A POLYHEDRAL-BASED SYSTEMC MODELING AND GENERATION FRAMEWORK FOR EFFECTIVE LOW-POWER DESIGN SPACE EXPLORATION

BY

WEI ZUO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

With the prevalence of systems-on-chips there is a growing need for automation and acceleration of the design process. A classical approach is to take a C/C++ specification of the application, convert it to a SystemC (or equivalent) description of hardware implementing this application, and perform successive refinement of the description to improve various design metrics. In this thesis, we present an automated SystemC generation and design space exploration flow alleviating several productivity and design time issues encountered in the current design process. We first automatically convert a subset of C/C++, namely affine program regions, into a full SystemC description through polyhedral model-based techniques while performing powerful data locality and parallelism transformations. We then leverage key properties of affine computations to design a fast and accurate latency and power characterization flow. Using this flow, we build analytical models of power and performance that can effectively prune away a large amount of inferior design points very fast and generate Pareto-optimal solution points. Experimental results show that (1) our SystemC models can evaluate system performance and power that is only 0.57% and 5.04% away from gate-level evaluation results, respectively; (2) our latency and power analytical models are 3.24% and 5.31% away from the actual Pareto points generated by SystemC simulation, with 2091x faster design-space exploration time on average. The generated Pareto-optimal points provide effective low-power design solutions given different latency constraints.

To my parents, for their love and support.

ACKNOWLEDGMENTS

This work is partially supported by an Intel grant. I would like to thank Dr. Mondira (Mandy) Pant, Dr. Andrey Ayupov, Dr. Taemin Kim, and Dr. Kyuntae Han of Intel for helpful discussions. I would also like to thank Dr. Louis-Noel Pochet from Ohio State University for his advice and help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The semiconductor industry has moved to high-level hardware-capable languages to model the entire SoC. In particular, transaction-level modeling (TLM) with SystemC is immensely popular and is being widely used currently [1]. Despite the popularity and advantages of SystemC, it has several limitations. One is the inherent difficulty associated with SystemC generation. Traditionally, the SoC design specification is often provided in a high-level language such as C/C++ by software engineers as a golden reference model. Also, hardware/software partitioning is manually done by system engineers, and the hardware portions are re-implemented in SystemC with software being run on microprocessor SystemC IPs in order to simulate the complete system [2]. This approach has a few drawbacks. First, additional design effort in SystemC is needed to re-implement the hardware portion of the SoC. Second, since this SystemC model is built manually, it is difficult to effectively and extensively explore different design decisions; this leaves designers uncertain about the optimality of the current design. Third, at such an early design stage, accurate power estimation becomes difficult with very little lower level implementation details [3]. Finally, the SystemC IPs are often written with the emphasis on code re-usability and are optimized for simulation speed. Thus, it may not be synthesizable by high-level synthesis (HLS – the process of automatic translation of high-level hardware description to RTL) tools. Furthermore, the SystemC model of the accelerator only provides fast modeling of a single design point. A designer seeking to design a new system should instead look at an optimal power and latency trade-off curve. Generating this trade-off curve requires a detailed exploration of the practical design space. Even with expedited simulation, the design space remains too large to consider while blindly iterating over the entire space.

As an illustration, *loop tiling* is a powerful transformation to improve data locality and parallelism. Different tile size determines the number of tiles

and the memory size associated with the tile. In addition, *loop unrolling* can also significantly improve the performance at the cost of area and power. Thus, finding the optimal tile size and unrolling degree is critical to optimize the accelerator design. For example, consider a given affine segment of code consisting of quadruple-nested *for* loops. Each loop has a possible iteration tile size of 1 to N. Additionally, if we assume no inter-loop dependence, each loop body can be independently replicated via unrolling. All these different configurations lead to different design decisions, hence the resulting design space consists of $(2N)^4$ points. If N=1024 and each point requires just 10 seconds to obtain, then complete design space exploration would require 5.4 million years!

In this work, we demonstrate that these limitations can be successfully addressed with an automated design flow when focusing on the class of affine programs. This important program class encompasses numerous computation methods used in image processing, medical imaging, statistics etc. [4, 5, 6]. It has the distinguishing feature of having a control-flow that can be exactly described at compile-time [4] thereby allowing the design of accurate power/latency models, and very powerful optimization frameworks to expose parallelism with data reuse that have already been developed [5, 6]. Our proposed framework takes a C/C++ application as an input, automatically extracts the region(s) which are affine programs, and for those: (1) automatically performs software transformations to expose parallelism and temporal data locality; (2) automatically emits two SystemC variants of the program region, a high-level model embedding accurate power and latency information, and a fully synthesizable version for HLS implementation; (3) automatically implements a design space exploration engine considering different loop tile sizes and parallelism degrees.

By tackling these challenges, our framework enables designers to have accurate SystemC-level accelerator models considering power and latency, which can be further seamlessly integrated in the SoC platform, and to speedily explore the tremendous design space considering different micro-architecture and design constraints. To the best of our knowledge, this is the first work with integration of software transformation, hardware modeling with SystemC generation and effective design space exploration. Our contribution is an automatic design flow that generates accelerator SystemC model and delivers effective design space exploration of different accelerator

micro-architectures. It has the following unique features:

- Automated C-to-SystemC transformation engine that generates SystemC code for regular loop-based applications, enabling both accurate high-level power/performance modeling and high-level synthesis solution;
- An effective characterization flow for latency and power estimation enabling fast and accurate high-level modeling for regular affine programs;
- Accurate analytical power and latency models for effective design space pruning;
- Efficient accelerator design space exploration for accurate power and latency Pareto curve generation to guide effective low-power design.

This thesis is organized as follows. Chapter 2 covers the background and related work. Chapter 3 presents the methodology and implementation of our SystemC modeling framework. Chapter 4 presents the methodology of our power and latency modeling framework. Chapter 5 presents experimental results, before concluding in Chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

Application modeling Transaction-level modeling (TLM) is a popular methodology for high-level system modeling [1]. In TLM, details of communication and computation models are separated, and various models are provided with different time approximations, namely loosely-timed, approximate-timed and cycle-accurate [7]. This mix of abstraction levels captures the function modules, micro-architectural details and essential timing information in one language, hence enabling hardware/software co-simulation and fast design space exploration. In practice, TLM is supported by SystemC where the computation and communication models are implemented as function calls and channels. However, the creation of such a SystemC model is not easy. Many previous works focused on generating the SystemC from Unified Modeling Language (UML) [8], which is a high-level language for hardware specification. However, the transformation from C/C++ to SystemC is less studied. Taking another approach, some HLS tools generate the SystemC simulation model after synthesis is done. Nonetheless, this model is mainly for representing the behavior of a particular synthesized hardware. Thus, it involves tedious implementation details and cycle-accurate information, and also lacks power annotation. This low-level abstraction is not a good option for high-level modeling.

Software transformations Efficient design requires a good mapping of the computation to the hardware resources. Data locality must be considered, to reduce data movement between computing elements to improve performance and energy. Coarse-grained parallelism is critical, typically to enable replication of modules for parallel execution. Other forms of parallelism-based optimizations (e.g., for pipelining or task parallelism) are also critical for good design, and automating these optimizations has proved to be a very difficult challenge. In this work, we consider *affine programs*, that is, the set

of (sub-)programs where the data- and control-flow can be expressed using affine functions of the surrounding loop iterators and variables invariant in the program region. These regions are known as Static Control Parts [4]. Numerous previous works have shown the significant advantages of operating on this program class, for instance for automatic data locality and parallelism extraction via loop tiling [5]. Recent work demonstrated the power of an integrated approach for high-level synthesis using the polyhedral framework [6], providing the PolyOpt software infrastructure to optimize affine programs for HLS. In this work, we use PolyOpt to automatically detect affine program regions and perform all software transformations needed to make the code suitable for tiling, a.k.a. loop blocking; and we use the Pluto algorithm [5] for additional transformations for parallelism and data locality improvement. Because of the static control flow properties of affine programs, the behavior of the program is essentially captured by iterating the same computation chunks (tiles), and this precise set of iterations can be exactly modeled at compile-time [6].

High-level power modeling Extensive studies have been done for power modeling at different abstraction levels. Among those, power macro-modeling has been widely adopted. However, most previous work is focused on building the power macro-models for primitive components [9, 10], and consists of lookup tables to estimate power for that circuit. The indices are different variables capturing the relationship of power and dependent variables such as switching probability and operating frequency, etc. However, this method has two drawbacks. First, a large offline characterizing effort is required to build a comprehensive library covering all basic circuits. Second, most of these studies stay at RTL level and the accuracy of this method is limited when applied at high level (also known as behavior level), where little hardware detail is available. Thus, some assumptions and predictions must be made on the hardware implementation, which limits the accuracy. Also, an online training is sometimes required to compensate for this inaccuracy [9].

Recently, SystemC level power modeling has attracted wide attention. Many studies focus on TLM power modeling [11, 12]. However, this often requires the users to generate their own power macros to plug into the model. In contrast, we automatically generate them. Others build power model for hardware directly at the SystemC level, but they are usually tar-

geted at communication structures and interconnecting networks [13]. On the software side, techniques to accelerate simulation of functional models using host-compiled techniques have also been developed [14]. But accurate behavior-level SystemC power modeling for hardware IPs is still a highly challenging problem. In this work, we also build models for each unique computation block (tile). But we leverage several fundamental properties of affine programs, using the ability to describe analytically their control and data flow in closed form yet exact representations [4], thereby bypassing the need to execute the model for the entire application. Accuracy is achievable because of the inherent regularity of this program class.

Design space exploration for power improvement Power is one of the key design constraints for SoC. Many works target analyzing and optimizing power at early design stage. Design space exploration for power improvement has been extensively adopted, considering different design factors such as communication cost [15], memory hierarchy [10], macro-architecture [10], application execution on certain processors [16], etc. However, most of these methods rely on a detailed power modeling for the primitive components, as well as carefully designed high-level intermediate representation/prediction of design implementation [10, 17], which limits the accuracy of power estimation or the universality of applications.

Numerous previous studies have been done on different aspects of considering high-level modeling and design space exploration separately. However, a fully automated and integrated solution for accelerator modeling and design space exploration is still a big challenge.

CHAPTER 3

SYSTEMC GENERATION FRAMEWORK

3.1 Overview of the Framework

Latency and power are two key constraints for energy-smart SoC designs. It is important to estimate these factors accurately at an early design stage so they can help guide system-level design space exploration. SystemC is widely used at the system level to deliver fast simulation speed, and many works have been done to create different levels of SystemC models to achieve the balance between accuracy and simulation speed. In this work, we propose a new system-level design flow to estimate power and latency accurately for SystemC models of electronic circuits, targeting IPs or hardware cores that contain multiple affine loops. Compared to previous work, it has the following distinguishing features. (1) It is fully associated with the PolyOpt compilation framework. Thus, we can use the powerful loop transformation in PolyOpt to expose atomic tiles in the source code and transfer them to SystemC modules. (2) It generates both communication and computation blocks and builds the corresponding latency and power models. Furthermore, it considers different input switching activities, and simulates the interaction between the accelerator and the system. (3) It characterizes the latency and power for one tile, and it is easy to integrate these tile-level models to a single framework for the whole design power/latency estimation. Thus, we can evaluate and explore different optimization / design options provided by PolyOpt. (4) Compared to Functional Level Power Analysis, our tile-based power/latency analysis flow has the flexibility to use IPs and blackboxes in the design, and to target different technologies. (5) It achieves fast runtime through the use of behavior-level SystemC models for fast simulation; this is because we build detailed power/latency models for tiles of the loops without the need to simulate the entire design at lower design levels. (6) With high

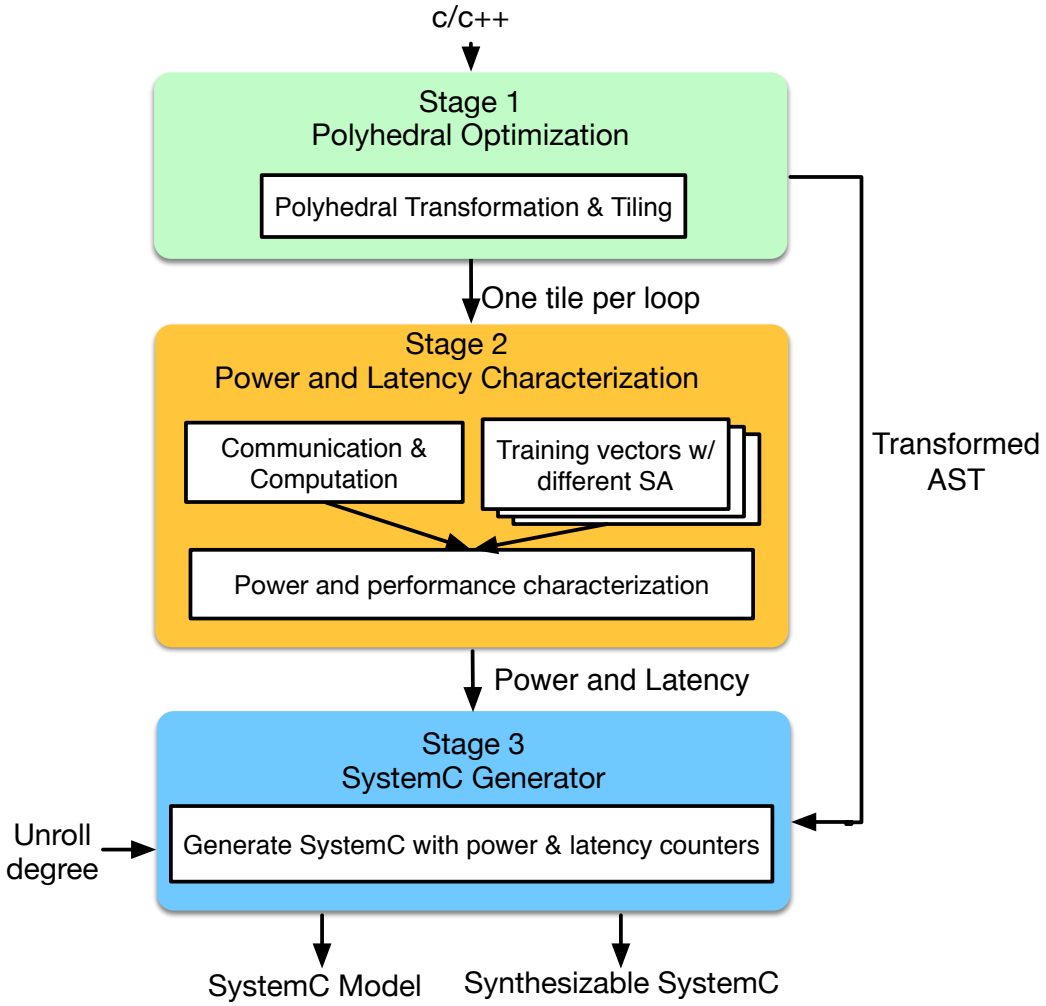

Figure 3.1: SystemC generation and analysis framework

speed and accuracy, our SystemC modeling flow can enable early design evaluation and design space exploration.

We first highlight the key features of our design flow, which are summarized in Fig. 3.1. Our framework is a multi-stage process to automatically generate SystemC codes and build power and performance models. First, we transform the input program to make loop tiling possible, and tile the loops using polyhedral transformations with the PolyOpt infrastructure. After that, we extract tiles and separate them into components: the computation blocks and the communication blocks (at the beginning/end of the tile, to transfer the data needed from/to memory to/from the local buffers). In the next stage, we separately characterize the power and latency of these parts with information extracted from gate-level simulation. Then, we build a power model for each part, considering different input switching activities. Finally, using polyhedral analysis, we generate a SystemC model for the tiled loop kernels and back-annotate the power and latency information to the SystemC model to compute the corresponding values for the entire design.

3.2 Architecture of Generated Accelerator

The general architecture of generated accelerator is shown in Fig. 3.2. It consists of computation modules (Acc_tile) and local memories (local Mem). The computation blocks in our model read data from local memories and execute computation operations; and the communication blocks take charge of transferring data between local memories and main memory. We have two input flags to control the generated hardware: (1) Tile size, which affects the loop tiling decision in software transformation stage, and further affects the local buffer size, which in turn influences the communication cost. (2) Parallelism degree, which decides at which loop level the loop unrolling is implemented, so that we can do complete unrolling for replication at that level if semantically possible. Thus, together with the tile size, parallelism degree decides the replication numbers of one tile. Within each block, we implement an input switching activity calculation function, which dynamically computes the input switching activity and guides the selection of power information associated with the tile.

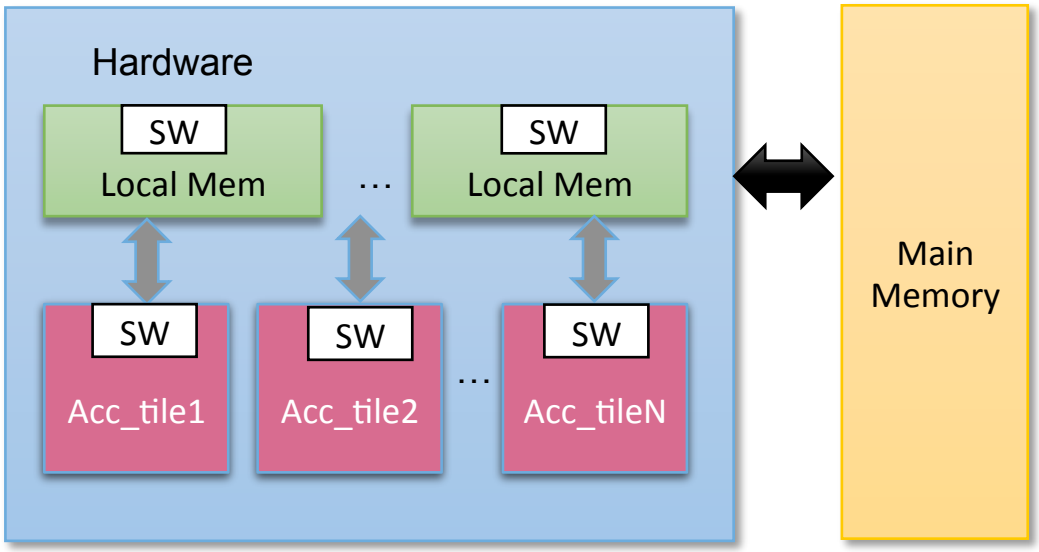

Figure 3.2: Architecture of the generated accelerator

3.3 Software Transformations

The first step of our framework is to apply software transformations to restructure the computation so as to expose (1) good temporal data locality to reduce data movements; (2) sufficient parallelism, to enable parallelization via replication; (3) explicit atomic computation tiles and the associated tile data reuse buffers and data transfers operations. All these transformations are available in the PolyOpt framework which uses a set of very powerful mathematical models to reorganize the operations in the computation while preserving the semantics [6].

It is important to note that reorganizing the computation into atomic tiles, or loop tiling, is central to the development of our framework. Most practical affine programs are tilable *provided the program is first transformed to make tiling possible*. Enabling the applicability of tiling in a fully automatic way has been previously studied and addressed in the state-of-the-art Pluto framework for instance [5], which has been ported for HLS purposes in the PolyOpt framework. A complex sequence of loop transformations including loop interchange, skewing, fusion, distribution, peeling, code motion, etc., is automatically computed and applied to make the program tilable whenever possible, dramatically broadening the class of programs that can be tiled. Fig. 3.3 shows an example of tiling.

3.4 Power and Latency Characterization

At this stage, we focus exclusively on particular tiles and construct a detailed power and latency characterization for each. These results are used to compute the total application latency and power in integration stage. Computation blocks of a tile are fed into a HLS tool to generate RTL code, and to generate the local memory library using a memory compiler. These two portions are combined together using an automatically generated wrapper and go through the logic synthesis to generate the netlist. Then, a gate-level simulation is applied followed by the power analysis to generate the latency and power information. Loop unrolling is used when specified by the user, otherwise loop pipelining is implemented. For the communication blocks, we generate a library of memory IPs with different sizes, and characterize the read/write latency and power according to different input switching activities and build the look-up table.

We automatically generate the testbench and the training input vectors with controllable switching activities. We construct a test vector pool with switching activities from 0.1 to 0.9 with a step size of 0.1. We use uniform distribution to decide the flipping bits in adjacent test vectors. Then, a gate-level simulation is done to obtain the latency and switching activity, which is then fed into the power analyzer to obtain the power profile. The power is composed of internal, switching and leakage power. The first two are due to capacitive charging/discharging of output load and internal transistors of the logic gates, respectively, and are the source of dynamic power. The latter represents the static power. This information is later back-annotated into the SystemC model. Note that since our framework is general, the user can provide real testing vectors for characterization to accommodate realistic switching activities.

3.5 SystemC Code Generation

The third step in the framework is the implementation of the *SystemC Generator*. This is integrated in the PolyOpt application and uses information acquired by polyhedral model analyses. The generator receives as input the transformed AST produced by PolyOpt, which has already been annotated

with polyhedral information on loop trip count, data dependences, parallelism, etc., as well as the unroll degrees selected by the user.

The generator proceeds recursively, following the AST structure. It first creates a Function Module (FM) node for each tile, and proceeds recursively (bottom up) for the surrounding loops, as illustrated in Fig. 3.3. If the loop is annotated by PolyOpt as parallel, then parallelism via replication can be implemented, and the user-provided information about the unrolling factor is used to replicate the tile modules in the loop body. For non-tile FMs, we remove the loop and replace it with SystemC initialization and synchronization signals. Latency will be emulated using the trip count expression (which can be a function of the FM parameters), which is computed for all loops by PolyOpt. At this stage, all the unrolled loops are implemented in FM format.

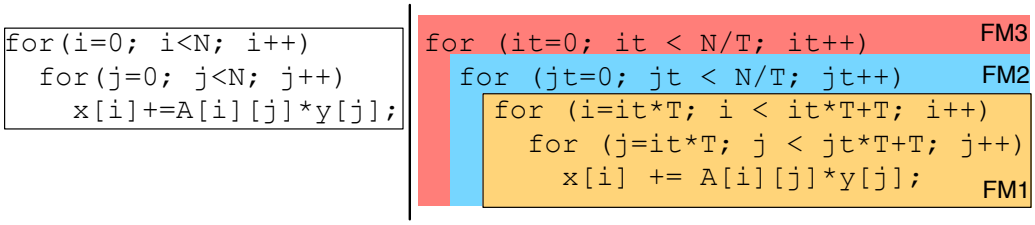

Figure 3.3: Left: C/C++ code fragment isolated by PolyOpt. Right: tiled code and equivalent Function Modules generated. FM1 is a tile of size $T \times T$.

To create the SystemC model, a SystemC module is created for each FM node, including creating ports and signal bindings and inserting the corresponding function body to this FM as a SC_THREAD. Timing information is embedded in `wait(...)` calls to accurately represent the latency of modules. Power information is obtained from the power model indexed by different input switching activities. To accurately calculate the input switching activities for each tile, we implement a switching activity calculation function by aggregating the Hamming distance between the input vectors of two adjacent iterations, and then divide the sum by the iteration numbers of one tile, which is obtained from the trip count expressions. Note that this function dynamically computes the input switching activities of one tile, which captures the propagation of switching activities among different tiles, as well

as the state probabilities of each tile.

Another back-end generates the fully synthesizable version to generate SystemC code suitable for HLS tools. This code differs from the SystemC modeling code in three aspects. First, to implement the parallelism via replication, we need to solve the memory access conflicts. We adopt a memory partitioning and banking strategy to divide the original array into different memory banks [18]. Given the memory banks and array partition degrees, our flow replaces the original array by the partitioned banks, and generates correct accessing order for each bank of the original arrays by constructing the corresponding branch conditions. Second, to solve the read/write conflicts to the same memory element in the same statement, we identify memory read and write access, and generate separate statements by inserting temporary variables and `wait()` statements to schedule them. Finally, we remove all the latency and power annotation functions used in SystemC modeling.

3.6 Integration

After generating the SystemC, the power and latency information for one tile is back-annotated into the SystemC model, and a fast SystemC simulation is then run for estimating the entire design. Latency is modeled using the SystemC wait() statements inserted in each tile. Power is modeled using the embedded monitoring function. A SystemC simulation is used for estimation. Once each tile module starts running, a function named "update_power()" is called, updating the power value by adding the module's dynamic power. The wait() function is then called with the function's execution time. Once the execution is done, the following execution update_power () is called again to accumulate the module's idle power value. We now present the computation of the values of the wait latencies and power annotations.

3.6.1 Latency computation for computation blocks

For computation blocks, the latency of executing one loop iteration is equal to the execution time of its body (i.e., the associated tile FM; in this section, we do not distinguish between *tile* and *function module*) plus the overhead of parameter setup time and the synchronization for the corresponding module.

We have found these types of latency overhead to be negligible for the kind of parallelism we implement. Thus, the overall execution time estimation is the sum of the latency of the tiles, considering the total number of tiles to be executed in sequence. Note that for unrolling via replication, as we only unroll a sync-free parallel loop, the total number of tiles executing serially is the number of tiles divided by the unroll factor. Within one tile, the iteration number of the tile body is proportional to the tile size.

Using polyhedral analysis makes it possible to compute exactly how many times a loop is executed [6] by counting the number of integer points in the iteration domain of the loop. We use this specific feature of affine programs to derive a simple yet precise latency model in our setup, where for each different tile module T_i having a latency $lat(T_i)$ computed via HLS, we compute the latency as:

$$L_{comp} = \sum_i |Domain(T_i)| * lat(T_i) * sfactor_i/ufactor_i \tag{3.1}$$

where $Domain(T_i)$ is the set of all executions of T_i, which is described using affine inequalities, that, is the iteration domain of the tile T_i [4]. $|S|$ denotes the cardinality of the set S and is an Ehrhart polynomial here [19]. This value is automatically computed by PolyOpt, and it multiplies the tile size factor $sfactor$, which indicates the difference between the given tile size and the tile size used in characterization flow. Then, the value is divided by the unroll factor $ufactor_i$ by which the tile is replicated (it is 1 for all tiles which are not replicated).

3.6.2 Latency computation for communication blocks

For communication blocks, we do not apply unrolling due to the memory port limit. Thus, for each read and write block Rd_i and Wr_i, given the read/write latency lat_{rd} and lat_{wr} per access, the total read/write latency L_{rd}/l_{wr} are computed as:

$$L_{rd} = \sum_i |Domain(Rd_i)|(\sum_{iter_i} iter_i * lat_{rd}) \tag{3.2}$$

$$L_{wr} = \sum_i |Domain(Wr_i)|(\sum_{iter_i} iter_i * lat_{wr}) \tag{3.3}$$

where $Domain\,(Rd_i)$ and $Domain\,(Wr_i)$ are computed in the same way as $Domain(Ti)$, and $iter_i$ are the number of iterations within one communication block. Thus, the total latency L_{total} is:

$$L_{total} = L_{comp} + L_{rd} + L_{wr} \tag{3.4}$$

The communication/computation overlap is not modeled in this work; we will further work on extending the flow to support that.

3.6.3 Power integration for computation blocks

To compute the average power consumed by the accelerator, we first aggregate the energy consumed by different components, and then divide the total energy by latency, which is estimated using previously mentioned method. Since tiles are being repeatedly invoked by modules, we assume the leakage power consumption of each computation block is identical for all executions. Therefore, given $Pow_{leak}(T_i)$, the static power for one tile module T_i as computed by the tools, the energy gained by static power $P_{comp_{leak}}$ is:

$$E_{comp_{leak}} = \sum_i Pow_{leak}(T_i) * ufactor_i * L_{total} \tag{3.5}$$

For dynamic power, the tiles only consume dynamic power when they are activated. As we need to embed in the SystemC file a measure of the average power, we cannot directly sum up the individual tile dynamic powers to calculate the total value. Instead, we provide an average for the design by first integrating to get the energy per module, and then normalize by the total execution time to get an average instantaneous dynamic power metric. Note that unlike the integration of latency (where we can directly multiply $lat\,(T_i)$ by the unroll factor, since all tile copies have the same latency for different module copies), the input switching activities of each tile copy may be different, thus different tiles consume different amount of energy. Thus, instead of multiplying the power for one tile by unrolling factor, we aggregate the power of all tile copies. Thus, given Pow_{dyn}, the dynamic power of one tile T_i, we compute the dynamic energy as:

$$E_{comp_{dyn}} = (\sum_i \sum_{j=0}^{ufactor_i} Pow_{dyn}(T_{ij}) * lat(T_i)) \tag{3.6}$$

where T_{ij} is the j-th copy of tile T_i.

3.6.4 Power integration for communication blocks

For each computation block, the power information of read/write operations of corresponding local memory is available, together with the read/write latency per access. For the static energy, the calculation is similar to that for $E_{comp_{leak}}$; given leakage power for each local memory M_i, the energy is

$$E_{comm_{leak}} = \sum_i Pow_{leak}(M_i) * L_{total} \tag{3.7}$$

For dynamic power, the blocks only consume dynamic power when communication happens since we do not have the unrolling factor for communication blocks. Thus, given the dynamic power for each read/write block $Pow_{dyn}(Rd_i)$ and $Pow_{dyn}(Wr_i)$, the energy is:

$$E_{rd_{dyn}} = \sum_i Pow_{dyn}(Rd_i) * L_{rd} \tag{3.8}$$

$$E_{wr_{dyn}} = \sum_i Pow_{dyn}(Wr_i) * L_{wr} \tag{3.9}$$

Therefore, the average power Pwr_{avg} is:

$$Pwr_{avg} = (E_{comp_{leak}} + E_{comp_{dyn}} + Ecomm_{leak} + Erd_{dyn} + Ewr_{dyn})/L_{total} \tag{3.10}$$

CHAPTER 4

POWER AND LATENCY MODELING AND DESIGN SPACE EXPLORATION

4.1 Power and Latency Modeling

Using the methodology provided above, we are able to efficiently and accurately acquire the power and latency associated with a given configuration of C/C++ code. However, the overall design space covers the entire range of possible code configurations. This includes all possible combinations of iteration tile sizing and loop unrolling configurations. Instead of iterating exhaustively over the entire design space, we recognize that the general relationships between code parameters and the resulting power and latency can be roughly extracted by sampling. We therefore implement the following methodology to effectively model the power and latency within the design space. The overall framework is shown in Fig 4.1.

First, we specify the loop structure of the C/C++ code that is being considered. For each loop, we indicate the range of valid iteration tile sizes. When considering nested loops, we also identify the maximum loop depth for which unrolling should be applied. While an exhaustive search would involve every possible combination of tile iteration sizes and loop unrolling configurations, we greatly condense the process by iterating over each loop range with a stride that increases exponentially. This sampling rate is primarily motivated by recognizing that power and latency trends in the memory IP incorporated into our accelerators generally correspond to sizes correlated with a log scale. Therefore, we sample the design space on a scale that is logarithmically proportional to the entire space, resulting in a significant reduction of the overall framework runtime.

Second, once the array of sampling vectors is generated, we iterate over each vector. Every vector is exported to the files corresponding with the SystemC generator inputs. A script automatically initiates SystemC code

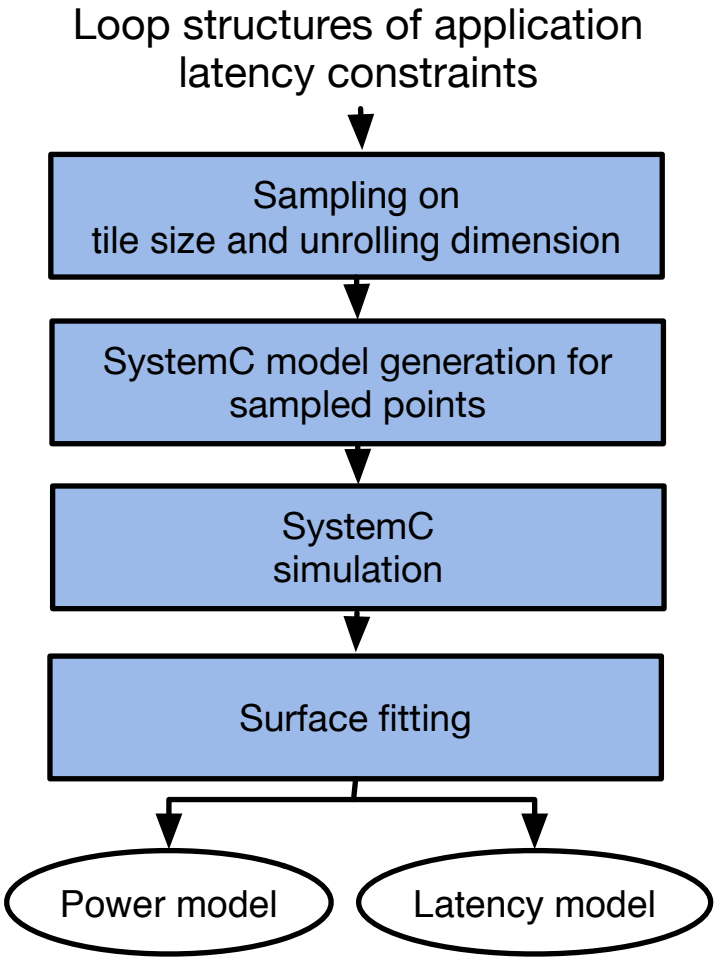

Figure 4.1: Analytical power/latency modeling framework

generation, compilation, and execution. The power and latency results for each sampling vector are output by the SystemC execution, and are combined into a single file.

Third, we feed this file into a MATLAB script. This script automatically identifies the sampling vectors and their corresponding power and latency. By specifying the range of sample vector points (ranges of tile sizes and loop unrolling), we generate a new array of sampling vectors. In generating these vectors, we iterate over each range with a constant stride (instead of an exponential stride), resulting in an array of sampling vectors that span the design space at a much finer granularity. This array is used for design space interpolation. Using the sampled data, we generate a two-surface curve: one

describes the relationship between the code parameters (tiles sizes and unrolling configuration) and power; the other describes the relationship with latency. As each sampling vector is multidimensional and can have very high dimensionality (some benchmarks have 6 dimensions), we use a general tool that allows the interpolation of hyper-surfaces. Specifically, we have chosen to utilize the MATLAB function *griddatan*, which uses the multidimensional sample points to perform a triangulation-based linear interpolation for each point specified in our high granularity array. So long as the input data remains continuous, this methodology results in power and latency estimations that are reasonably acculturate. The combination of all power and latency estimations at a high granularity constitutes our power and latency models.

To illustrate the accuracy of our model, Fig. 4.2 (Left) shows our model of the design space of benchmark AtAx. As described above, this model is generated using our interpolation methodology applied to points acquired by logarithmically sampling the design space. The high granularity interpolation is set to model all loop unrolling configurations and all iteration tile sizes of striding 4. In comparison, Fig. 4.2 (Right) shows the complete design space as acquired directly from SystemC simulation. We evaluate the accuracy of our model by comparing each point in our interpolated model with the value calculated directly using our SystemC framework. In doing so, we find that the average error in power across all points is 4.10% while the average error in latency is 3.28%. Therefore, despite the small subset of sample points, the model maintains accuracy and preserves the general curve trends shown in the real design space.

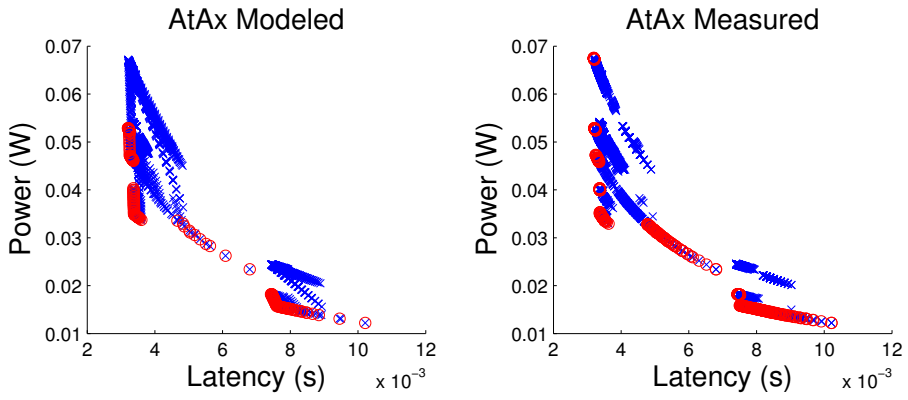

Figure 4.2: Modeled design space and measured design space

4.2 Design Space Exploration

Combining the power and latency modeling together with the SystemC generation framework, we develop a fast design space exploration technique to effectively optimize the design regarding power and latency budgets. Figure 4.3 shows the key components.

While it is theoretically interesting to consider the entire design space, the true value of design space exploration comes from identifying points that correspond to the optimal trade-offs. The notion of trade-off between power and latency can be seen by again referencing Fig. 4.2. For any blue point, there exists a red point that either has less latency for an equivalent level of power or uses less power for the same latency cost. To choose a design corresponding to a blue point would therefore waste power or result in unnecessary additional latency. Thus, all red points, which mathematically correspond to the Pareto front, are considered optimal, while all blue points shown are considered non-optimal and should be trimmed from the design space.

In order to trim the design space, a script provided by [20] automatically extracts the Pareto points from the high granularity modeled design space. To ensure that we do not prune away real optimal points due to model accuracy limitation, we add a thickness to the curve to preserve the correctness. This is accomplished by performing a linear stepwise curve fitting to the Pareto points. We then translate this curve vertically and horizontally for a user specified distance. By comparing the power and latency data of all points in our design space with this translated curve, and trimming all points that lie above it, we effectively generate a "thickened" Pareto front to describe the optimal power and latency trade-offs.

All points in the thickness-adjusted Pareto-curve are the optimal solution candidates and are sent back to the SystemC generation flow and the simulation is run. In completing this simulation, we collect the real power and latency values associated with the candidate Pareto points. As we have now eliminated the error associated with our design space model relative to our extracted points, we confidently prune away all points that no longer fit the Pareto frontier without considering thickness. The finalized results consist of a set of optimized points that are ready for the designer's consideration.

To illustrate the viability of our Pareto front extraction process, we again consider the example described in Fig. 4.2. We now further consider the

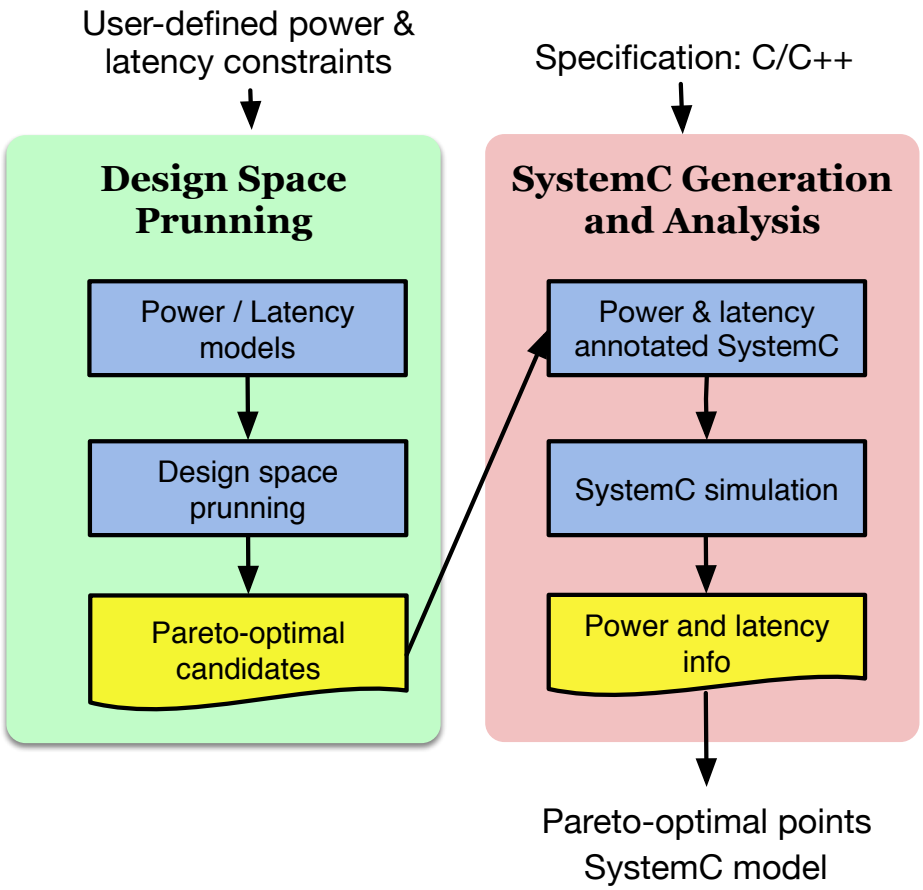

Figure 4.3: Overview of the DSE framework

173 points common to each Pareto frontier. When analyzing these points, the average error in power is approximately 2.28% while the average error in latency is approximately 0.51%. Thus, our accuracy corresponding to the Pareto curve points is generally higher than our overall model accuracy, indicating that our model effectively captures trends along the Pareto frontier.

CHAPTER 5

EXPERIMENTS

5.1 Experimental Setup

All initial power and latency characterization is accomplished utilizing industry-standard tools, whose names omitted due to industry policies. All experiments are implemented using 45-nm standard cell library for computation blocks, and 45-nm memory compiler for memory blocks. All experiments target a frequency of 1GHz. Benchmarks are synthesized using fixed-point arithmetic.

Table 5.1: Benchmark description

Benchmark	Description	N	Dim
GEMM	Matrix multiplication	1024	6
AtAx	Matrix transpose & vector multiplication	1024	4
GEMVER	Matrix vector products and addition	1024	4
Jacobi-2d	2-D Jacobi stencil computation	1024	5
Correlation	Correlation matrix computation	1024	6
Covariance	Covariance computation	1024	6
Sobel Filter	Sobel operator edge detection algorithm	1024	4

We evaluate our framework using six benchmarks from the PolyBench/C test suite. To achieve diversity, we choose benchmarks from computation kernels with different applications. AtAx, GEMM and GEMVER are the numerical kernels; Jacobi-2d is a stencil algorithm, and is widely used in the image processing field. Correlation and Covariance benchmarks contain computations that are frequently used in data-mining algorithms. In addition to the applications from the PolyBench/C test suite, we also evaluate the Sobel-filter, which is a common image processing algorithm used for edge detection. The benchmarks are described in Table 5.1. Column N refers to the problem size for the input code, and column *Dim* refers to the maximal

loop nesting depth of each benchmark *after tiling*. Taking $AtAx$ as an example, the input code has 2 loops, each containing 1024 iterations, while after tiling it has $Dim = 4$ loops.

Our experiments consist of two parts. We first test the accuracy of our SystemC generation and power/latency characterization framework. Since all tiles are repetitive, we can, without loss of accuracy, decrease the number of the loop iterations (thus reduce the replicated tile numbers) by a factor of 8 to shorten the gate-level simulation time. In the second portion of our experiments, we evaluate our framework for accelerator design space exploration (DSE).

5.2 Latency and Power Modeling Results

In order to evaluate the accuracy of our SystemC model, we compare the estimated latency and power with a golden model. The golden model is the SystemC code of the entire design generated by our SystemC generator. We first input our SystemC code through HLS tool which is followed by a logic synthesis process. We then run a gate-level simulation to generate the power and latency results. The iteration tile size is set to 32 in all benchmarks for this study, but our characterization flow works for any reasonable tile sizes.

The results are shown in Table 5.2. We verify our results in two stages. First, in order to verify the accuracy of our one-tile based estimation method, we run the framework iteration once with a fixed input vector and compare the results with the golden model. The results of this comparison for latency are shown in columns 2-4; results for power are in columns 5-7. Second, in order to verify the accuracy of our model relative to different switching activities, we randomly generate 20 input vector sets with different switching activities ranging from 0.1 to 0.95 with a step size 0.05, to cover the possible switching activity range, and each set includes 10000 input vectors. The values are used as input to the framework as well as to the golden model. We then calculate the harmonic mean of the 18 error rates with the results shown in columns 8-9.

From the results we see that the latency estimation is highly accurate (within 1%). The accuracy is achieved primarily for two reasons. First, all the benchmarks are regular, affine programs with predictable loop bound-

Table 5.2: Latency and power modeling results

Benchmark	Latency (ns)			Power (w)			SystemC modeling error rate (%)	
	Golden	Estimated	Err_rate (%)	Golden	Estimated	Err_rate (%)	Latency	Power
GEMM	5086744	5087336	0.012	0.0736	0.072	2.17	0.02	2.29
AtAx	55287	55726	0.43	0.046	0.043	6.53	0.0439	4.98
GEMVER	63742	63934	0.30	0.069	0.065	5.80	0.38	5.68
Jacobi-2d	182338	182998	0.36	0.0803	0.0843	4.98	0.38	4.55
Correlation	8435027	8476829	0.50	0.12	0.116	3.33	0.68	4.36
Covariance	8349287	8375059	0.31	0.1119	0.1175	5.00	0.42	6.01
Sobel	162060	163376	0.81	0.0821	0.0874	6.46	0.94	7.32
Average	-	-	0.51	-	-	4.96	0.57	5.04

aries. Second, our framework ensures both the parallel execution among tiles and the sequential execution between loops. Thus, the latency is highly predictable at the SystemC-level.

However, relative to latency, power estimation is significantly more challenging due to complex dependencies relating to hardware specific implementation. Nevertheless, we obtain an average error rate of 5.04%. This accuracy is achieved for several reasons. After the polyhedral transformation and SystemC generation stages, the design consists primarily of tiles with identical shapes and loop bodies (except some partial tiles at the loop boundaries). Additionally, the replicated tiles are independent of each other and are executed in parallel, which are implemented by resource duplication. By recognizing these properties of our hardware implementation, we are able to maintain high accuracy in using our tile-based power model.

5.3 Details of Design Space Exploration

We now use our framework for design space exploration. It explores design candidates with different tile sizes and parallelism degrees. For each benchmark, a Perl script provides a wrapper around the SystemC generation framework. The loop dimensions and a vector describing the range of iteration tile sizes associated with each loop are given as the input. We use the memory compiler to generate different memory IPs, with sizes from 64B to 16 KB.

Following our analytic models, we iterate over the design space with a log_2 sampling density and acquire power and latency values for each sample vector. Using these results we construct power and latency models for each benchmark which can be seen in Fig. 5.1, and due to the space limitation, we can only display details of four benchmarks. We have chosen representative benchmarks from different application suits. In all figures, we have highlighted the Pareto frontier in red. In all experiments, we compensate for possible model error by including in our Pareto frontier all points whose corresponding latency and power values deviate up to 10μs and 10μW respectively from the original Pareto curve. We then run SystemC simulation with all the points in the thickened curve, and prune away the inferior points to generate the final Pareto curve. By comparing the measured power and la-

tency of these points with the results generated from our model, we calculate the accuracy for each benchmark.

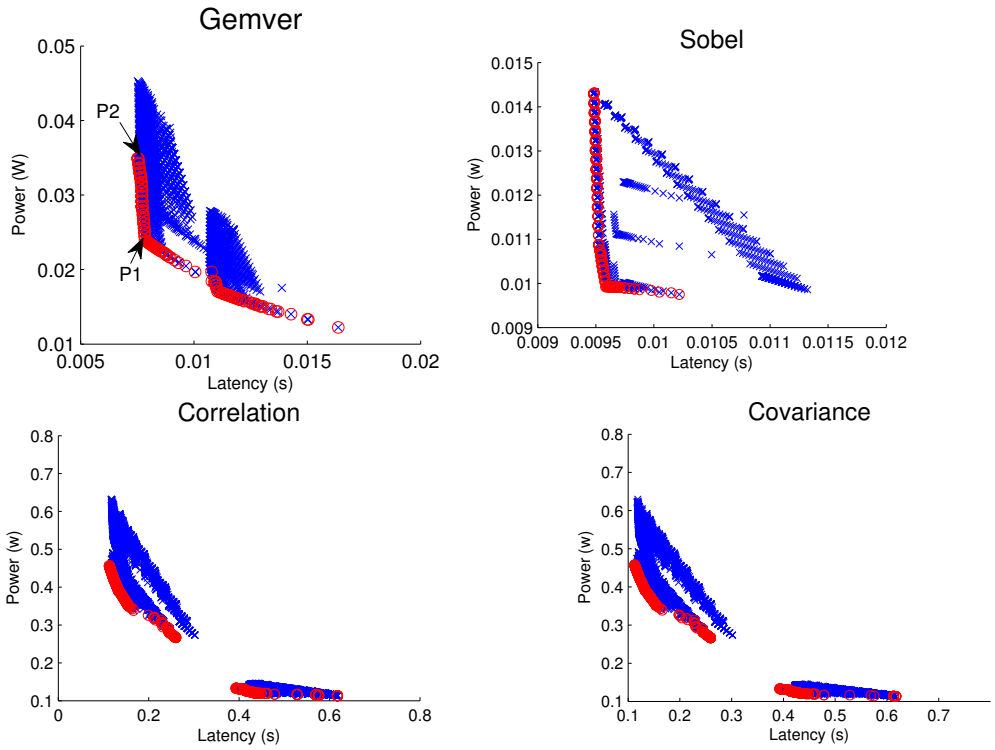

Figure 5.1: Modeled design space and Pareto points

The results are shown in Table 5.3. Column *Accuracy* lists the respective error rates of power and latency. We observe that the average error for the power model is 5.30%, and for the latency model, the average error is 3.24%. Column *# simulation points* shows the total number of sample points that are run through our SystemC generation and simulation process. We subdivide this number into two parts: the sampling points for model generation, and the Pareto points for final verification.

In order to illustrate the benefit of utilizing a thickness-adjusted Pareto-curve prior to our secondary simulation run, we consider the points that would have been eliminated without thickness consideration. For each benchmark, we identify all points added due to the effect of curve thickening and determine if any of such points are contained within the final set of optimized points. Any such points represent true Pareto points present in the final optimized set that, without thickness consideration, would have been prematurely pruned due to modeling limitations. Column *Thickness Addition* represents the number of additional points considered due to curve thick-

Table 5.3: Complete results of DSE

Benchmark	Accuracy (%)		# Simulation points		# Total points	Speed-Up	Thickness Addition	True Pareto	Average Runtime per Point (mins)
	Power	Latency	Sample	Pareto					
GEMM	4.83	3.38	250	719	7086244	7313	60	4	11.46
AtAx	2.14	1.01	324	314	58564	92	58	17	0.16
GEMVER	5.34	0.89	100	195	58564	199	52	9	0.39
Jacobi-2d	7.68	0.11	25	263	14641	51	127	3	0.39
Correlation	8.66	4.56	625	642	28344976	22372	125	13	9.19
Covariance	3.25	3.19	625	784	28344976	19877	217	27	9.36
Sobel	5.30	0.89	25	257	14641	52	132	4	0.23
Average	5.31	3.24	-	-	-	2091	-	-	4.45

ening. Column *True Pareto* represents the number of true Pareto points that were identified as a result. We see that in all cases, adding a degree of thickness proved beneficial.

In order to illustrate the DSE speed-up associated with our methodology, we list the number of total points in the design space associated with each experiment in column $\#Totalpoints$. This is computed as the product of the number of all possible iteration tile sizes and possible unroll degrees. For the loop levels that cannot be unrolled due to dependency, we do not consider them as valid points. We estimate the speed-up obtained by our flow compared with the solution by exhaustively traversing the design space, which is listed in column $SpeedUp$. We observe that our flow provides design-space exploration speed-up ranging from 51x to 22372x, with the average value of 2091x. We also list the SystemC simulation time for one point for each benchmark in column *Average runtime per Point*. We observe that for certain benchmarks, the runtime are several minutes, and extensive exploration is not a feasible solution.

From Fig 5.1, we first observe a clear trade-off between power and latency across the design space. For example, if we consider the Pareto points of *Covariance* and compare these points, we can see a 6x difference of latency and 5x difference of power. We also observe the differences in Pareto-curves between different benchmarks. For example, the curves of correlation and covariance are more gradual, while the curves found in *GEMVER* and *Sobel* are distinctly sharper. By further inspection, we observe that for computation-intensive kernels, the Pareto curves have shapes similar to those of Correlation and Covariance, while the communication-intensive kernels possess Pareto curves similar to those of *GEMVER* and *Sobel*.

When we consider communication-intensive kernels such as *GEMVER*, we observe that the points corresponding to the smallest latency have a large variance along the power axis with only minimal variance in latency. Through further analysis of these points, we discover that they exhibit complete unrolling of all non-dependent loops. In such conditions, the latency is dominated by serialized memory communication and we have already achieved the maximum speed-up available by paralleling the computation. Since the runtime performance is bound by serialized memory communication, the decrease in latency is trivial, resulting in the steep curve. Therefore, for these types of designs, the designer can pay the price of a small additional latency

in return for great power savings. For example, in *GEMVER*, the design point P1 is 1.7X more power-efficient than the design point P2 with only 4% longer latency, as shown in Fig 5.1. These observations confirm that our design space exploration flow does provide significant insight into the behavior of applications, hence effectively guiding the design choices.

CHAPTER 6

CONCLUSION

In this thesis, we proposed (1) a new automatic SystemC-level modeling and synthesis framework, offering fast and accurate power and performance estimation at the early design stage, made possible by exploiting the control and data flow regularities in affine programs; (2) accurate analytical models providing power and latency information for all points in the design space, which effectively optimize the accelerator design decisions; (3) a fast design space exploration to generate accurate power and latency Pareto curves to guide effective low-power design. Our proposed framework combines SoC modeling and design space exploration, as well as enabling the HLS implementation. It has the potential to significantly improve design productivity and quality for highly power-efficient accelerator designs, which are essential for modern low-power SoC chips. Our future work will include carrying out accurate and fast system-level modeling and design space exploration for the entire SoC with an automated software/hardware co-design engine.

REFERENCES

[1] J. Bandler, A. Mohamed, and M. Bakr, "Tlm-based modeling and design exploiting space mapping," *Microwave Theory and Techniques*, vol. 53, no. 9, Sept 2005.

[2] I.-Y. Chuang, T.-Y. Fan, C.-H. Lin, C.-N. Liu, and J.-C. Yeh, "HW/SW co-design for multi-core system on ESL virtual platform," in *VLSI-DAT'11*, 2011.

[3] S. Ahuja, "High level power estimation and reduction techniques for power aware hardware design," Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2010.

[4] P. Feautrier, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in *PLDI'08*, 2008.

[6] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *FPGA'13*, 2013.

[7] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *CODES+ISSS'03*, 2003.

[8] C. Xi, L. JianHua, Z. ZuCheng, and S. YaoHui, "Modeling SystemC design in UML and automatic code generation," in *Proc. ASP-DAC'05*, 2005.

[9] A. Bogliolo, L. Benini, and G. De Micheli, "Regression-based RTL Power Modeling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 3, pp. 337–372, July 2000.

[10] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: a Pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ISCA'14*. IEEE, 2014.

[11] D. Greaves and M. Yasin, "Tlm power3: Power estimation methodology for systemc tlm 2.0," in *FDL'12*, 2012.

[12] G. Vece and M. Conti, "Power estimation in embedded systems within a SystemC-based design context: The PKtool environment," in *Intelligent solutions in Embedded Systems*, June 2009, pp. 179–184.

[13] B. Talwar and B. Amrutur, "A System-C based microarchitectural exploration framework for latency, power and performance trade-offs of on-chip Interconnection Networks," in *NoCArc'08*, 2008.

[14] S. Chakravarty, Z. Zhao, and A. Gerstlauer, "Automated, retargetable back-annotation for host compiled performance and power modeling," in *CODES+ISSS'13*, 2013.

[15] 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 *DATE'09*, 2009.

[16] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO'09*.

[17] D. Chen, J. Cong, Y. Fan, and Z. Zhang, "High-level power estimation and low-power design space exploration for FPGAs," in *ASP-DAC'07*, 2007.

[18] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *FPGA'14*, 2014, pp. 199–208.

[19] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Counting integer points in parametric polytopes using Barvinok's rational functions," *Algorithmica*, vol. 48, no. 1, June 2007.

[20] http://www.mathworks.com/matlabcentral/fileexchange/17251-pareto-front.