Chapter 2
Design Space Exploration of Compiler Passes: A Co-Exploration Approach for the Embedded Domain

Abstract Very Long Instruction Word (VLIW) processors represent an attractive solution for embedded computing, offering significant computational power with reduced hardware complexity. However, they impose higher compiler complexity since the instructions are executed in parallel based on the static compiler schedule. Therefore, finding a promising set of compiler transformations and defining their effects have a significant impact on the overall system performance. In this chapter, we provide a methodology with an integrated framework to automatically (i) generate optimized application-specific VLIW architectural configurations and (ii) analyze compiler level transformations, enabling application-specific compiler tuning over customized VLIW system architectures. We based the analysis on a Design of Experiments (DoEs) procedure that statistically captures the higher order effects among different sets of activated compiler transformations. Applying the proposed methodology onto real-case embedded application scenarios, we show that (i) only a limited set of compiler transformations exposes high confidence level (over 95%) in affecting the performance and (ii) using them we could be able to achieve gains between 16–23% in comparison to the default optimization levels. In the next chapters, we go deeper in building machine learning models to tackle the problem.

2.1 VLIW

Embedded system design traditionally exploits the knowledge of the target domain, e.g., telecommunication, multimedia, home automation etc., to customize the HW/SW coefficients found onto the deployed computing devices. Although the functionalities of these devices differ, the computational structure and design are tightly connected with the platform they rely on. Platform-based designs have been proposed as a promising alternative for designing complex systems by redefining the problem of tuning specific design parameters of the platform template.

The scientific and commercial urge to use VLIW technology seems to be raised again after three decades of existence [1]; VLIW processor templates are being used particularly in embedded processors, designed to perform special-purpose functions, usually for real-time or hardware acceleration. Being able to use VLIW power-saving

A. H. Ashouri et al., *Automatic Tuning of Compilers Using Machine Learning*,
PoliMI SpringerBriefs, https://doi.org/10.1007/978-3-319-71489-9_2

cores in CPUs seems to be using day by day. However, the trade-offs between right parallel execution and the speedup managed by compiler instead of hardware are becoming a very complex task. VLIW can potentially achieve greater performance, offering high degree of Instruction Level Parallelism (ILP) with low silicon and power costs. On the one hand, architecture configurability of VLIW platforms offers significant advantages regarding portability, sizing and parameter tuning provided to the designer [1, 2]. On the other hand, it introduces a lot of complexity during optimization due to multi-objective nature of the solution space and the multi-parametric structure of the design space.

Although a significant amount of research has been conducted on exploring and optimizing VLIW architectural parameters [3] and introducing specific compiler optimization for VLIW processors [4, 5], there are limited references regarding the analysis of the impacts of conventional compiler transformations onto VLIW architectures and moreover how these transformations are correlating with the underlying architectural configuration. Nowadays, the existence of modular and reusable compiler tool-chains LLVM and ROSE [6] raises the opportunity for system designers to exploit sophisticated compiler passes and customize their compiler infrastructure accordingly. Given the large optimization space provided by the modern compiler infrastructures, the designer has to traverse to find the best trade-off points, thus a fine-grained and automatic characterization of the effects that each compiler transformation has onto the application's behavior, is considered of great importance. Empirical evaluation of the effects, by simply activating and deactivating compiler passes cannot be considered adequate, since a lot of inter-transformation interactions and second order effects are neglected. Due to the complexity of characterizing the solution space, there is a necessity to extend conventional exploration approaches by applying sophisticated analysis and data-mining for extracting knowledge from statistical results [7]. The problem becomes more demanding in the embedded computing domain, which requires different optimizations related to each platform configuration customized for a specific application domain. The main contribution of this chapter consists of proposing a compiler/architecture methodology that provides to the designer an integrated environment to automatically (i) generate optimized application specific architectural configurations of VLIW-based platforms, and (ii) a statistical analysis of the effects of compiler level transformations.

We target the design problem of compiler/architecture co-exploration in embedded computing. Thus, we focus on enabling application-specific compiler tuning over customized VLIW system architectures. First, a multi-objective exploration loop targeting application-specific micro-architectural customization is applied for extracting the best VLIW architecture candidates. We utilize the newly introduced *Roofline* processor architecture model [8] for characterizing the differing architectural solutions onto various resource constraints. The optimized VLIW architectural configurations are then propagated to the compiler analysis phase in which the statistical effects of the applied compiler transformations are characterized in a fine-grained manner. The developed exploration framework integrates the LLVM compiler infrastructure [9] as a source to source code transformation tool together with the VEX compiler-simulator for mapping the transformed code onto custom VLIW

architecture instances. We evaluated the overall methodology (customized architecture selection and statistical compiler level analysis) using a GSM codec application as the driving use case. We show that only a limited set of compiler transformations has a significant effect on optimizing performance across a set of GSM specific VLIW processors. In addition to the application specific scenario, we present results regarding the multiple embedded applications onto a single VLIW instance, showing that the proposed analysis can be used to extract promising compiler transformations in cross-application manner.

The rest of the chapter is organized as follows. Section 2.2 provides a brief discussion on related work and the current state of the art in the field. In Sect. 2.3, we introduce the basic methodology for architecture customization and statistical compiler level analysis. Section 2.3.2.1 presents the experimental evaluation of the proposed methodology on differing customized VLIW architectures and benchmark applications. Section 2.4 summarizes the work and concludes this chapter.

2.2 Background

Although we have entered the era of multi-core systems, the high degree of instruction parallelism offered by VLIW architectures seems to make them an interesting alternative for a large set of commercial embedded systems [1, 10, 11]. VLIW architectures are also emerging in the modern many-core embedded accelerator devices, i.e., KALRAY MPPA256 for image and signal processing applications.

Several research works have targetted the generation of Pareto optimal VLIW architectural configurations [3, 11] by exploring the space using pre-allocated compiler sequences over differing architecture instances. Towards the same direction of VLIW architectural configuration, Wong et al. [12] introduced r-VEX, a reconfigurable and extensible VLIW processor. The source code is mapped using the VEX (VLIW Example) environment [13], which forms a compilation-simulation system that targets a broad class of VLIW processor architectures, and enables compiling, simulating, analyzing and evaluating C programs [2].

In current literature, there is a lot of attention on iterative compilation and predictive compiler modeling to predict the potential speedup of compiler transformed programs utilizing code features provided by static program analysis as mentioned in the Chap. 1. However, there is a lack of comprehensive analysis regarding the impact of applying differing conventional compiler transformations on customized VLIW architectures. Although, in VLIW compilation infrastructures [13] there are available batch compiler optimization modes, fine-grained analysis of compiler effects for VLIW architectures and its relation with architecture customization is not adequately targeted.

2.3 Methodology for Compiler Analysis of Customized VLIW Architectures

In this section, we describe the proposed methodology for compiler analysis of customized VLIW architectures. The proposed methodology comprises of two phases: (i) Customized VLIW architecture selection and (ii) Statistical analysis of compiler transformations. From a high-level point of view, we first generate a set of promising VLIW architectural candidates that tailor to the characteristics of the target application, optimizing on the performance-intensity trade-off curve with respect to the overall hardware allocated resource. Then, statistical analysis of distributions generated over the compiler transformation space is performed on the set of these selected customized VLIW solutions. This enables the designer to characterize the effects of each compiler transformation in both an architecture specific manner and a cross-architecture manner.

We used the Roofline performance model [8] as the basis for both generating the custom architecture configurations and characterizing the effect of the compiler passes. Roofline relates processor performance to off-chip memory traffic. It characterizes processor architectures in a two-dimensional space, i.e., performance (Mops/sec) versus operational intensity (ops/Byte). *Operational intensity* is defined as operations per byte of DRAM traffic, defining total byte accessed as those bytes that go to the main memory after been filtered by the cache hierarchy. The advantage of using Roofline model is twofold: (i) it provides the designer with an intuitive insight visual metric for fast evaluation of the architectural optimality of the configuration and (ii) it is useful to characterize the impact of applied compiler transformations onto a specific architecture. For example, Fig. 2.1 depicts the Roofline model of a specific VLIW configuration and the superposition of application configurations

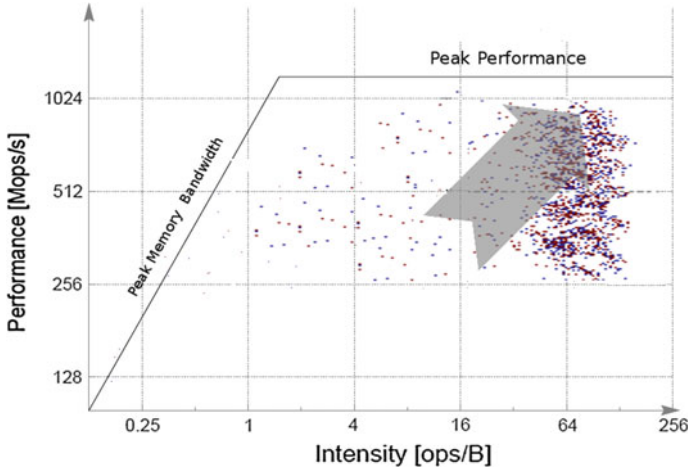


Fig. 2.1 Roofline example

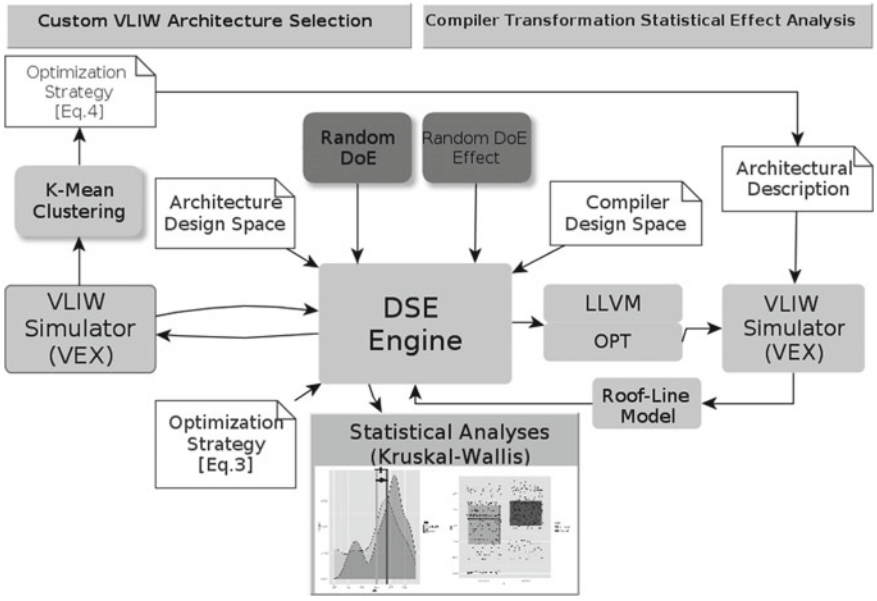


Fig. 2.2 Tool-chain implementing the proposed methodology

derived from an experimental campaign of 4 K different compiler parameter combinations. A general trend (highlighted by the arrow in Fig. 2.1) can be easily detected towards higher performance and operational intensity points. Given this visual representation, a designer can identify promising compiler passes to be applied. A custom exploration and analysis framework (Fig. 2.2) has been developed based on the integration of open source tools to implement the proposed methodology. Specifically, we used Multicube Explorer [14, 15] as the central DSE engine. Given the architectural and compiler design space descriptions, it manages to automatically generate configuration vectors according to the specified DoE – random DoE during the phase of custom VLIW architecture selection and random effect DoE during the compiler transformation analysis phase. The LLVM compiler infrastructure[1] is integrated within the framework – specifically the LLVM C front-end and the opt tool – as a source to source transformation tool of the original application code after applying the compiler transformations instructed by the DSE engine. The transformed code of the application is mapped onto the VLIW processor using the VEX [13] VLIW compiler-simulator tool, which is used for both generating different VLIW architectural configurations and mapping code onto these custom VLIW processors. Custom scripts have been developed to evaluate each examined configuration according to the Roofline model. Statistical analysis and visualization of results are performed using the statistical language R [16].

[1] LLVM projected supported its C source-to-source compiler frontend till v2.8.

2.3.1 Custom VLIW Architecture Selection

Application-specific customization of architecture's parameters is one of the early system design optimization phases for defining platform configurations that meet the desired performance specifications. Given the large number of parameters that usually defines a processor architecture and the delay required for simulating each possible configuration, the task of optimal micro-architectural parameter selection forms an extremely challenging exploration problem that for reasonably representative design space definitions becomes intractable, regarding the time required for exhaustive evaluation. Several research works utilizing well-known meta-heuristics [3, 17] have been already proposed for generating the Pareto optimal sets of the aforementioned optimization problem.

In this chapter, however, we slightly shift the focus of exploration from delivering the optimal set of architectural configurations to discover custom architecture configurations that do not correspond to the boundaries of Pareto regions, i.e., very low-cost architectures with very poor performance or very expensive architectures that deliver very high gains regarding performance. Thus, here we invoke a relaxed optimization search strategy that is based on a random sampling of the targeted design space rather than on an optimization oriented strategy, e.g. simulated annealing or NSGA-II genetic optimization [17] etc.

Table 2.1 shows the micro-architectural design space, Ω, considered for the custom VLIW architecture selection phase. In the first step, we randomly sample the Ω design space. Each explored solution is stored in the database of explored solutions, X after being characterized according to the performance and operational intensity metrics defined within the Roofline model, where:

$$Performance(\mathbf{x}) = \frac{\#Operations(\mathbf{x})}{\#NumCycles(\mathbf{x}) \times ClkFreq(\mathbf{x})} \tag{2.1}$$

$$Intensity(\mathbf{x}) = \frac{\#Operations(\mathbf{x})}{\#CacheMisses(\mathbf{x}) \times CacheLineSize(\mathbf{x})} \tag{2.2}$$

After the formation of the X, we are interested in finding those explored architectures that maximize the performance and operational intensity of the application while using minimum computational and memory resources. In order to extract the desired architectural configurations, we perform Pareto filtering on the solution space defined with the X, by considering the following multi-objective optimization problem:

$$\min_{\mathbf{x}\in\Omega} \begin{bmatrix} \frac{1}{Performance(\mathbf{x})} \\ \frac{1}{Intensity(\mathbf{x})} \\ \#CompResources(\mathbf{x}) \\ \#MemResources(\mathbf{x}) \end{bmatrix} \tag{2.3}$$

Table 2.1 VLIW microarchitectural design space

Parameters	Values (integer range)
lg2CacheSize	[11–30]
lg2Sets	[0,3]
lg2LineSize	[5,9]
lg2ICacheSize	[11,30]
lg2ICacheSets	[0,3]
lg2ICacheLines	[5,9]
ClkFreq	[300,500]
NumCaches	[1,2]
IssueWidth	[1,16]
NumAlus	[1,16]
NumMuls	[1,4]
RegisterFile	[32,128]
BranchRegister	[32,128]

where computational resources are (i) number of ALUs and (ii) number of multipliers, while memory resources are (i) data cache size, (ii) instruction cache size and (iii) register file size. Although, in Eq. 2.3 we present the unconstrained version of the target optimization problem, we note that our exploration infrastructure permits also the inclusion of arbitrary constraints either on the objectives itself or on specific parameter combinations that the designer has a priori evaluated as not interesting.

The outcome of the optimization procedure defined in Eq. 2.3 is a Pareto surface, X_p, of the explored X, thus exhibiting a large number of VLIW architectural configurations. In order to restrict the number of VLIW configuration that will be characterized as the representative customized VLIW solutions that will be propagated to the statistical compiler analysis phase, we perform a clustering on the performance - intensity solution space. We used k-means [18] clustering for the aforementioned procedure, with a configurable number of clusters, k, decided by the designer. The clustering procedure partitions the X_p solution space into k regions of interest, $X_p^{c_i}$, e.g. region of high intensity and high performance, or region of low intensity and high performance etc. Eventually, each cluster should deliver one representative VLIW architecture, that forms the optimal solution within the cluster. We define this optimal solution per cluster as the architectural configuration that minimizes area cost of the processor while maximizing both the metrics of performance and operational intensity. In order to extract this optimal configuration from each cluster, we iteratively apply the following single-objective minimization problem in every $X_p^{c_i}$ produced by the k-means clustering:

$$\min_{\mathbf{x} \in X_p^{c_i}} \frac{Area(\mathbf{x})}{Performance(\mathbf{x}) \times Intensity(\mathbf{x})} \tag{2.4}$$

For the calculation of the area cost in Eq. 2.4, the area model provided by the McPAT [19] micro-architecture framework has been used, assuming a processor technology of 90 nm.

Finding an architectures which is optimized by using the right set of compiler optimizations is an essential task to mitigate. 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. 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. Compilers perform optimization based on the program knowledge. Not all optimizations are controlled directly by an optimization pass. In this work, we select 15 compiler passes supported by LLVM compiler are as described in the Table 2.2.

2.3.1.1 DoE

Given a huge multi-objective optimization problem, it is necessary to use the design of experiment (DoE) methods, i.e., Taguchi Design of experiment [20]. DoEs are the basic components for building the exploration strategies. The DoE used in this work was based on random factors which generated a set of random designed points. In addition, the optimization algorithm used here was parallel DoE (PDoE) which was based on the possibility of performing concurrent evaluation of the different design points, i,e, in the experimental analyses, for each compiler transformations per benchmark, the number of exploration was 500, therefore, it would have given enough points for the system to use for DoE and Optimizer to generates the effects and metrics besides the Pareto points (if exists).

2.3.2 Compiler Transformation Statistical Effect Analysis

The second phase of the proposed methodology receives as input the generated custom VLIW architectures as described in the previous section, and for each of the set of micro-architectural points, it evaluates the statistical effects of the compiler transformations in a fine-grained manner. In this research work we focus on 15 of the compiler passes supported by LLVM (see Table 2.2).

As a first step in our analysis, we have to determine a reasonable number of samples to produce a robust analysis of the main effects associated with the 15 compiler parameters. In the following, each configuration of these compiler parameters, or set of compiler options, will be defined as a vector of 15 values, where each value represents a compiler pass option.

Table 2.2 Selected compiler transformations from LLVM framework

Compiler transformation	Abbreviation	Short description
Constant propagation	Constprop	Constant operands instructions are replaced with a constant value and propagated
Dead code elimination	Dce	Checks instructions that were used by removed instructions to see if they are newly dead
Function integration/Inlining	Inline	Bottom-up inlining of functions into callees
Combine redundant instruction	Instcombine	Combine instructions to form fewer, simple instructions. This pass does not modify the CFG and applies algebraic simplification
Loop invariant dode motion	Licm	Removes 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 strength reduction	Loop-reduce	Strength reduction on array references inside loops that have as one or more of their components the loop induction variable
Rotates loops	Loop-rotate	A simple loop rotation transformation
Unroll loops	Loop-unroll	A simple loop unrolling
Unswitch loops	Loop-unswitch	Transforms loops that contain branches on loop-invariant conditions to have multiple loops
Promote memory to register	Mem2reg	It promotes memory references to be register references
Memory copy optimizations	Memcpyopt	Performs various transformations related to eliminating memcpy calls, or transforming sets of stores into memset's
Reassociate expressions	Reassociate	It reassociates commutative expressions in an order that is designed to promote better constant propagation
Scalar replacement of aggregates	Scalarrepl	It breaks up alloca instructions of aggregate type (structure or array) into individual alloca instructions for each member if possible
Sparse cond const propagation	Sccp	It assumes values are constant and Basic Blocks are dead unless proven otherwise. It proves values to be constant, and replaces them with constants and Proves conditional branches to be unconditional
Simplify the control flow graph	Simplycfg	Performs dead code elimination and basic block merging

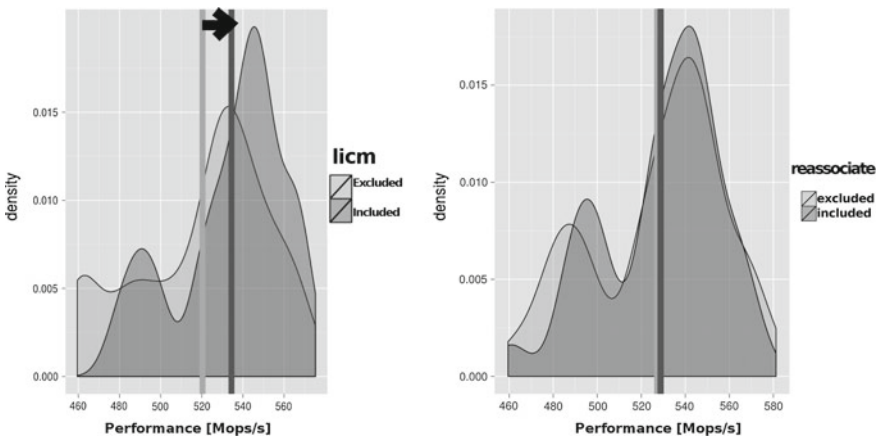
To accommodate our goal, we defined a randomized design of experiments $D_N(p)$ for each compiler parameter p. $D_N(p)$ is a list of *options sets*:

$$D(p) = \left[o_{1+}, o_{1-}, o_{2+}, o_{2-}, \ldots, o_{N+}, o_{N-}\right] \tag{2.5}$$

where o_{n+} corresponds to the n-th random option set in which compiler pass $p \in \{OFF, ON\}$ is set to its maximum value (ON) while all the others compiler passes are randomly chosen. In a dual way, o_{n-} is equal to o_{n+} except that p assumes its minimum value (OFF).

By applying this DoE, we can easily measure how much the impact of the transition $(- \rightarrow +)$ for parameter p impacts (in average over all the considered options sets) on the performance without requiring a full-factorial design. As an example, Fig. 2.3 depicts the generated performance distributions by activating and deactivating the 'licm' and 'reassociate' compiler transformations for a GSM codec application. It can be observed that while the activation of 'licm' has a clear positive effect on performance—the median is shifted towards higher performance values, this is not the case for the 'reassociate' transformation for which the activation and deactivation distributions have almost the same shape and density, thus not permitting the designer to recognize a clear trend.

As the second step, for each optimization set in $D(p)$ we evaluate the vector of performance responses with the actual architecture synthesis after the compilation and simulation of the target application. We consider the hypothesis whether the mean of the performance given by the options sets where p was minimum (or *off*) is *different* from the mean where p was maximum (or *on*). In practice, this is framed as a *null-hypothesis statistical test*, which, given the *non-parametric* (or non-gaussian)



(a) licm (b) reassociate

Fig. 2.3 **a** licm's having significant positive effect, **b** reassociate's causing no significant effect

nature of the underlying distributions,[2] cannot be assessed with as a simple ANOVA but, instead, with a Kruskal-Wallis test [21]. To complete the hypothesis test, the designer sets an acceptance ratio of $p - value\%$ meaning that the probability of 'measuring' different means when the underlying distributions are equal (or the chance of a false positive) is less than 5%.

2.3.2.1 Statistical Analysis

As mentioned in Sect. 2.2, there have been several works involving the machine learning techniques and predictions [22–24]. In this research work we have focused on analyzing the effects of applying the specific compiler transformations on the design space. The probability of certainty about the effects of a specific compiler transformation on performance metric could be done using some statistical tests; ANOVA, Kruskal-Wallis. ANOVA [25] test has been widely used as a reliable tester for normal distributions. In addition, using Kruskal-Wallis [21], is a good test tool as it assumes the distribution to be non-parametric. This method is used for testing whether samples originated from the same distribution or not. In this work, since dealing with empirical data on experimental results, we assumed the models as non-parametric, therefore, Kruskal-wallis was employed. The algorithm goes as:

- 1- Rank all the groups from 1 to N together
- 2- Statistical test is elaborated among the group to calculate the value K which contains the square of the average ranks
- 3- Finally the *p-value* is approximated as $Pr(\chi^2_{g-1} \geq K)$
- 4- If the statistic is not significant, then there is no real evidence of difference between samples and could be deduced the samples comply with the model.

In this chapter, the global threshold was set as high as 5% in order to increase the robustness of the results. Therefore, a test is deduced as passed regarding Kruskal-wallis test in which it has the p-value smaller than 0.05. In this case, a model is passed if and only if it had confidence threshold over 95%; experimental analyses represented in Fig. 2.5 will be focused later in this chapter.

In this section, we experimentally evaluate the proposed methodology. We consider the GSM codec embedded application as the driving use case, automatically generating four representative application specific architectures after applying the custom VLIW architecture selection. We use these VLIW architectures for statistically analyzing the effects of compiler transformations across differing VLIW configurations. Furthermore, we analyze the compiler transformation effects in a cross-application manner, by considering a larger set of embedded applications mapped onto a default (application independent) VLIW processor configuration.

The first subsection, introduce the experimental setup and the framework. The second subsection will contain the architectural selection based on the method described

[2] Since the distributions are built based on empirical/experimental data, the distribution is considered in general non-parametric.

in Sect. 2.3.1 and exploration on standard benchmark regarding the derived configurations will be presented. Eventually, there will be a comparison of the default architecture among 5 other benchmarks will be discussed and depicted with the statistical consolidations.

We apply the overall proposed methodology considering the GSM codec as the driving application. We apply the custom VLIW architecture selection phase to generate optimized representative VLIW architectures in an application specific manner. The considered architectural design space is depicted in Table 2.1. We configure the search procedure to randomly generate and evaluate 30 K configurations, using a uniform sampling over the targeted configuration space (Table 2.1). Applying the multi-objective optimization problem defined in Eq. 2.3 over the 30 K solutions, the Pareto surface of the configurations that maximize performance and operational intensity while minimizing resources is generated. Without loss of generality, we consider the generation of k = 4 clusters over the generated Pareto surface, aiming at generation of four GSM-specific VLIW architectures. Figure 2.4 shows results of clustering of the extracted Pareto surface and its mapping onto the two-dimensional performance versus intensity space. Each cluster has been characterized according to its position on the performance versus intensity space as: (i) HH for the cluster placed to the high intensity and high performance region, (ii) LH for the low intensity and high performance region, (iii) LL for the low intensity and low performance region, and (iv) HL for the high intensity and low performance region, respectively.

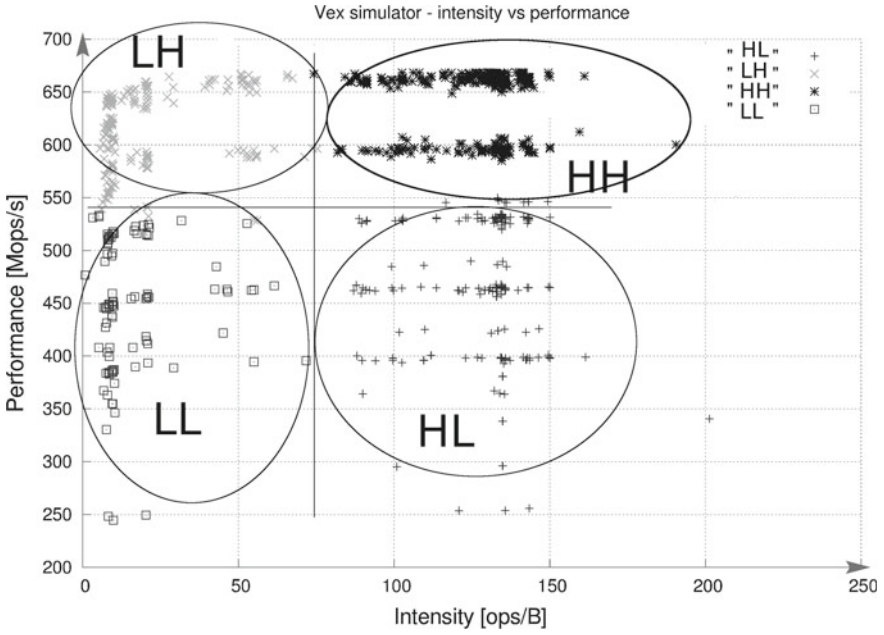


Fig. 2.4 Four clustered Pareto-sets

Table 2.3 VLIW architecture configurations

Parameters	Arch-HL	Arch-LH	Arch-HH	Arch-LL	Arch-User
lg2CacheSize	15	12	13	12	16
lg2Sets	1	3	0	1	2
lg2LineSize	7	5	5	5	5
lg2ICacheSize	16	14	16	14	16
lg2ICacheSets	1	3	3	2	2
lg2ICacheLines	6	8	7	5	6
ClkFreq	400	450	450	300	500
NumCaches	2	1	1	1	1
IssueWidth	6	6	14	9	8
NumAlus	4	6	7	3	8
NumMuls	1	4	4	14	2
MemLoad	4	3	6	5	4
MemStore	2	8	4	6	4
RegisterFile	104	100	32	76	64
BranchRegister	76	84	88	48	64

The final k = 4 representative VLIW architectures are derived after applying within each cluster the optimization operator of Eq. 2.4. Table 2.3 reports the architectural configuration for each of the k = 4 application specific VLIW architectures.

For each of the k = 4 application specific VLIW architectures, we explore the compiler level design space, defined in Table 2.2. We generate the non-parametric distribution of the performance and intensity for each compiler transformation considering 500 samples per transformation. As described in Sect. 2.3.2, the non-parametric distributions are analyzed based on Kruskal-Wallis test to specify the statistical effect, i.e. if the inclusion or exclusion of a specific transformation impacts in a specific and robust manner the two considered metrics. Table 2.4 summarizes the results of Kruskal-Wallis statistical tests for each compiler transformation over the four examined architecture configurations. As shown, four compiler passes (*inline, licm, loop-reduce and loop-rotate*), over the fifteen initially considered, had a significant impact on performance when activated. In addition, Fig. 2.5, shows the confidence level for each of the considered compiler transformations. It is shown that the four mentioned compiler transformations exhibit a high confidence level $>99\%$. Therefore, it could be implied that activating these specific transformations, the designer can be around 99% confident that the effect on performance will be the same as the one determined by the exploration.

In the second set of experiments, we perform statistical analysis in a cross-application manner. For this experimental campaign, we assume a larger set of applications (namely GSM, AES encryption engine, ADPCM codec, JPEG decoder and

Table 2.4 Summary of Kruskal-Wallis analysis on performance for GSM-specific VLIW architectures

Compiler transformation	Arch-HL	Arch-LH	Arch-HH	Arch-LL
Constprop	–	–	–	–
Dce	–	–	–	–
Inline	√	√	√	√
Instcombine	–	–	–	–
Licm	√	√	√	√
Loop reduce	√	√	√	√
Loop rotate	√	√	√	√
Loop unroll	–	–	–	–
Loop unswitch	–	–	–	–
Mem2reg	–	–	–	–
Memcpyopt	–	–	–	–
Reassociate	–	√	√	√
Scalarrepl	–	–	–	–
Sccp	–	–	–	–
Simplyfycfg	–	–	–	–

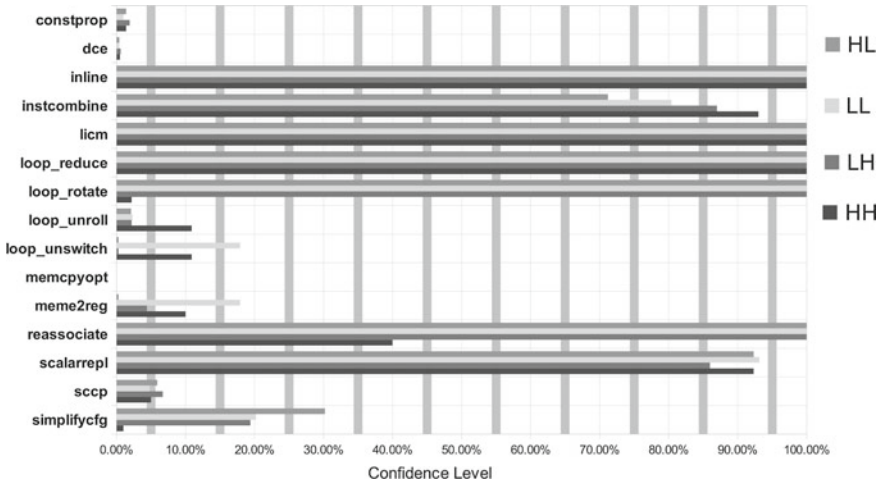


Fig. 2.5 Confidence level characterization of compiler transformations regarding the effect on performance for each on of the GSM specific VLIW architectures, resulted after Kruskal-Wallis statistical test

Table 2.5 Kruskal-Wallis analysis on performance for multiple applications

Compiler transformation	GSM	AES	ADPCM	JPEG	Blowfish
Constprop	–	–	–	–	–
Dce	–	–	–	–	–
Inline	√	–	√	√	–
Instcombine	√	–	√	√	√
Licm	√	√	√	√	√
Loop reduce	√	√	√	√	√
Loop rotate	√	√	√	√	–
Loop unroll	–	–	–	–	–
Loop unswitch	–	–	–	–	–
Mem2reg	√	√	√	√	√
Memcpyopt	–	–	–	–	–
Reassociate	√	–	–	–	–
Scalarrepl	√	–	–	–	√
Sccp	–	–	–	–	–
Simplyfycfg	–	–	–	–	–

Blowfish block cipher). The performance of each applications has been evaluated considering a user specified VLIW architecture, Arch-User, defined in the last column of Table 2.3. For each benchmark the compiler transformation statistical effect analysis (Sect. 2.3.2) is applied, considering distributions of 500 samples per compiler transformation. Table 2.5 summarizes in an aggregated manner the results of the Kruskal-Wallis analysis considering in each case a confidense level $\geq$5%. For the specific setup, we observe that there is a set of four compiler parameters (*licm, loop reduce, loop rotate and mem2reg*) with significant effect on the performance and with a high confidence level over all the examined application use cases. Furthermore, examining each application in isolation, the designer can derive which are the compiler parameters that need to be pre-allocated, thus reducing significantly the design-time required to optimize the performance of the targeted application during iterative compilation exploration. As an example, we depict in the Fig. 2.6, the normalized speedup gains achieved by activating the compiler transformations proposed by our methodology in comparison with several well-known compilation strategies. It is shown that the proposed methodology defined speedup gains in all the examined cases between 16 and 23%.

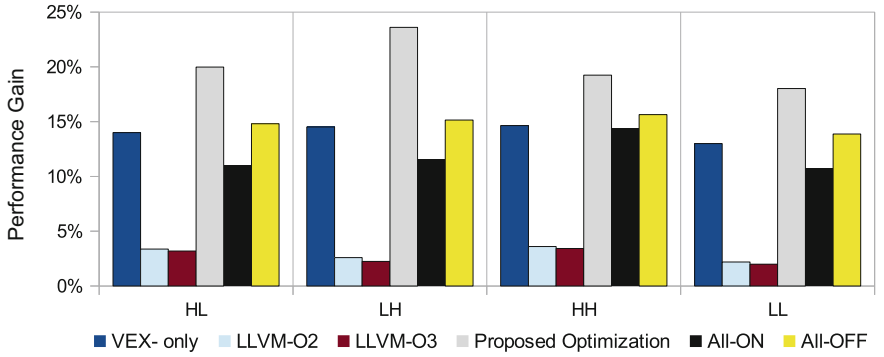


Fig. 2.6 The gained speed-up we gained comparing to the default LLVM-O1 optimization level in GSM benchmark

2.4 Conclusions and Future Work

This chapter presented a methodology for a compiler/architecture co-exploration of VLIW platform design. It provides the designer with an integrated environment to automatically (i) generate optimized application specific VLIW architectural configurations and (ii) analyze in a fine-grained manner the effects of compiler level transformations regarding the performance and operational intensity trade-offs. Being focused more on the analysis part, we showed that the adoption of the specific methodology either in a cross-architecture and/or cross-application manner, can deliver significant application specific insights thus enabling the designer to guide through decisions regarding the architecture and the compilation optimization strategy. Future work is aligned with our strong belief that the proposed methodology can be exploited in a straightforward manner within automated design frameworks focusing on performance optimization through iterative compilation and architecture specialization.

References

1. Fisher JA, Faraboschi P, Young C (2009) VLIW processors: once blue sky, now commonplace. IEEE Solid-State Circuits Mag 1(2):10–17
2. Fisher JA, Faraboschi P, Young C (2004) Embedded computing: a VLIW approach to architecture, compilers and tools. Morgan Kaufmann, Burlington, MA
3. Ascia G, Catania V, Palesi M, Patti D (2005) A system-level framework for evaluating area/performance/power trade-offs of vliw-based embedded systems. Design automation conference. In: Proceedings of the ASP-DAC 2005. Asia and South Pacific, vol 2., pp 940–943
4. Fisher JA (1981) Trace scheduling: a technique for global microcode compaction. IEEE Trans Comput 30(7):478–490
5. Hwu WMW, Mahlke SA, Chen WY, Chang PP, Warter NJ, Bringmann RA, Ouellette RG, Hank RE, Kiyohara T, Haab GE et al (1993) The superblock: an effective technique for VLIW and superscalar compilation. J Supercomput 7(1–2):229–248

6. Quinlan D (2000) Rose: compiler support for object-oriented frameworks. Parallel Process Lett 10:215–226
7. Fenacci D, Franke B, Thomson J (2010) Workload characterization supporting the development of domain-specific compiler optimizations using decision trees for data mining. In: Proceedings of the 13th international workshop on software & compilers for embedded systems, p 5. ACM
8. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76
9. The LLVM website (2013). http://www.llvm.org/
10. Faraboschi P, Homewood F (2000) ST200: a VLIW architecture for media-oriented applications. In: Microprocessor Forum. San Jose, CA
11. Saptono D, Brost V, Yang F, Prasetyo E (2008) Design space exploration for a custom VLIW architecture: direct photo printer hardware setting using VEX compiler. In: Proceedings of the 2008 IEEE international conference on signal image technology and internet based systems, SITIS '08, pp 416–421, Washington, DC, USA. IEEE Computer Society
12. Wong S, Van As T, Brown G (2008) ρ-vex: a reconfigurable and extensible softcore VLIW processor. In: International conference on ICECE Technology. FPT 2008, pp 369–372. IEEE
13. Hewlett-packard laboratories. vex toolchain. [online], available. http://www.hpl.hp.com/downloads/vex/
14. Multicube explorer. http://m3explorer.sourceforge.net/
15. Zaccaria V, Palermo G, Castro F, Silvano C, Mariani G (2010) Multicube explorer: an open source framework for design space exploration of chip multi-processors. In: 23rd International conference on architecture of computing systems (ARCS), pp 1–7. VDE
16. R Core Team et al. (2013) R: a language and environment for statistical computing. Vienna, Austria
17. Palermo G, Silvano C, Valsecchi S, Zaccaria V (2003) A system-level methodology for fast multi-objective design space exploration. In: Proceedings of the 13th ACM Great Lakes symposium on VLSI, pp 92–95. ACM
18. Kanungo T, Mount DM, Netanyahu NS, Piatko CD, Silverman R, Wu AY (2002) An efficient k-means clustering algorithm: analysis and implementation. IEEE Trans Pattern Anal Mach Intell 24(7):881–892
19. Li S, Ahn JH, Strong RD, Brockman JB, Tullsen DM, Jouppi NP (2009) McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: International symposium on microarchitecture. MICRO-42. 42nd Annual IEEE/ACM, pp 469–480. IEEE
20. Roy RK (2001) Design of experiments using the Taguchi approach: 16 steps to product and process improvement. Wiley, Hoboken
21. Breslow N (1970) A generalized Kruskal-Wallis test for comparing k samples subject to unequal patterns of censorship. Biometrika 57(3):579–594
22. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MF, Thomson J, Toussaint M, Williams CK (2006) Using machine learning to focus iterative optimization. In: Proceedings of the international symposium on code generation and optimization. IEEE Computer Society, pp 295–305
23. Cavazos J, Dubach C, Agakov F (2006) Automatic performance model construction for the fast software exploration of new hardware designs. In: Proceedings of the 2006 international conference on compilers, architecture and synthesis for embedded systems, pp 24–34
24. Dubach C, Cavazos J, Franke B (2007) Fast compiler optimisation evaluation using code-feature based performance prediction. In: Proceedings of the 4th international conference on computing frontiers, pp 131–142
25. Thompson B (2002) Statistical, practical, and clinical: how many kinds of significance do counselors need to consider? J Couns Dev 80(1):64–71

http://www.springer.com/978-3-319-71488-2

Automatic Tuning of Compilers Using Machine Learning

Ashouri, A.H.; Palermo, G.; Cavazos, J.; Silvano, C.

2018, XVII, 118 p. 23 illus., 6 illus. in color., Softcover
ISBN: 978-3-319-71488-2