Language Support for Navigating Architecture Design in Closed Form

WEILONG CUI and GEORGIOS TZIMPRAGOS, University of California, Santa Barbara
YU TAO, Peking University
JOSEPH MCMAHAN, DEEKSHA DANGWAL, and NESTAN TSISKARIDZE, University of California, Santa Barbara
GEORGE MICHELOGIANNAKIS and DILIP P. VASUDEVAN, Lawrence Berkeley National Laboratory
TIMOTHY SHERWOOD, University of California, Santa Barbara

As computer architecture continues to expand beyond software-agnostic microarchitecture to specialized and heterogeneous logic or even radically different emerging computing models (e.g., quantum cores, DNA storage units), detailed cycle-level simulation is no longer presupposed. Exploring designs under such complex interacting relationships (e.g., performance, energy, thermal, frequency) calls for a more integrative but higher-level approach. We propose Charm, a modeling language supporting closed-form high-level architecture modeling. Charm enables mathematical representations of mutually dependent architectural relationships to be specified, composed, checked, evaluated, reused, and shared. The language is interpreted through a combination of automatic symbolic evaluation, scalable graph transformation, and efficient compiler techniques, generating executable DAGs and optimized analysis procedures. Charm also exploits the advancements in satisfiability modulo theory solvers to automatically search the design space to help architects explore multiple design knobs simultaneously (e.g., different CNN tiling configurations). Through two case studies, we demonstrate that Charm allows one to define high-level architecture models in a clean and concise format, maximize reusability and shareability, capture unreasonable assumptions, and significantly ease design space exploration at a high level.

CCS Concepts: • **Computing methodologies** → **Modeling methodologies**; • **Computer systems organization** → *Architectures*; • **General and reference** → Cross-computing tools and techniques;

Additional Key Words and Phrases: High-level models, modeling language, design space exploration

ACM Reference format:
Weilong Cui, Georgios Tzimpragos, Yu Tao, Joseph McMahan, Deeksha Dangwal, Nestan Tsiskaridze, George Michelogiannakis, Dilip P. Vasudevan, and Timothy Sherwood. 2019. Language Support for Navigating Architecture Design in Closed Form. *J. Emerg. Technol. Comput. Syst.* 16, 1, Article 9 (October 2019), 28 pages.
https://doi.org/10.1145/3360047

This is an extended version of the original conference paper, "Charm: A Language for Closed-Form High-Level Architecture Modeling," which appeared in the proceedings of ISCA 2018.
This material is based on work supported by the National Science Foundation under grants 1740352, 1730309, 1717779, 1563935, 1444481, 1341058, and 1660686, and gifts from Cisco Systems.
Authors' addresses: W. Cui, G. Tzimpragos, J. McMahan, D. Dangwal, N. Tsiskaridze, and T. Sherwood, University of California, Santa Barbara, Santa Barbara, CA 93106; Y. Tao, Peking University, 5 Yiheyuan Road, Haidian District, Beijing, China 100871; G. Michelogiannakis and D. P. Vasudevan, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720.

1550-4832/2019/10-ART9 $15.00
https://doi.org/10.1145/3360047

1 INTRODUCTION

Computer architecture is evolving into a field asked to evaluate a tremendous space of designs. From small embedded systems to warehouse-scale computing infrastructure and from well-characterized CMOS technology nodes to emerging devices at the edge of our understanding, computer architects are expected to be able to speak to the interdependent 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 and quantum architectures are to be considered. Although there are a great deal of 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 much less structured.

Careful application of detailed simulation can accurately estimate the potential of a specific microarchitecture, but exploration across higher-level questions always involves analytic models. For example, "Given some target cooling budget, how much more performance can I get out of an ASIC versus an FPGA for this application given my ASIC will be two technology 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. At the same time, when you break these questions down, they require a combination of a surprisingly complex set of assumptions or models. How do technology node and performance relate? What is the relationship between energy use and performance? ASIC and FPGA performance? Dynamic and leakage power? Any result computed from these relationships will rely on the specific set of 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 these relationships being executable to the degree necessary to explore a set of options (e.g., for a varying parameter such as total cooling budget). A thorough understanding of these high-level relationships not only provides insights for analyzing system tradeoffs but can also guide the search from an initial guess toward improved designs by efficiently exploring the design space.

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 get, the lack of a common engineering basis for these models has kept them from being swiftly and correctly constructed, understood, and applied toward guiding new system designs. Some models share a set of common relationships, but they redefine those symbols and equations with subtle differences that can be misleading; some have implicit constraints on one or more architectural quantities that may lead to pitfalls if not respected, such as a proper range of operating voltage. To automate the evaluation, one has to manually convert these mathematical equations to executable functions and handcraft many-fold for loops and interpretation logic, which can be tedious, error prone, and inefficient.

More importantly, as the design shifts toward a landscape of less understood ASICs and beyond Moore technologies, unlike traditional systems such as CPUs where architects have tens of years of existing designs and experience to draw upon, we usually are faced with a large set of free variables both at the architecture level (e.g., issue width, buffer sizes) and application level (e.g., neural network structure, tiling configuration), as well as a broad set of constraints (e.g., thermal limits, classification accuracy). The sheer number from the combinatorics of these parameters imposes a significant computational challenge when searching for an optimal design. Sweeping the vast space, even in closed-form only, incurs high costs in both time and resources.

To address these issues, we design and explore a declarative modeling language, Charm [20], that serves as a unified layer for the representation, execution, and optimization of closed-form

high-level architecture models. Charm provides a concise and natural abstraction to clearly express architectural relationships, automatically check model consistency, and easily declare analysis goals. Charm can also transparently search the design space for optimal configurations utilizing state-of-the-art constraint solvers. Building and evaluating closed-form high-level architecture models using Charm has the following major benefits:

Clarity through 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 and 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 shared in a variety of modeling scenarios.

Flexibility through Automation. Rather than treating the mathematical relations as functions, like in traditional programming languages, Charm keeps the abstraction at the mathematical level; hence, it is able to generate corresponding dataflow graph on-the-fly without requiring the user to rewrite the model when the same model is used for different high-level studies. To assist design space exploration, Charm also transparently transforms the system model into a satisfiability modulo theory (SMT) instance, if it is underdetermined (there are one or more free variables in the model), and utilizes SMT solvers (i.e., z3 [22] in our implementation) to efficiently explore the design space through bounding the infinite search space and approximation.

Safety through Type Checking. Charm enables new static and runtime checking capabilities on high-level architecture models by enforcing a type system. One example is that many architecturally meaningful variables have inherent physical bounds that they must satisfy; otherwise, although mathematically viable, the solution is not realistic. With the type system built in, Charm can dynamically check if all variables are within user-defined bounds to ensure a meaningful modeling result. The type system also helps prune the design space based on the bounds, without which a declarative analysis might end up wasting a huge amount of computing effort in less meaningful subspaces. Charm also incorporate physical unit as an optional part of variable definition and will check and convert physical units dynamically.

Efficiency through 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 nontrivial amount of time to naively evaluate the set of equations in 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, Charm promotes collaboration between application designers, computer architects, and hardware engineers because they can now share and refine models using the same formal specification and a common set of abstractions. For example, to reason about the energy consumption of an application on a platform, with his or her own application model written in Charm, the application developer will not have to implement an energy model from scratch and can simply plug in an existing one written also in Charm.

We release Charm as an open source tool on GitHub[1] to serve as a framework the architecture community can utilize to define and share analytical models. We also provide a wide collection of established architecture models from literature for quick use/reference, including the dark silicon model [24], a resource overhead model for implementing magic state distillation on surface

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

code [11], mechanistic CPU models [12, 26], a TCAM power model [2], the LogCA model for accelerators [5], the adder/multiplier models from PyRTL [16], a widely used CNN roofline model [73], dynamic power and area models for NoC [37], specifications of Xilinx 7-series FPGA [71], and the extended Hill-Marty model [21].

To describe Charm, we begin in Section 2 with a motivating example high-level model to show the problems with ad hoc modeling in practice. Then we introduce the design of Charm in Section 3, followed by two case studies demonstrating the application and benefits of building closed-form high-level architectural models with Charm in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 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 [24]. 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 Esmaeilzadeh et al. [24], 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.

2.1 A Brief Review of the Dark Silicon Model

To forecast the degree to which dark silicon will become prevalent on CMPs under process scaling, Esmaeilzadeh et al. [24] 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 that are essentially very different models: $CmpM_U$ and $CmpM_R$. $CmpM_U$ is an extension of the Hill-Marty CMP model [33], and $CmpM_R$ is a mechanistic model [28].

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 different topologies, scaling predictions, and core configurations. They then plot the scaling curves for the dynamic topology/$CmpM_R$ with both ITRS and conservative scaling predictions.

2.2 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 4.1). Figure 1 plots two variables explored as an example output of Charm. At a high level, we can see that the code is split into three major components: type definition (Lines 3–8[2]), model specification (Lines 11–52), and analysis declaration (Lines 55–61).

Specifically, we first define commonly used domains as Charm types on the architectural quantities that we care about (Lines 3–8). For example, the parallelism parameter in the model has a physical meaning related to 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. Although this is a simple example, more complex constraints are possible.

[2]All line numbers in Section 2 refer to Listing 1 unless otherwise specified.

```
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), (2.38, t=22), (3.21, t=16), (4.17, t=11), (3.85, t=8))
27     b = piecewise((1., t=45), (0.66, t=32), (0.54, t=22), (0.38, t=16), (0.25, t=11), (0.12, t=8))
28
29 # Pollock's Rule Extended with Power (CorM).
30 define ExtendedPollacksRule:
31     ref_core_performance : R+ as perf
32     ref_core_area : R+ as area
33     ref_core_power : R+ as power
34     area = 0.0152*perf**2 + 0.0265*perf + 7.4393
35     power = 0.0002*perf**3 + 0.0009*perf**2
36             + 0.3859*perf - 0.0301
37     perf < 50
38
39 # Amdahl's Law under Symmetric Multicore (CmpM_U).
40 define SymmetricAmdahl:
41     speedup : R+ as sp
42     core_performance : R+ as perf
43     core_area : R+ as a
44     core_power : R+ as power
45     core_num : R+ as N
46     chip_area : R+ as A
47     thermal_design_power : R+ as TDP
48     fraction_parallelism : Fraction as F
49     dark_silicon_ratio : Fraction as R
50     sp = 1 / ((1 - F) / perf + F / (perf * N))
51     N = min(floor(A / a), floor(TDP / power))
52     R * A = A - N * a
53
54 # Assumptions are now explicit and composable.
55 given ITRS, ExtendedPollacksRule, SymmetricAmdahl
56 assume chip_area = 111.0
57 assume thermal_design_power = 125.0
58 assume fraction_parallelism = [0.999, 0.99, 0.97, 0.95, 0.9, 0.8, 0.5]
59 assume tech_node = [45, 32, 22, 16, 11, 8]
60 assume ref_core_performance = linspace(0, 50, 0.05)
61 explore speedup, dark_silicon_ratio, core_num
```

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

We then formally specify (Lines 11–52) the three models (*DevM*, *CorM*, *CmpM*). Taking the *ExtendedPollacksRule* model (Lines 34–41) as an example, we declare up front all of 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

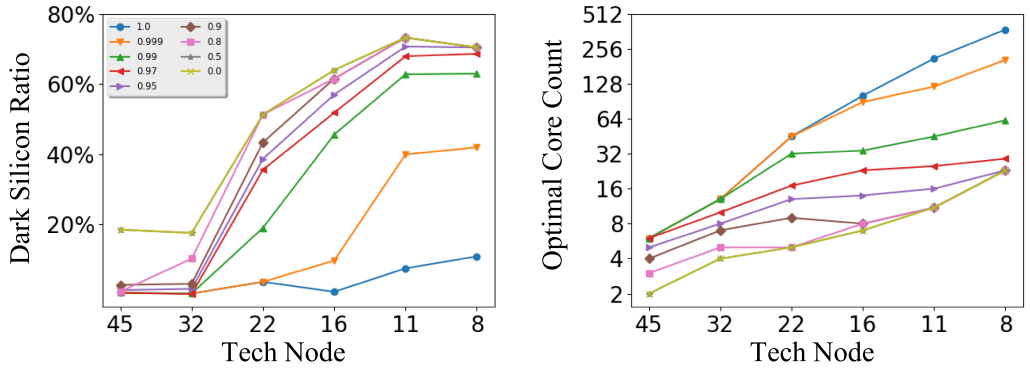

Fig. 1. Upper-bound ITRS scaling with symmetric topology. Each line corresponds to a different type of application characterized by its parallelizable portion of execution from 0 to 1.

the relationships between the architectural quantities (e.g., $area = 0.0152perf^2 + 0.0265perf + 7.4393$; the constants come directly from the original dark silicon paper [24]).

Once the models are defined, it is straightforward to declare the analysis in Charm (Lines 55–61). 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 55); we then provide values such as the area (Line 56) and power (Line 57) constraints; and finally we tell Charm what quantities we care to explore, in this case *speedup*, *dark_silicon_ratio*, and *core_num* (Line 61).

2.3 Pitfalls with Unstructured High-Level Architecture Modeling

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 ensure that it is mistake free) is a function of the degree of clarity, flexibility, safety (both type and unit in Charm), and automation supported by the tool, along with the complexity of the model under investigation. 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. Without a clear, unified representation, it takes nontrivial effort for one to figure out how to turn these models into executable code, and the clarity of the resulting code (in a traditional scripting language) is heavily dependent on the practice of good programming. To be more specific, ad hoc modeling ends up with the following issues.

Composition. It is hard to link the models' I/Os together or even check if the models can be connected properly at all. Architectural models are usually connected to each other through some common system parameters or physical quantities. The chain of data movement and dependencies among the dark silicon models is not explicitly exposed by the models. This issue of mismatched

[3]With all of the potential issues, unstructured methods in architectural modeling may not be as correct as one tends to believe [8, 53].

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 up front within each model and Charm "wires them up" through these channeling I/O variables. More importantly, Charm throws an error when the models cannot be properly linked.

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 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 (Lines 11–52) from the analysis procedure declaration (Lines 55–61). Such iterations over input values are declarative and transparent (as opposed to writing *for* loops, often many-fold, imperatively) by simply providing a list of values (Lines 58–60) in Charm.

Second, typical high-level architectural models are not "functions" but rather a set of mathematical *relationships*. The distinction is quite important. Traditional lvalue/rvalue style assignments (common to both functions and spreadsheets) create the following issue.

Restructuring and Reorientation. The models cannot be evaluated in a flexible way. Even though the model is a relation 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 for a given value of core performance. If the control quantity changes 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, that 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. However, in Charm, models *are* interpreted as a set of *mutually* dependent relationships without a fixed direction, and Charm runtime will generate the corresponding dataflow graph and needed functions based on the provided controls and quantities to explore.

Third, many complicated models do not have a full specification at early design cycles, and specifications of certain parameters, configurations, and design knobs are not deterministic by nature. Using traditional methods, the following issues come up.

Automatic Design Space Searching. Rather than having a fully specified system, architects usually are given a set of constraints (e.g., a target IPC, or TDP) and are expected to search the entire space, consisting of a large number of free variables, to find a viable configuration (for all architectural knobs) for further evaluation when designing complex or unconventional systems. Charm uses z3 [22] to transparently support such effort by transforming an underspecified model into an SMT instance. With any (or all) of the assumptions (Lines 56–60) taken away, Charm automatically detects the free variables, transforms the relationships into bounded SMT instances, and calls out to z3 to search for a viable configuration (if there exists a satisfiable solution). Furthermore, Charm can also iteratively optimize the design by tightening constraints in the SMT instance (e.g., lower bound on performance, upper bound on power) on-the-fly.

Reasoning under Uncertainty. Architectural models usually involve some uncertainties [21], such as how technology may scale over the next 10 to 15 years. It is natural for computer architects to first evaluate the model with some concrete values (e.g., the scaling factors in Lines 26 and 27)

and then model the uncertain quantity as some distribution (e.g., Gaussian distribution), as in our case studies in Section 4. It requires nontrivial programming efforts with spreadsheets and scripting languages to support uncertain random variables. Charm supports different forms of input values such as scalars, vectors, and distributions by design to greatly ease modeling and exploring with uncertainties in mind.

Fourth, 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 *mathematically* valid, but it *will* lead to meaningless and misleading results 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 a range of operation. Aside from the physical constraints, implicit domain constraints also come from how the model is built in the 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, the result derived from that is much less accurate and trusted. These 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 reimplement the model from natural language descriptions (e.g., in a published paper). To address this issue, Charm encourages model specification to include these implicit constraints explicitly as constraints built in the model specifications (e.g., Line 37), then automatically checks if these constraints are violated during evaluation.

Unbounded Distributions. Many architectural quantities, such as *core frequency*, follow normal distribution due to process variability [46, 54, 56]. However, the use of these types of unbounded distributions sometimes violates the physical constraints of the quantity (e.g., *frequency* must be positive). In unstructured modeling, this check is completely up to the user and, if overlooked, will lead to incorrect 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 configuration 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 nontrivial amount of time. It only becomes worse when one tries to evaluate models with uncertainties [21]. 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 on the model builders and others who wish to use existing models through reimplementation. As we show in Section 4, for both fully determined systems (with the invariant hoisting and memoization techniques) and underdetermined ones (through SMT), Charm greatly speeds up the exploration without additional effort from the model builders.

Although the preceding code example is for a mature general-purpose chip-multiprocessor model, Charm is definitely not limited by the type of technology that the model represents. As long as the models can be expressed in closed form, Charm can be applied to encode analysis with such models. Another simple snippet to showcase Charm's expressiveness is given in Listing 2. We also actively use Charm to manage many different analytical models including data-center performance models (similar to Kaanellou et al. [38]) and cost models for emerging race-logic architecture for decision trees [66].

```
Define BenefitModel:
    AverageNumCacheLines : I+ as N
    AverageCacheMissRate : R+ as MR
    CacheLineSize : I+ as S
    SIMDWidth : I+ as W
    NumStoreInsts : I as NI
    WordSize : I+ as w
    Benefit : R+ as B

    B = ceiling(N * MR) * S * W + NI * w * W
```

Listing 2. Code snippet to encode equation from Section 7.2 in Kim et al. [39].

3 CHARM DESIGN

Charm provides a simple, domain-specific modeling language to express both closed-form models and the design space exploration logic. 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 nonlinear algebra, like exponentiation. Basic non-closed-form functions like summation and product are also supported. To extend the design space exploration to uncertain domains, Charm also supports distributional values to be set and propagated through the models transparently. If the system is fully determined, the interpreter is able to transform the mathematical relationships into a series of dataflow graphs for fast evaluation; otherwise, an SMT instance is created by transforming the relations in the model into proper domains (bit vectors and floating points in our implementation). A type system is applied to ensure that all architecturally meaningful quantities operate in the correct domain and provide type conversion guidance when transforming into the SMT domain. Charm also optimizes the design space exploration procedure using compiler techniques to eliminate redundant computation.

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

3.1 Language Abstractions

To address all of the issues described in Section 2.3, Charm provides a common layer with the following three key abstractions: *types*, *models*, and *analysis*.

Charm language is strongly typed. To express the *type* of a variable, the keyword *typedef* must be used. The model abstraction enforces explicit type declaration to ensure that no implicit assumptions about data types and domains across models exist. 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 Lines 3 and 4). There are only two base types, *Real* and *Integer*, standing for real and integer numbers, respectively. Internally, real numbers are represented by *float* and integers by *int*.

To construct a *model*, the keyword that must be used is *define*. More specifically, a model specification in Charm encapsulates the following two pieces in a high-level architecture model:

- ***A Set of Variables.*** Each variable has a universally unique and consistent full name. Each variable also has a local short name (optional), as well as an explicitly declared type and physical unit (optional, e.g., chip_area : R+ in μm^2). For instance, in Listing 1 line 31, the short name is "*perf*"; the type is "R+", which means positive real number; and the long name is "*ref_core_performance.*" Short names live only within the definition of a model, whereas full names are exported to other models, as well as to the analysis logic.
- ***A Set of Relations.*** In Charm, both equations and inequalities are considered *relations*. Relations define mathematical relationships between variables using either their full or short names (e.g., Listing 1 Lines 34–35 and 50–52). Both linear and nonlinear systems are

present in architectural models that we care about. The general problem of determining the definability of and solving such systems is theoretically hard and beyond the scope of this work. Given the limitations and solving capabilities of existing backend solvers, some very complicated nonlinear equations cannot be symbolically solved (e.g., Charm will throw an error if one tries to solve for x in $y = (a^{1/x})^{2^x}$). Fortunately, however, we find that most models computer architects care about are well within the limit. Equations can also bind variables to constant quantities as assumptions defined in the model specification (e.g., $kBoltzmann = 8.6173303 \times 10^{-5}$). Relations can be value generative, if values of all but one variable are given. A relation can also be nonvalue generative, which includes all inequalities and equations that have all of their variables assigned a value.

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

Before computation starts, *given* statement selects the model(s) to be used in the analysis. If multiple models are selected, they are linked together automatically by the interpreter using the full names of their variables.

In general, many algebra systems can be solved without additional assumptions. However, it is also common for computer architecture models to have some control quantities (e.g., design options, like *core size*, and system configurations, like *cache associativity*) given by system designers. The keyword *assume* serves such purpose by differentiating assignment equal signs from the mathematical equal signs found inside the model specification. For instance, *assume* statements are assignments much like in other programming languages, whereas equations in model specification are merely mathematical relationships that do not imply a direction of data movement. Charm also constrains *assume* statements to be assignment with constants. For example, they can only be used to express external inputs to the model rather than defining additional relations outside of the model specification.

Moreover, Charm supports both scalar and vector value assignments, as well as values derived from commonly used distributions, such as *Gaussian distribution*. More specifically, iterations can be expressed either in a Pythonic list-like syntax or with functions that generate a list, such as *linspace*, and they are assigned to some input variable just like a normal *assume* statement (e.g., Listing 1 Line 60). Charm handles iteration naturally by selecting combinations of all iterative input values nonrepeatedly 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 a 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 the special "list" notation after its variable name (e.g., *assume* $L[] = [1, 2]$, which means $L[0] = 1$ and $L[1] = 2$).

Finally, to complete the analysis specification, the variables that we would like to solve for should be provided. For that purpose, the keyword *explore* must be used. Given this information, Charm transforms the undirected dependency graph (representing the way the selected models are connected) into a directed acyclic function graph, and then it starts the evaluation process by propagating the given values (via *assume* statements or assumptions in model specification) through the DAG.

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

3.2 Language Internals

To evaluate the models and optimize the evaluation logic, Charm internally uses two graph structures to represent and transform the computation. Figure 4 graphically shows the interpretation process. In this section, we first define the core graph data structures and then describe how we

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

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.

can perform type checking, function generation, evaluation, and optimization with these graph structures.

Dependency Graph. A dependency graph is a bipartite graph $G = \langle V_{var}, V_{rel}, E \rangle$, 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.

Function Graph. A function graph is a *directed acyclic* dataflow graph D for a determined system, in which

- Every node in V_{var} has at most one incoming edge (i.e., its in-degree being 0 or 1).
- Every node in V_{eq} has at most one 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 variable with a unique full name and an equation/constraint node to every equation/constraint. When creating relation node, Charm creates an edge between the equation/constraint node and a variable node if the variable appears in the equation/constraint. Finally, Charm scans the analysis statements and marks variable nodes being assigned as input nodes.

Physical unit check and conversion is performed as part of the type checking process. Charm uses Pint [1] as a backend engine to check type consistency and modify the relations when parsing based on the necessary unit conversion rate (e.g., mm to μm). Charm will throw an error if the units in one relation cannot be converted into a single base unit.

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,

$$C, D, E, \Omega \in \mathit{RelationSet} \quad \Gamma \in \mathit{TypeEnvironment} = \mathit{Name} \to \mathit{RelationSet}$$
$$\Theta \in \mathit{RuleEnvironment} = \mathit{Name} \to \mathit{RelationSet} \quad \mu \in \mathit{VariableMap} = \mathit{Name} \to \mathit{Value}$$

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

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

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

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

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, whereass `Ext` returns all extensions of a variable (portion of the name appearing after a dot when multi-instanced). `isConsistent` ensures that the relation set is consistent. `isFullyDetermined` ensures that 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 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 3.2). 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.

Charm checks that their defined types are identical (both base type and constraints associated). For inconsistent types, Charm aborts execution and issues an error message.

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 3 Lines 5 and 6. Both of them follow the equation shown in Listing 1 Line 23 when plugged in for evaluation. We discuss their physical meanings later in Section 4.1, but they are essentially two variables following the same mathematical relationship. We refer to 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.

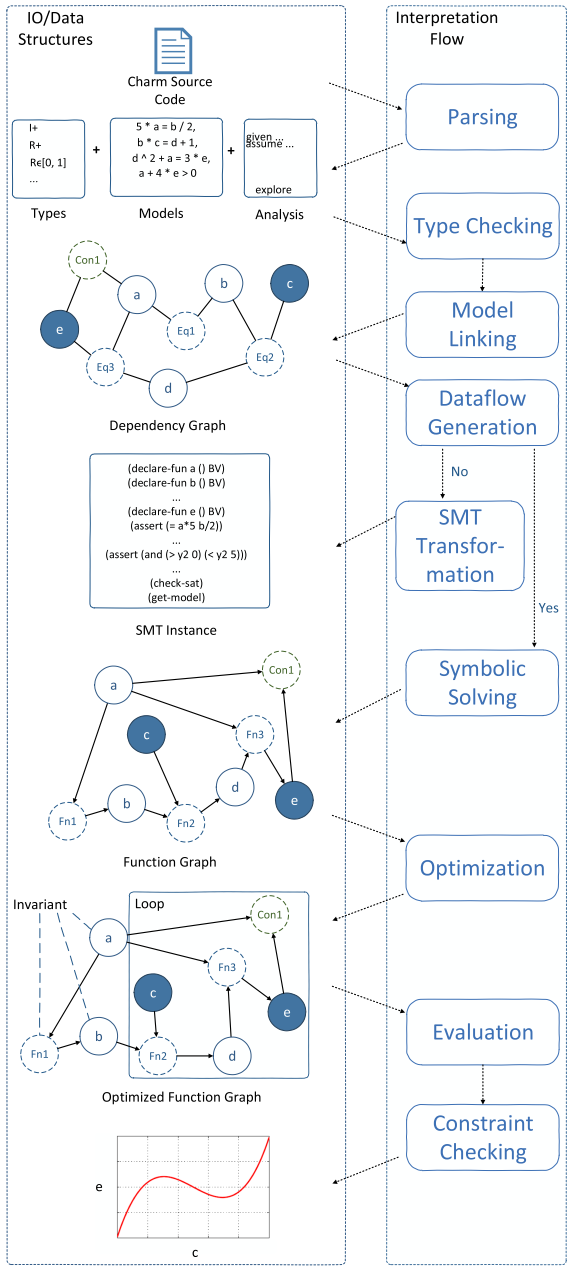

Fig. 4. Overview of the Charm interpreter. The parser takes Charm source code and breaks it into a set of types, a set of model definitions, and a set of analysis statements. Charm then links types, models, and assignments in a dependency graph after they go through the type checker. The graph then is fed to a function generator and a symbolic solver to convert it into a function graph. If the conversion is successful, it then gets evaluated after optimization and checked against model constraints. If the conversion from dependency graph to function graph is unsuccessful, Charm then transparently generates an SMT instance based on the variables and relations in the model specification and calls out to the SMT solver to find a possible solution.

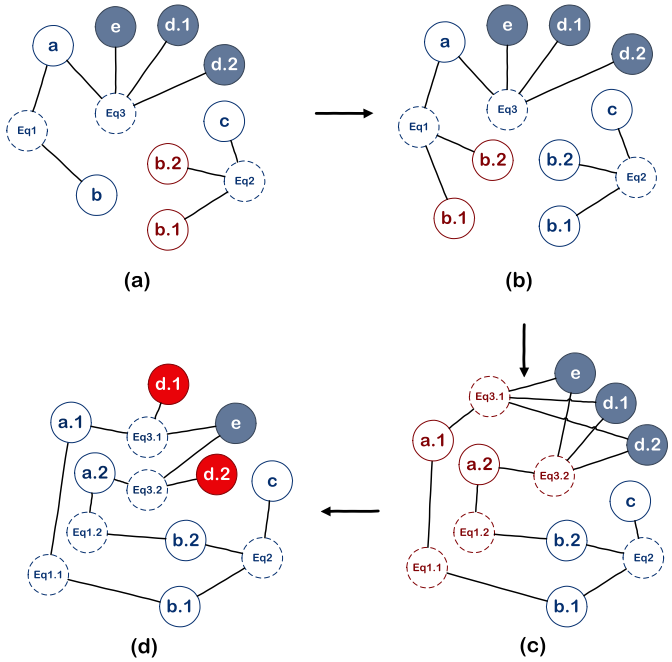

Fig. 5. Relation multi-instancing when generating a dependency graph. (a) The initial graph has extended names ($b.1$, $b.2$). (b) Charm finds and splits the corresponding base name node (b). (c) Charm propagates the multi-instancing. For instance, all nodes along the path to the base name node (b) are also split to create different "instances" of relationships for different quantities that follow the same relationship definition (in this case, $Eq1$ and $Eq3$). 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 nonconsistent 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 ($d.2$, $Eq3.1$).

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 while trying to invoke multi-instancing.

Function Graph Building and Function Generation. After building the dependency graph G, we try to convert G into a function graph F. If this conversion is successful, by the definition of function graph, the system is fully- or overdetermined and Charm uses Sympy [62] as the backend 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 relation and output being the variable pointed at by the equation node (inequalities do not generate values). Otherwise, the system is underdetermined and Charm generates an SMT instance (after unit check and conversion) from the specification of the models and feeds it to the SMT solver. As part of the dynamic type checking, each variable is also associated with the bounds from its type while being evaluated.

The conversion from G to F takes three steps:

- The first step builds an intermediate graph $R = G - V_{input} - E_{input}$. For instance, we take out all variable nodes that are treated as inputs to the system (values provided in *assume* statements by the user) and edges associated with them.

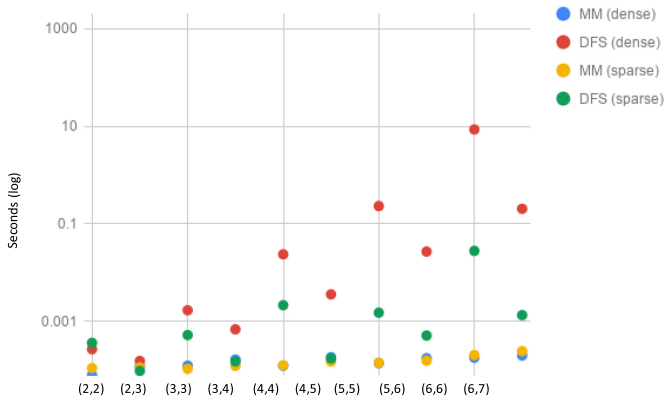

Fig. 6. Scalability of the conversion from dependency graph to function graph. The *x*-axis is the number of (equations, variables) in the system model. We use random dense graphs (with a density around 0.4) and sparse graphs (with a density around 0.2) to evaluate the time overhead of the graph transformation. In dense graphs, each equation can connect up to all of the variables, whereas in sparse graphs, each equation can connect up to half of the variables, and hence fewer dependencies.

- In the second step, we try to convert R to a function graph F. The space of all possible labeling of edges is $2^{|E_R|}$, where E_R is the edge set of R. Fortunately, given the constraints from the definition of function graph, the conversion problem here maps to the problem of finding a *maximum matching* of R and checking that it has a matching number equal to $|V_{var}|$ of R. Finding such a maximum matching in a bipartite graph is a classic problem in graph theory and has a polynomial-time heuristic algorithm [35]. In our implementation, we use networkx [29] to find the maximum matching. Figure 6 shows the scalability of this conversion compared to a DFS-based algorithm we initially implemented on randomly generated dependency graphs [20]. As we can tell, the conversion cost using the new matching algorithm scales linearly as the graph gets larger, whereas the searching algorithm we had before has an exponentially growing cost. This difference matches the expectation from the theoretic complexity of the two algorithms ($O(|E|)$ for matching vs. $2^{|E|}$ for searching, where E is the set of edges in the dependency graph). Hence, in our case studies in Section 4, we see a huge reduction in terms of the execution time of interpretation (i.e., the time cost of performing analysis with Charm).
- Last, F must be *acyclic* to evaluate. However, when there are codependent equations, they form cycles after the conversion. In case of a cycle, all equation nodes in the cycle must be converted to functions altogether. We pass the equations in a cycle to the backend solver at once and then replace the cycle with pairs consisting of a function node and a variable node; 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 of the variable nodes in the cycle. Each function node generated by the cycle elimination has one unique variable in the cycle as its output. All functions generated by this step are from the same set of equations, only with different variables as inputs/outputs. Figure 7 shows an example of cycle elimination in F.

Computational Constraints. A special computational constraint is applied when building a function graph: some mathematical operators are not reversible or have infinite solutions, such as $\sum$ and $\prod$. In addition, some are computationally hard for the solver, like solving x in $y = (a^{1/x})^{2^x}$.

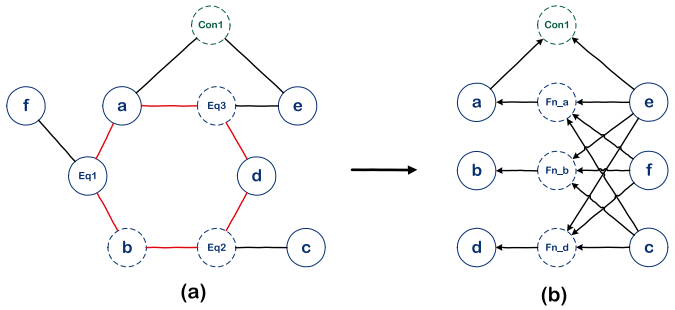

Fig. 7. Cycle elimination when generating a function graph. Equations in a cycle are solved at once and are replaced with three functions, each of which generates a different variable value.

For the nonreversible equation, its direction is fixed. For instance, its edges have fixed direction and are not subject to the function converter.

Evaluation and Constraint Checking. Once we have a viable function 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.

Performance Optimization. Oftentimes, architects explore the relationship between two variables by iterating over different input values. One simple yet effective performance optimization is invariant hoisting. With the function graph structure, it is straightforward to optimize for invariant hoisting in Charm. From each iterative variable node, Charm simply traverses the graph from that node. 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 no paths from c to them.

Each function node also caches a mapping table between inputs and its output. Such memoization optimizes away costly recomputation over the same set of input values.

Transforming an Underdetermined System into an SMT Problem. If the conversion from dependency graph to function graph fails (i.e., the algorithm does not find a matching with a size equal to the number of variables in R), Charm aggregates all relations and exports a quantifier-free SMT instance to z3. It is well known that some of the SMT theories are undecidable, but, fortunately, the theory of bit vectors and floating points is bounded and decidable. We approximate each real variable in the model into a floating point (fp) using the IEEE 754 encoding since most of time we do not need infinite precision (e.g., we typically do not need an IPC to the 10th decimal), and each integer variable is approximated by a bit vector (bv). The transformation also bounds the search space by the number of bits we use. We set 32 bits as the default length because it achieves good balance between the applicable range (most design configurations fit within the range) and synthesis speed. The computation is done in the fp domain, and we dynamically cast bv to/from fp values by rounding to the nearest even number. We find in our experiments that this approximation works well for a wide variety of architecture models.

Optimizing Underdetermined Systems. In a complex system design space, sometimes a manual search (even from a synthesized configuration) is not favorable because it can quickly become tedious and inefficient if each iteration requires a combination of hundreds of parameters. With the SMT solver, the optimization can be automatized by iteratively tightening/loosing bounds of the system constraints (e.g., iteratively asking the question "Is there a configuration that has better

```
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 3. Asymmetric model and the changes in code.

performance?" or "Is there a configuration that consumes less power/energy?"). To quantify how much better each iteration should be targeting, Charm allows users to specify a step coefficient (e.g., speedup@0.1) to control the granularity of the search.

4 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 execution times with and without optimization. The second case study demonstrates how Charm deals with underspecified system models and assists design space exploration with the use of SMT solvers. We use a well-cited analytic model for convolutional neural networks (CNNs) [74] along with the roofline models of a set of FPGA platforms to explore tiling configurations for different CNN architectures.

4.1 Dark Silicon and Beyond

Listing 3 highlights all of the changes we need to implement in Charm to model and switch the exploration from symmetric topology to asymmetric. Note that in the asymmetric model, "relation multi-instancing" comes in handy when expressing two coexisting types of a core. To switch the analysis, all we need to do is to change the models that are *given* (Listing 3 Line 22) and provide values for two types of cores instead of one (Listing 3 Lines 23 and 24). We also write a new constraint (Listing 3 Line 20) to specify the fact that the big core should have better performance than the small core.

It is even simpler to switch from ITRS scaling predictions to conservative predictions [10]. Listing 4 shows all of the changes needed. Figure 8 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?" In that case, we can explore the design space with a distribution of scaling factors. We use a Gaussian distribution for the scaling factor, the mean of which is set to the average value of the two predictions and the standard deviation set to the difference between the mean and the predictions. Listing 5 shows the necessary changes in Charm code. It

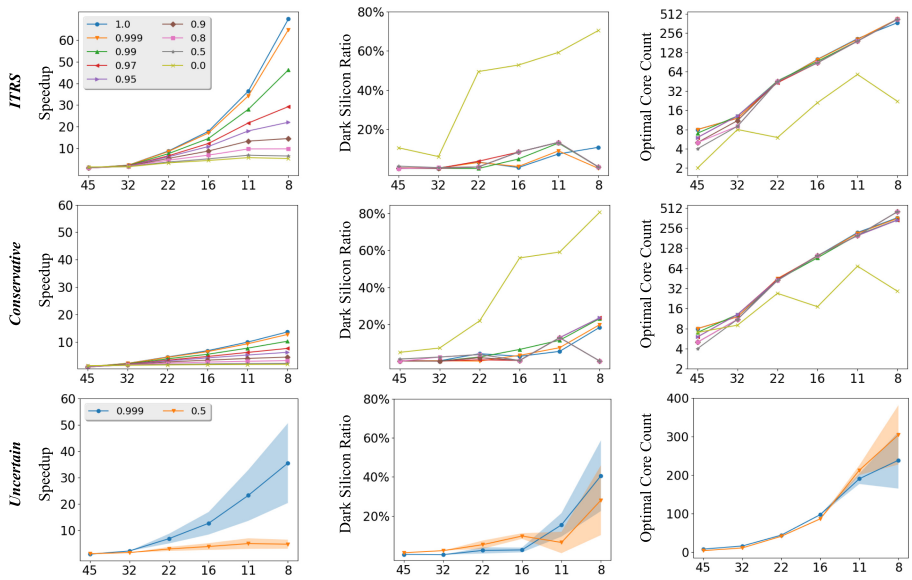

Fig. 8. Upper-bound scaling with asymmetric topology with a tech node on the *x*-axis. Note that the last figure of optimal core count has a linear-scale *y*-axis to better demonstrate the variance. For clarity, we only plot two regions in the uncertain scaling results, but the trends for other f values are similar.

```
# 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 4. Conservative scaling and the changes in code. This definition would replace ITRS in Listing 3.

```
# 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 5. Uncertain scaling and the changes in code.

is important to note that although the Gaussian distribution is not bounded, the scaling factors have a bounded domain. The type checking in Charm ensures that the scaling factors a and b operate only in their defined domains (see Listing 1 Lines 20 and 21) and the provided Gaussian distribution is converted to a truncated Gaussian distribution with the same mean and standard deviation within Charm. From Figure 8, we can see that with technology scaling, the more parallel

workload (with an f close to 1) shows higher sensitivity toward technology uncertainties, whereas the more serial workload is less sensitive to changes in 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 11 nm and beyond. The uncertainty grows sharply from 16 nm to 11 nm mainly because below 11 nm, 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 technology node scales to 11 nm and beyond, the CMP becomes *power bounded* and is extremely sensitive to the power uncertainties propagated from the uncertainty of the power scaling factor.

In terms of execution performance, we compare Charm execution to an unoptimized baseline in which all computation is redone per iteration (similar to a straightforward implementation with other scripting languages without additional programmer effort, i.e., no invariant hoisting nor memoization). For ITRS or conservative scaling with an asymmetric topology (a design space of 150K design points), full-blown Charm finishes on average within 120.5 seconds, whereas the unoptimized implementation uses 159.5 seconds (1.3x speedup). For the uncertain scaling with a MC sample size of 200 ($\tilde{1}$ million design points), optimized Charm uses 1,562.5 seconds, whereas it takes 5,703.1 seconds for the baseline implementation (3.6x speedup) on a single Intel i7 core at 3.3 GHz to finish. We expect the speedup to only grow dramatically as the design space gets larger because the redundant computation will scale exponentially with the number of parameters.

4.2 Exploration of CNN Tiling on FPGAs

In this case study, we explore an underdetermined system with Charm. To evaluate CNN tiling on a set of Xilinx FPGA boards analytically, we take the analytical model from previous work [74] and use it to demonstrate the SAT-based searching capabilities of Charm. We let Charm automatically explore the configuration space for tiling and discover several new findings in this parameter space. We first describe the model for CNNs and the FPGA boards we are targeting. Then we ask this question: What is the optimal tiling configuration of a CNN architecture to maximize performance on a specific FPGA board?

4.2.1 CNN and FPGA Models. The roofline model is an intuitive tool used to visualize a design's performance under the constraints of the platform's peak performance and maximum bandwidth [69]. To enable simultaneous multiplatform exploration, we allow an overlapping of a variety of "rooflines." Moreover, besides the compute and bandwidth ceilings defined by the platform's specifications, Charm allows the user to set "floors": the lowest acceptable performance and the computation-to-communication ratio, which is directly related to the energy cost per operation.

For CNNs, the convolution layers take up most of the computation time [18]. Many methods for efficient implementations of these networks on hardware have been proposed, building FPGA accelerators being one of them. Zhang et al. [74] build a roofline model based on memory bandwidth and logic resources with tiling:

$$\begin{aligned} \textit{computational roof} &= \frac{\textit{total \# of operations}}{\textit{\# of execution cycles}} \\ &\approx \frac{2 \times R \times C \times M \times N \times K \times K}{\left\lceil \frac{M}{T_m} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times R \times C \times K \times K} \end{aligned} \tag{1}$$

$$\begin{aligned} \textit{CTC ratio} &= \frac{\textit{total \# of operations}}{\textit{total external data access}} \\ &= \frac{2 \times R \times C \times M \times N \times K \times K}{\alpha_{in} \times B_{in} + \alpha_{wght} + \alpha_{out} \times B_{out}}, \end{aligned} \tag{2}$$

```
define xilinx_xc7vh870t_3:
    computation : R+ as cmpt
    computation_roof : R+ as roof
    bandwidth : R+ as bw
    bram_usage : R+
    roof = 3734.0
    cmpt <= roof
    bw <= 420.0/8
    bram_usage <= 50760.0 * 1000 * 1000/8
```

Listing 6. Hardware constraints from an FPGA board.

where

$$B_{in} = T_n(ST_r + K - S)(ST_c + K - S) \tag{3}$$

$$B_{wght} = T_m T_n K^2 \tag{4}$$

$$B_{out} = T_m T_r T_c \tag{5}$$

$$0 < B_{in} + B_{wght} + B_{out} < BRAM_{capacity} \tag{6}$$

$$\alpha_{in} = \alpha_{weight} = \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c} \tag{7}$$

$$\alpha_{out} = \frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c}. \tag{8}$$

Here, the tiling design space consists of tile dimensions T_r, T_c, T_m, and T_n. These parameters are bounded by the corresponding dimensions of the NN structure. $\alpha_{in}, \alpha_{out}, \alpha_{wght}$ and $B_{in}, B_{out}, B_{wght}$ denote the trip counts and buffer sizes for accesses to the input feature maps, output feature maps, and the weights, respectively.

With the help of this analytic model, we can now explore the tiling configuration space given any platform in the roofline space. An example FPGA board (xilinx_xc7vh870t_3) in Charm is presented in Listing 6. The full Charm code for the CNN model is shown in Listing 7.

4.2.2 Optimizing Design under Constraints. Here, we try to find an optimal configuration, in terms of (a) performance, (b) bandwidth requirement, and (c) both performance and bandwidth requirement, by generating constraints on-the-fly while exploring the design space.

We use the CNN model of AlexNet [40]. AlexNet consists of five convolution and three fully connected layers. For this experiment, we consider only the convolution part of this model. We look for the optimal (T_m, T_n, T_r, T_c) configuration for the second layer of the CNN model by dynamically adding lower-bound performance and/or upper-bound bandwidth constraints to make the SMT solver find a better configuration iteratively.

Figure 9 presents the search result. We can tell that the lower-/upper-bound constraints quickly guide the search to a more interesting area. By utilizing a state-of-the-art SMT solver, without additional effort from the model builder, Charm greatly reduces the time spent on sweeping the design parameters. In the preceding evaluation, a brute force search over the space ($\langle T_m \times T_n \times T_r \times T_c \rangle$, Lines 38–45 in Listing 7) takes more than 5 hours to finish, whereas automatic exploration finds optimality in roughly 30 minutes, achieving more than 10x speedup. We expect this speedup to only grow as the model gets more complicated with more than four parameters to search for. Figure 10 transforms the roofline space into a performance-bandwidth space.

```
define CNN:
    R : I+
    C : I+
    M : I+
    N : I+
    K : I+
    T_m : I+
    T_n : I+
    computation : R+ as comp
    comp = (2 * M * N) / (ceiling(M / T_m) *
        ceiling(N / T_n))

    a_in : R+
    a_weight : R+
    a_out : R+
    B_in : I+
    B_weight : I+
    B_out : I+
    T_r : I+
    T_c : I+
    S : I+
    bram_usage : R+ as bram
    computation_to_communication_ratio : R+ as ctc
    bandwidth : R+ as bw
    a_in = (M * N * R * C) / (T_m * T_n * T_r * T_c)
    a_weight = a_in
    a_out = (M * R * C) / (T_m * T_r * T_c)
    B_in = T_n * (S * T_r + K - S) *
        (S * T_c + K - S)
    B_weight = T_m * T_n * K * K
    B_out = T_m * T_r * T_c
    bram = B_in + B_weight + B_out
    bw = comp / ctc
    ctc = (R * C * M * N * K * K) /
        (a_in * B_in + a_weight * B_weight +
            a_out * B_out)

    T_m > 0
    T_n > 0
    T_r > 0
    T_c > 0
    T_m <= M
    T_n <= N
    T_r <= R
    T_c <= C
```

Listing 7. CNN roofline model.

There are a few interesting observations from these two viewpoints of the space:

- From Figure 10, we can tell that performance does not have a clear linear correlation with bandwidth. For instance, to achieve better performance, a balance between processing resource and communication is more important rather than simply dumping logic transistors or using wide communication channels.
- The preceding said balance can be seen from the most performant few configurations. All of these configurations are "asymmetric" and retain some ratio between T_m and T_n (a possible interpretation maps to the on-chip processing resource and the communication channel width). As this ratio deviates from the "balance," performance degrades as we can tell from the configuration of "BW only" and "Perf + BW."
- From Figure 10, we can tell that this FPGA board is clearly bounded by its computation resources rather than communication bandwidth as designs within less than 99% of the computation roof consume less than half of the available bandwidth.

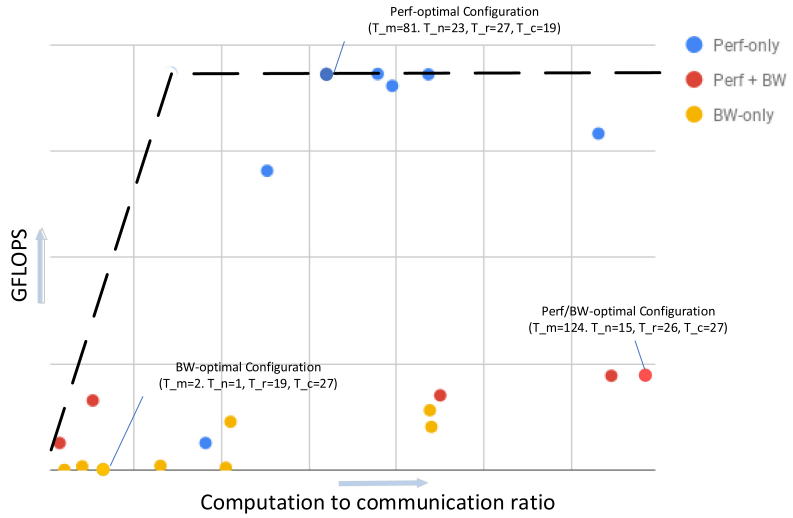

Fig. 9. Finding the optimal tile configuration for the second convolution layer of AlexNet under the roofline constraint when targeting xc7vh870t. "Perf-only" is the exploration trajectory when we only optimize for performance, whereas "Perf + BW" shows the trajectory when we simultaneously optimize for both performance and the computation to communication overhead. In the first case, the SMT instance is iteratively bounded toward the upper part of the graph (i.e., the subspace with higher performance), whereas the second case pushes the search to the upper-right corner where designs have both better performance and a lower requirement on bandwidth.

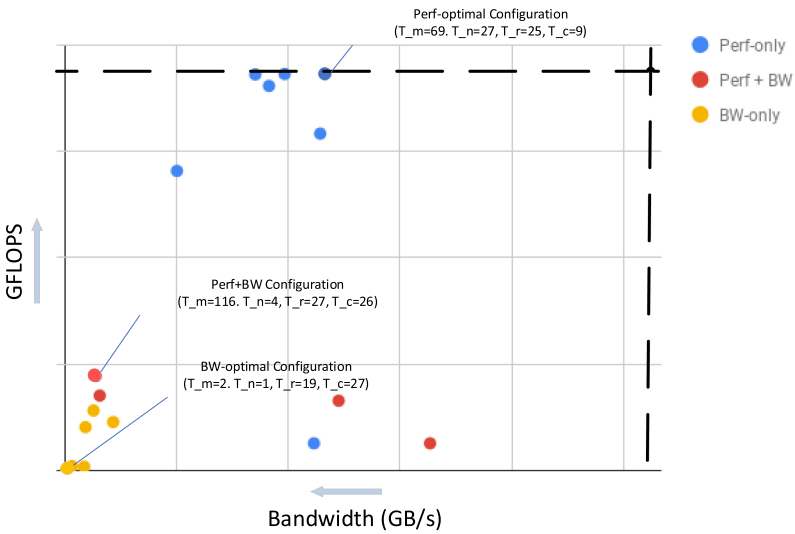

Fig. 10. Projecting the search results into the performance-bandwidth space.

5 RELATED WORK

5.1 Closed-Form Architecture Models

Many of the recently developed high-level analytical models are conceptually inherent from Amdahl's law [7], 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 [33]. A long line of research work using extensions of their closed-form model focuses on different aspects of the system, including application [61], communication and synchronization [25, 72], energy and power consumption [24, 70], heterogeneity [5, 15], chip reliability [59], architectural risk [21], 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 preceding models.

Another set of analytical performance models are built directly from the mechanisms of the specific system [12–14, 26, 27, 34, 51, 58]. These models usually rely on simulations or hardware counters to collect the necessary inputs to their core closed-form equations. Our language can also express and manage these equations. Moreover, empirical modeling [9, 36, 41–44] is used to discover correlation between two or more architectural quantities. These correlations can usually be expressed as parameterized equations in closed form. The resulting models of such empirical methods can also be managed by and benefit from Charm.

5.2 Systems and Languages Supporting Analytical Modeling

There exist systems and languages that support structured analytical modeling. Modelica [23] supports multidomain analytical modeling with an emphasis on object-oriented model composition, but the connection of models needs to be explicitly dictated and the design space exploration requires user intervention. However, Charm is more restricted, and thus it is able to automatically link models and generate exploration loops. Aspen [60] provides a DSL to express applications along with an abstract machine organization to model performance. Palm [64] utilizes source code annotation to build analytical models for target applications. LSE [68] 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 automatically performing performance modeling [3, 4, 67]. Most of these languages and systems serve a different purpose of expressing the mapping between performance/power models and specific detailed application/architecture and are not well suited for high-level analytical design space exploration. In contrast, Charm is tailored for structured yet flexible exploration of the interactions between architectural variables and their ramifications at a high level. There are also a few systems exploiting the power of symbolic execution for modeling [8, 21], but Charm provides more capabilities around formalizing, checking, and evaluating the models. There also exists a tool [65] 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 dataflow-centered programming languages in the field of incremental/reactive programming [30, 31, 45, 48, 63] but differ in that Charm is highly restrictive. This restrictiveness means that Charm is more of a modeling language than a programming language. For instance, 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).

5.3 Exploring Design Space with Constraint Solving

Mohanty et al. [50] use constraint solving as a first step to prune the entire design space of embedded SoCs at a coarse granularity for later evaluation. CoBaSA [49] compiles the component-based software development design space and system constraints into a pseudo-Boolean satisfiability problem to find feasible solutions with a large number of constraints. Haubelt et al. [32] encode the system synthesis problem into an SAT instance to find a feasible binding between processes and resources. Regarding the use of SMT solvers [47, 52, 55, 57], a plethora of research work has explored task scheduling and resource management using an SMT solver including methods with

high-level language or custom DSL as the frontend [52, 57]. In Charm, we use the SMT solver to explore the design space formed by analytic architecture models. We utilize the theories of bit vectors and floating-point numbers to bound the infinite design space and approximate the architectural quantities.

6 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 critical for guiding the processes of design and evaluation than ever before. Already today we are seeing machine learning [19], cryptography [6], 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 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. Although our language already supports symbolic restructuring, memoization, hoisting (and several other optimizations), consistency checks, and the capability of automatic exploration, Charm is merely the first step toward 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 vs. pJ errors) or backends connecting models to nonlinear optimizers. Most importantly though, by giving the sets of mutually dependent architectural relationships a common language, Charm, along with the collection of established models, has 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 ahead, we see that tools such as this hold significant promise in enabling more collaborative and community-driven efforts that can make our best thinking on the future of architecture more readily and easily accessible to all who are interested. Furthermore, although Charm focuses on facilitating models analysis, we believe that by combining Charm with other hardware design toolsets (e.g., PyRTL [17]), we can automate the process of developing actual hardware designs directly from performance studies (e.g., via templates).

REFERENCES

[1] Python Software Foundation. 2017. Pint: Makes Units Easy. Retrieved September 23, 2019 from https://pypi.org/project/Pint/.

[2] B. Agrawal and T. Sherwood. 2006. Modeling TCAM power for next generation network devices. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*. 120–129. DOI: https://doi.org/10.1109/ISPASS.2006.1620796

[3] Sadaf R. Alam and Jeffrey S. Vetter. 2006. A framework to develop symbolic performance models of parallel applications. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE, Los Alamitos, CA, 320–320. http://dl.acm.org/citation.cfm?id=1898699.1898852

[4] Sadaf R. Alam and Jeffrey S. Vetter. 2006. Hierarchical model validation of symbolic performance models of scientific kernels. In *Proceedings of the European Conference on Parallel Processing*. 65–77.

[5] Muhammad Shoaib Bin Altaf and David A. Wood. 2017. LogCA: A high-level performance model for hardware accelerators. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 375–388. DOI: https://doi.org/10.1145/3079856.3080216

[6] Joël Alwen and Jeremiah Blocki. 2016. Efficiently computing data-independent memory-hard functions. In *Proceedings of the Annual Cryptology Conference*. 241–271.

[7] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference (AFIPS'67 (Spring))*. ACM, New York, NY, 483–485. DOI: https://doi.org/10.1145/1465482.1465560

[8] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. 2004. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE, Los Alamitos, CA, 439–448. http://dl.acm.org/citation.cfm?id=998675.999448

[9] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. 2010. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 26–36. DOI: https://doi.org/10.1145/1815961.1815967

[10] Shekhar Borkar. 2010. The exascale challenge. In *Proceedings of 2010 International Symposium on VLSI Design, Automation, and Test*. 2–3. DOI: https://doi.org/10.1109/VDAT.2010.5496640

[11] Sergey Bravyi and Jeongwan Haah. 2012. Magic-state distillation with low overhead. *Physical Review A* 86, 5 (2012), 052329.

[12] M. Breughe, S. Eyerman, and L. Eeckhout. 2012. A mechanistic performance model for superscalar in-order processors. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems Software*. 14–24. DOI: https://doi.org/10.1109/ISPASS.2012.6189202

[13] D. Brooks, V. Tiwari, and M. Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of 27th International Symposium on Computer Architecture*. 83–94.

[14] X. E. Chen and T. M. Aamodt. 2008. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. 59–70. DOI: https://doi.org/10.1109/MICRO.2008.4771779

[15] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. 2010. 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 (MICRO'43)*. IEEE, Los Alamitos, CA, 225–236. DOI: https://doi.org/10.1109/MICRO.2010.36

[16] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *Proceedings of the 2017 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–7. DOI: https://doi.org/10.23919/FPL.2017.8056860

[17] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *Proceedings of the 2017 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–7. DOI: https://doi.org/10.23919/FPL.2017.8056860

[18] Jason Cong and Bingjun Xiao. 2014. Minimizing computation in convolutional neural networks. In *Proceedings of the International Conference on Artificial Neural Networks*. 281–290.

[19] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[20] Weilong Cui, Yongshan Ding, Deeksha Dangwal, Adam Holmes, Joseph McMahan, Ali Javadi-Abhari, Georgios Tzimpragos, Frederic T. Chong, and Timothy Sherwood. 2018. Charm: A language for closed-form high-level architecture modeling. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 152–165. DOI: https://doi.org/10.1109/ISCA.2018.00023

[21] Weilong Cui and Timothy Sherwood. 2017. Estimating and understanding architectural risk. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*.

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[23] Hilding Elmqvist and Sven Erik Mattsson. 1997. An introduction to the physical modeling language Modelica. In *Proceedings of the 9th European Simulation Symposium (ESS'07)*. 110–114.

[24] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 365–376. DOI: https://doi.org/10.1145/2000064.2000108

[25] Stijn Eyerman and Lieven Eeckhout. 2010. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 362–370. DOI: https://doi.org/10.1145/1815961.1816011

[26] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems* 27, 2 (May 2009), Article 3, 37 pages. DOI: https://doi.org/10.1145/1534909.1534910

[27] S. Eyerman, K. Hoste, and L. Eeckhout. 2011. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (IEEE ISPASS'11)*. 216–226. DOI: https://doi.org/10.1109/ISPASS.2011.5762738

[28] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. 2009. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Computer Architecture Letters* 8, 1 (Jan. 2009), 25–28. DOI: https://doi.org/10.1109/L-CA.2009.4

[29] Aric Hagberg, Pieter Swart, and Daniel S. Chult. 2008. *Exploring Network Structure, Dynamics, and Function Using NetworkX*. Technical Report. Los Alamos National Lab, Los Alamos, NM.

[30] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE* 79, 9 (Sept. 1991), 1305–1320. DOI: https://doi.org/10.1109/5.97300

[31] Matthew A. Hammer, Umut A. Acar, and Yan Chen. 2009. CEAL: A C-based language for self-adjusting computation. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, NY, 25–37. DOI: https://doi.org/10.1145/1542476.1542480

[32] Christian Haubelt, Jurgen Teich, Rainer Feldmann, and Burkhard Monien. 2003. SAT-based techniques in system synthesis. In *Proceedings of the Conference on Design, Automation, and Test in Europe—Volume 1 (DATE'03)*. IEEE, Los Alamitos, CA, 11168. http://dl.acm.org/citation.cfm?id=789083.1022903

[33] Mark D. Hill and Michael R. Marty. 2008. Amdahl's law in the multicore era. *Computer* 41, 7 (July 2008), 33–38. DOI: https://doi.org/10.1109/MC.2008.209

[34] Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 280–289. DOI: https://doi.org/10.1145/1815961.1815998

[35] John. Hopcroft and Richard. Karp. 1973. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing* 2, 4 (1973), 225–231. DOI: https://doi.org/10.1137/0202019 arXiv:https://doi.org/10.1137/0202019

[36] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 2006. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, 195–206. DOI: https://doi.org/10.1145/1168857.1168882

[37] A. B. Kahng, Bin Li, L. S. Peh, and K. Samadi. 2009. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the 2009 Design, Automation, and Test in Europe Conference and Exhibition*. 423–428. DOI: https://doi.org/10.1109/DATE.2009.5090700

[38] Eleni Kanellou, Nikolaos Chrysos, Stelios Mavridis, Yannis Sfakianakis, and Angelos Bilas. 2018. GPU provisioning: The 80–20 rule. In *Proceedings of Euro-Par 2018: The 24th International Conference on Parallel and Distributed Computing*. 352–364. DOI: https://doi.org/10.1007/978-3-319-96983-1_25

[39] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward standardized near-data processing with unrestricted data placement for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'17)*. ACM, New York, NY, Article 24, 12 pages. DOI: https://doi.org/10.1145/3126908.3126965

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems—Volume 1 (NIPS'12)*. 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

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

[42] B. C. Lee and D. M. Brooks. 2007. Illustrative design space studies with microarchitectural regression models. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 340–351. DOI: https://doi.org/10.1109/HPCA.2007.346211

[43] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. 2007. 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 (PPoPP'07)*. ACM, New York, NY, 249–258. DOI: https://doi.org/10.1145/1229428.1229479

[44] Benjamin C. Lee, Jamison Collins, Hong Wang, and David Brooks. 2008. CPR: Composable performance regression for scalable multiprocessor models. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*. IEEE, Los Alamitos, CA, 270–281. DOI: https://doi.org/10.1109/MICRO.2008.4771797

[45] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. 1991. Programming real-time applications with SIGNAL. *Proceedings of the IEEE* 79, 9 (Sept. 1991), 1321–1336. DOI: https://doi.org/10.1109/5.97301

[46] X. Liang and D. Brooks. 2006. Mitigating the impact of process variations on processor register files and execution units. In *Proceedings of the 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-06)*. 504–514. DOI: https://doi.org/10.1109/MICRO.2006.37

[47] Weichen Liu, Zonghua Gu, Jiang Xu, Xiaowen Wu, and Yaoyao Ye. 2011. Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 22, 8 (Aug 2011), 1382–1389. DOI: https://doi.org/10.1109/TPDS.2010.204

[48] Louis Mandel and Marc Pouzet. 2005. ReactiveML: A reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'05)*. ACM, New York, NY, 82–93. DOI: https://doi.org/10.1145/1069774.1069782

[49] Panagiotis Manolios, Daron Vroon, and Gayatri Subramanian. 2007. Automating component-based system assembly. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*. ACM, New York, NY, 61–72. DOI: https://doi.org/10.1145/1273463.1273473

[50] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. 2002. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/SCOPES'02)*. ACM, New York, NY, 18–27. DOI: https://doi.org/10.1145/513829.513835

[51] Arun Arvind Nair, Stijn Eyerman, Jian Chen, Lizy Kurian John, and Lieven Eeckhout. 2015. Mechanistic modeling of architectural vulnerability factor. *ACM Transactions on Computer Systems* 32, 4, Article 11 (Jan. 2015), 32 pages. DOI: https://doi.org/10.1145/2669364

[52] Steffen Peter and Tony Givargis. 2015. Component-based synthesis of embedded systems using satisfiability modulo theories. *ACM Transactions on Design Automation of Electronic Systems* 20, 4, Article 49 (Sept. 2015), 27 pages. DOI: https://doi.org/10.1145/2746235

[53] Stephen G. Powell, Kenneth R. Baker, and Barry Lawson. 2008. A critical review of the literature on spreadsheet errors. *Decision Support Systems* 46, 1 (Dec. 2008), 128–138. DOI: https://doi.org/10.1016/j.dss.2008.06.001

[54] A. Rahimi, L. Benini, and R. K. Gupta. 2016. Variability mitigation in nanometer CMOS integrated systems: A survey of techniques from circuits to software. *Proceedings of the IEEE* 104, 7 (July 2016), 1410–1448. DOI: https://doi.org/10.1109/JPROC.2016.2518864

[55] Felix Reimann, Michael Glaß, Christian Haubelt, Michael Eberl, and Jürgen Teich. 2010. Improving platform-based system synthesis by satisfiability modulo theories solving. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'10)*. ACM, New York, NY, 135–144. DOI: https://doi.org/10.1145/1878961.1878986

[56] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. 2008. VARIUS: A model of process variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing* 21, 1 (Feb. 2008), 3–13. DOI: https://doi.org/10.1109/TSM.2007.913186

[57] Timothy E. Sheard. 2012. Painless programming combining reduction and search: Design principles for embedding decision procedures in high-level languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP'12)*. ACM, New York, NY, 89–102. DOI: https://doi.org/10.1145/2364527.2364542

[58] S. Song, C. Su, B. Rountree, and K. W. Cameron. 2013. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 673–686. DOI: https://doi.org/10.1109/IPDPS.2013.73

[59] William J. Song, Saibal Mukhopadhyay, and Sudhakar Yalamanchili. 2016. Amdahl's law for lifetime reliability scaling in heterogeneous multicore processors. In *Proceedings of the 2016 International Symposium on High-Performance Computer Architecture (HPCA-22)*.

[60] Kyle L. Spafford and Jeffrey S. Vetter. 2012. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. IEEE, Los Alamitos, CA, Article 84, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389110

[61] Xian-He Sun and Yong Chen. 2010. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing* 70, 2 (Feb. 2010), 183–188. DOI: https://doi.org/10.1016/j.jpdc.2009.05.002

[62] SymPy Development Team. 2016. *SymPy: Python Library for Symbolic Mathematics*. Available at http://www.sympy.org.

[63] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. ACM, New York, NY, 320–331. DOI: https://doi.org/10.1145/2970276.2970298

[64] Nathan R. Tallent and Adolfy Hoisie. 2014. Palm: Easing the burden of analytical performance modeling. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*. ACM, New York, NY, 221–230. DOI: https://doi.org/10.1145/2597652.2597683

[65] Karl-Heinz Temme. 1989. CHARM: A synthesis tool for high-level chip-architecture planning. In *Proceedings of the 1989 IEEE Custom Integrated Circuits Conference*. 4.2/1–4.2/4. DOI: https://doi.org/10.1109/CICC.1989.56684

[66] Georgios Tzimpragos, Advait Madhavan, Dilip Vasudevan, Dmitri Strukov, and Timothy Sherwood. 2019. Boosted race trees for low energy classification. In *Proceedings of the 24th International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS'19). ACM, New York, NY, 215–228. DOI: https://doi.org/10.1145/3297858.3304036

[67] Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell, and John Shalf. 2015. ExaSAT: An exascale co-design tool for performance modeling. *International Journal of High Performance Computing Applications* 29, 2 (May 2015), 209–232. DOI: https://doi.org/10.1177/1094342014568690

[68] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, Sharad Malik, and David I. August. 2006. The liberty simulation environment: A deliberate approach to high-level system modeling. *ACM Transactions on Computer Systems* 24, 3 (Aug. 2006), 211–249. DOI: https://doi.org/10.1145/1151690.1151691

[69] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52, 4 (April 2009), 65–76. DOI: https://doi.org/10.1145/1498765.1498785

[70] Dong Hyuk Woo and Hsien-Hsin S. Lee. 2008. Extending Amdahl's law for energy-efficient computing in the many-core era. *Computer* 41, 12 (Dec. 2008), 24–31. DOI: https://doi.org/10.1109/MC.2008.494

[71] Xilinx. 2018. 7 Series Product Tables and Product Selection Guide. Retrieved September 23, 2019 from https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf.

[72] L. Yavits, A. Morad, and R. Ginosar. 2014. The effect of communication and synchronization on Amdahl's law in multicore systems. *Parallel Computing* 40, 1 (2014), 1–16. DOI: https://doi.org/10.1016/j.parco.2013.11.001

[73] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. ACM, New York, NY, 161–170. DOI: https://doi.org/10.1145/2684746.2689060

[74] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, 161–170.

Received January 2019; revised June 2019; accepted September 2019