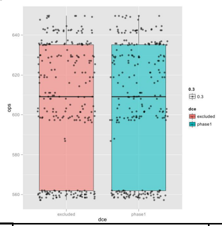
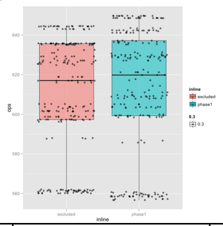
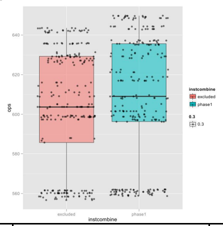
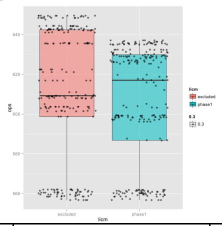
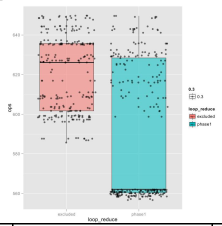
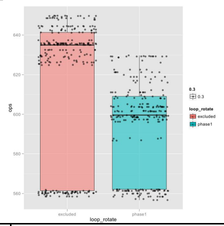
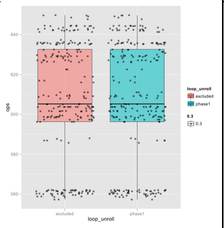
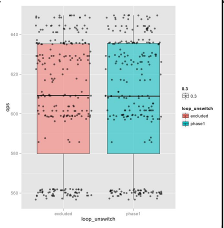
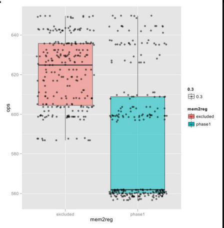
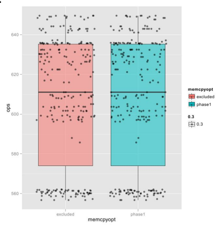
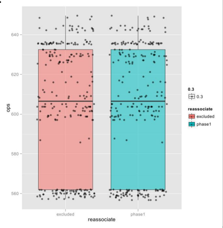
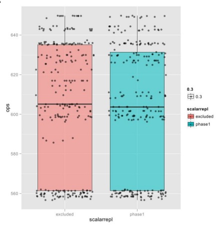
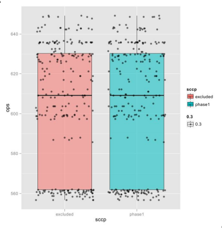
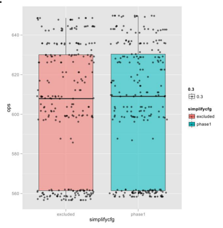
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
GSM Correlation on deltas							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

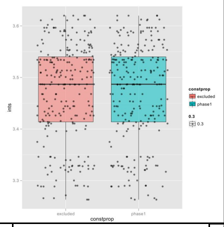
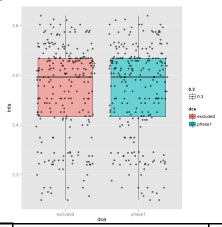
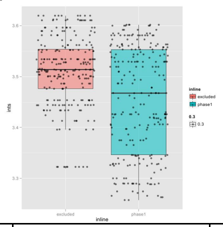
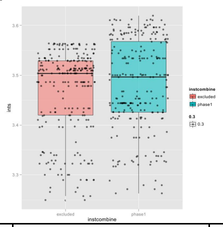
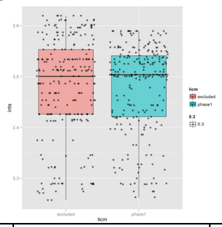
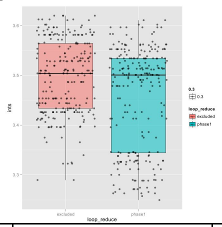
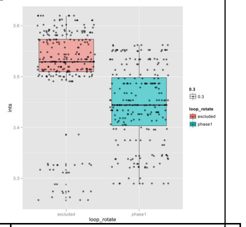
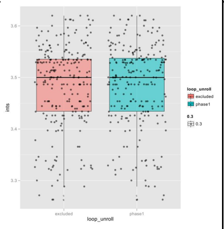
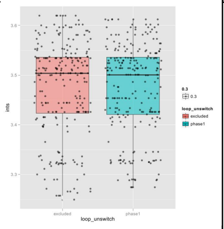
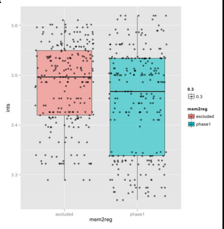
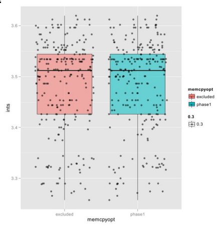
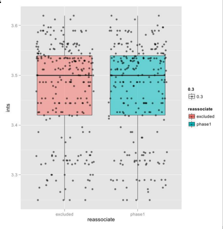
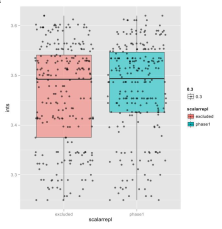
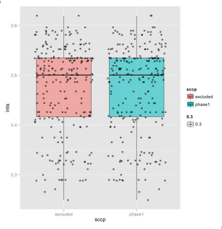
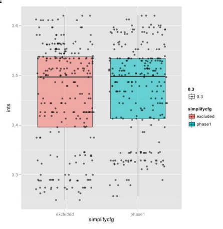
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
GSM Scatter-plot							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

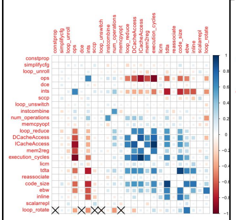
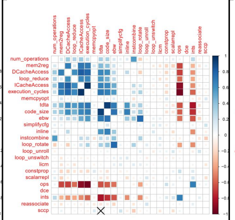
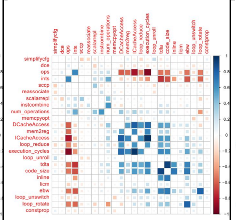
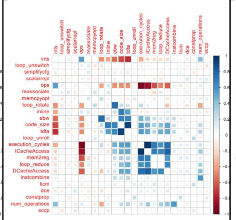
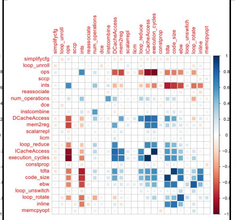
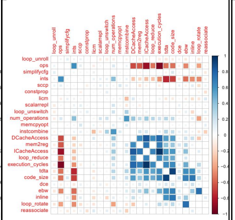
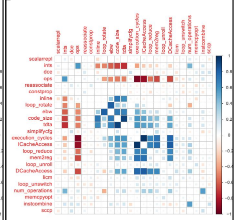
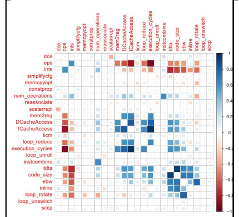
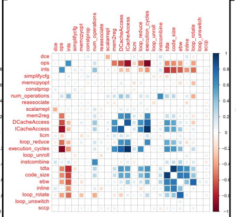
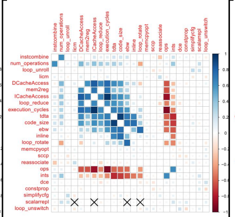
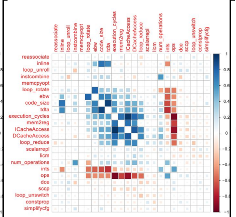
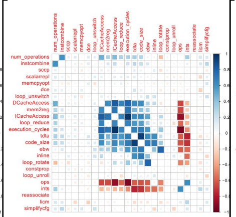
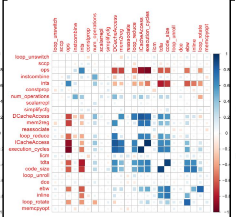
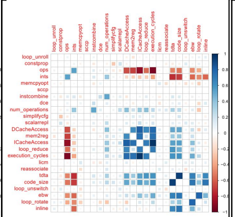
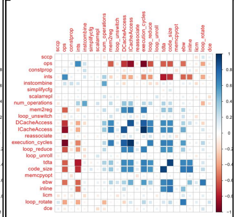
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
GSM Principal Component Analysis (PCA)							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

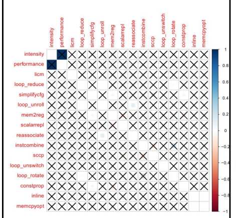
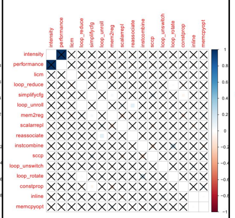
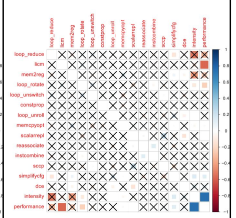
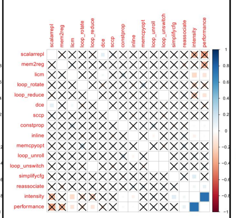
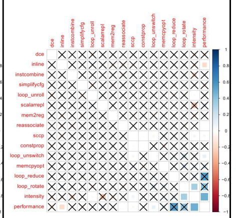
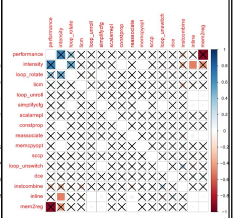
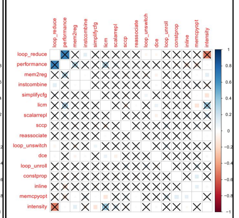
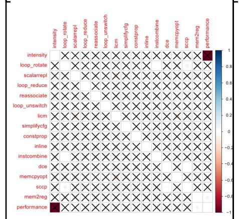
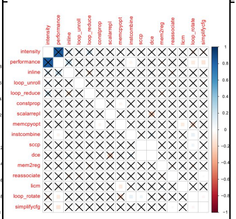
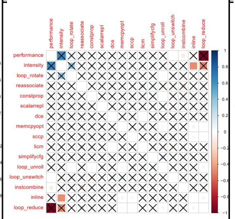
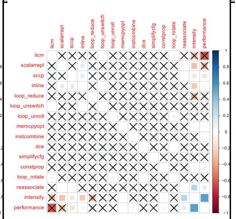
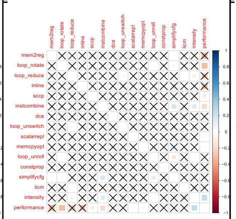
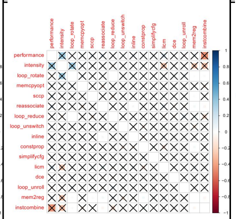
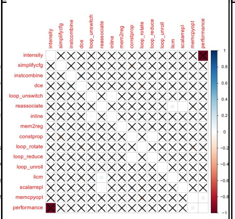
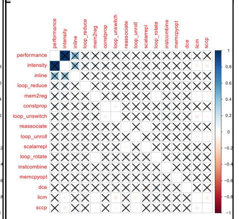
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
AES Distributions Performance							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

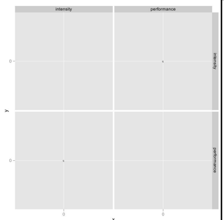
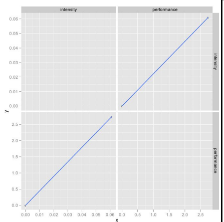
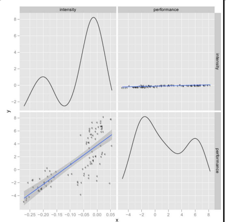
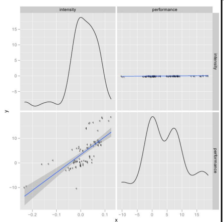
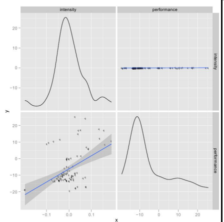
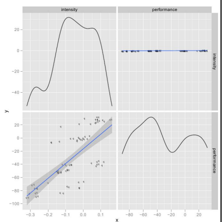
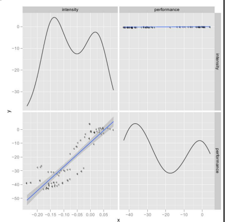
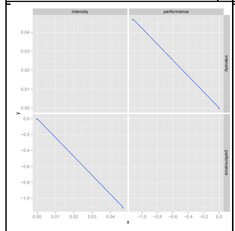
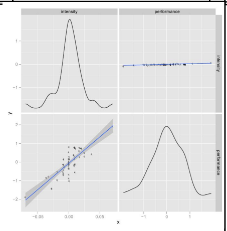
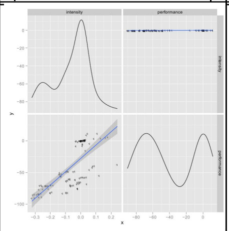
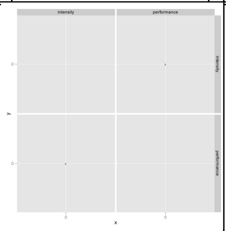
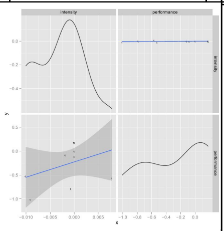
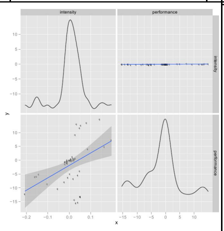
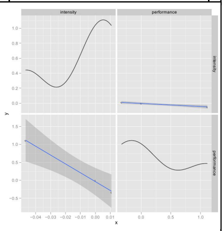
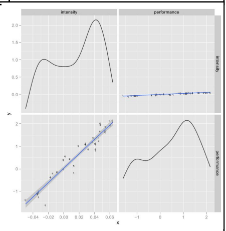
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
AES Distributions Intensity							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

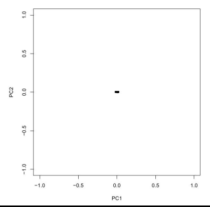
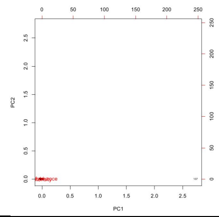
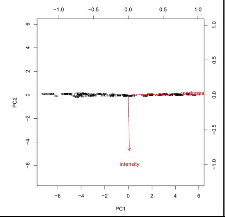
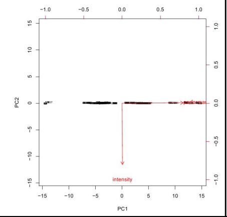
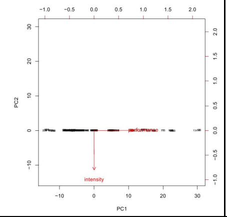
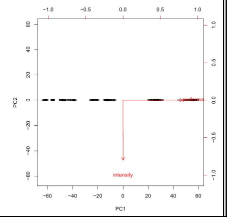
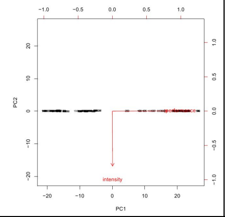
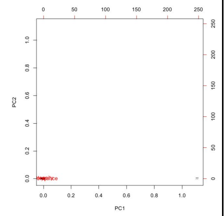
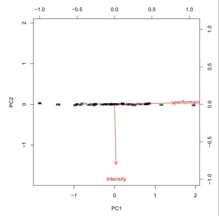
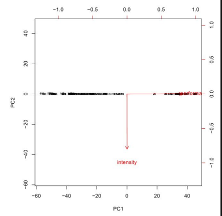
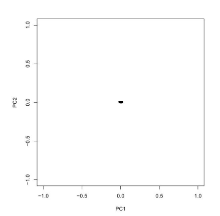
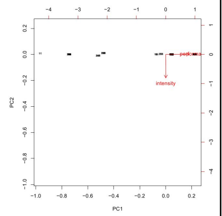
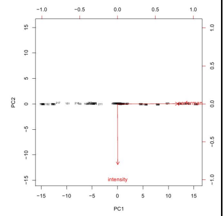
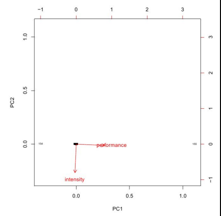
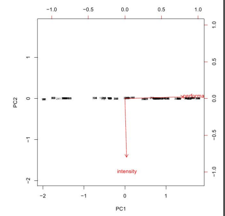
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
AES Box-plot Performance							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

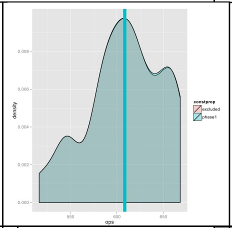
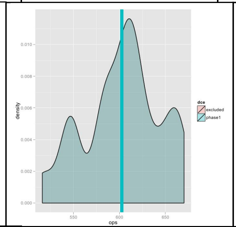
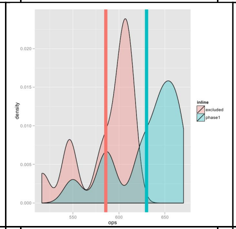
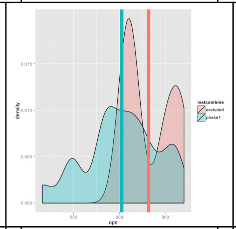
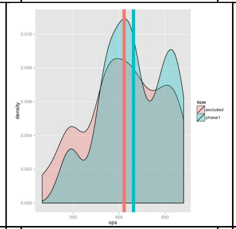
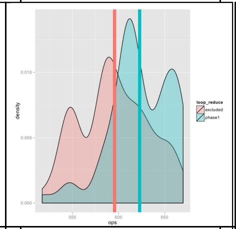
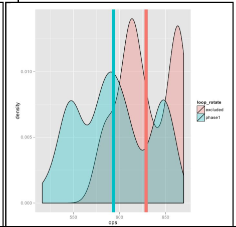
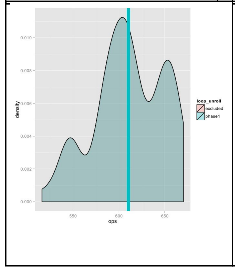
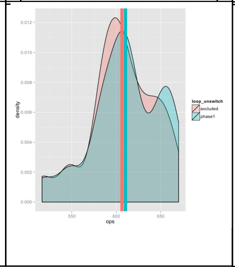
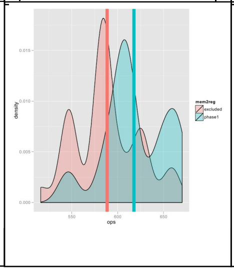
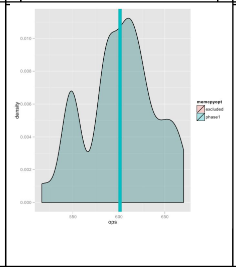
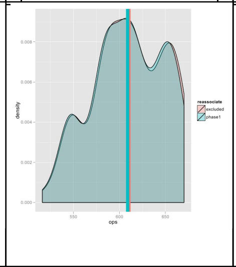
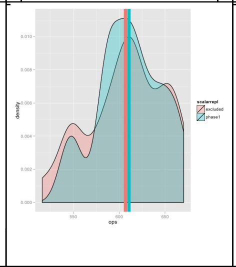
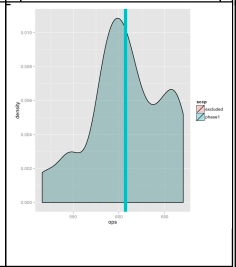
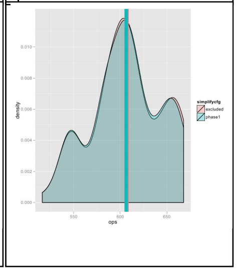
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
AES Box-plot Intensity							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

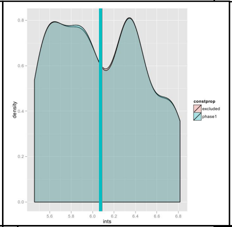
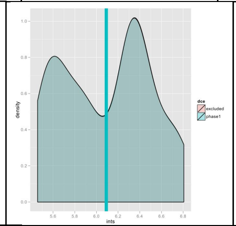
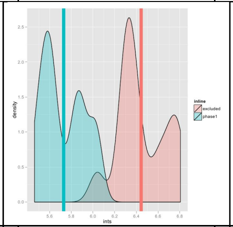
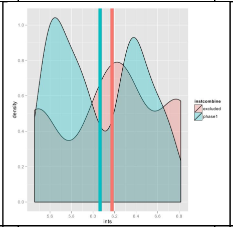
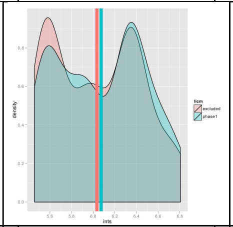
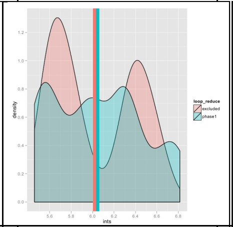
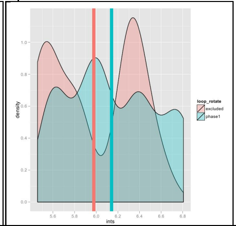
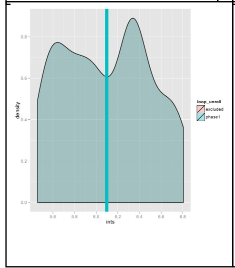
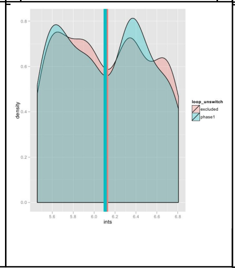
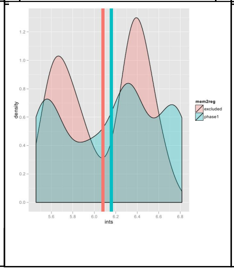
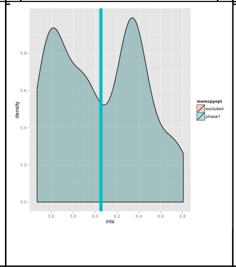
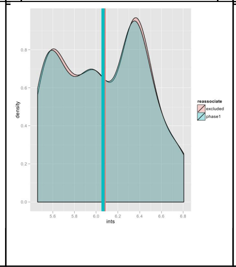
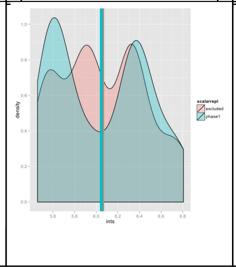
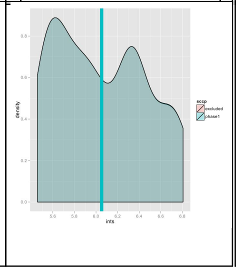
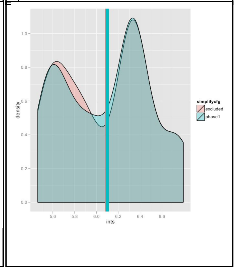
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
AES Correlation on raw data							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

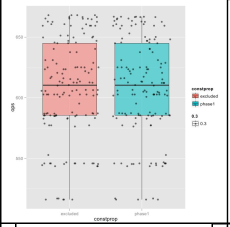
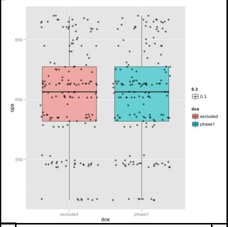
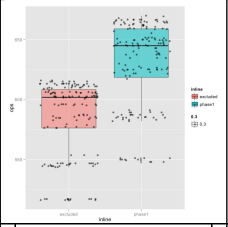
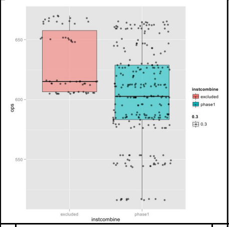
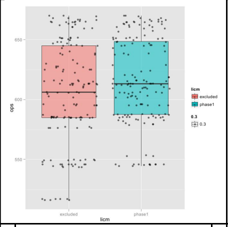
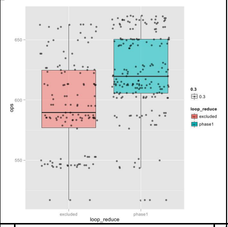
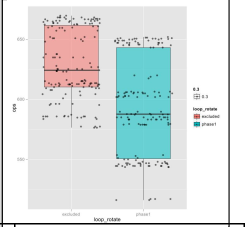
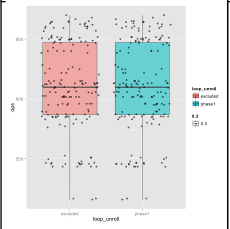
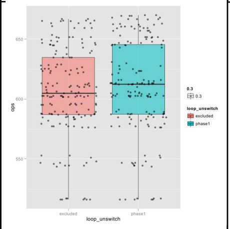
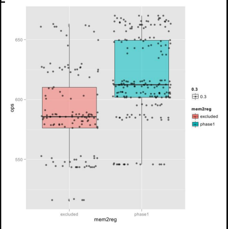
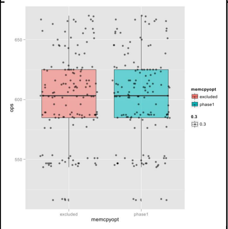
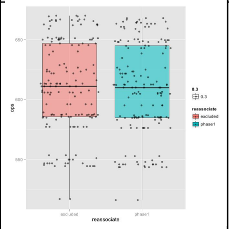
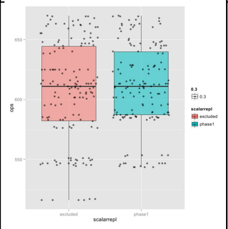
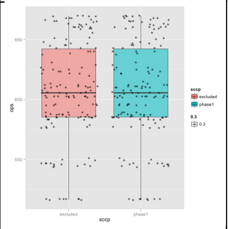
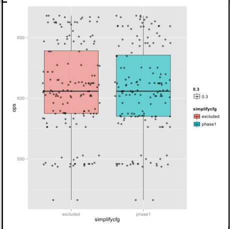
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
AES Correlation on deltas							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

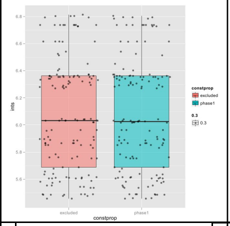
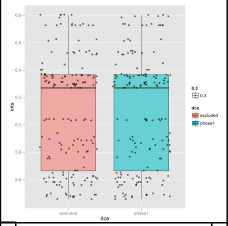
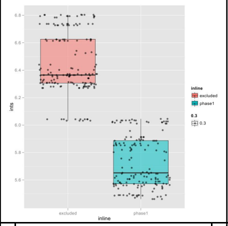
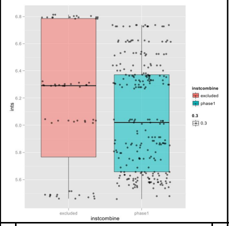
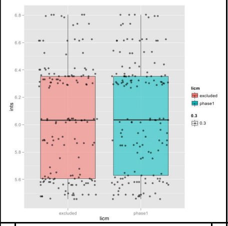
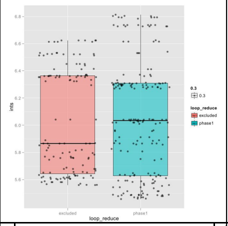
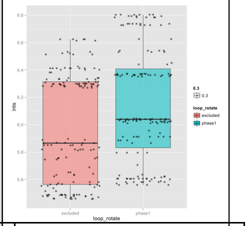
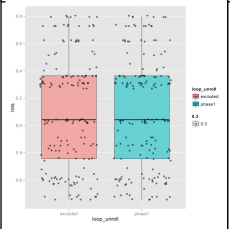
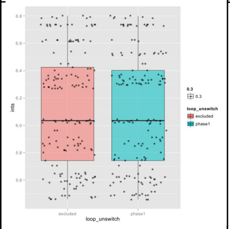
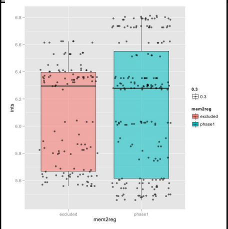
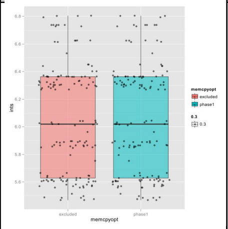
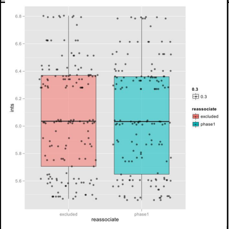
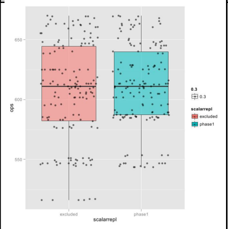
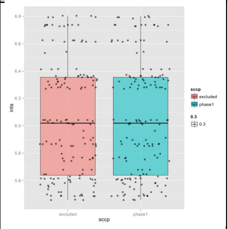
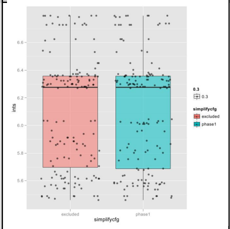
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
AES Scatter-plot							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

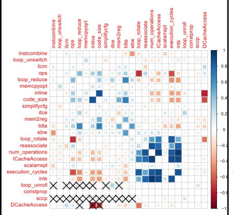
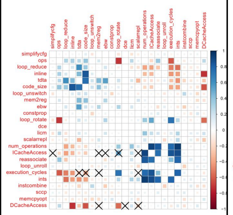
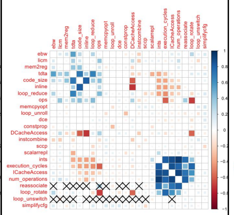
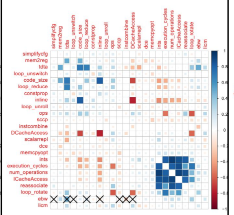
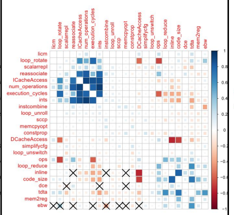
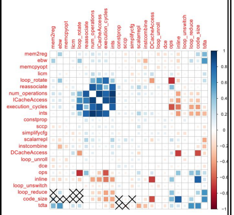
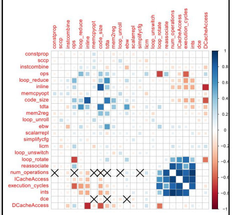
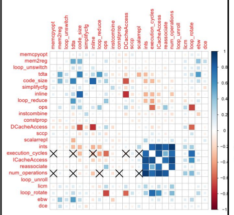
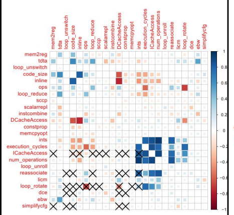
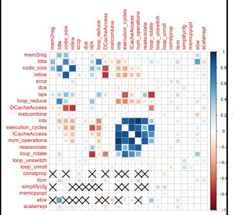
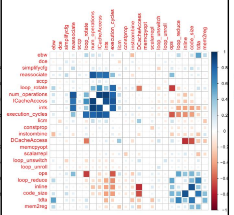
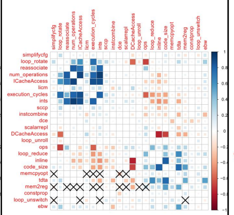
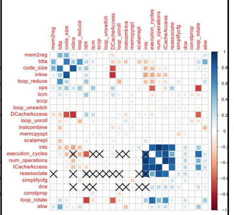
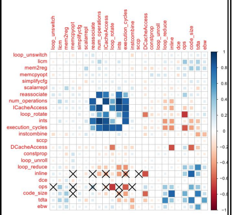
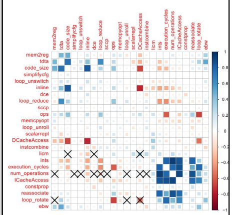
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
AES Principal Component Analysis (PCA)							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

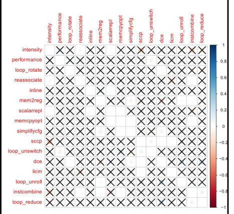
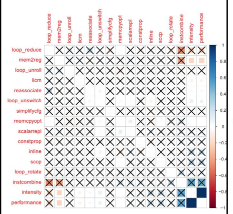
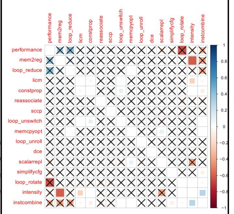
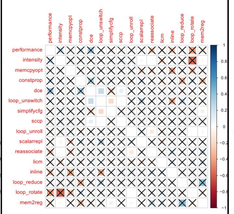
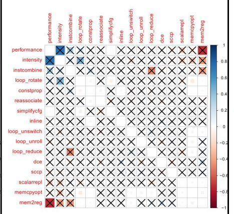
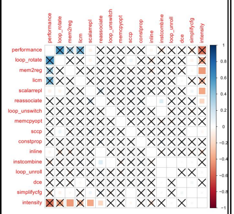
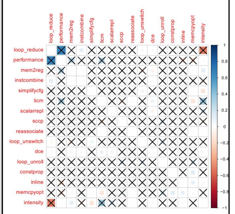
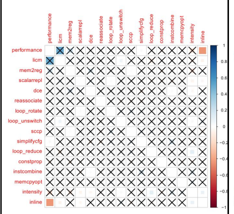
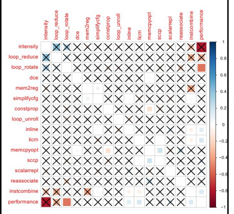
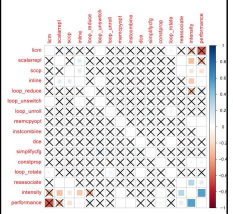
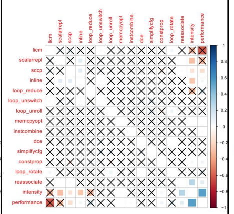
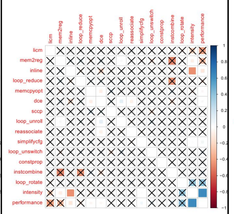
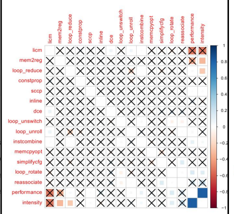
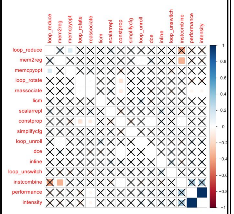
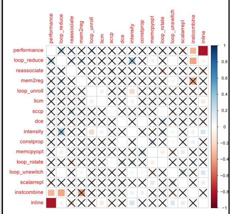
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
ADPCM Distributions Performance							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

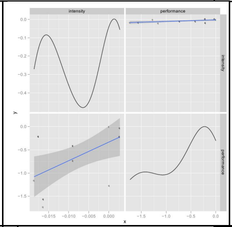
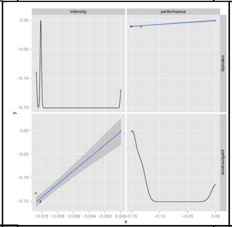
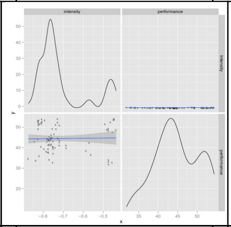
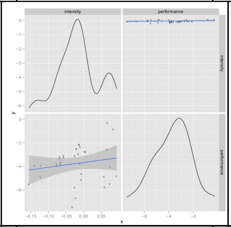
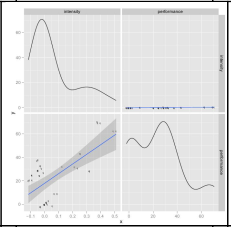
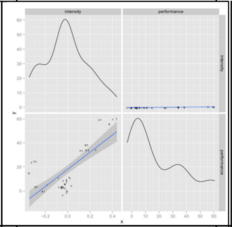
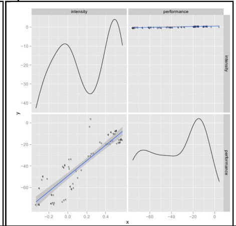
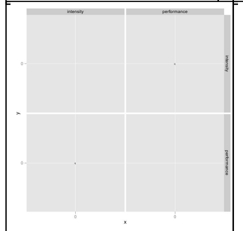
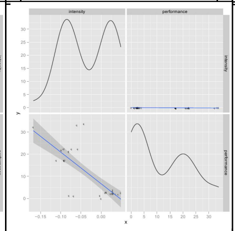
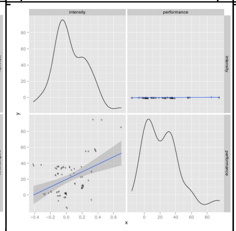
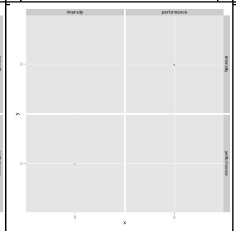
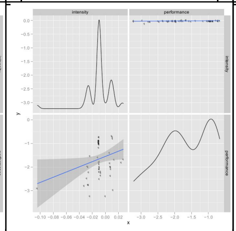
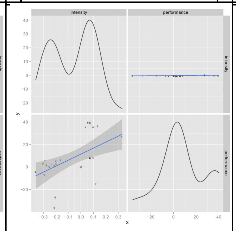
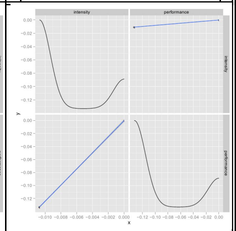
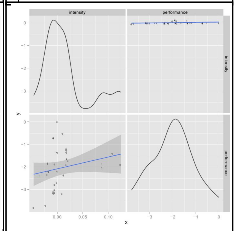
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
ADPCM Distributions Intensity							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

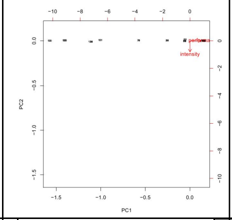
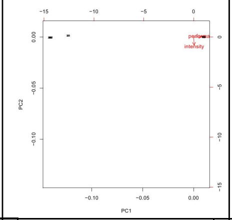
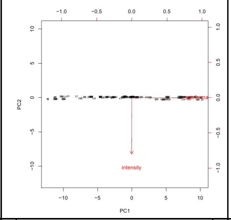
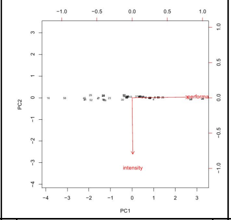
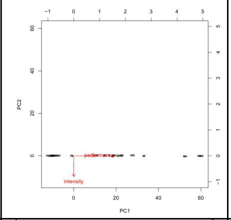
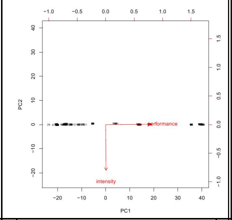
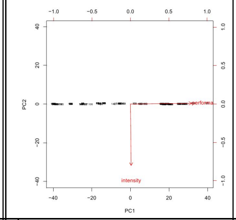
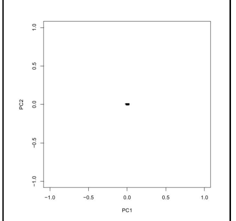
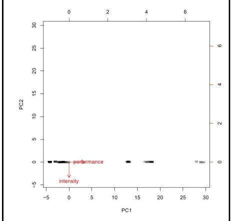
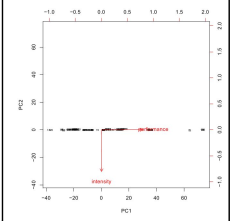
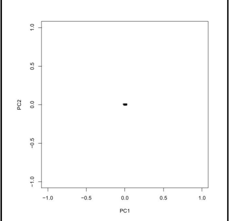
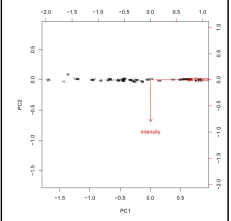
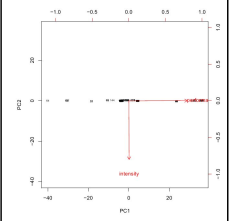
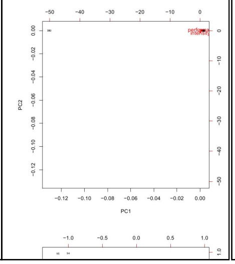
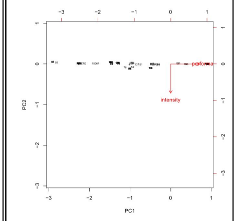
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
ADPCM Box-plot Performance							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

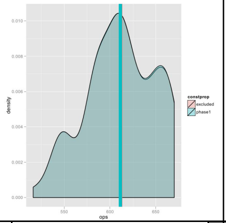
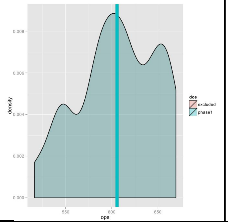
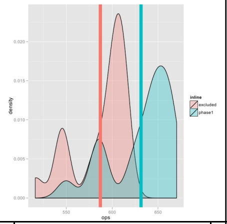
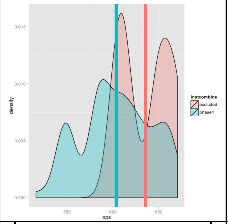
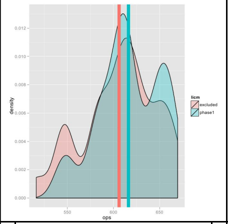
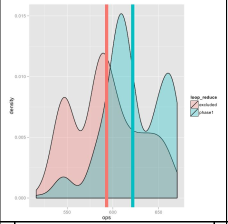
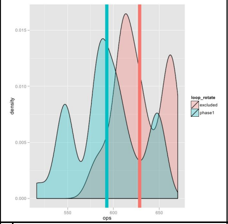
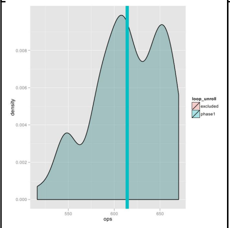
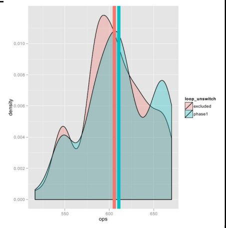
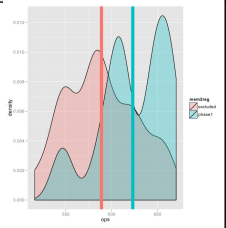
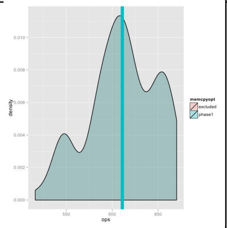
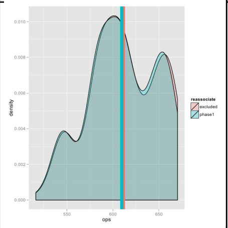
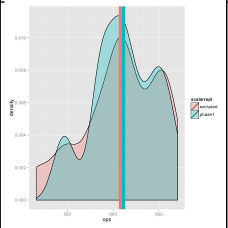
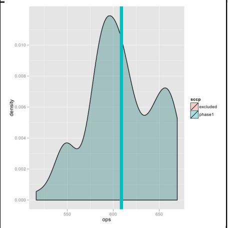
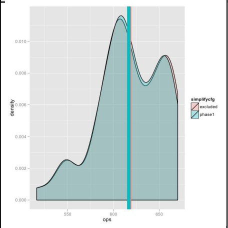
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
ADPCM Box-plot Intensity							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

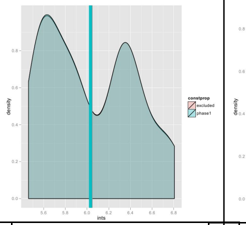
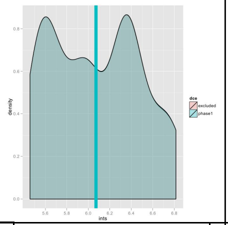
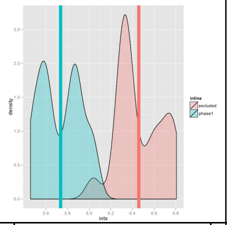
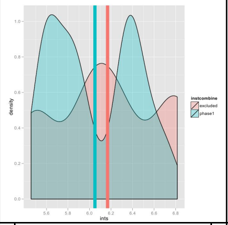
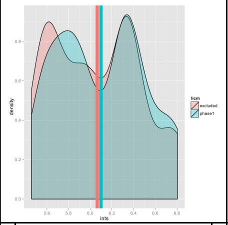
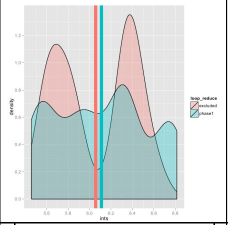
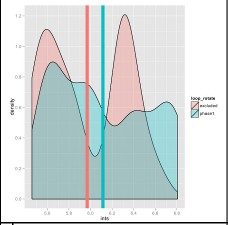
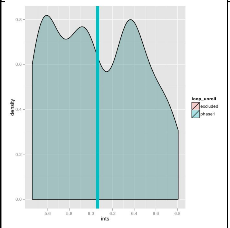
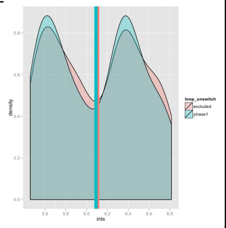
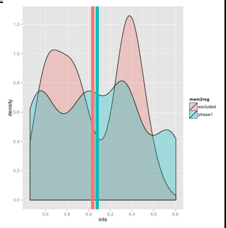
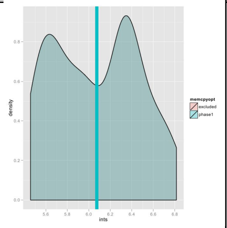
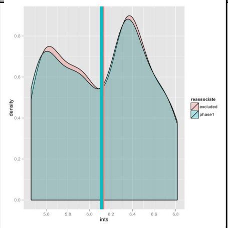
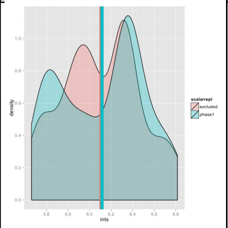
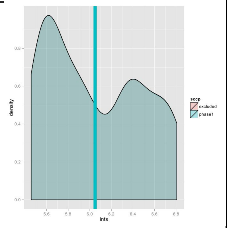
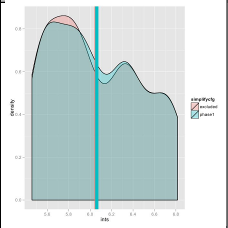
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
ADPCM **Correlation on raw data**							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

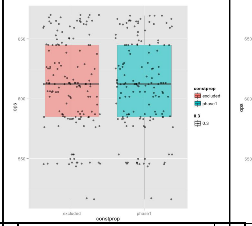
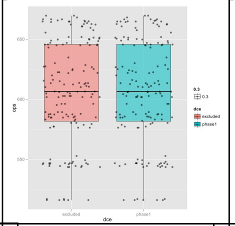
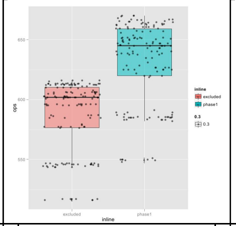
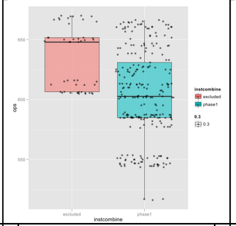
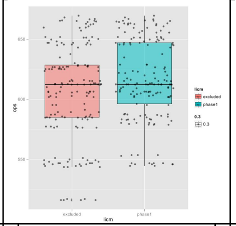
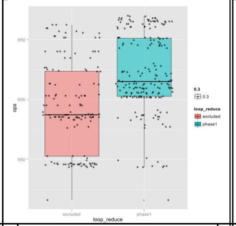
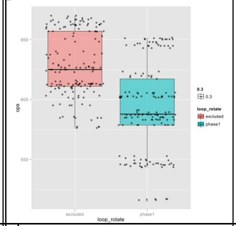
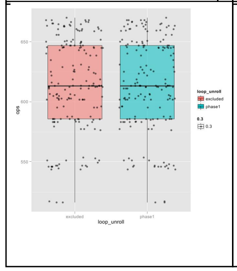
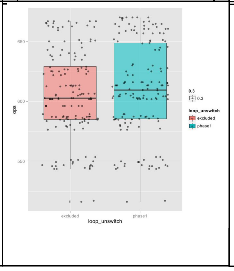
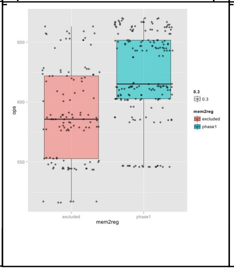
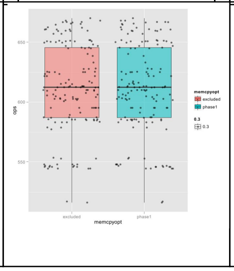
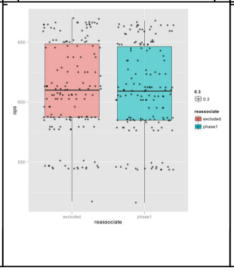
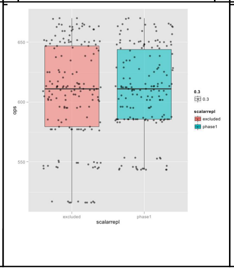
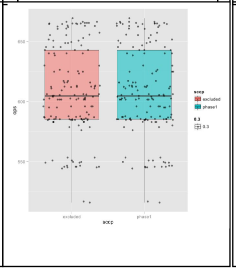
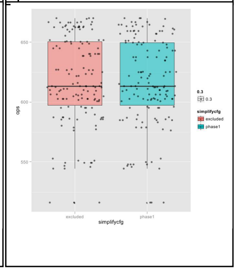
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
ADPCM **Correlation on deltas**							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

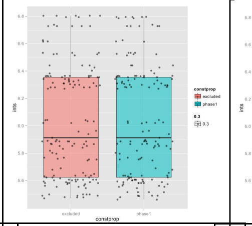
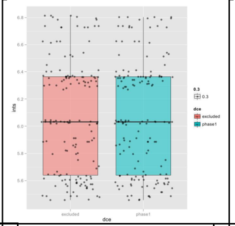
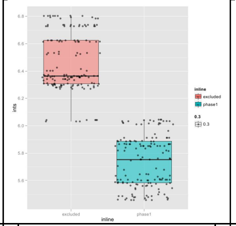
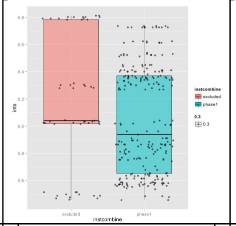
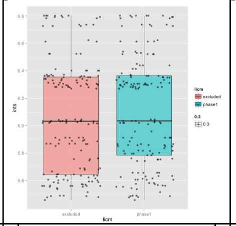
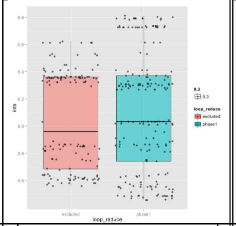
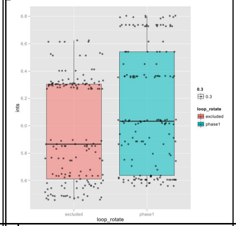
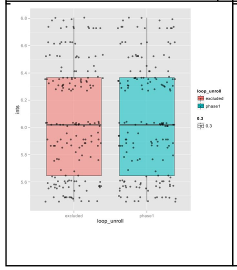
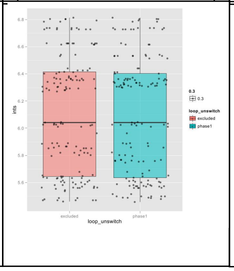
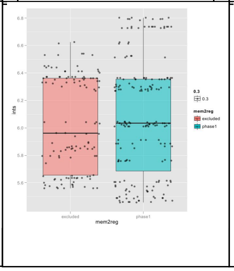
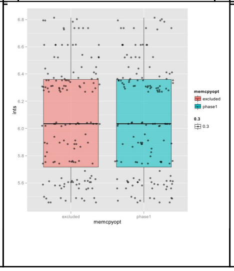
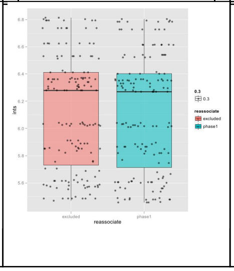
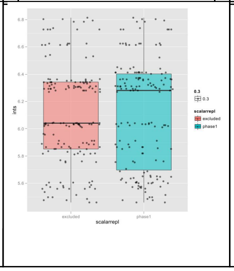
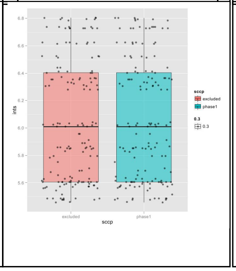
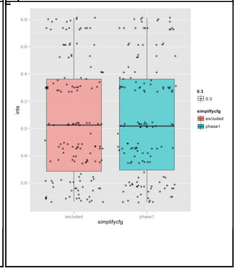
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
ADPCM **Scatter-plot**							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

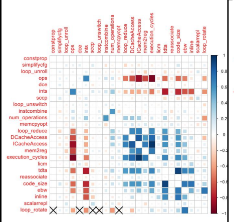
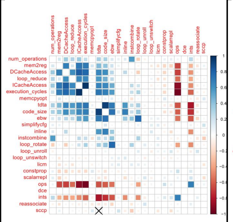
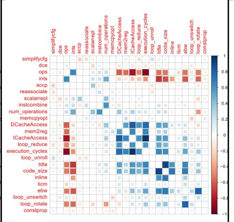
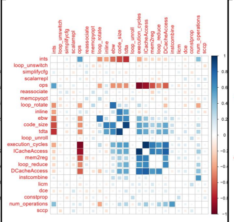
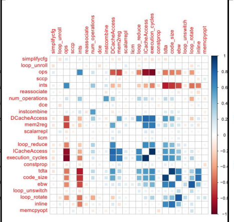
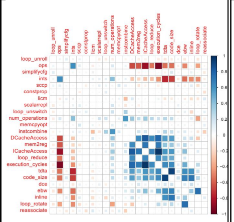
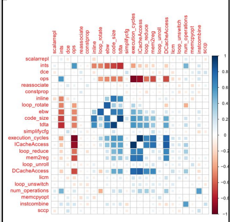
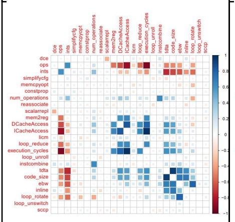
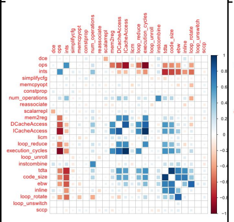
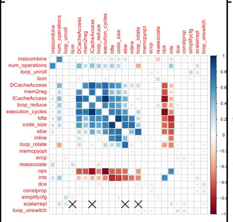
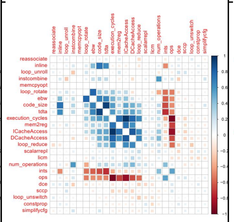
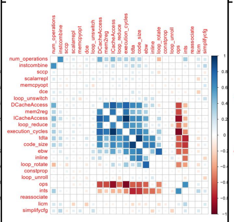
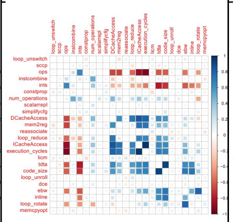
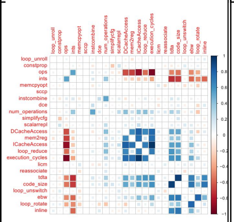
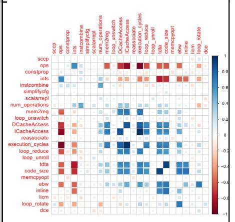
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
ADPCM **Principal Component Analysis (PCA)**							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

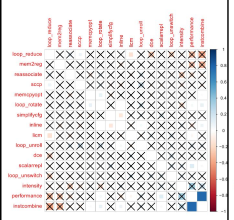
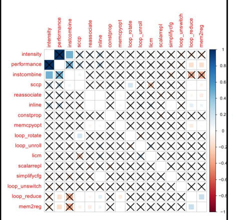
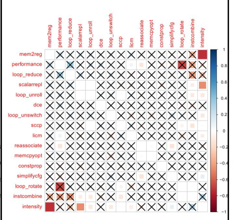
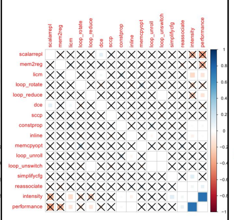
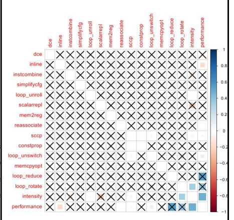
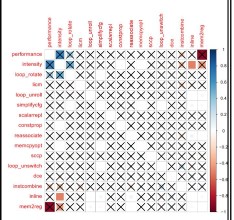
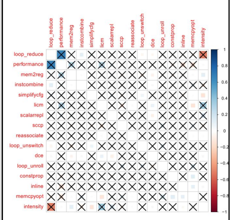
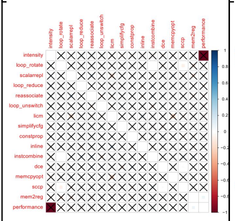
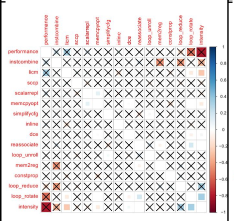
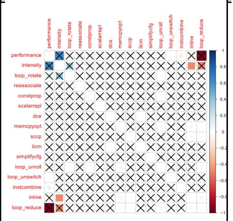
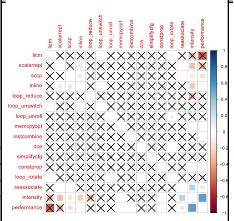
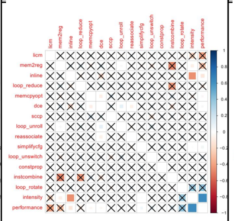
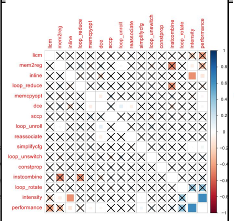
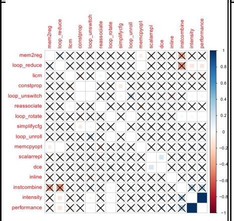
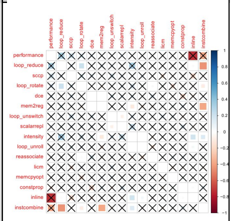
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
JPEG Distributions Performance							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

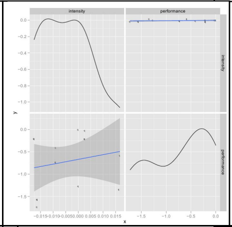
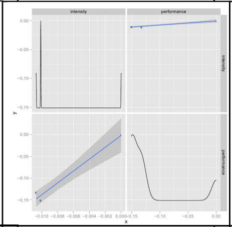
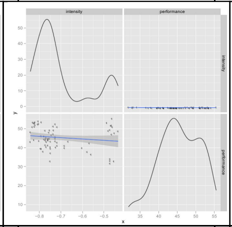
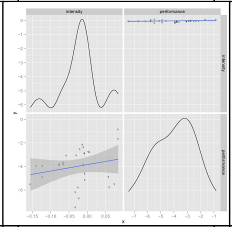
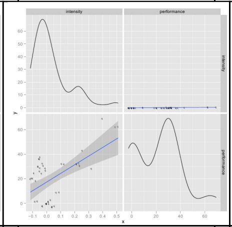
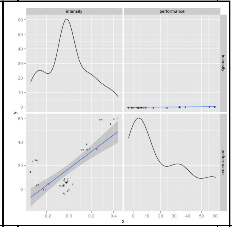
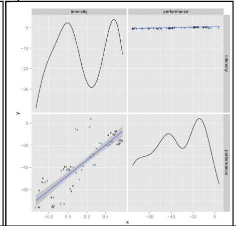
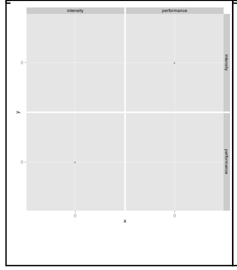
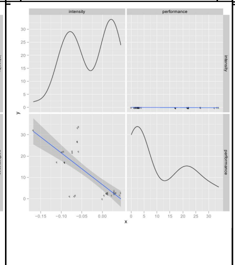
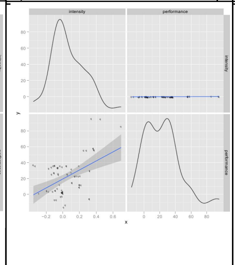
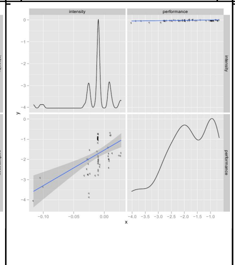
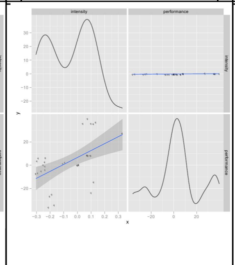
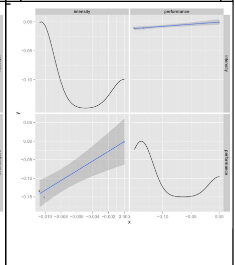
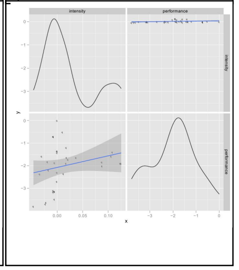
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
JPEG Distributions Intensity							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

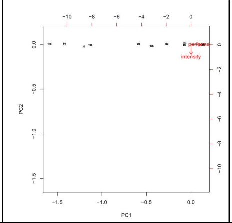
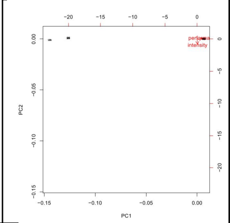
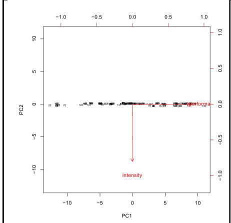
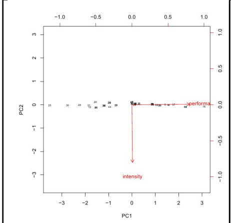
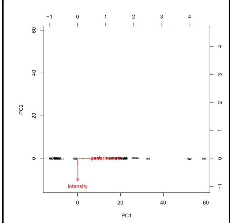
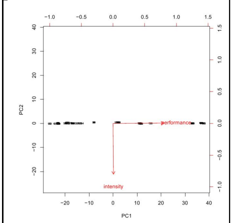
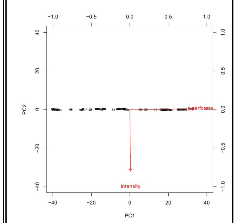
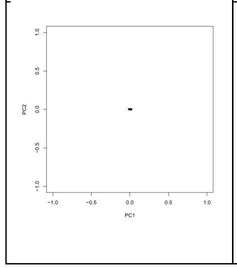
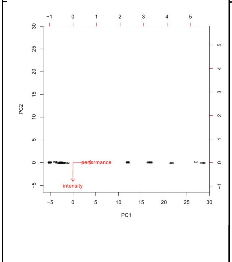
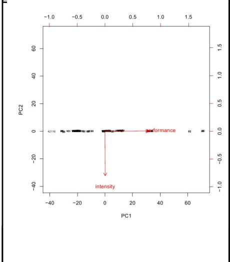
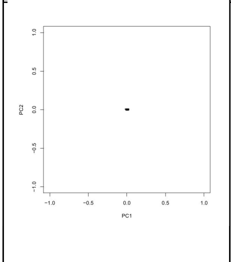
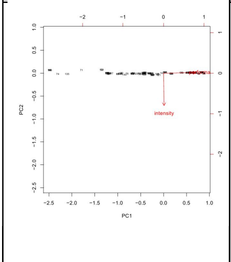
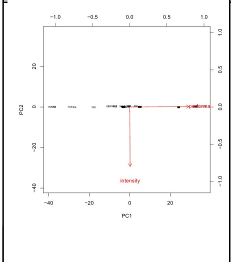
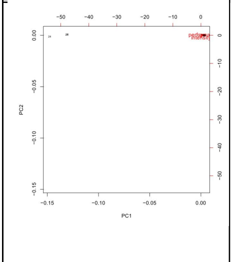
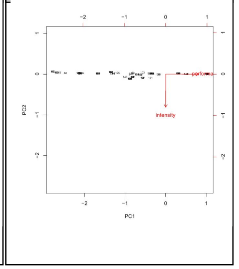
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
JPEG Box-plot Performance							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

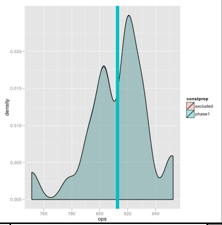
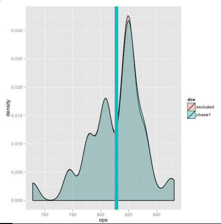
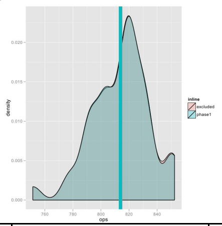
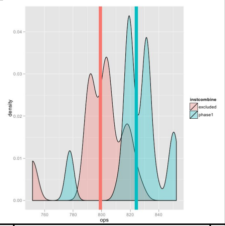
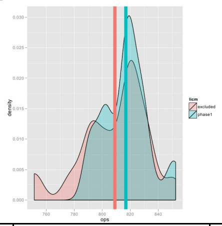
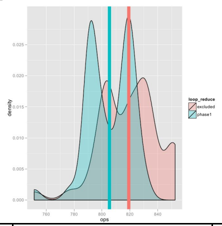
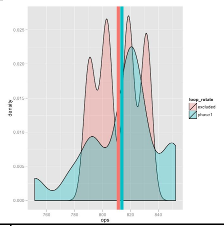
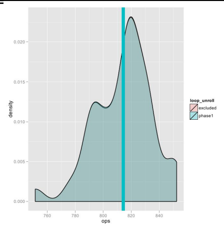
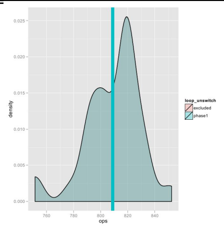
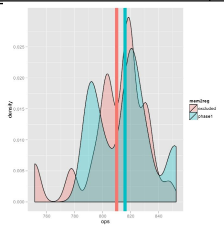
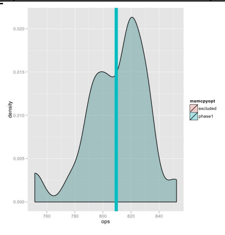
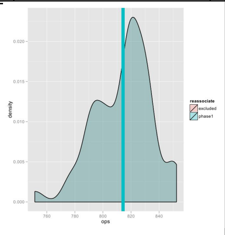
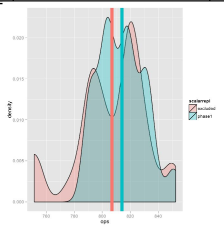
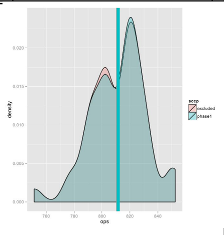
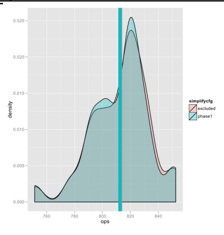
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
JPEG Box-plot Intensity							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

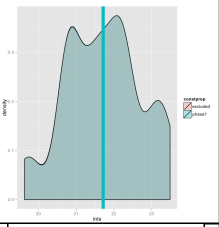
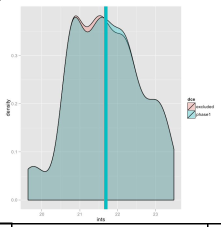
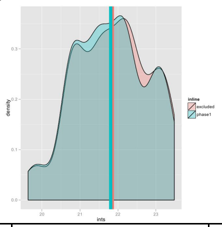
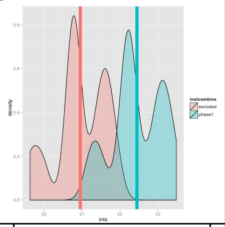
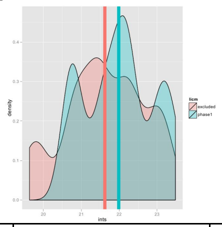
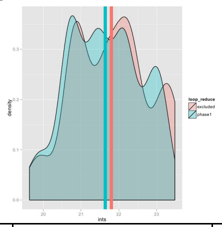
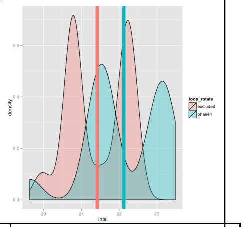
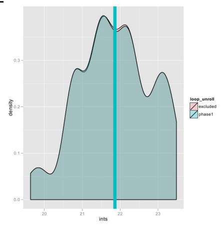
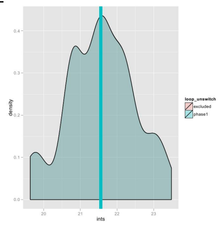
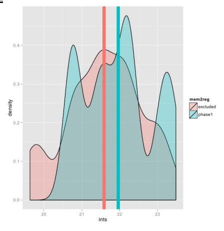
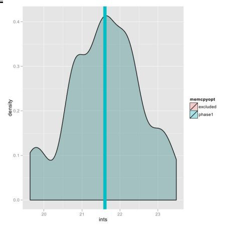
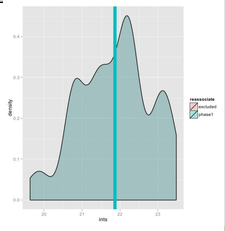
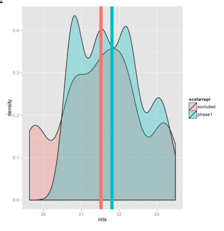
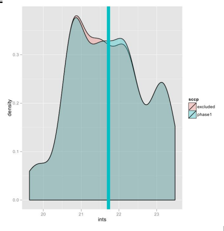
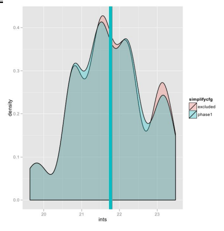
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
JPEG Correlation on raw data							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

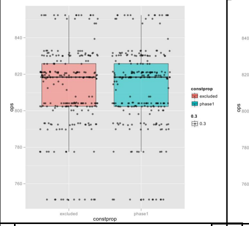
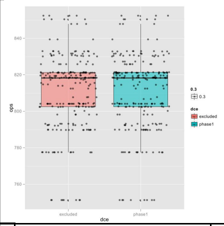
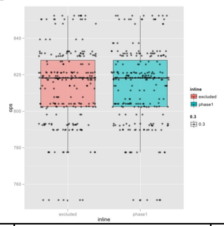
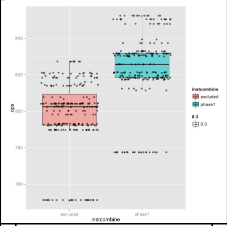
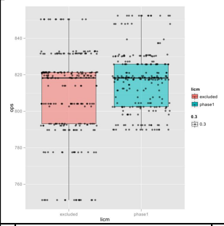
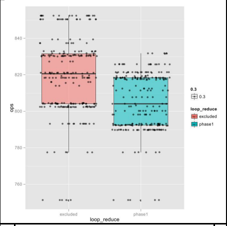
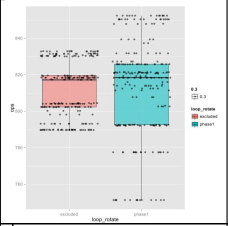
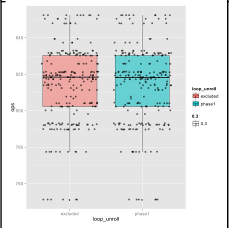
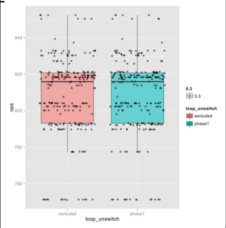
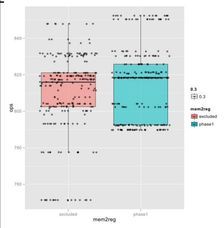
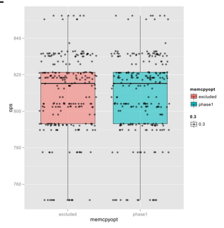
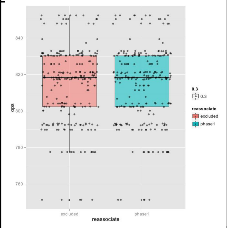
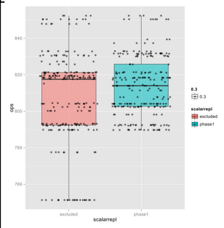
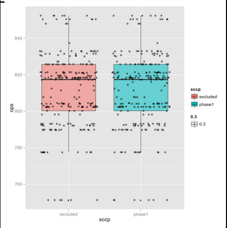
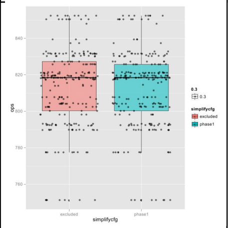
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
JPEG Correlation on deltas							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

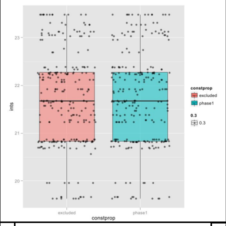
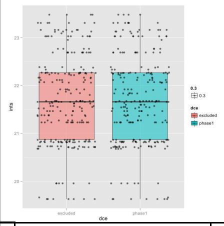
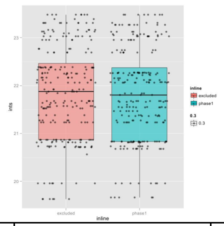
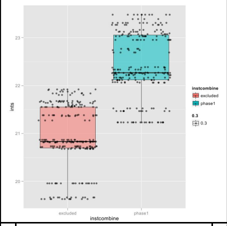
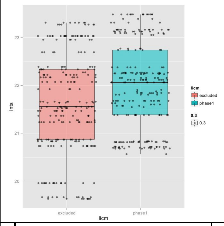
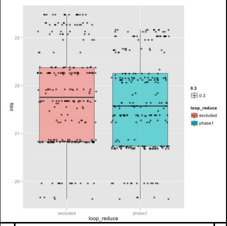
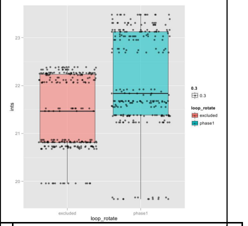
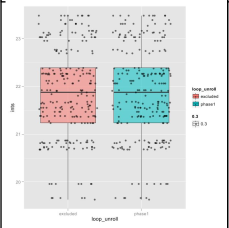
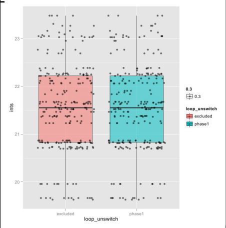
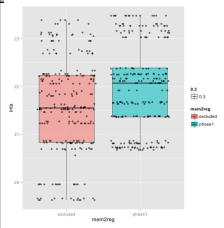
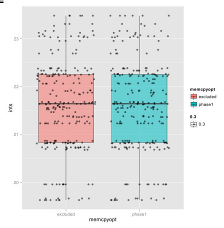
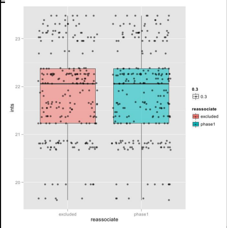
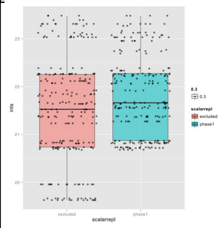
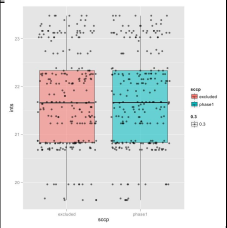
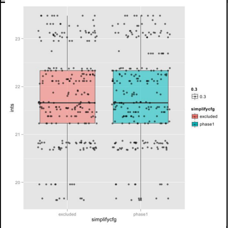
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
JPEG **Scatter-plot**							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

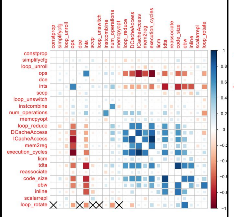
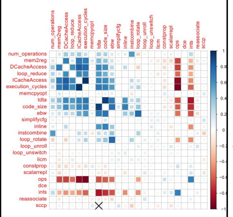
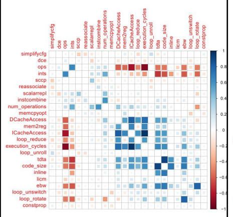
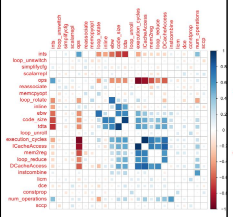
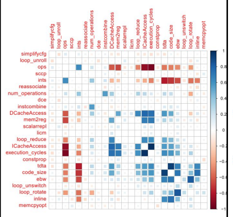
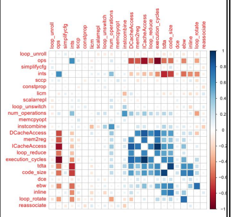
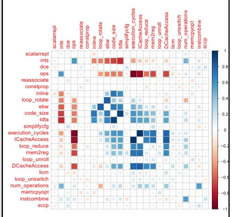
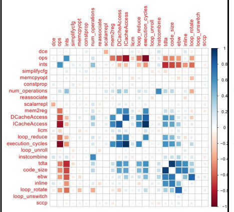
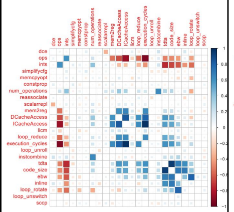
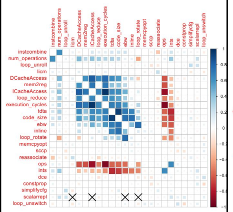
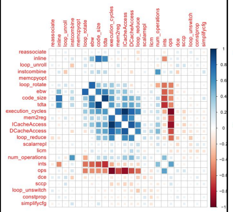
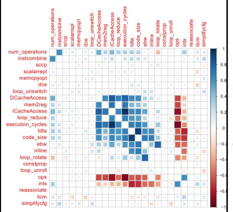
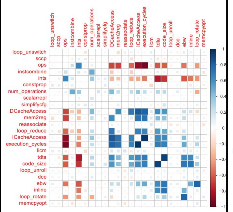
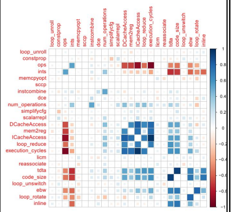
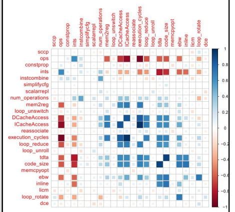
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
JPEG **Principal Component Analysis (PCA)**							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

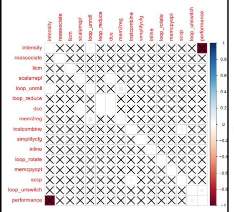
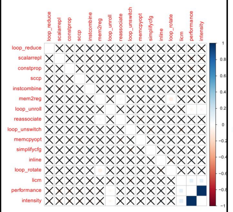
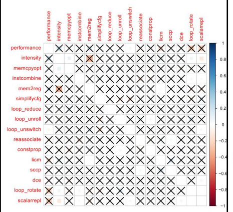
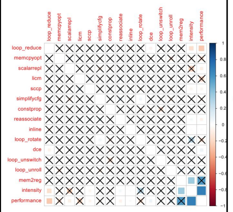
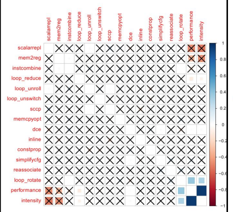
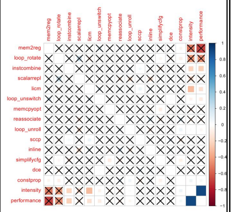
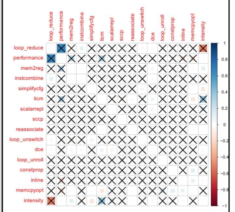
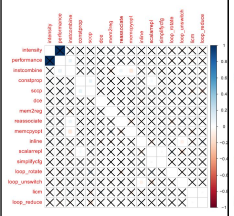
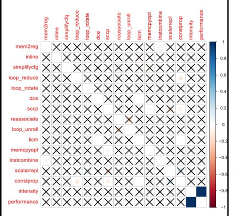
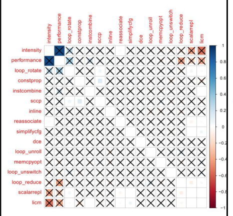
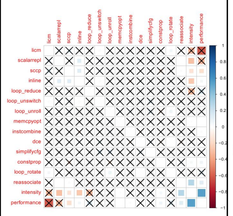
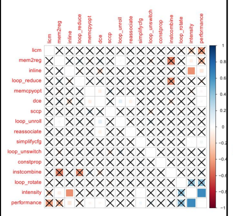
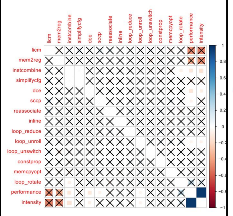
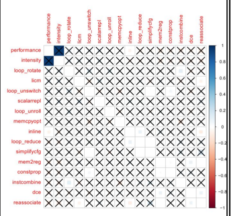
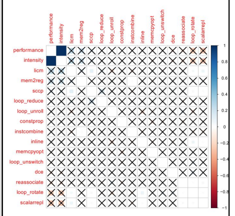
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
BLOWFISH Distributions Performance							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

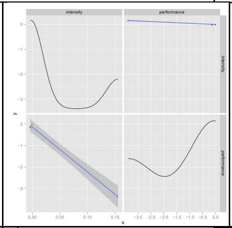
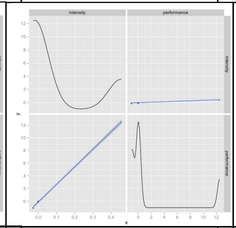
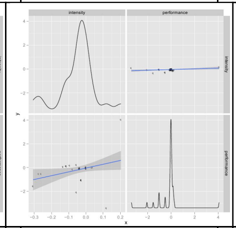
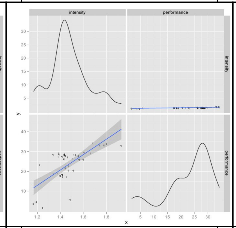
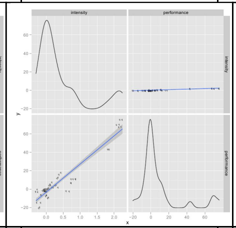
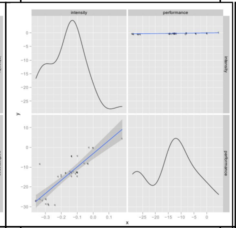
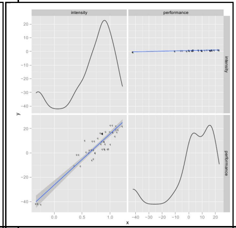
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
BLOWFISH Distributions Intensity							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

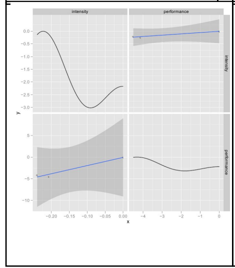
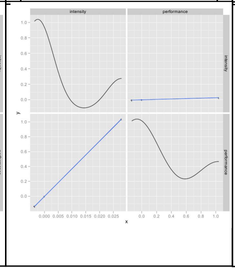
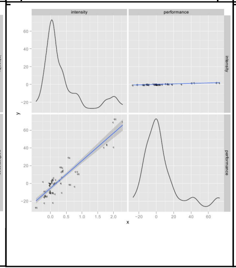
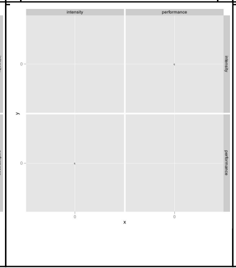
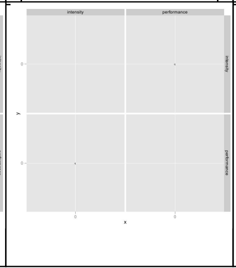
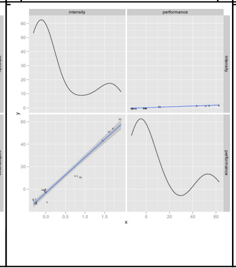
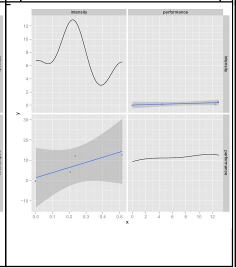
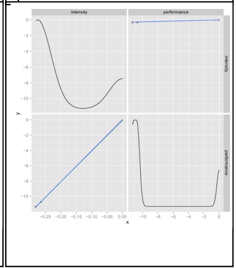
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
BLOWFISH Box-plot Performance							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

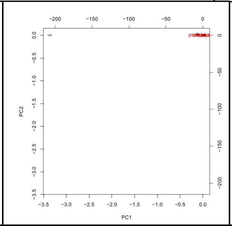
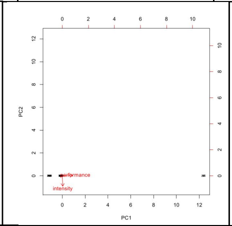
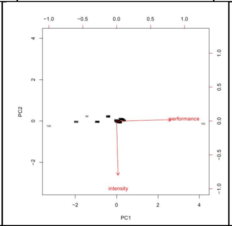
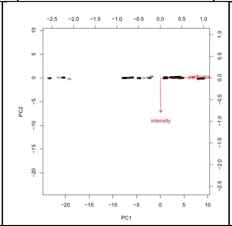
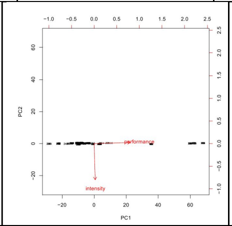
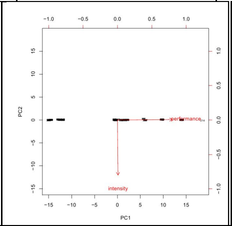
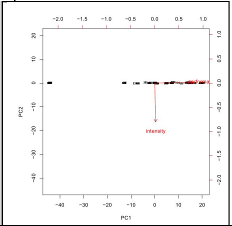
	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
BLOWFISH Box-plot Intensity							
Loop_unroll	**Loop_unswitch**	**mem2reg**	**memcpyopt**	**reassociate**	**scalarrepl**	**sccp**	**simplifycfg**
							

	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
BLOWFISH Correlation on raw data							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
BLOWFISH Correlation on deltas							
Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
BLOWFISH **Scatter-plot**							

Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg
							

	constprop	dce	inline	instcombine	licm	Loop_reduce	Loop_rotate
BLOWFISH **Principal Component Analysis (PCA)**							

Loop_unroll	Loop_unswitch	mem2reg	memcpyopt	reassociate	scalarrepl	sccp	simplifycfg