Charm: A Language for Closed-form High-level Architecture Modeling

Weilong Cui*, Yongshan Ding†, Deeksha Dangwal*, Adam Holmes†, Joseph McMahan*,
Ali Javadi-Abhari‡, Georgios Tzimpragos*, Frederic T. Chong† and Timothy Sherwood*
*University of California, Santa Barbara
{cuiwl, deeksha, jmcmahan, gtzimpragos, sherwood}@cs.ucsb.edu
†University of Chicago
{yongshan, adholmes}@uchicago.edu, chong@cs.uchicago.edu
‡IBM Research
ali.javadi@ibm.com

***Abstract*—As computer architecture continues to expand beyond software-agnostic microarchitecture to data center organization, reconfigurable logic, heterogeneous systems, application-specific logic, and even radically different technologies such as quantum computing, detailed cycle-level simulation is no longer presupposed. Exploring designs under such complex interacting relationships (e.g., performance, energy, thermal, cost, voltage, frequency, cooling energy, leakage, etc.) calls for a more integrative but higher-level approach. We propose Charm, a domain specific language supporting Closed-form High-level ARchitecture Modeling. Charm enables mathematical representations of mutually dependent architectural relationships to be specified, composed, checked, evaluated and reused. The language is interpreted through a combination of symbolic evaluation (e.g., restructuring) and compiler techniques (e.g., memoization and invariant hoisting), generating executable evaluation functions and optimized analysis procedures. Further supporting reuse, a type system constrains architectural quantities and ensures models operate only in a validated domain. Through two case studies, we demonstrate that Charm allows one to define high-level architecture models concisely, maximize reusability, capture unreasonable assumptions and inputs, and significantly speedup design space exploration.**

***Keywords*-abstraction; modeling; DSL;**

I. Introduction

Computer architecture is evolving into a field asked to cover a tremendous space of designs. From the smallest embedded system to the largest warehouse-scale computing infrastructure, from the most well-characterized CMOS technology node to novel devices at the edge of our understanding, computer architects are expected to be able to speak to the non-orthogonal concerns of energy, cost, leakage, cooling, complexity, area, power, yield, and of course performance of a set of designs. Even radical approaches such as DNA-based computing [1] and quantum architectures [2], [3] are to be considered. While there are a great deal of well considered infrastructures to build around when detailed cycle-level simulation is required, for engineering questions that span multiple interacting constraints or to extreme scales the best approaches are more ad-hoc.

Careful application of detailed simulation can accurately estimate the potential of a specific microarchitecture, but exploration across higher level questions always involves some analytic models. For example, "given some target cooling budget, how much more performance can I get out of an ASIC versus an FGPA for this application given my ASIC will be 2 tech nodes behind the FPGA?" The explosion of domain-targeted computing solutions means that more and more people are being asked to answer these questions accurately and with some understanding of the confidence in those answers. While any Ph.D. in Computer Architecture should be able to answer this question, when you break it down, it requires a combination of a surprisingly complex set of assumptions. How do tech node and performance relate? What is the relationship between energy use and performance? ASIC and FPGA performance? Dynamic and leakage power? Temperature and leakage? Any result computed from these relationships will rely on the specific relationships chosen, on those relationships being accurate in the range of evaluation, on a sufficient number of assumptions being made to produce an answer (either implicitly or explicitly), and finally on that the end result be executable to the degree necessary to explore a set of options (such as for a varying parameter e.g., total cooling budget).

Such analysis today is not supported in any structured form. Typically it exists as a set of equations in an Excel spreadsheet or perhaps as a set of handwritten functions in a scripting language. Unfortunately, this comes with some issues. As simple as sets of mathematical relationships between quantities, the lack of a common engineering basis for these models have kept them from being swiftly and correctly constructed, understood and applied in guiding new system designs. Some models share a set of common relationships but they redefine those symbols and equations often with subtle differences that can be misleading if one is not careful. Some have implicit constraints on one or more architectural quantities which may lead to pitfalls if overlooked. Finally, one has to manually convert these mathematical equations to executable functions in order to evaluate the model and perform the design space exploration, which can be error-prone and inefficient.

To address these issues we design and explore a declarative domain specific language, Charm, to serve as a unified basis for the representation, execution, and optimization of closed-form high-level architecture models. Charm provides a concise and natural abstraction to express architectural relationships and declare analysis goals. By combining symbolic manipulation, constraint solving, and compiler techniques, Charm bridges the gap between mathematical equations and executable, optimized evaluation functions and analysis procedures. The benefit of building and evaluating closed-form high-level architecture models using Charm is threefold:

Abstraction – Charm encapsulates a set of mutually dependent relationships and supports flexible function generation. It enables representation of architecture models in a mathematically consistent way. Depending on which metric the model is trying to evaluate, Charm can generate corresponding functions without requiring the user to re-write the equations. It also modulates high-level architecture models by packing commonly used equations, constraints and assumptions in modules. These architectural "rules of thumb" can then be easily composed, reused and extended in a variety of modelling scenarios.

Type Checking – Charm enables new static and run-time checking capabilities on high-level architecture models by enforcing a type system in such models. One example is that many architecturally meaningful variables have inherent physical bounds that they must satisfy; otherwise, although mathematically viable, the solution is not reasonable from an architectural point of view. With the type system built-in, Charm can dynamically check if all variables are within defined bounds to ensure a meaningful modelling result. The type system also helps prune the design space based on constraints, without which a declarative analysis might end up wasting a huge amount of computing effort in less meaningful sub-spaces.

Optimization – Charm opens up new opportunities for compiler-level optimization when evaluating architecture models. Although high-level architecture models are usually several orders of magnitude faster than detailed simulations, as the model gets complicated or is applied many times to estimate a distribution, it can still take a non-trivial amount of time to naively evaluate the set of equations every iteration. By expressing these complicated models in Charm, we are able to identify common intermediate results to hoist outside of the main design option iteration and/or apply memoization on functions.

Finally, and perhaps most importantly to the community, it promotes collaboration between application designers, computer architects, and hardware engineers because they can share and refine models using the same formal specification and a common set of abstractions.

We release Charm as an open-source tool on github[1] and we provide a wide collection of established architecture models for quick use/reference, including: the dark silicon model [4], a resource overhead model for implementing magic state distillation on surface code [5]–[7], mechanistic cpu models [8], [9], a TCAM power model [10], the LogCA model for accelerators [11], the adder/multiplier models from PyRTL [12], a widely-used CNN model [13], dynamic power and area models for NoC [14], specifications of Xilinx 7-series FPGA [15] and the extended Hill-Marty model [16].

To describe Charm we begin in Section II with a motivating example high-level model to show the problems with ad-hoc modelling in practice. Then we introduce the design of Charm in Section III followed by two case studies demonstrating the application and benefits of building closed-form high-level architectural models with Charm in Section IV. Finally we discuss the related works in Section V and conclude in Section VI.

II. Charm by Example

To understand Charm it is useful to have a running example. In this section, we present an implementation of the model and analysis from a well-cited study of dark silicon scaling [4]. After a brief review of the models, we show the complete code in Charm performing the same analysis of symmetric topology with ITRS technology scaling predictions. As we extend this model to cover more analysis provided in [4], it leads to a discussion of the potential issues with less structured approaches and highlights some of the features of the language that help architects avoid these pitfalls.

A. A Brief Review of the Dark Silicon Model

To forecast the degree to which dark silicon will become prevelent on CMPs under process scaling, Esmaeilzadeh et al. first construct three models: a device model (*DevM*), a core model (*CorM*) and a CMP model (*CmpM*). *DevM* is the technology scaling model relating *tech node* to *frequency scaling factor* and *power scaling factor*. It is a composite model combining a scaling prediction with a simple dynamic power model ($P = \alpha C V_{dd}^2 f$). *CorM* is the model relating *core performance*, *core power*, and *core area*. It is empirically deduced by fitting real processor data points. *CmpM* has two flavors which are essentially very different models: $CmpM_U$ and $CmpM_R$. $CmpM_U$ is an extension of the Hill-Marty CMP model [17] and $CmpM_R$ is a mechanistic model [18].

A composition of the three models is then used to drive the design space exploration. The authors combine *DevM* and *CorM* to look at *CorM* for different *tech node* and combine *DevM*, *CorM*, and *CmpM* to iterate over a collections of topologies, scaling predictions and core configurations. They then plot the scaling curves for the dynamic topology/$CmpM_R$ with both ITRS and a conservative scaling [19].

B. A Complete Charm Code Example

Listing 1 gives the complete code in Charm DSL to run the design space exploration with ITRS predictions on the symmetric topology (we later extend the analysis to other topologies and predictions in Section IV-A). At a high level, we can see that the code is split into three major components:

[1]https://github.com/UCSBarchlab/Charm.git

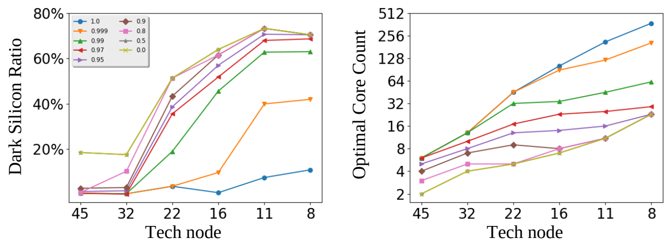

Fig. 1. Upper-bound ITRS scaling with symmetric topology.

type definition (Line 3-8[2]), model specification (Line 11-56) and analysis declaration (Line 59-66).

Specifically, we first define commonly used domains as Charm types on the architectural quantities we care about (Line 3-8). For example, the parallelism parameter in the model has a physical meaning of the proportion of the algorithm that can be parallelized and it naturally falls between $[0, 1]$. We thus define a type *Fraction* to encapsulate this domain constraint. While this is a simple example, more complex constraints are possible.

We then formally specify the three models (*DevM*, *CorM*, *CmpM*) to evaluate (Line 11-56). Taking the *ExtendedPollacksRule* model (Line 34-41) as an example, it declares upfront all the architectural quantities that are involved in the model (e.g., *ref_core_area* which is the core size at the reference technology node), their types (e.g., *ref_core_area* is a real number on the positive domain) and the relationships between the architectural quantities, e.g., $area = 0.0152 perf^2 + 0.0265 perf + 7.4393$ (the constants come directly from the original dark silicon paper [4]).

Once the models are defined, it is straightforward to declare the analysis in Charm (Line 59-66). One simply selects the given models in the study, supplies the inputs and specifies the target metrics to explore. For example, in this case, we select *ITRS*, *ExtendedPollacksRule* and *SymmetricAmdahl* models (Line 59), provide values such as the area (Line 60) and power (Line 61) constraints, and finally tell Charm what quantities we care to explore, in this case *speedup*, *dark_silicon_ratio* and *core_num* (Line 66).

C. Unstructured High-level Architecture Modeling Pitfalls

Building and executing an architectural model with an unstructured approach (e.g., in a spreadsheet or some general purpose scripting language) is clearly possible[3], but the lack of a common abstraction introduces some issues as one tries to scale the analysis. Each additional interacting component is a set of new opportunities to make an uncaught mistake.

The degree to which these mistakes end up in the final model (and the amount of effort required to make sure it is mistake-free) is a function of the degree of composability, reusability, consistency and completeness checking supported

[2]Line numbers in Section II all refer to Listing 1 unless otherwise specified.

[3]With all the potential issues, unstructured methods in architectural modeling may not be as correct as one tends to believe [20], [21].

```
 1 # Type definitions.
 2 # A real number greater than 0.
 3 typedef R+ : Real r
 4     r > 0
 5
 6 # A real number between [0, 1].
 7 typedef Fraction : Real f
 8     0 < = f, f < = 1
 9
10 # Simple Fit of the ITRS Scaling (DevM).
11 define ITRS:
12     ref_tech_node : R+ as ref_t
13     ref_core_performance : R+ as ref_perf
14     ref_core_power : R+ as ref_power
15     ref_core_area : R+ as ref_area
16     tech_node : R+ as t
17     core_performance : R+ as perf
18     core_power : R+ as power
19     core_area : R+ as area
20     perf_scaling_factor : R+ as a
21     power_scaling_factor : R+ as b
22     ref_t = 45
23     perf = a * ref_perf
24     power = b * ref_power
25     area / t**2 = ref_area / ref_t**2
26     a = piecewise((1., t=45),    (1.09, t=32),
27                   (2.38, t=22), (3.21, t=16),
28                   (4.17, t=11), (3.85, t=8))
29     b = piecewise((1., t=45),    (0.66, t=32),
30                   (0.54, t=22), (0.38, t=16),
31                   (0.25, t=11), (0.12, t=8))
32
33 # Pollock's Rule Extended with Power (CorM).
34 define ExtendedPollacksRule:
35     ref_core_performance : R+ as perf
36     ref_core_area : R+ as area
37     ref_core_power : R+ as power
38     area = 0.0152*perf**2 + 0.0265*perf + 7.4393
39     power = 0.0002*perf**3 + 0.0009*perf**2
40         + 0.3859*perf - 0.0301
41     perf < 50
42
43 # Amdahl's Law under Symmetric Multicore (CmpM_U).
44 define SymmetricAmdahl:
45     speedup : R+ as sp
46     core_performance : R+ as perf
47     core_area : R+ as a
48     core_power : R+ as power
49     core_num : R+ as N
50     chip_area : R+ as A
51     thermal_design_power : R+ as TDP
52     fraction_parallelism : Fraction as F
53     dark_silicon_ratio : Fraction as R
54     sp = 1 / ((1 - F) / perf + F / (perf * N))
55     N = min(floor(A / a), floor(TDP / power))
56     R * A = A - N * a
57
58 # Assumptions are now explicit and composable.
59 given ITRS, ExtendedPollacksRule, SymmetricAmdahl
60 assume chip_area = 111.0
61 assume thermal_design_power = 125.0
62 assume fraction_parallelism = [0.999, 0.99, 0.97,
63                                0.95, 0.9, 0.8, 0.5]
64 assume tech_node = [45, 32, 22, 16, 11, 8]
65 assume ref_core_performance = linspace(0, 50, 0.05)
66 explore speedup, dark_silicon_ratio, core_num
```

Listing 1. Dark silicon analysis on symmetric topology with ITRS scaling.

by the tool. It is easiest to see this if we talk specifically again about the code of our example dark silicon analysis.

We first note that, although clearly defined conceptually, the three models needed are each of a different *form*: *DevM* is essentially a table of different scaling factors, *CorM* is an empirical set of points and a regression curve and *CmpM* is in the form of mathematical equations relating a set of high-level architectural quantities. Furthermore, even if they were of the same form, they are not "functions" but rather a set of mathematical *relationships*. The distinction is quite important. With traditional lvalue / rvalue style assignments (common to both functions and spreadsheets) you end up with four issues:

Composition: It is hard to link the models' I/O together or even check if the models can be connected properly at all. Architectural models usually are connected to each other through some common system parameters or physical quantities. In this example, to do the dark silicon analysis, one needs to take scaling factors from tables in *DevM*, pass them as inputs to *CorM*, apply the values and re-fits the curve for different *tech node*, after which one then has to sample from the two Pareto curves in *CorM* and supply the samples to $CmpM_U$ for final evaluation. This chain of data movement and dependency is not explicitly exposed by the models, and it takes some effort to understand how these models link together. This issue of mismatched form is even more acute when one wishes to switch out the *CmpM* core model with the $CmpM_R$ core model because $CmpM_R$ takes a completely different set of inputs. With unstructured methods, one has to explicitly program these connections typically by function call chains. With Charm, one simply specifies all variables upfront within each model, and as long as the full variable names are consistent, Charm "wires up" the models through these channelling I/O variables. Perhaps most importantly, Charm throws an error when the models cannot be properly linked. For example, if one forgets to provide values for technology node (Line 64), Charm will complain that too many variables are free, or if the scaling model is about transistor rather than processor core, as long as the variables are properly named (e.g., one does not name transistor performance as *core* performance), Charm will capture this mismatch and warn that the models cannot be connected.

Restructuring and Reorientation: The models cannot be evaluated in a flexible way. Even though the model is a relationship between quantities, in spreadsheets or scripting languages one has to implement the evaluation as functions with fixed arguments. In this example, one typically codes up to evaluate the *speedup* from given value of *core performance*. If the control quantity is changed to another, say *core area*, one has to fix the code. An even worse, and probably more interesting, case is when the control becomes the one under investigation, i.e., the input/output of the functions are reversed. In our example here, it happens when one wishes to explore the core count constraint given a target dark silicon ratio. There is no easy way for ad hoc methods to deal with this kind of flexibility but to completely reprogram. While in Charm, models *are* interpreted as a set of *mutually* dependent relationships without a fixed direction, and Charm runtime will generate the needed functions based on the provided controls and the quantities to explore.

Reasoning under Uncertainty: Architectural models usually involve some uncertainties [16], such as how technology may scale over the next 10-15 years. It is natural for computer architects to first evaluate the model with some concrete values (e.g., the scaling factors in Line 26, 29) and then model the uncertain quantity as some distribution, e.g., Gaussian distribution, as in our case studies in Section IV. It requires non-trivial programming effort with spreadsheets and scripting languages to support uncertain random variables. Charm supports different forms of input values such as scalars, vectors as well as distributions to ease architectural exploration.

Exploration: The analysis procedure is often coupled with the model definition. A common practice for computer architects is to explore the design space by iterating over a set of design options or different values for some system configuration knobs. With the high-level models, architects usually write imperative instructions to iterate over specific variables, and when the iterative variable changes to another, it quickly becomes tedious and error-prone to break and reconstruct the many-fold nested *for* loops. Charm decouples the model specification (Line 11-56) from the analysis procedure declaration (Line 59-66). Such iterations over input values are declarative and transparent (as opposed to writing *for* loops imperatively) by simply providing a list of values as inputs (Line 62, 64 and 65) in Charm.

Secondly, computer architectural quantities often have certain physical meanings. For example, *core performance* typically cannot be negative. A potential issue with unstructured methods is that these boundaries are usually only programmed ad hoc in spreadsheets or scripting languages. A negative *core performance* may be totally *mathematically* valid and *will* lead to meaningless misleading result if not captured in the unstructured implementation. This issue is even more likely to occur in the following two cases.

Implicit Domain Constraints: Architectural models typically have their range of operation. Aside from the physical constraints, implicit domain constraints also come from how the model is built at first place. In the dark silicon example, the normalized performance of the real data points that the authors used to generate the *CorM* is in the range of $(0, 50)$. Even though one can argue that a core with *normalized performance of 100* generally follows that regression but the result derived from that is much less accurate and trusted. This type of constraints are at most times only implicitly conveyed through the model building process, where it leads to a potential pitfall when the model is reused, especially when one only tries to interpret and re-implement the model from natural language descriptions (like in a published paper). While Charm encourages model builders to put in these implicit constraints explicitly as constraints built in the model specifications, e.g., Line 41. Charm will automatically check to see if these

$$\begin{aligned}
&var, rn, tn \in Name \qquad rel \in Relation\\
&\qquad val \in Value\\
p \in Program &:= \overrightarrow{td}\ \overrightarrow{rdef}\ \overrightarrow{a}\ \textbf{explore}\ \overrightarrow{var}\\
td \in TypeDefinition &:= \textbf{typedef}\ tn\ \overrightarrow{rel}\\
rdef \in RuleDefinition &:= \textbf{define}\ rn\ \overrightarrow{rdecl}\\
rdecl \in RuleDeclaration &:= var\ tn \mid rel\\
a \in AnalyzeStatement &:= \textbf{given}\ \overrightarrow{rn} \mid \textbf{assume}\ \overrightarrow{asmt}\\
asmt \in Assignment &:= var = val
\end{aligned}$$

Fig. 2. Abstract syntax of charm. A program is a sequence of type definitions, rule definitions, analysis statements, and a list of variables to explore. Relations are atomic with respect to the semantics; they use the syntax and semantics of the backend solver. They use the standard arithmetic and comparison operators, and allow lists, tuples, and real numbers as possible values.

constraints are violated during evaluation.

Unbounded Distributions: Many architectural quantities follow normal distribution such as *core frequency* due to process variability [22]–[24]. When using these types of unbounded distributions, it sometimes violates the physical constraints of the quantity (*frequency* must be positive). In unstructured modeling, this check is completely ad hoc and, if overlooked, will lead to meaningless results. With Charm, this issue is automatically handled by the type checker, as long as one specifies a correct type for the quantity, e.g., *frequency : R+*.

Last but not least, the design space to cover is typically huge with high-level models. In the dark silicon model, the authors explore a hundred core configurations for each combination of a scaling trend in $DevM$ and a CMP model from $CmpM_U$ or a workload with $CmpM_R$. The models are often to be evaluated hundreds of thousands, if not millions, of times which will take a non-trivial amount of time. It only becomes worse when one tries to evaluate models with uncertainties [16]. Without a structured system, a quick spreadsheet or naive prototyping will end up with unacceptable performance when the problem is scaled up and the burden of optimization falls upon the model builders and others who wish to use existing models through re-implementation. As we show in Section IV, with the invariant hoisting and memoization techniques, Charm greatly speeds up the exploration without additional effort from the model builders.

III. Charm Design

Charm provides a simple domain specific modeling language to express both closed-form models and the design space exploration dimensions. The DSL has an easy-to-use Python-like syntax. In terms of mathematical expressiveness, Charm supports all common closed-form algebra that computer architects often resort to, including linear algebra like polynomials and simple non-linear algebra like exponentiation. Basic non-closed-form functions like summation and product are also supported. To enhance the design space exploration to uncertain domains, Charm also supports distributional values to be set and propagated through the models transparently. Once written in Charm DSL, the interpreter is able to transform the mathematical relationships and constraints into a series of data-flow graphs for fast evaluation. A type system is applied to make sure all architecturally meaningful quantities operate in the correct domain. Charm also optimizes the design space exploration procedure using compiler techniques to eliminate redundant computation. Figure 4 graphically shows the interpretation process.

In this section, we first describe the abstractions Charm provides and formalize the syntax and semantics of Charm DSL. We then articulate the internal design of the interpreter and how type checking, definability checking, evaluation and optimization are done in Charm.

A. Language Abstractions

Charm provides a common layer with three key abstractions to address all the potential issues in Section II-C. In Charm DSL, five keywords are reserved to express three abstractions: **types**, **models** and **analysis**.

Keyword *typedef* translates into the first abstraction: **type**. The type system is designed to be simple but useful: each type is essentially a base type with constraints, e.g., *R+* is defined as a positive number of base type *real* in Listing 1 Line 3-4. There are only two base types, *Real* and *Integer* standing for real numbers and integer numbers respectively. Internally, real numbers are represented by *float* and integers by *int*.

The second key abstraction is **model**. Keyword *define* constructs a model. A model specification in Charm encapsulates the following three pieces in a high-level architecture model.

A set of variables. Each variable has a universally consistent full name. Each variable also has a local short name (optional), as well as explicitly declared types. The short names only live within the definition and the full names are exported to other models and the analysis.

A set of equations. Equations define mathematical relationships between variables using either their full or short names (e.g., Listing 1 Line 54-56). Both linear and nonlinear systems are present in the common architectural models we care about. The general problem of trying to determine the definability of and solving such systems is theoretically hard and beyond the scope of this work. Given the limitations of the solving capabilities of the back-end solvers, some very complicated non-linear equations cannot be symbolically solved (e.g., solve for x in $y = (a^{1/x})^{2^x}$). Fortunately, we find that most models computer architects care about (even complicated as quantum computing in Section IV-B) are well within the limit. Equations can also bind variables to constant quantities as assumptions defined within the model specification (e.g., $kBoltzmann = 8.6173303 \times 10^5$).

A set of constraints. Inequalities define additional constraints on variables or expressions (e.g., Listing 1 Line 41). The difference between equations and constraints in Charm is that equations can be value generative if all but one variable are

$$C, D, E, \Omega \in RelationSet \quad \Gamma \in TypeEnvironment = Name \rightarrow RelationSet$$
$$\Theta \in RuleEnvironment = Name \rightarrow RelationSet \quad \mu \in VariableMap = Name \rightarrow Value$$

$$\frac{C = \left\{c \mid c \in \overrightarrow{rel}\right\}}{\textbf{typedef}\ tn\ \overrightarrow{rel} \Downarrow_T (tn, C)}\ \textsc{typedef} \qquad \frac{\begin{array}{c}(\Gamma, rdecl_i) \Downarrow C_i \quad C = \bigcup C_i \\ i \in 1..|\overrightarrow{rdecl}|\end{array}}{\left(\Gamma, \textbf{define}\ rn\ \overrightarrow{rdecl}\right) \Downarrow_R (rn, C)}\ \textsc{ruledef} \qquad \frac{\Gamma(tn) = C}{(\Gamma, var\ tn) \Downarrow C\,[var/tn]}\ \textsc{rd-var}$$

$$\frac{}{(\_, rel) \Downarrow \{rel\}}\ \textsc{rd-rel} \qquad \frac{C_i = \Theta(rn_i) \quad C = \bigcup C_i \quad i \in 0..|\overrightarrow{rn}|}{\left(\Theta, \textbf{given}\ \overrightarrow{rn}\right) \Downarrow_A C}\ \textsc{given} \qquad \frac{}{\left(\_, \textbf{assume}\ \overrightarrow{asmt}\right) \Downarrow_A \left\{e \mid e \in \overrightarrow{asmt}\right\}}\ \textsc{assume}$$

$$\frac{\begin{array}{c}\mathtt{Ext}(x) = \emptyset \bigvee \mathtt{Ext}(y) = \emptyset \bigvee \mathtt{Ext}(x) = \mathtt{Ext}(y), \forall x, y \in \mathtt{vars}(rel) \\ \omega = \left\{\alpha(a) \mid a \in \bigcup \mathtt{Ext}(b_i), \forall b_i \in \mathtt{vars}(rel)\right\} \quad \alpha(a) = rel[x.a/x, \forall x \in \mathtt{vars}(rel)]\end{array}}{rel \Downarrow_M \omega}\ \textsc{multi-instance}$$

$$\frac{\begin{array}{c}\Gamma(tn_i) = C_i \text{ where } td_i \Downarrow_T (tn_i, C_i) \qquad \Theta(rn_j) = D_j \text{ where } \left(\Gamma, rdef_j\right) \Downarrow_R (rn_j, D_j) \\ \Omega = \bigcup E_k \text{ where } (\Theta, a_k) \Downarrow_A E_k \qquad \Omega' = \bigcup\{\omega \mid rel \Downarrow_M \omega \ \bigwedge\ rel \in \Omega\} \qquad \mathtt{isConsistent}(\Omega') \\ \mathtt{isFullyDetermined}\left(\Omega', \overrightarrow{var}\right) \quad \mu = \mathtt{SOLVE}\left(\Omega', \overrightarrow{var}\right) \quad i \in 1..|\overrightarrow{td}| \quad j \in 1..|\overrightarrow{rdef}| \quad k \in 1..|\overrightarrow{a}|\end{array}}{\overrightarrow{td}\ \overrightarrow{rdef}\ \overrightarrow{a}\ \textbf{explore}\ \overrightarrow{var} \Downarrow_P \mu}\ \textsc{program}$$

Fig. 3. Operational semantics of Charm. Relations are here taken as atoms; they use the semantics of the backend solver engine. An overhead arrow indicates a sequence of one or more elements. $C[x/y]$ indicates to substitute all instances of y in C with x. `vars` returns the names of all variables used in the relation set, while `Ext` returns all extensions of a variable (portion of the name appearing after a dot when multi-instanced). `isConsistent` ensures the relation set is consistent. `isFullyDetermined` ensures the relation set is fully determined with respect to $\overrightarrow{var}$. `SOLVE` is an instance of the backend solver; it returns a mapping of all specified variables to values (real numbers, lists, and tuples). TYPEDEF takes a type definition and returns a tuple with type name and relation set. RULEDEF takes a rule definition and the type environment and returns a tuple with rule name and relation set. RD-VAR takes a type rule declaration and the type environment and returns a relation set, where relations on the indicated type now apply to the indicated variable. RD-REL takes a relation rule declaration and returns the same relation in a set. GIVEN takes a **given** analyze statement and the rule definitions and returns the relation set of the indicated rule. ASSUME takes an **assume** analyze statement and returns a relation set of all the declared equalities. MULTI-INSTANCE takes a relation and returns a set of relations, where the original relation is duplicated once for each extension possessed by its variables, with the names of the variables replaced by their extended version (as discussed in section III-B). PROGRAM takes a program and returns a map for the list of exploration variables, mapping each to real numbers, lists, and tuples determined by the backend solver.

given, while constraints require all variables given during evaluation. Inequalities are by definition constraints and, when all variables are given, an equation is over-determined and turns into a constraint. We refer to the set of both equations and constraints as *relations*.

Charm DSL accepts different mathematically equivalent forms of relations, so that different modelers with different background expertise can write the math in the conventional way of their own fields and use other models directly as they are without rewriting.

The Charm DSL is strongly typed. The model abstraction enforces explicit type declaration to make sure there are not implicit assumptions about data types and domains across models.

Charm abstracts the common structure of an **analysis** with three keywords: *given*, *assume* and *explore*.

Before computation, *given* statement selects the model in the analysis. If multiple models are selected, they are linked together automatically by the interpreter. Full names of variables are used to connect each other across models.

Although in general, many algebra systems can be solved without additional inputs, for computer architecture models, at most times, some control quantities need to be given (e.g., design options like *core size* and system configurations like *cache associativity*) in order to solve for the quantities under investigation (e.g., *speedup* of a CMP). Keyword *assume* serves such purpose by differentiating assignment equal signs from mathematical equal signs inside model specification, i.e., *assume* statements are assignments much like in other programming languages while equations in model specification are merely mathematical relationships which do not imply a direction of data movement. Charm also constrains *assume* statements to be assignment with constants, i.e., they can only be used to express external inputs to the model rather than defining additional relations outside of the model specification.

Charm supports both scalar and vector value assignments, as well as random variable of commonly used distributions, e.g., *Gaussian Distribution*.

Iteration is expressed in a Pythonic list-like syntax or functions that generates a list, e.g., *linspace*, and assigned

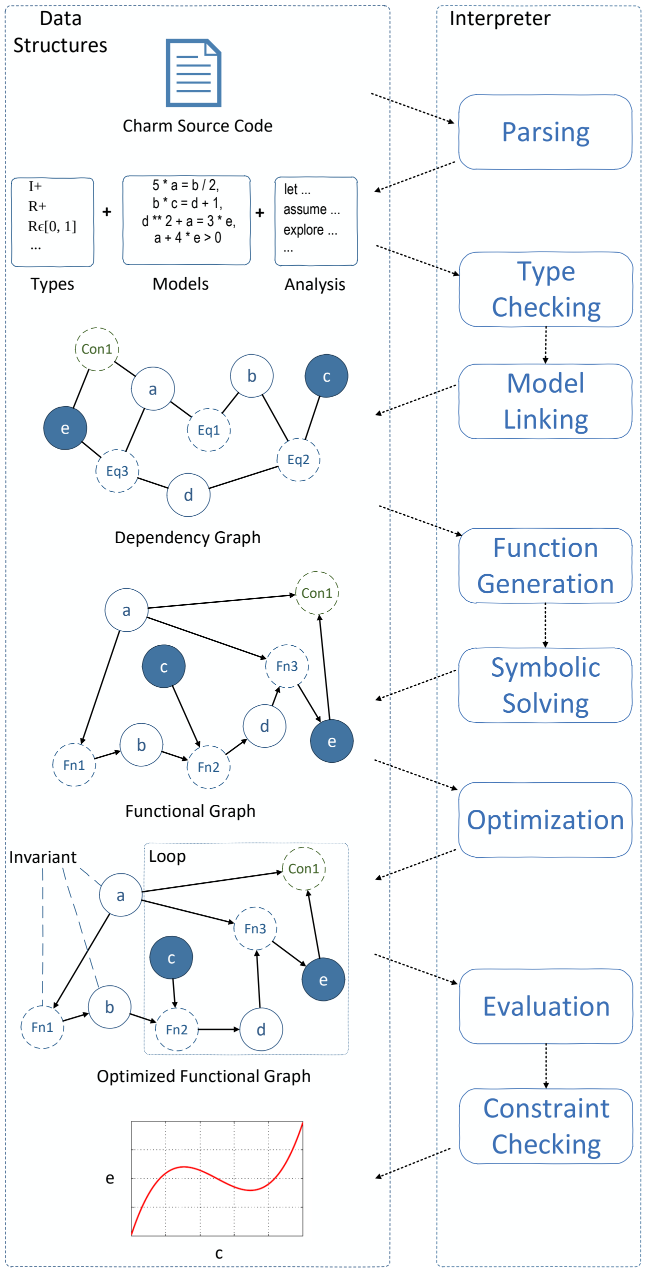

Fig. 4. Overview of Charm interpreter. The example system has 3 equations (Eq1, Eq2 and Eq3), 1 constraint (Con1) and 5 variables in which c is the iterative input. In this example, we are tying to explore the relationship between e and c given a. The parser takes Charm source code and breaks it into a set of types, a set of model definitions and a set of analysis statements. Charm then links types, models and assignments in a dependency graph after they go through the type checker. The graph then is fed to a function generator and a symbolic solver to convert it into a functional graph. The optimizer finally takes the functional graph and annotate it with hints for execution before it finally gets evaluated and checked against model constraints.

to some input variable just like a normal *assume* statement (e.g., Listing 1 Line 65). Charm handles iteration naturally by selecting combinations of all iterative input values non-repeatedly from their Cartesian space in a Gray code fashion. Two special cases are: a), if two or more input variables are dependent, they can be expressed like Python tuple assignment, e.g., $assume\ (tech\_node, freq\_scaling\_factor) = [(45, 1.), (32, 1.09)]$ and b), if a variable is indexed, it can be expressed using special "list" notation after its variable name, e.g., $assume\ L[] = [1, 2]$, which means $L[0] = 1$ and $L[1] = 2$. These notations become handy when we write the quantum models with Charm in Section IV-B.

Finally, an analysis is completed by specifying which quantities to solve for symbolically and evaluate using *explore*. Charm exploits a data-flow centric approach and builds a directed acyclic functional graph internally to propagate given values through linked models to the final responsive variables architects wish to explore.

Figure 2 gives the abstract syntax of Charm and Figure 3 formalizes the semantics.

B. Language Internals

In order to evaluate the models and optimize the evaluation logic, Charm uses two data-flow graph structures internally to represent and transform the computation. In this section, we first define the core graph data structures and then describe how we can perform type checking, function generation, evaluation and optimization with these graph structure.

Dependency Graph. A dependency graph is a bipartite graph $G =< V_{var}, V_{rel}, E >$, where:

- V_{var} is the variable node set in which every variable in the selected models is a vertex.
- V_{rel} is the relation node set and $V_{rel} = V_{eq} \cup V_{con}$, where V_{eq} is the set of vertices in which every equation in the selected models is a vertex; V_{con} is the set of vertices in which every constraint in the selected models is a vertex.
- E is the set of edges and there exists an edge between vertices in V_{var} and V_{rel} if and only if the variable name appears in the relation.

Functional Graph. A functional graph is a *directed acyclic* dependency graph D in which:

- Every node in V_{var} has at most 1 incoming edge, i.e., its in-degree being 0 or 1.
- Every node in V_{eq} has at most 1 outgoing edge, i.e., its out-degree being 0 or 1.
- Every node in V_{con} has no outgoing edge, i.e., its out-degree being 0.

Dependency graph building and static type checking. To build the dependency graph from the models, Charm performs a single scan over all relations in the models. It assigns a variable node to every variables with a unique full name (including variables automatically generated by multi-instancing) and an equation/constraint node to every equation/constraint. When creating relation node, Charm creates an edge between the equation/constraint node to a variable node if the variable is used in the equation/constraint. Finally, Charm scans the analysis statements and marks variable nodes being assigned as input nodes.

Charm performs simple type checking both statically when building the dependency graph after parsing and dynamically when checking constraints at runtime. Static type checking is done by tracking the variable names and types when building the dependency graph. Each variable must be declared with an explicitly defined type. If a variable name is used by two or more relations, we check that their defined types are identical (both base type and constraints associated). Charm aborts execution and issues an error message for inconsistent types.

Relation Multi-instancing. When building a dependency graph, different variables sometimes follow the same mathematical relationships. An example is *core_performance.big* and *core_performance.small* defined in Listing 2 Line 5-6. Both of them follow equation in Listing 1 Line 23 when plugged in for evaluation. We discuss their physical meanings later in Section IV-A, but they are essentially two variables following the same mathematical relationship. We refer this behavior as "relation multi-instancing" and use the dot notation (a variable name and a name extension concatenated by dot, e.g., *core_area.big*) to invoke multi-instancing. Charm internally creates variable nodes and relation nodes for multiple instances with different name extensions. Figure 5 shows how these nodes in the dependency graph are created. The model is ill-defined if Charm fails to find extended input variables with consistent name extensions or discovers inconsistent name extension sets for different variables trying to invoke multi-instancing.

Functional graph building and function generation. After building the dependency graph G, the function converter tries to convert G into a functional graph F. If it can convert successfully, there is a viable solution when all equations or sets of equations can be solved and lambdified by the back-end symbolic solver, and therefore the models can be evaluated by Charm.

The function converter backtracks through G in a DFS manner and tries to label all the edges with a direction without introducing a conflict. A conflict occurs when an equation node has more than one outgoing edges or when an inequality node has any outgoing edge or when a variable node (excluding input nodes) does not have exact one incoming edge. If there is a successful labeling of all edges, Charm uses Sympy [25] as the back-end solver to convert all equations and constraints (all inequalities and equation nodes with an out-degree of 0 are considered as constraints at this point) into callable functions with inputs being the variables directly pointing to the equation/constraint and output being the variable pointed at by the equation node. As part of type checking, each variable node is also associated with the constraint from its type. These type constraints are also lambdified and evaluated during evaluation. The search space

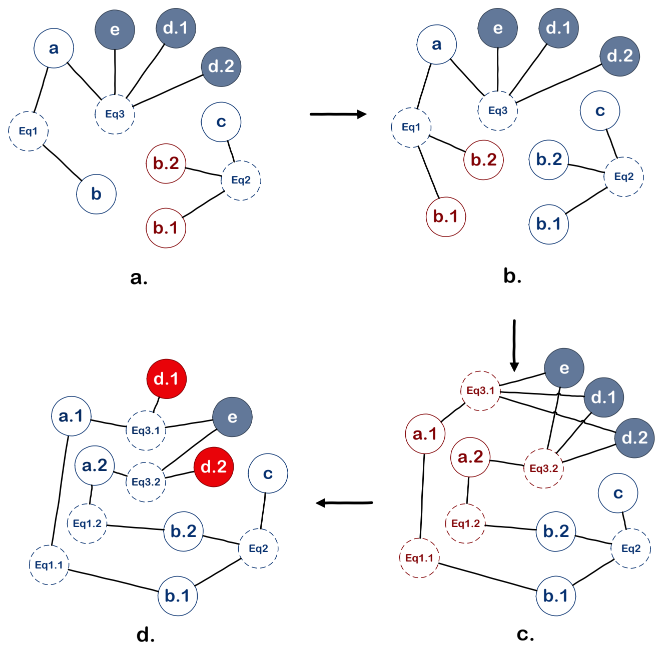

Fig. 5. Relation multi-instancing when generating dependency graph. a) The initial graph has extended names ($b.1$, $b.2$). b) Charm finds and splits the corresponding base name node. c) Charm propagates the multi-instancing, i.e., all nodes connected to the base name node (b) are also split. Then Charm merges names with same extension together. d) The multi-instancing ends with checking input nodes for identical name extensions and removing edges between non-consistent name extensions. In this case, it ends when the split process reaches d and e, successfully finds $d.1$ and $d.2$ which are extended names with consistent name extension set ($\{.1, .2\}$ in this example) and removes the edges between ($d.1$, $Eq3.2$) and ($d.2$, $Eq3.1$).

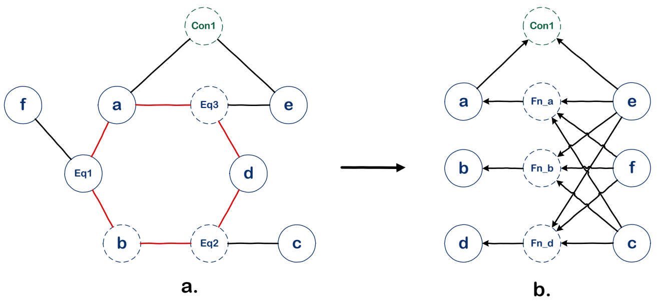

Fig. 6. Cycle elimination when generating functional graph. Equations in a cycle are solved at once and are replaced with three functions, each of which generates a different variable value.

for conversion is in practice greatly reduced by the following heuristics:

- All edges with one node being input node have fixed direction (from the input node).
- All edges with one node being a dangling variable node (variable node that has only one edge) have fixed direction (to the variable node).
- All edges with one node being a constraint have fixed direction (to the constraint node).

Cycle elimination. A functional graph F must be *acyclic* in order to evaluate. However, when there are codependent equations, they form cycles. In case of a cycle, all equation nodes in the cycle must be solved altogether. We pass the equations in a cycle to the solver at once and then replace the cycle with pairs of function node and variable node, where each pair is a mapping between all inputs to the cycle (a dummy input node is created if there are no inputs from other parts of the graph to the cycle) and one variable node inside the cycle. Each function node generated by the cycle has one variable along the cycle as its output and all functions generated by the cycle are from the same set of equations, only with different variables as output. Figure 6 shows an example of cycle elimination in F.

Computational constraints. A special computational constraint is applied when building a functional graph: some mathematical operators are not reversible or have infinite solutions, such as $\sum$ and $\prod$, some are computationally hard for the solver, like solving x in $y = (a^{1/x})^{2^x}$. For the non-reversible equation, its direction is fixed, i.e. its edges have fixed direction not subject to the function converter.

Evaluation and constraint checking. Once we have a viable functional graph F, a feasible solution is to derive from all input nodes and propagate the given values by traversing F. Each following function/constraint node is transformed using higher-order functions to "remember" propagated partial values before all inputs are ready and it can be evaluated.

Optimization. Oftentimes architects explore the relationship between two variables by iterating over different input values. One simple yet effective optimization is invariant hoisting. With the functional graph structure, it is straightforward to optimize for invariant in Charm. From each iterative variable node, Charm simply traverse from that node, then all nodes that cannot be reached from the iterative input nodes are invariant to iteration over that input. In the simple illustrative example in Figure 4, c is iterative and a, b, $Fn1$ are invariant because there are not paths from c to them.

Each function node also caches a mapping table between inputs and its output. Such memoization optimizes away unnecessary re-computation over same set of input values.

IV. Case Studies

In this section, we demonstrate the application of Charm using two case studies. In the first case study, we show the benefits of Charm by extending the dark silicon analysis with a different topology and a distribution of technology scaling. We also compare the execution times with and without optimization.

The second case study focuses more on the problem of modeling a critical resource in fault-tolerant quantum computing and performs exploration with varying physical error rate. Interestingly, when validating Charm results in the second case study, Charm helps find inconsistent model definition errors, which are silently propagated through by Mathematica [26] and would have led to incorrect results.

```
1  # Amdahl's Law under Asymmetric Multicore (CmpM_U).
2  define AsymmetricAmdahl:
3      speedup : R+ as sp
4      # here we need two types of perf, area, power
5      core_performance.big : R+ as big_perf
6      core_performance.small : R+ as small_perf
7      core_area.big : R+ as big_a
8      core_area.small : R+ as small_a
9      core_power.big : R+ as big_power
10     core_power.small : R+ as small_power
11     core_num : R+ as N
12     chip_area : R+ as A
13     thermal_design_power : R+ as TDP
14     fraction_parallelism : Fraction as F
15     dark_silicon_ratio : Fraction as R
16     sp = 1 / ((1-F)/big_perf + F/(N*small_perf+
       big_perf))
17     N = min(floor((A - big_a)/small_a),
18             floor((TDP - big_power)/small_power))
19     R * A = A - (N * small_a + big_a)
20     big_perf >= small_perf
21
22 given ITRS, ExtendedPollacksRule, AsymmetricAmdahl
23 assume ref_core_performance.big=linspace(0,50,0.05)
24 assume ref_core_performance.small=linspace
       (0,50,0.05)
```

Listing 2. Asymmetric model and the changes in code.

```
# Conservative scaling model (DevM).
define ConservativeScaling:
    ...
    a = piecewise((1., t=45),   (1.10, t=32),
                  (1.19, t=22), (1.25, t=16),
                  (1.30, t=11), (1.34, t=8))
    b = piecewise((1., t=45),   (0.71, t=32),
                  (0.52, t=22), (0.39, t=16),
                  (0.29, t=11), (0.22, t=8))

given ConservativeScaling, ExtendedPollacksRule,
    AsymmetricAmdahl
```

Listing 3. Conservative scaling and the changes in code.

A. *Dark Silicon and Beyond*

Listing 2 highlights all the changes that we need to implement in Charm to model and switch the DSE from symmetric topology to asymmetric. Note that in the asymmetric model, "relation multi-instancing" comes in handy when expressing two co-existing types of core. To switch the analysis, all we need to do is to change the models that are *given* (Listing 2 Line 22) and provide values for two types of core instead of one (Listing 2 Line 23-24). We also write a new constraint (Listing 2 Line 20) to specify the fact that the big core should have better performance than the small core.

It's even simpler to switch from ITRS scaling predictions to the conservative predictions [19]. Listing 3 shows all the changes needed. Figure 7 plots the resulting scaling trends for the asymmetric topology.

One interesting question one may ask is *"what if the actual technology scaling is somewhere in between the two predictions?"* We explore the design space with a distribution of scaling factors. We use a Gaussian distribution for the scaling factor, the mean of which being the average value

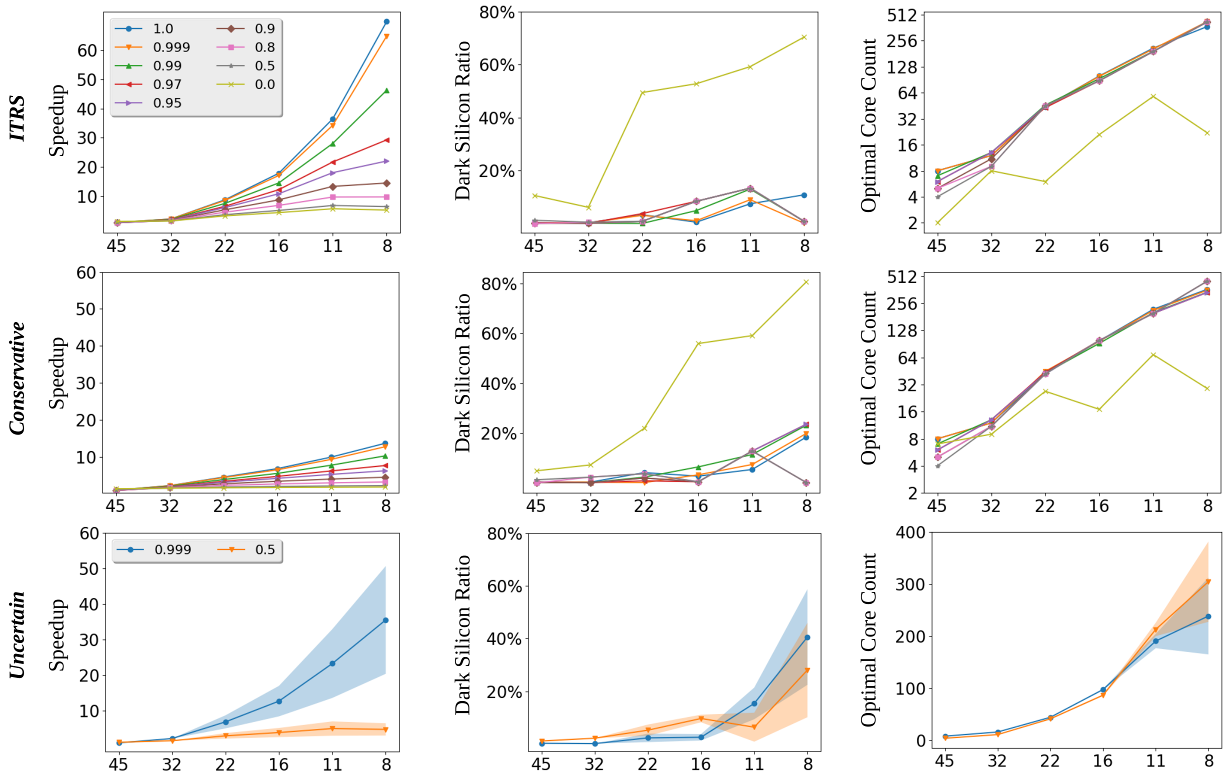

Fig. 7. Upper-bound scaling with asymmetric topology with tech node on x-axis. Note that the last figure of optimal core count has a linear-scale y-axis to better demonstrate the variance. For clarity we only plot two regions in the uncertain scaling results, but the trends for other f values are similar.

between the two extremities and the standard deviation being the difference between the mean and the extremities. Listing 4 shows the necessary changes in Charm code. It is important that although Gaussian distribution is not bounded, the scaling factors have a bounded domain. The type checking in Charm makes sure that the scaling factors a and b operate only in their defined domains (see Listing 1 Line 20-21), and the provided Gaussian distribution is converted to a truncated Gaussian distribution with the same mean and standard deviation within Charm. From Figure 7, we can see that with the technology scaling, the more parallel workload (with an f close to 1) shows more sensitivity towards technology uncertainties while the more serial workload is less sensitive to the changes in the core performance and power. Another probably even more interesting observation is that the optimal core count of the most performant configuration becomes very uncertain once we hit 11nm and beyond. The uncertainty grows sharply from 16nm to 11nm mainly because below 11nm, the CMP is mainly **area bounded**, and since the area scaling is certain (Listing 1 Line 25), it limits the amount of uncertainty that gets propagated to the optimal core count. Meanwhile, when the tech node scales to 11nm and beyond, the CMP becomes **power bounded** and is extremely sensitive to the power uncertainties propagated from the uncertainty of the power scaling factor.

Figure 8 shows the actual functional graph generated by Charm. In terms of execution performance, we compare

```
# Distributional scaling model (DevM).
define DistScaling:
    ...
  a = piecewise((1.,t=45),(Gauss(1.095,0.005),t=32),
  (Gauss(1.785,0.595),t=22),(Gauss(2.23,0.98),t=16),
  (Gauss(2.735,1.435),t=11),(Gauss(2.595,1.255),t=8)
    )
  b = piecewise((1.,t=45),(Gauss(0.685,0.025),t=32),
  (Gauss(0.53,0.01),t=22),(Gauss(0.385,0.005),t=16),
  (Gauss(0.27,0.02),t=11),(Gauss(0.17,0.05),t=8))

given DistScaling, ExtendedPollacksRule,
    AsymmetricAmdahl
```

Listing 4. Uncertain scaling and the changes in code.

Charm execution to an unoptimized baseline in which all computation is re-done per iteration (no invariant hoisting nor memoization). For ITRS or conservative scaling with asymmetric topology (a design space of 150K design points), full-blown Charm finishes on average within 120.5s, while the unoptimized implementation uses 159.5s (1.3X speedup). For the uncertain scaling with a MC sample size of 200 ($\tilde{1}$ million design points), optimized Charm uses 1562.5s, and it takes 5703.1s for the baseline implementation (3.6X speedup) on a single Intel i7 core at 3.3GHz to finish.

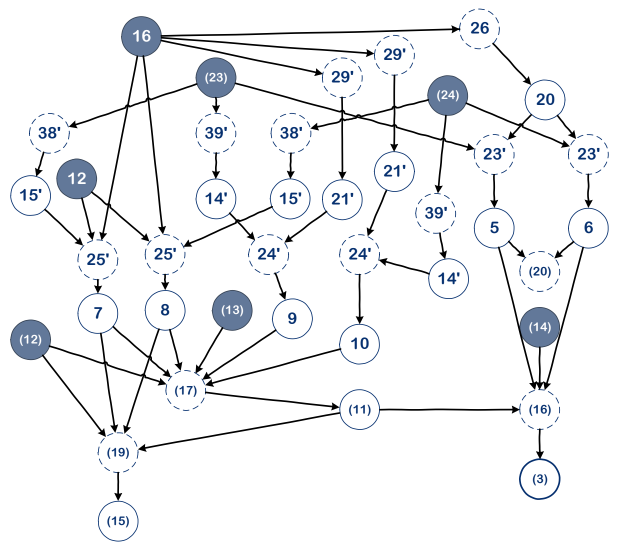

Fig. 8. Functional graph generated by Charm for asymmetric dark silicon model. Node labels correspond to line numbers in Charm source code. As we present the asymmetric model code separately from the rest, plain numbers correspond to lines in Listing 1, and numbers in parenthesis correspond to lines in Listing 2. Numbers with prime (e.g., 21') are cloned names/equations generated by Charm from the same line of code.

B. Surface Code Error Corrected Quantum Application and Architecture Co-optimization

In this section, a high level model for the resource overhead for implementing magic state distillation on surface code [5]–[7] is described and implemented within the Charm framework, which is used to pinpoint nontrivial interactions between fundamental system parameters.

For this study, we focus primarily on the Bravyi-Haah "$3k + 8 \rightarrow k$" procedure [5] augmented with the block-code protocol. By recursively stacking magic state distillation protocols in a tree-like fashion, one can generate arbitrarily high-fidelity magic states, which is required by a quantum program [7]. The space required by one round of Bravyi-Haah magic state distillation is given by the number of physical qubits required to run the circuit. Using block code, the procedure will consume $(3k+8)^{\ell-1}(6k+14)d^2$ physical qubits, where d is the surface code distance we are using.

Adding more factory capacity K results in more output magic state capacity (higher effective rate). However this also adds more components to the factory that may fail. In fact, a magic state factory has a **yield rate** proportional to the output capacity K that is caused by uncertainty in the success probability of the underlying Bravyi-Haah protocol. This yield rate scales as:

$$K_{\text{output}} = k^{\ell} \times \prod_{r=1}^{\ell} \Big[1 - (3k+8)\epsilon_r \Big] \tag{1}$$

where $\epsilon_r = (1+3k)^{2^r-1}\epsilon_{\text{in}}^{2^r}$, because each level of the process results in incrementally higher fidelity (i.e., lower error rate).

Given a T gate request distribution D representing a program, the number of iterations needed to distill is:

$$\sum_{t=0}^{T_{\text{peak}}} \Big(s \cdot \sqrt{K'} + \sqrt{\frac{t - sK'}{2}} \cdot R \Big) \cdot L_{\text{cp}} \cdot D[t], \tag{2}$$

where $s = \left\lfloor \frac{t}{K'} \right\rfloor$, and $R = \frac{7d+15}{24d\ell}$. All of these equations combine to form a high level space-time estimate of the resources required to execute a quantum application on a machine with a specified magic state distillation factory architecture.

Using Charm, we are able to analyze the underlying sensitivity of different magic state factory architectures to variations in the underlying error rate of the physical system. We examine two different design cases, one where the factory is designed assuming a 10^{-3} error rate, and one assuming a 10^{-5} error rate. Figure 9 illustrates that while the time-optimal factory does show a lower expected space-time volume, it also shows significantly higher uncertainty and spreads in performance values over the space-optimal factory. This design point clearly motivates that quantifying the uncertainty of a physical device is necessary to lead to risk-optimal system designs that perform well on a given system.

Charm is able to discover and quantify this trend with minimum effort, and allows for a quantitative analysis to be performed on these designs that will aid the construction of physical systems. Additionally, implementing this high level performance model in Charm allows for validation and more domain-specific error catching that previous implementations in Mathematica has been unable to catch. Specifically, a previous implementation has an incorrect parameter passed into a distance calculation function that Mathematica allowed to flow through. Charm is able to detect this error, warns that the models cannot be connected properly which helped correct the results of the model.

V. Related Work

A. Closed-form Architecture Models

Many of the recently developed high-level analytical models are conceptually inherent from the well known Amdahl's Law [27], which is often expressed as a closed-form performance model of parallel programs. The most well studied derivative is the multicore performance model by Hill and Marty [17]. A long line of research work using extensions of their closed-form model focus on different aspects of the system, including application [28], communication and synchronization [29], [30], energy and power consumption [4], [31], heterogeneity [11], [32], chip reliability [33], architectural risk [16] and so on. Our language consumes these models and provides a systematic way to establish new high-level models either by constructing new equations and constraints or reusing those from the above models.

Another set of analytical performance model is built directly from the mechanisms of the specific system [8], [9], [34]–[42]. These models usually rely on some simulations/hardware counter to collect the necessary inputs to their core closed-form equations. Our language can also express and manage these equations. Empirical modeling [43]–[51] is also used to

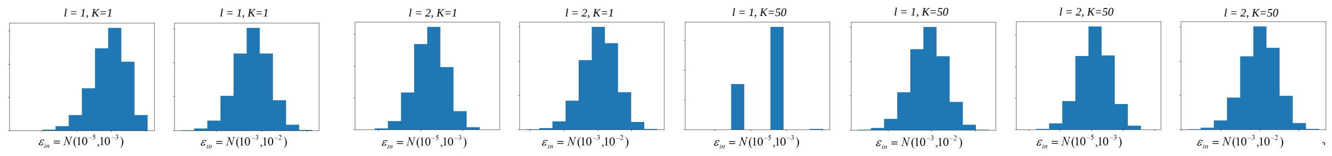

Fig. 9. Factories designed with an implied low physical error rate only require concatenation up to $\ell = 1$ level, while more pessimistic factories require $\ell = 2$. While larger factories with $K = 50$ consistently show lower mean space-time consumption, they also suffer from large performance uncertainty when the assumed design error rate varies.

discover correlation between two or more architectural quantities. They can usually be expressed as parametrized equations in closed-form, the resulting models of such empirical methods can also be managed by and benefit from Charm.

B. Systems and Languages Supporting Analytical Modeling

There exist systems and languages that support structured analytical modeling. Modelica [52] supports multi-domain analytical modeling with an emphasis on object-oriented model composition, but the connection of models need to be explicitly dictated and the design space exploration require user intervention, while Charm is more restricted and thus able to automatically link models and generate exploration loops. Aspen [53] provides a DSL to express application and an abstract machine organization in order to model performance. Palm [54] utilizes source code annotation to build analytical model for the application. LSE [55] is a fully concurrent-structural modeling framework designed to maximize reusability of components. There are also many other works in the field of HPC for automatic performance modelling extracting [56]–[58]. Most of these languages and systems serve a different purpose of expressing mapping between performance/power model and specific detailed application/architecture and are not well-suited for high-level analytical design space exploration. While Charm is tailored for structured yet flexible exploration of the interactions between architectural variables as well as their ramifications at a high level. There are also a few systems exploiting the power of symbolic execution for modeling [16], [20], but Charm provides more capabilities around formalizing, checking and evaluating the models. There also exits a tool [59] of the same name CHARM (Chip-architecture Planning Tool) which uses a knowledge-based scheme to ease high-level synthesis.

The internals of Charm resemble some of the data-flow centered programming languages in the field of incremental/reactive programming [60]–[64] but differ in that Charm is highly restrictive. The restrictiveness means that Charm is more of a modeling language rather than a programming language, i.e., Charm does not support general purpose structures like loops and function calls but supports a malleability useful for exploration (e.g., reversing input/output dependencies).

VI. Conclusion

Computer architecture is a rapidly evolving field. Complex and intricately interacting constraints around energy, temperature, performance, cost, and fabrication create a web of relationships. As we move toward more heterogeneous and accelerator-heavy techniques, our understanding of these relationships is more fundamental to the process of design and evaluation than ever before. Already today we are seeing machine learning [65], cryptography [66], and other fields attempting to pull architectural analysis into their own work – sometimes introducing serious bugs along the way. Architecture is now a field that is expected to make scientific statements connecting nanoscale device details to the largest warehouse scale computers and everything in between. Spanning these 11 orders of magnitude will require more complex analytic approaches to be used in tandem with the traditional simulation and prototyping tools that computer architects have long relied on.

Charm provides domain specific language support for architecture modeling in a way that leads to more flexible, scalable, shareable, and correct analytic models. While our language already supports symbolic restructuring, memoization, hoisting, and several optimization and consistency checks, Charm is merely the first step towards a more powerful and useful modeling language for computer architects. It is easy to imagine other useful additions in the future such as checks on the consistency of physical types (e.g., nJ versus pJ errors) or back-ends connecting models to non-linear optimizers. Most importantly though, by giving the sets of mutually dependent architectural relationships a common language, Charm along with the collection of established models have the potential to enable more complete and precise specification, easier composition, more through checking, and (most importantly) broader reuse and sharing of complex analytic models. Looking forward we see that tools such as this hold significant promise in enabling more collaborative and community driven efforts that make our best thinking on the future of architecture more readily and easily accessible to all that are interested.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grants No. 1740352, 1730309, 1717779, 1563935, 1444481, 1341058, 1730449, 1660686, Los Alamos National Laboratory and the U.S. Department of Defense under subcontract 431682 and gifts from Cisco Systems and Intel Corporation.

The authors would like to thank Michael Christensen for his help with formalizing the semantics and the anonymous reviewers for their invaluable feedback.

References

[1] J. Bornholt, R. Lopez, D. M. Carmean, L. Ceze, G. Seelig, and K. Strauss, "A dna-based archival storage system," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 637–649. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872397

[2] A. Javadi-Abhari, P. Gokhale, A. Holmes, D. Franklin, K. R. Brown, M. Martonosi, and F. T. Chong, "Optimized surface code communication in superconducting quantum computers," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 692–705. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123949

[3] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, "An experimental microarchitecture for a superconducting quantum processor," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 813–825. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123952

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108

[5] S. Bravyi and J. Haah, "Magic-state distillation with low overhead," *Physical Review A*, vol. 86, no. 5, p. 052329, 2012.

[6] A. G. Fowler, S. J. Devitt, and C. Jones, "Surface code implementation of block code state distillation," *Scientific Reports*, vol. 3, p. 1939, jun 2013.

[7] J. O'Gorman and E. T. Campbell, "Quantum computation with realistic magic-state factories," *Physical Review A*, vol. 95, no. 3, p. 032338, 2017.

[8] M. Breughe, S. Eyerman, and L. Eeckhout, "A mechanistic performance model for superscalar in-order processors," in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, April 2012, pp. 14–24.

[9] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 3:1–3:37, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1534909.1534910

[10] B. Agrawal and T. Sherwood, "Modeling tcam power for next generation network devices," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2006, pp. 120–129.

[11] M. S. B. Altaf and D. A. Wood, "Logca: A high-level performance model for hardware accelerators," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 375–388. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080216

[12] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–7.

[13] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[14] 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 *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 423–428.

[15] Xilinx, "7 series product tables and product selection guide," February 2018, online. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf

[16] W. Cui and T. Sherwood, "Estimating and understanding architectural risk," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, 2017.

[17] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.209

[18] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 25–28, Jan 2009.

[19] S. Borkar, "The exascale challenge," in *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*, April 2010, pp. 2–3.

[20] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen, "Validating the unit correctness of spreadsheet programs," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 439–448. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999448

[21] S. G. Powell, K. R. Baker, and B. Lawson, "A critical review of the literature on spreadsheet errors," *Decis. Support Syst.*, vol. 46, no. 1, pp. 128–138, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.dss.2008.06.001

[22] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Varius: A model of process variation and resulting timing errors for microarchitects," *IEEE Transactions on Semiconductor Manufacturing*, vol. 21, no. 1, pp. 3–13, Feb 2008.

[23] X. Liang and D. Brooks, "Mitigating the impact of process variations on processor register files and execution units," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec 2006, pp. 504–514.

[24] A. Rahimi, L. Benini, and R. K. Gupta, "Variability mitigation in nanometer cmos integrated systems: A survey of techniques from circuits to software," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410–1448, July 2016.

[25] SymPy Development Team, *SymPy: Python library for symbolic mathematics*, 2016. [Online]. Available: http://www.sympy.org

[26] W. R. Inc., "Mathematica, Version 11.2," champaign, IL, 2017.

[27] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[28] X.-H. Sun and Y. Chen, "Reevaluating amdahl's law in the multicore era," *J. Parallel Distrib. Comput.*, vol. 70, no. 2, pp. 183–188, Feb. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2009.05.002

[29] L. Yavits, A. Morad, and R. Ginosar, "The effect of communication and synchronization on amdahl's law in multicore systems," *Parallel Computing*, vol. 40, no. 1, pp. 1 – 16, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819113001324

[30] S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 362–370. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816011

[31] D. H. Woo, D. H. Woo, D. H. Woo, D. H. Woo, H. H. S. Lee, H. H. S. Lee, H. H. S. Lee, and H. H. S. Lee, "Extending amdahl's law for energy-efficient computing in the many-core era," *Computer*, vol. 41, no. 12, pp. 24–31, Dec 2008.

[32] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 225–236. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.36

[33] W. J. Song, S. Mukhopadhyay, and S. Yalamanchili, "Amdahls law for lifetime reliability scaling in heterogeneous multicore processors," in *The 2016 International Symposium on High-Performance Computer Architecture (HPCA-22)*, March 2016.

[34] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[35] ——, "An integrated gpu power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer*

Architecture, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 280–289. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815998

[36] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 673–686.

[37] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 338–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006729

[38] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and mshrs," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, Nov 2008, pp. 59–70.

[39] S. Eyerman, K. Hoste, and L. Eeckhout, "Mechanistic-empirical processor performance modeling for constructing cpi stacks on real hardware," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 216–226.

[40] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, June 2000, pp. 83–94.

[41] 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 *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 469–480.

[42] A. A. Nair, S. Eyerman, J. Chen, L. K. John, and L. Eeckhout, "Mechanistic modeling of architectural vulnerability factor," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 11:1–11:32, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2669364

[43] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 7–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545217

[44] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168881

[45] B. Lee and D. Brooks, "Statistically rigorous regression modeling for the microprocessor design space," in *ISCA-33: Workshop on Modeling, Benchmarking, and Simulation*, 2006.

[46] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168882

[47] C. Dubach, T. Jones, and M. O'Boyle, "Microarchitectural design space exploration using an architecture-centric approach," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 262–271. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.26

[48] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 249–258. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229479

[49] B. C. Lee and D. M. Brooks, "Illustrative design space studies with microarchitectural regression models," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Feb 2007, pp. 340–351.

[50] B. C. Lee, J. Collins, H. Wang, and D. Brooks, "Cpr: Composable performance regression for scalable multiprocessor models," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 270–281. [Online]. Available: https://doi.org/10.1109/MICRO.2008.4771797

[51] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815967

[52] H. Elmqvist, S. Mattsson, H. Elmqvist, and D. Ab, "An introduction to the physical modeling language modelica," in *Proc. 9th European Simulation Sympossium ESS97, SCS Int*, 1997, pp. 110–114.

[53] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 84:1–84:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389110

[54] N. R. Tallent and A. Hoisie, "Palm: Easing the burden of analytical performance modeling," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 221–230. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597683

[55] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, "The liberty simulation environment: A deliberate approach to high-level system modeling," *ACM Trans. Comput. Syst.*, vol. 24, no. 3, pp. 211–249, Aug. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151690.1151691

[56] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, "Exasat: An exascale co-design tool for performance modeling," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 2, pp. 209–232, May 2015. [Online]. Available: http://dx.doi.org/10.1177/1094342014568690

[57] S. R. Alam and J. S. Vetter, "A framework to develop symbolic performance models of parallel applications," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 320–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898699.1898852

[58] ——, "Hierarchical model validation of symbolic performance models of scientific kernels," in *European Conference on Parallel Processing*. Springer, 2006, pp. 65–77.

[59] K. H. Temme, "Charm: a synthesis tool for high-level chip-architecture planning," in *1989 Proceedings of the IEEE Custom Integrated Circuits Conference*, May 1989, pp. 4.2/1–4.2/4.

[60] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep 1991.

[61] L. Mandel and M. Pouzet, "Reactiveml: A reactive extension to ml," in *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '05. New York, NY, USA: ACM, 2005, pp. 82–93. [Online]. Available: http://doi.acm.org/10.1145/1069774.1069782

[62] M. A. Hammer, U. A. Acar, and Y. Chen, "Ceal: A c-based language for self-adjusting computation," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 25–37. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542480

[63] T. Szabó, S. Erdweg, and M. Voelter, "Inca: A dsl for the definition of incremental program analyses," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 320–331. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970298

[64] P. LeGuernic, T. Gautier, M. L. Borgne, and C. L. Maire, "Programming real-time applications with signal," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, Sep 1991.

[65] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[66] J. Alwen and J. Blocki, "Efficiently computing data-independent memory-hard functions," in *Annual Cryptology Conference*. Springer, 2016, pp. 241–271.