# Customized Polyhedral Compilation for Low-Power High-Level SoC Synthesis

## Shonan Seminar 134

Deming Chen[1], **Louis-Noël Pouchet[2]**,
Wei Zuo[1], Warren Kemmerer[1], Jong Bin Lim[1], Te Mu[2]

1: University of Illinois Urbana-Champaign
2: Colorado State University

# Disclaimer

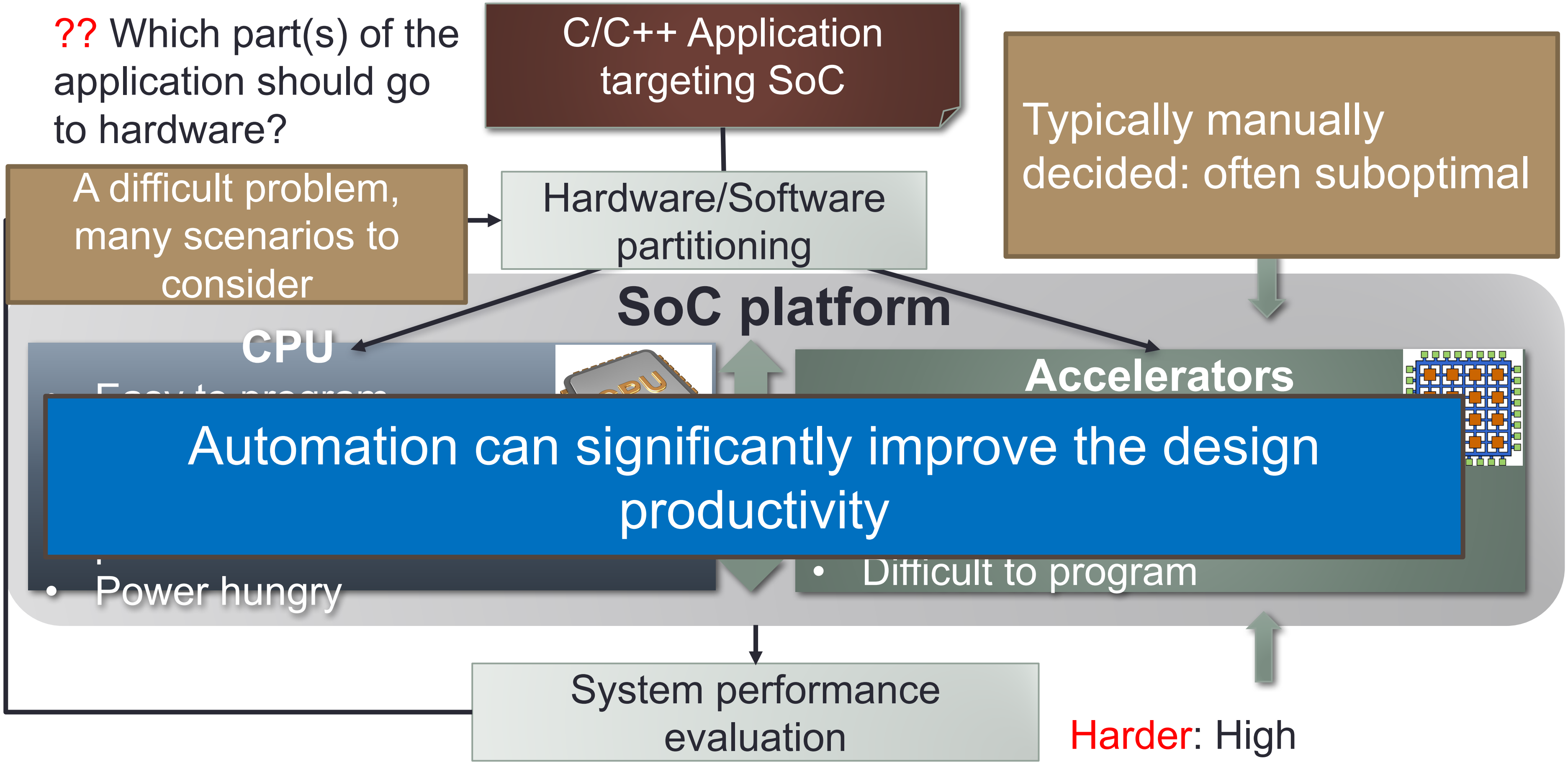
Contributors to this work include Wei Zuo, Warren Kemmerer, Jong Bin Lim, Louis-Noel Pouchet, Andrey Ayupov, Taemin Kim, Kyungtae Han, and Deming Chen.

This work was supported in part by an Intel University Research Office grant.The PolyOpt and PoCC compilers are also supported in part by the U.S. National Science Foundation and the U.S. Department of Energy.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the Department of Energy, nor Intel.

# SoC Modeling and Hardware/Software Co-design

?? Which part(s) of the application should go to hardware?


C/C++ Application targeting SoC
A difficult problem, many scenarios to consider
Hardware/Software partitioning
Typically manually decided: often suboptimal
SoC platform
CPU
• Easy to program
• Power hungry
Accelerators
• Difficult to program
System performance evaluation

Automation can significantly improve the design productivity

Harder: High communication cost, complicated dependencies …

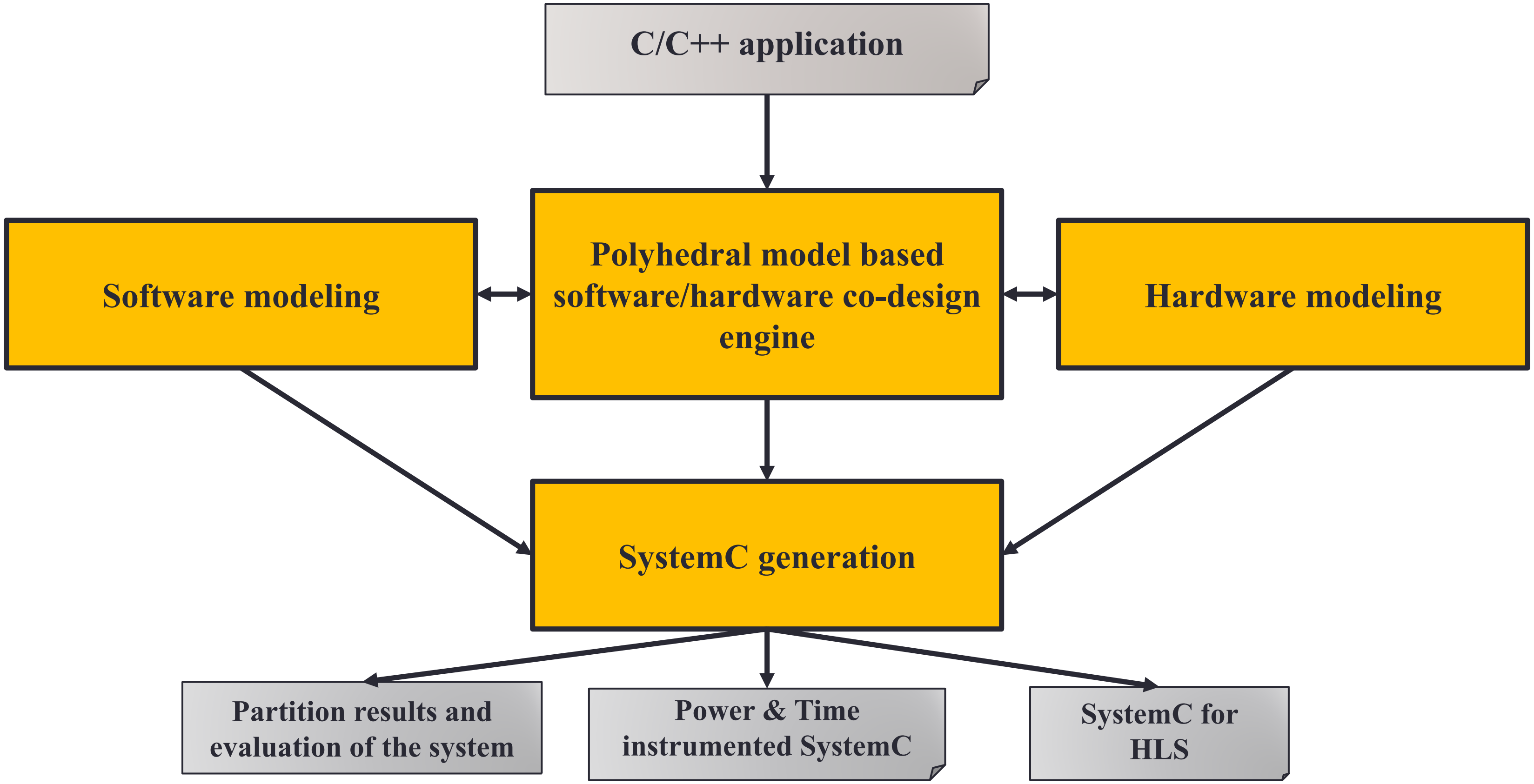
# Some Key Challenges

- How to accurately model system performance and power early on in the design process?
- How to significantly improve design productivity?
  - High-level modeling and synthesis
  - Automatic transformations/optimizations for different hardware targets
- How to explore the large hardware/software co-design space and automate hardware/software partitioning?
- Coarse-grain knobs
  - Which / how many processors?
  - Partitioning among CPU, DSP, accelerators?
  - Accelerators for which functions?
  - Hardwired vs. programmable accelerators?
  - Re-use IPs?
- Fine-grain knobs
  - Voltage, clock speed, bus bandwidth, communication topology, memory hierarchy, …

# Our Approach in a Nutshell

- Build a new SoC synthesis environment
  - Take C/C++ as input specification of the application
  - Go through automatic transformations and model-guided software/hardware partitioning
  - Output SystemC code for modeling the entire SoC
- This flow has the following unique features
  - A customized polyhedral compilation environment to extract program features, implement target-specific optimizations, and generate code
  - Use modular design and HLS to evaluate RTL performance and power details quickly
  - Two types of SystemC outputs
    - 1st: assisting the system engineers to achieve more accurate system-level performance and power evaluations for the SoC
    - 2nd: fully synthesizable, enabling HLS to generate RTL automatically

# Overview of Workflow


C/C++ application
Software modeling
Polyhedral model based software/hardware co-design engine
Hardware modeling
SystemC generation
Partition results and evaluation of the system
Power & Time instrumented SystemC
SystemC for HLS

# Automation of Hardware/Software Co-Design

- A
  - ch code region for acceleration

How to efficiently model and explore component designs and also support different platforms?

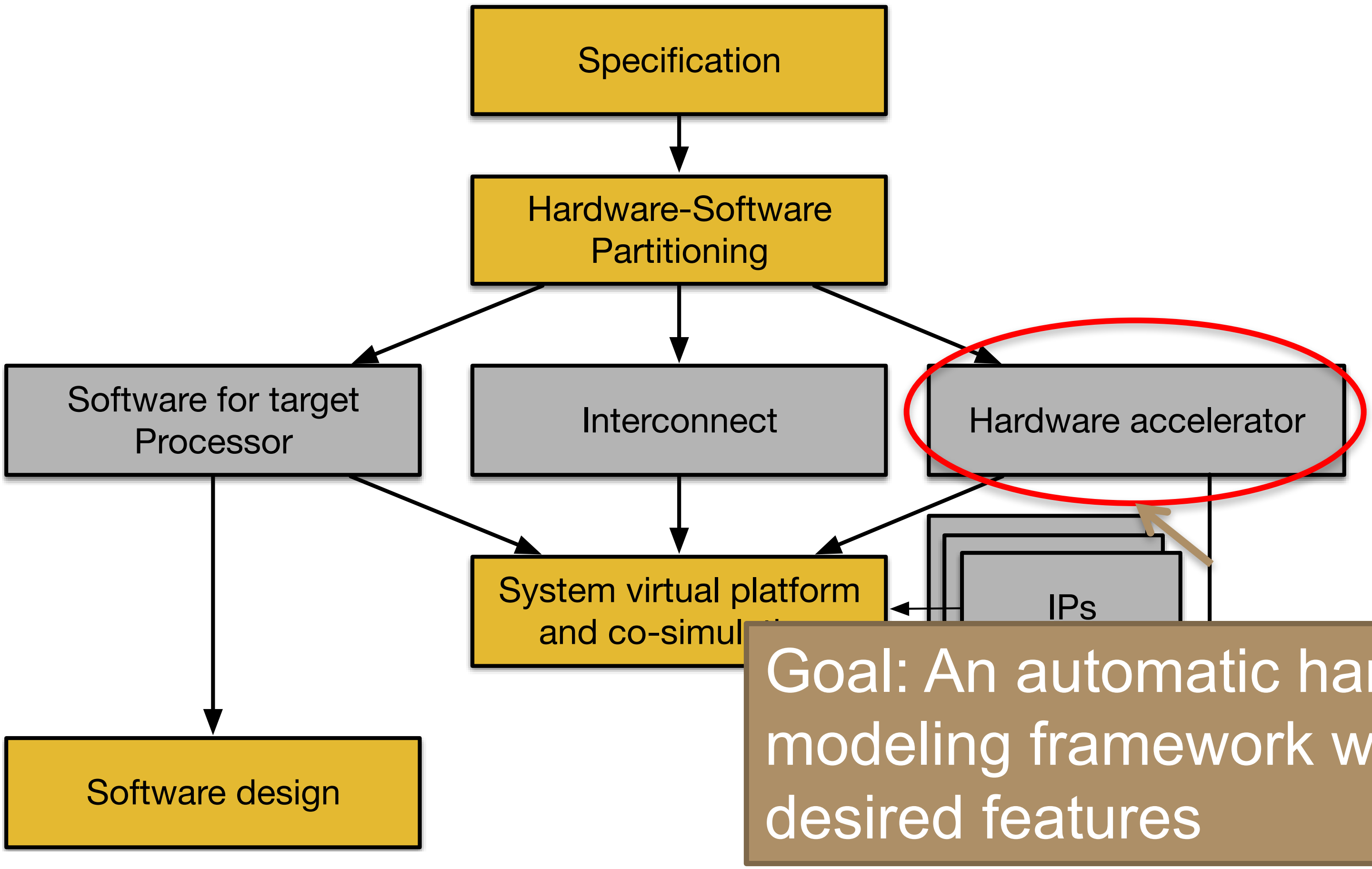
How to accurately model the whole system?

- Enabling to design globally optimal partitioning solutions under specific area

How to effectively and efficiently explore the huge system-level design space?

- Our contribution: an automated software/hardware partitioning framework by tackling these three challenges
  - High-level accurate hardware and software modeling
  - A task graph generation algorithm to capture key features of the system (e.g., parallelism, communication cost, resource sharing)
  - A randomized ILP algorithm to explore the design space efficiently and effectively

# SoC Design Framework


Specification
Hardware-Software Partitioning
Software for target Processor
Interconnect
Hardware accelerator
IPs
System virtual platform and co-simul
Software design

Goal: An automatic hardware modeling framework with desired features

# Hardware Modeling

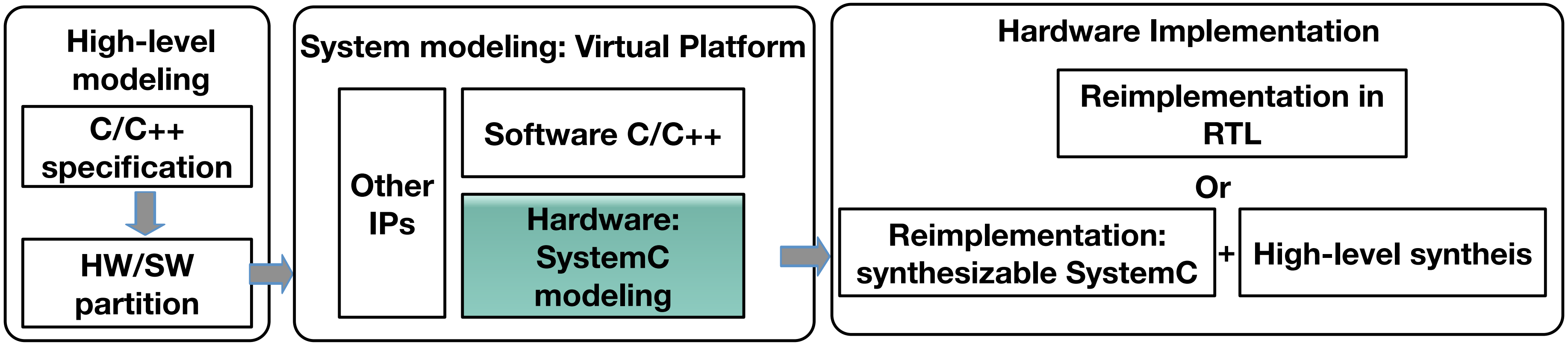
- **How to accurately model performance and power early on?**
  - Essential to enable rapid prototyping of SoC devices

    No gate-level/physical information available ☹

# Hardware Modeling

- **How to accurately model performance and power early on?**
  - Essential to enable rapid prototyping of SoC devices

    No gate-level/physical information available ☹
- **How to automate system design process?**
  - Typical design flow:

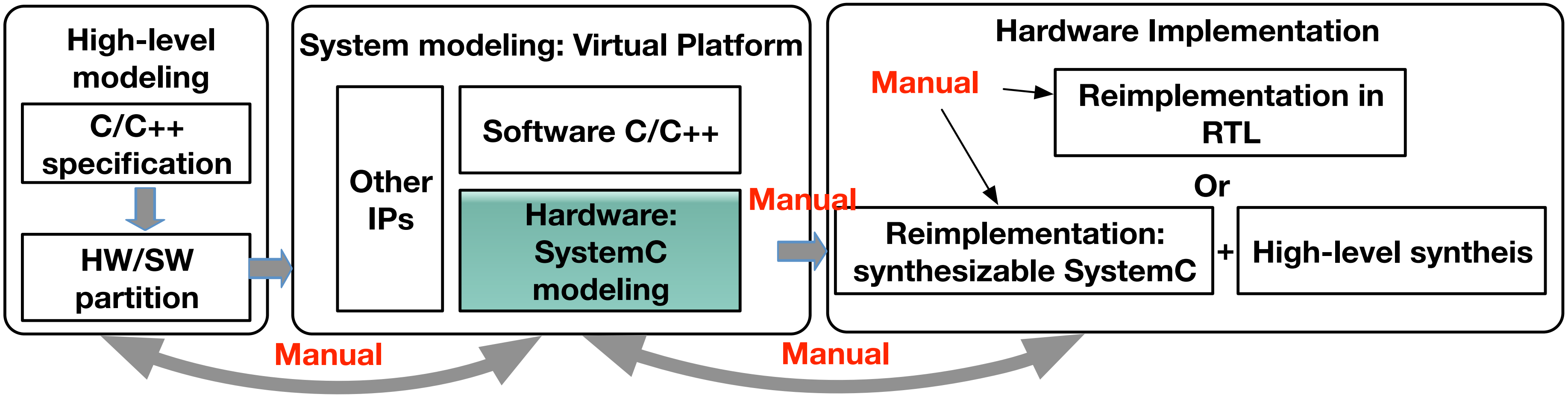


# Hardware Modeling

- **How to accurately model performance and power early on?**
  - Essential to enable rapid prototyping of SoC devices

No gate-level/physical information available ☹

- **How to automate system design process?**
  - Typical design flow:



Slow, manual process ☹

# Hardware Modeling

- **How to accurately model performance and power early on?**
  - Essential to enable rapid prototyping of SoC devices

No physical information available ☹

- **How to automate system design process?**

Slow, manual process ☹

- **How to explore the large design space?**
  - Architecture selection flexibility to achieve different design goals

Huge design space, hard to find optimal implementation ☹

**Our approach directly tackles these three problems ! ☺**

# Our Solution: SystemC Generation, Modeling and DSE

- **How to accurately model performance and power early on?**
  - Essential to enable rapid prototyping of SoC devices

Polyhedral-based power & latency characterization and estimation

- **How to automate system design process?**

Automatic SystemC generation and analytical power & latency modeling

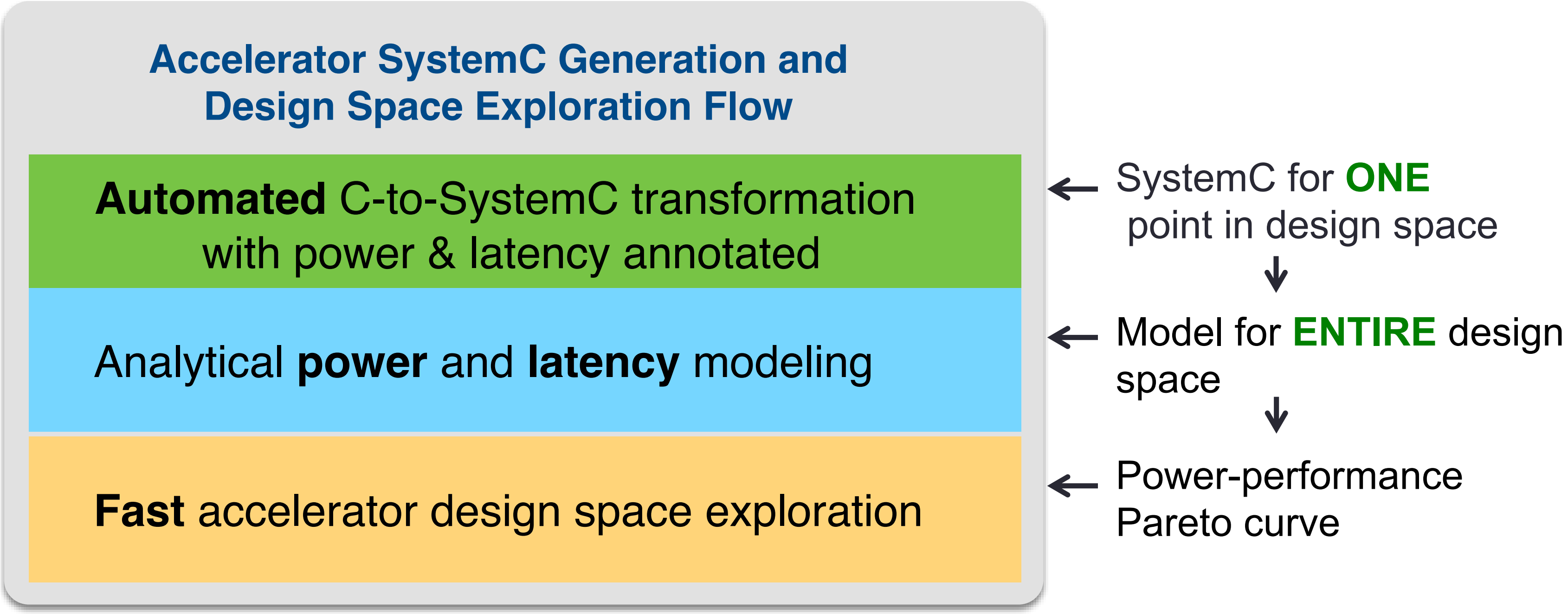
- **How to explore the large design space?**
  - Architecture selection flexibility to achieve different design goal

Fast accelerator design space exploration

**Our approach directly tackles these three problems ! ☺**

**Zuo et al., A Polyhedral-based SystemC Modeling and Generation Framework for Effective Low-power Design Space Exploration, ICCAD'15**

# Our Approaches and Contributions



**Accelerator SystemC Generation and Design Space Exploration Flow**

**Automated** C-to-SystemC transformation with power & latency annotated

Analytical **power** and **latency** modeling

**Fast** accelerator design space exploration

← SystemC for **ONE** point in design space

↓

← Model for **ENTIRE** design space

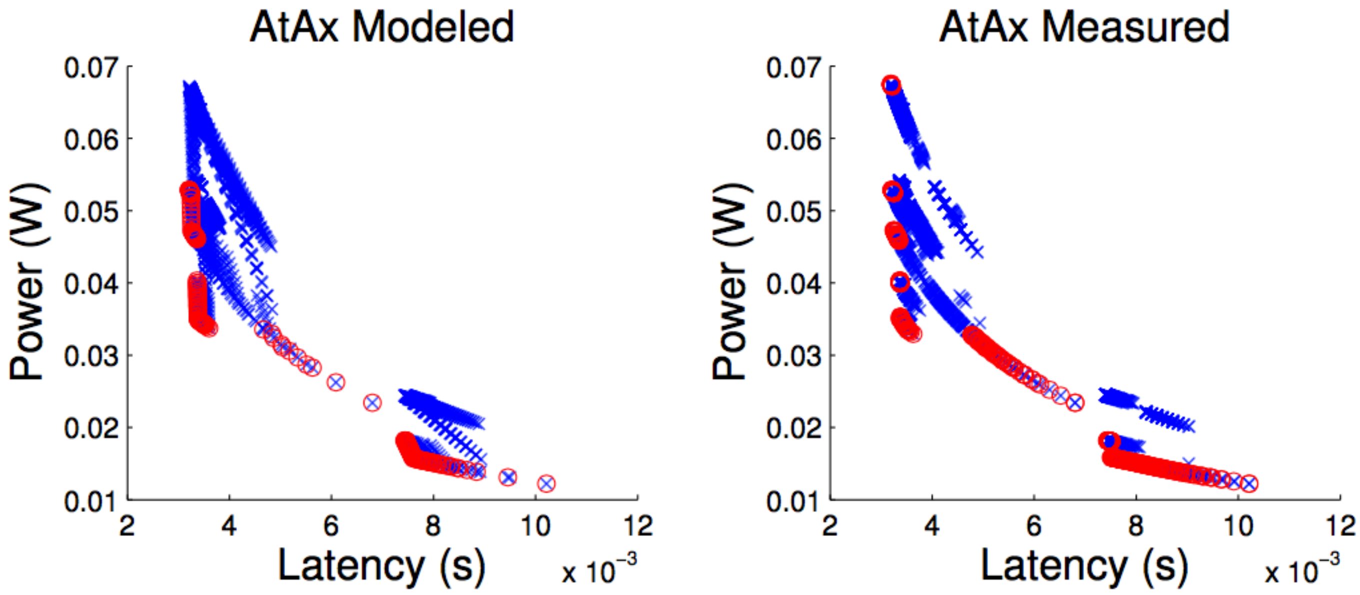
↓

← Power-performance Pareto curve

- Stages help each other for a final holistic solution
  - Each previous stage enables the effective operations of the next stage
  - Eventually achieve the high-performance & lower-power design solutions and tradeoffs

# An Example

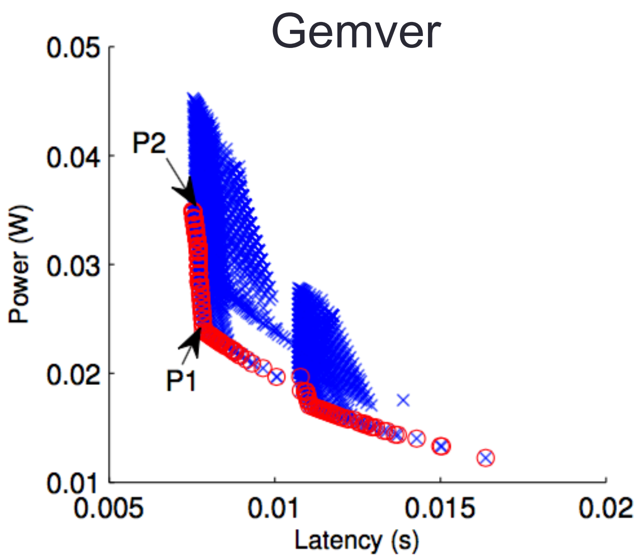
- Blue dots form the design space
- Red dots are the frontiers

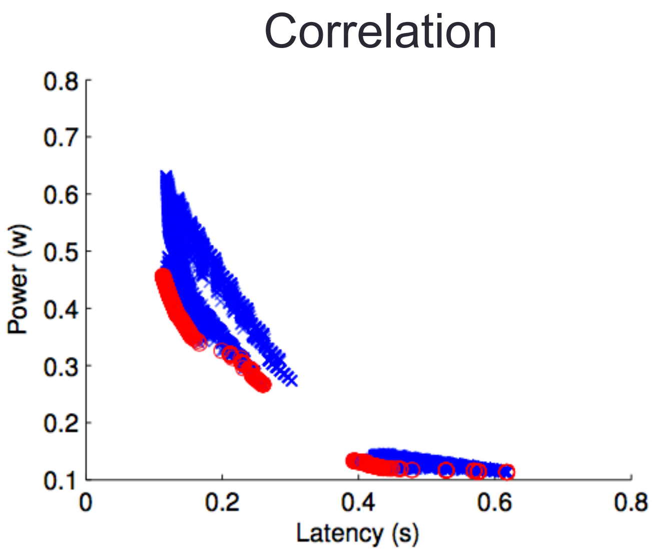


**Error Rate: Power: 4.1%; Latency: 3.28%**

# Analysis of the DSE Results

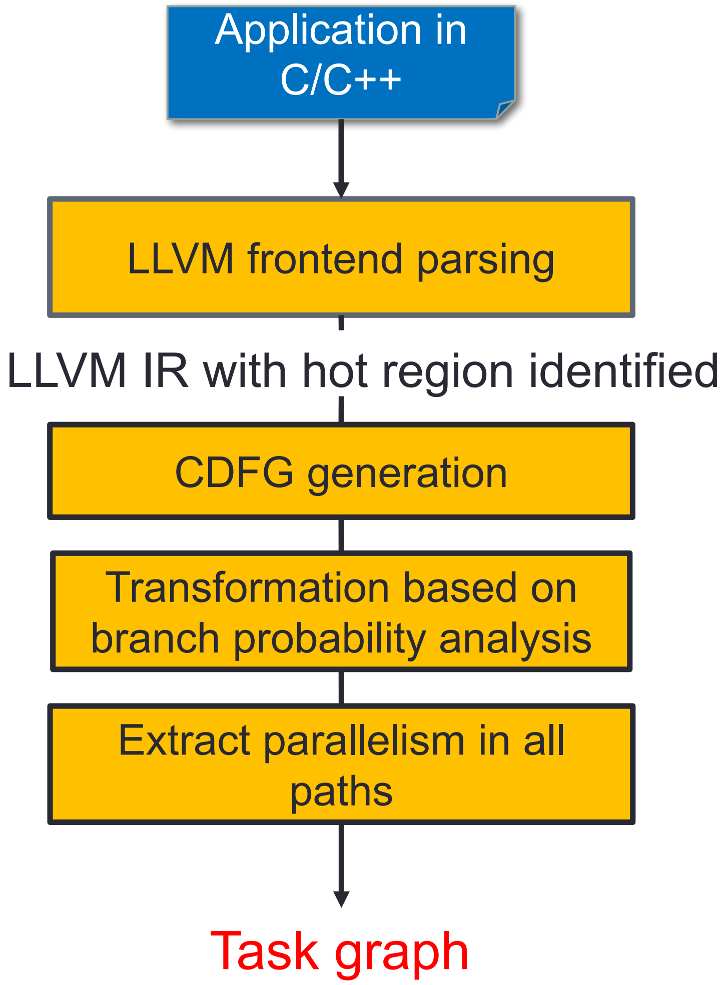
- Communication dominated design (Gemver)
  - Increase parallel computations causes minor latency decrease
  - Optimization opportunity:
    - P1 vs. P2: 1.7 x less power, 4% longer latency
- Computation dominated design (Correlation)
  - Effective power-latency trade-off


Gemver

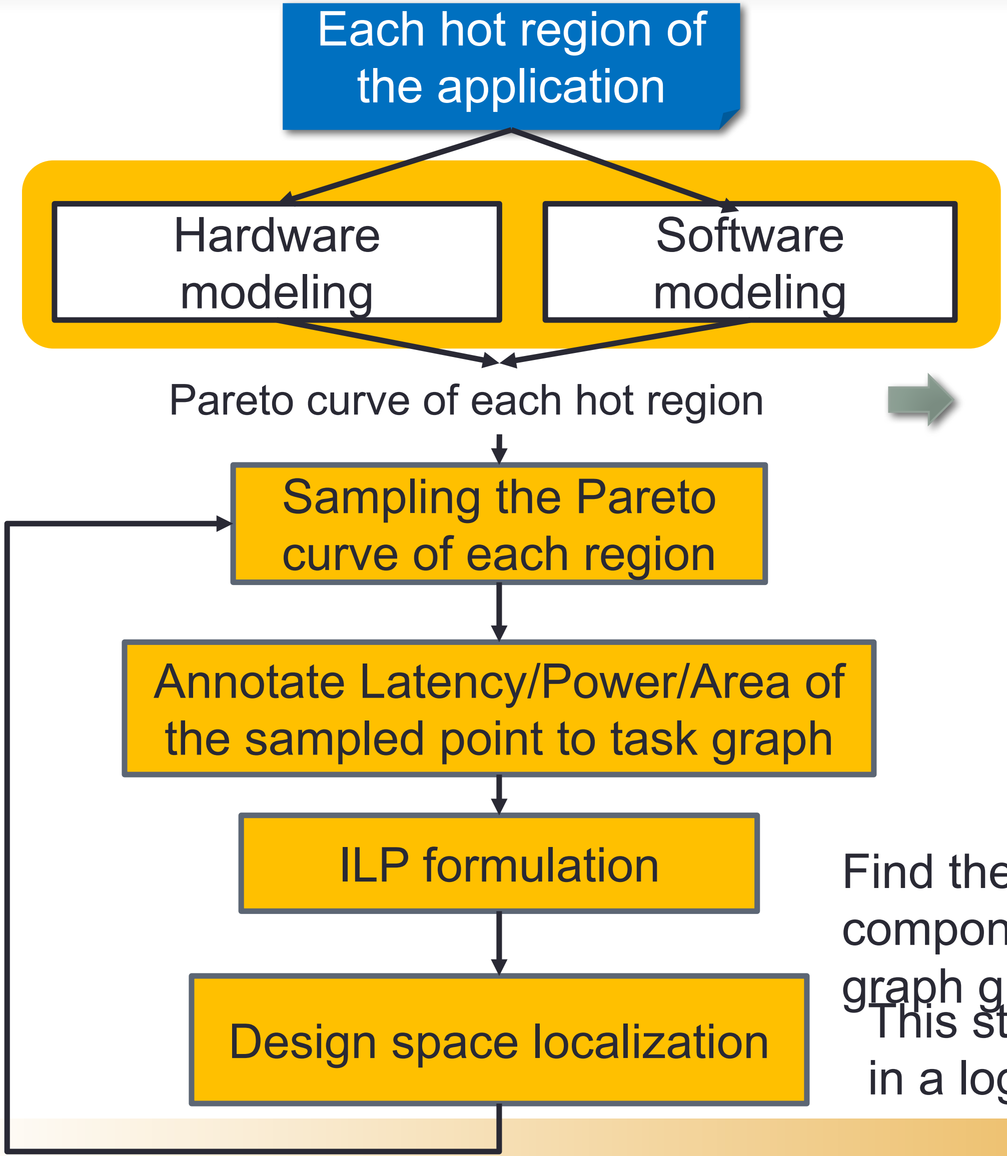

Correlation

# Now: the System Co-design!

- Build on top of LLVM infrastructure and leverage the compiler technique
- Factor in control flow and data flow dependency, resource reuse possibilities, control and data dependencies
- Extract and expose different parallelisms as different branches
  - Dramatically reduce the complexity for ILP
- Output the task graph for efficient ILP formulation


Application in C/C++
LLVM frontend parsing
LLVM IR with hot region identified
CDFG generation
Transformation based on branch probability analysis
Extract parallelism in all paths
Task graph

# Explore the Design Space: Characterization and Randomized ILP



Each hot region of the application

Hardware modeling

Software modeling

Pareto curve of each hot region

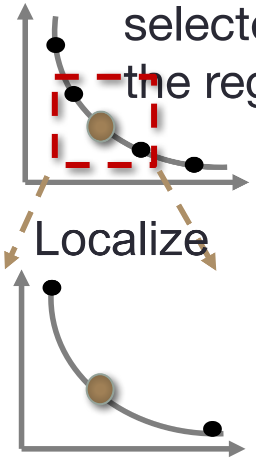
Sampling the Pareto curve of each region

Annotate Latency/Power/Area of the sampled point to task graph

ILP formulation

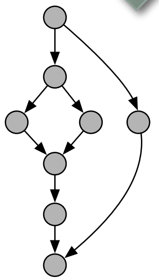
Design space localization

Leverages our hardware and software modeling framework



Assume brown dot is the selected implementation of the region

N sample points and associated latency, power and area



Find the implementation of each component for shortest latency of the task graph given power and area constraints

This step shrinks the design space in a logarithmic manner

# Experiments

- Platform: Xilinx Zynq-7000 XC7Z045 SoC
  - CPU: ARM Cortex-A9 (800 MHz)
  - FPGA: 100HMz
- Benchmarks:
  - Covariance, Correlation, 3-mm, RSA, AlexNet
- Experimental setup
  - **Efficiency**: Compare the speedup with Simulated Annealing (SA)
  - **Optimality**: Compare the partitioning result with brute-force search
  - **Accuracy**: Compare latency, power and resource with FPGA synthesis results
  - **Speedup over CPU**: Compare the latency with the single-thread Intel Xeon(R) CPU E3-1240

# Experiments I

- **Efficiency**: Compare the speedup with Simulated Annealing (SA)

| Benchmarks | Randomized ILP Runtime (S) | SA Runtime(s) | Speedup to SA |
|---|---|---|---|
| Correlation | 41.87 | 2667.59 | 63x |
| Covariance | 45.36 | 2375.66 | 52x |
| 3-mm | 40.10 | 1140.55 | 27x |
| RSA | 42.43 | 4689.72 | 110x |
| AlexNet | 3284.03 | 48620.93 | 14x |
| Average | -- | | **53x** |

- Our tool on average achieves 56.95× speedup over SA
- For covariance, both algorithms found the same results, while for the other benchmarks, randomized ILP outperforms SA

# Experiments II

- **Optimality**: Compare the partitioning result with brute-force
  - Can only run the brute-force search for the first 3 benchmarks
  - Randomized ILP can find the optimal results in the three cases
- **Accuracy**: Compare with FPGA synthesis results

| Benchmarks | FPGA Implementation | | |
|---|---|---|---|
| | Latency Error (%) | Power Error (%) | Resource Error (%) |
| Correlation | 4.88 | 6.30 | 0.54 |
| Covariance | 7.00 | 6.52 | 0.71 |
| 3-mm | 6.66 | 4.69 | 2.09 |
| RSA | 8.17 | 10.71 | 1.72 |
| AlexNet | 9.09 | 6.72 | 5.06 |
| Average | **7.16** | **6.98** | **2.02** |

# Experiments III

- **Speedup over CPU:** Compare the latency with the single-thread (default input code, non optimized) Intel Xeon(R) CPU E3-1240

| Benchmark | Randomized ILP | | CPU | Speedup |
|---|---|---|---|---|
| | Latency (s) | Power (W) | Latency (s) | |
| Correlation | 0.0818 | 2.36 | 3.797 | 46.4 |
| Covariance | 0.0834 | 2.54 | 4.111 | 49.3 |
| 3-mm | 0.0644 | 2.84 | 5.237 | 81.3 |
| RSA | 3.69 | 2.25 | 31.447 | 8.5 |
| AlexNet | 0.0458 | 19.5 | 10.90 | 238.0 |
| Average | | | | **84.7** |

# Polyhedral Tools

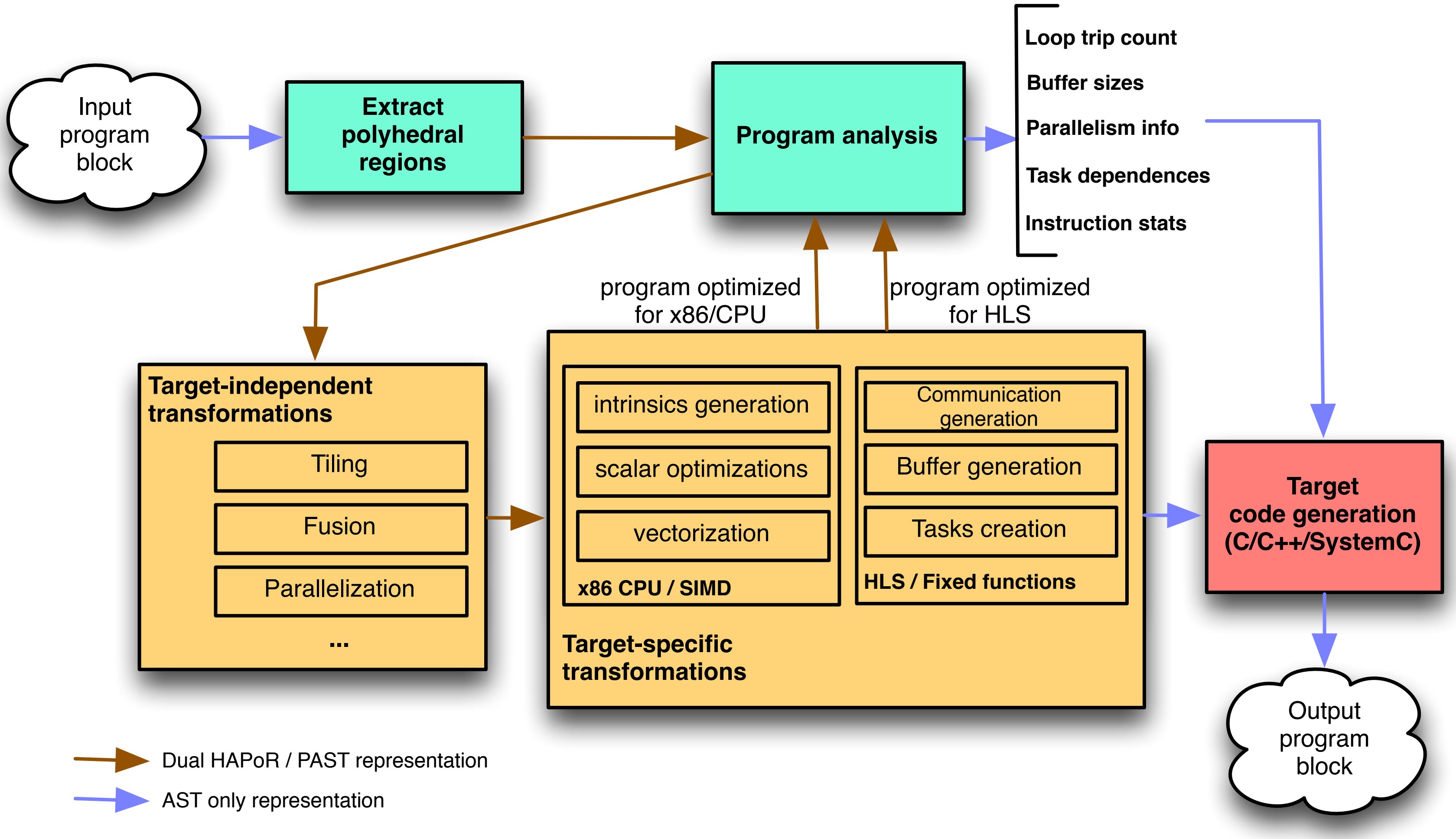
- Polyhedral compilation tools, hierarchical compilation
  - **Source code transformations toolbox (PolyOpt & PoCC)**
  - Target-independent optimizations
  - Target-specific optimizations
  - **Information for hardware mapping (loop trip count, data footprint, etc.)**

⇒**Objective: Provide program transformation environment, with automatic optimizations for data locality / parallelism**

- Program transformations for improved energy efficiency
  - Automatic program transformations to reduce data movements, improve parallelism
  - **For multi-core CPUs: tune the transfo. for a given frequency/core setup**
  - Compile-time approximation of cache misses and parallelism metrics

⇒**Objective: Demonstrate frequency-aware program transformation**

# Overview of Polyhedral Compilation Tools



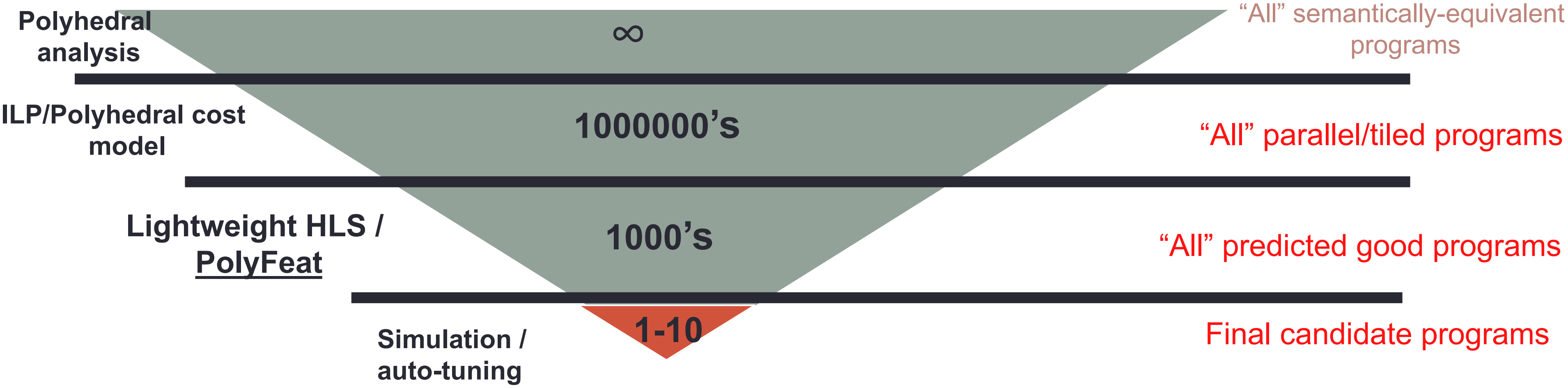
→ Dual HAPoR / PAST representation

→ AST only representation

# Transformation Selection

**Claim: smart H/S partitioning requires the code to be optimized for each possible target (CPU/GPU/DSP/IP/…)**

- Finding the best optimization needs non-linear cost models
- Our approach: build search spaces of candidate implementations, and progressively prune them



Polyhedral analysis
∞
"All" semantically-equivalent programs
ILP/Polyhedral cost model
1000000's
"All" parallel/tiled programs
Lightweight HLS / PolyFeat
1000's
"All" predicted good programs
Simulation / auto-tuning
1-10
Final candidate programs

# Transformations for CPU Energy Optimization

- **Central idea: the program transformation to minimize CPU energy may depend on the CPU frequency, <u>need to generate adaptive binaries</u>**
  - If the frequency is low, the code may not issue enough load/store per second to saturate the RAM bandwidth
    - Example: Harris corner detection, from OpenCV 3.0
  - If the frequency is high, the code may not need multi-cores to saturate the RAM bandwidth
- **Research problem: how to select the program transformation achieving best speed or CPU energy <u>for a particular frequency</u>?**
  - Multi-core parallelism may be essential at low frequencies
  - But it may provide little speedup at high frequencies (e.g., bandwidth-bound codes)
  - And what if there is a trade-off multi-core parallelism vs. data locality in the space of possible transformations?
- **How to address this problem at compile-time? We need to:**
  - Model the run-time behavior of the program (e.g., L3 cache misses)
  - Model program features in performance estimators, that consider CPU frequency and RAM bandwidth

# PolyFeat: Overview and Motivation

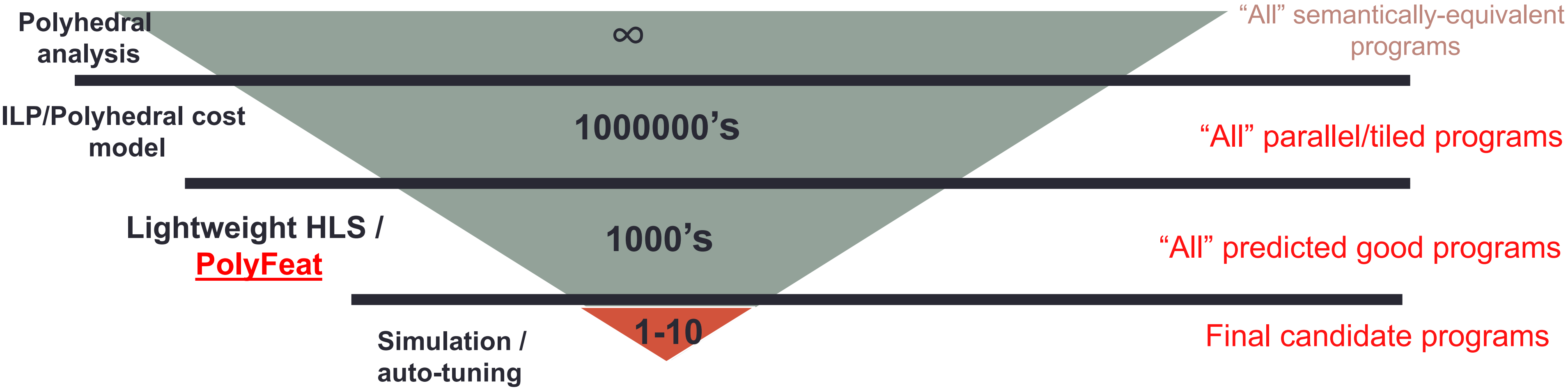
**Develop a purely compile-time approach to quickly evaluate thousands of candidate transformations, to determine quickly the best one(s)**

1. **Generate space of useful/important candidate optimizations**
   - Combine trade-offs between multi-core parallelization, data locality, pre-SIMD, etc.
   - Uses the polyhedral compiler PoCC (part of PolyOpt)
     - Limited to affine programs for the optimization process
     - Explore 1000s of candidates via exploration of loop fusion/distribution, tile size selection, …
2. **For each candidate C code generated, extract metrics**
   - Approximate cache misses (private/shared caches)
   - Count floating point operations (in each OpenMP thread, varying the number of threads)
3. **Build performance estimator, using CPU frequency and RAM bandwidth**
   - Estimate CommCyc: number of cycles spent communicating with RAM
   - Estimate CompCyc: number of cycles spent computing
   - Simple performance estimator: max(CommCyc, CompCyc)

# Where Is PolyFeat in the Food Chain?

**Claim: smart H/S partitioning requires the code to be optimized for each possible target (CPU/GPU/DSP/IP/…)**

- Finding the best optimization needs non-linear cost models
- Our approach: build search spaces of candidate implementations, and progressively prune them


Polyhedral analysis
∞
"All" semantically-equivalent programs
ILP/Polyhedral cost model
1000000's
"All" parallel/tiled programs
Lightweight HLS / PolyFeat
1000's
"All" predicted good programs
Simulation / auto-tuning
1-10
Final candidate programs

# Extracting Features from Source Code

- **Overview of the process:**
  1. Extract polyhedral representation of the program (including OpenMP doall)
  2. Inline parameter values (many of our analyses are not parametric)
  3. Count the number of operations in each loop / region of interest
  4. Compute the data space (in cache lines) accessed by each loop / region
  5. Run various ad-hoc algorithms to estimate cache misses, thread workload, etc.

- **Core features currently extracted:**

| | |
|---|---|
| Number of FLOPs (scalar, vectorizable, and scalar-equivalent) | **EXACT** |
| Data footprint (read/written) | **EXACT** |
| Data cache misses (at each level, inc. shared/private) | **APPROX** |
| OpenMP thread workload | **EXACT** |

# Kernel Categorization

- Run PolyFeat on 60 benchmarks (30 PolyBench x 2)
  - Objective: **categorize** benchmarks, show categorization changes w/ code transfo.

| Benchmarks | seq/par | bw-bound | poor scale | comp-bound |
|---|---|---|---|---|
| polybench-parrallel | 12/18 | 27 | 4 | 1 |
| polybench-poly | 5/25 | 15 | 1 | 10 |

Table V: Summary of Features

| Benchmarks | version | seq | bw-bound | poor scale | comp-bound |
|---|---|---|---|---|---|
| correlation | par | | | ✔ | |
| | poly | | | | ✔ |
| gemm | par | | ✔ | | |
| | poly | | | | ✔ |
| jacobi-2d | par | | ✓ | ✔ | |
| | poly | | ✔ | | |
| seidel-2d | par | ✔ | ✓ | | |
| | poly | | | | ✓ |

Table VI: Benchmark features

# Performance Estimators

- **Computation:**
  - Speedup by doubling the number of cores: number of ops in the largest thread using T threads divided by number of ops in the largest thread using 2T threads
  - Computation cycles: number of ops, assume 1 cycle per flop and 1 cycle per vector flop
- **Communication:**
  - Off-chip communication cycles: Freq * (LLCMiss * LLCLineSize / RAM_bandwidth)
- **Performance estimator:**
  - Max(ComputationCycles, CommunicationCycles)
- **Model for selecting the number of cores:**
  - If Speedup(T, 2T) > PowerIncrease(T, 2T) then use 2T cores
  - PowerIncrease: measured using Intel MKL benchmark (over-estimate)

# Experimental Results [1/2]

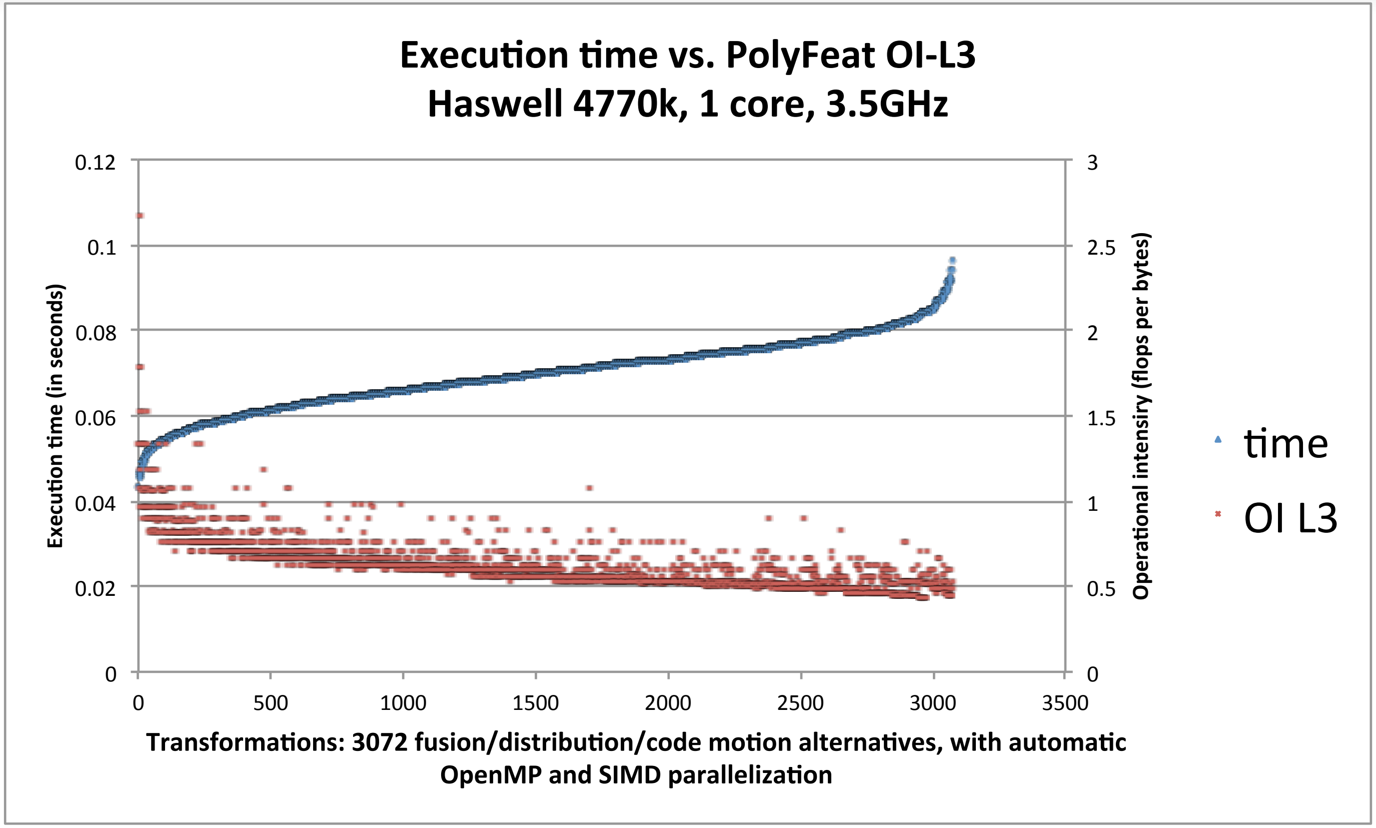


**Figure 1: Harris on a 4k UHD image. Original code from OpenCV 3.0 performs in 0.14s, 3x slower than the best transformation we output.**

# Experimental Results [2/2]

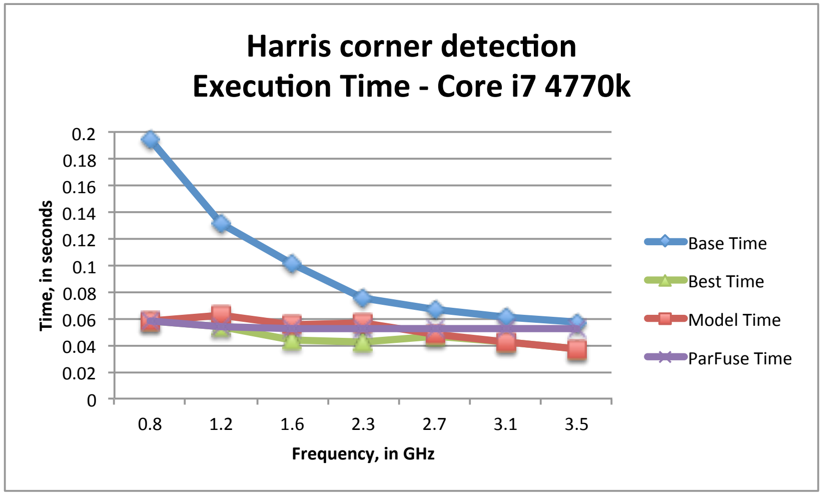


**Figure 2: Execution time comparison, original (base) versus best (in design space of 9216 points) for each frequency, our model, and ParFuse**

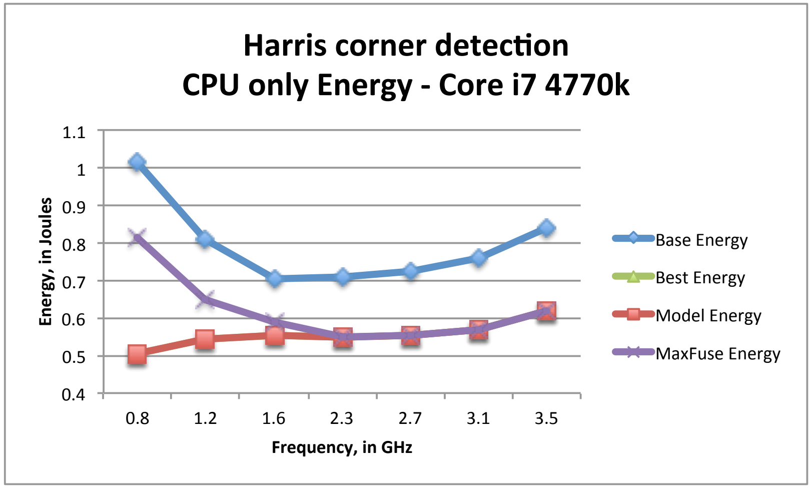


**Figure 3: CPU energy comparison, original (base) versus best (in design space of 9216 points) for each frequency, our model, and MaxFuse**

| | base | MaxFuse | ParFuse | model | best |
|---|---|---|---|---|---|
| T @ 0.8GHz | 0.35s | 0.74s | 0.098s | 0.098s | 0.98s |
| T @ 3.5GHz | 0.15s | 0.18s | 0.094s | 0.097s | 0.094s |
| E @ 0.8GHz | 1.86J | 3.52J | 0.86J | 0.86J | 0.86J |
| E @ 3.5GHz | 2.65J | 5.00J | 2.78J | 1.55J | 1.48J |

**Table 1: Summary for FDTD-2D on Core i7-4770k**

# Conclusions

- Polyhedral/affine programs form only a restricted set of computations, but this set can be effectively analyzed and optimized at compile-time
  - Polyhedral programs include tensor operations (e.g., deep learning), many dense linear algebra algorithms (e.g., most of BLAS), stencil computations (e.g., image processing), etc.
  - Ability to determine latency/power with good precision, using light HLS
- Hardware/software co-design needs quality code optimizations!
  - CPU code optimizations are often neglected by hardware specialists!
  - Polyhedral compilers provide fully automatic transformations for data locality and parallelism (both coarse- and fine-grain)
- Technical merits:
  - Polyhedral scheduling (aka automated loop transformations)
  - Multi-target code generation and optimization (CPU/GPU/FPGA/fixed fun.)
  - Automated data communication generation
  - Compile-time performance and energy modeling
  - Quick design space exploration