Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon

QIAOSHI ZHENG, University of California, San Diego and Northwestern Polytechnical University, China
NATHAN GOULDING-HOTTA, SCOTT RICKETTS, STEVEN SWANSON, and MICHAEL BEDFORD TAYLOR, University of California, San Diego
JACK SAMPSON, University of California, San Diego and The Pennsylvania State University

As chip designers face the prospect of increasingly dark silicon, there is increased interest in incorporating energy-efficient specialized coprocessors into general-purpose designs. For specialization to be a viable means of leveraging dark silicon, it must provide energy savings over the majority of execution for large, diverse workloads, and this will require deploying coprocessors in large numbers. Recent work has shown that automatically generated application-specific coprocessors can greatly improve energy efficiency, but it is not clear that current techniques will scale to *Coprocessor-Dominated Architectures* (*CoDAs*) with hundreds or thousands of coprocessors.

We show that scaling CoDAs to include very large numbers of coprocessors is challenging because of the energy cost of interconnects, the memory system, and leakage. These overheads grow with the number of coprocessors and, left unchecked, will squander the energy gains that coprocessors can provide. The article presents a detailed study of energy costs across a wide range of tiled CoDA designs and shows that careful choice of cache configuration, tile size, coarse-grain power management and transistor implementation can limit the growth of these overheads. For multithreaded workloads, designer must also take care to avoid excessive contention for coprocessors, which can significantly increase energy consumption. The results suggest that, for CoDAs that target larger workloads, amortizing shared overheads via multithreading can provide up to 3.8× reductions in energy per instruction, retaining much of the 5.3× potential of smaller designs.

Categories and Subject Descriptors: C.1.3 [**Other Architecture Styles**]: Heterogeneous (hybrid) systems

General Terms: Design, Performance

Additional Key Words and Phrases: CoDA, coprocessor, conservation core, dark silicon, energy efficiency, scalable specialization

ACM Reference Format:
Qiaohi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. 2014. Exploring energy scalability in coprocessor-dominated architectures for dark silicon. ACM Trans. Embedd. Comput. Syst. 13, 4s, Article 130 (March 2014), 24 pages.
DOI: http://dx.doi.org/10.1145/2584657

This work was partially supported by NSF awards 06483880, 0811794, 1018850, 0811794, and 1228992, and by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.
Author's addresses: Q. Zheng, N. Goulding-Hotta, Computer Science and Engineering Department, University of California, San Diego; S. Ricketts (Current address), Nvidia; S. Swanson, M. B. Taylor, Computer Science and Engineering Department, University of California, San Diego; J. Sampson (corressponding author; current address), Computer Science and Engineering Department, The Pennsylvania State University; email: jsampson@cs.ucsd.edu.

DOI: http://dx.doi.org/10.1145/2584657

1. INTRODUCTION

The end of Dennard scaling [Dennard et al. 1974], combined with fixed power budgets, has resulted in designs where larger and larger fractions of a chip's silicon area must remain inactive in order to stay within its power budget. This *dark silicon* results from the *utilization wall* [Venkatesh et al. 2010; Goulding et al. 2010; Esmaeilzadeh et al. 2011; Hardavellas et al. 2011; Taylor 2012, 2013; Govindaraju et al. 2012; Semiconductor Industries Association 2012]: the observation that the percentage of a chip that can switch at full frequency is dropping precipitously with each process generation.

As progressively decreasing portions of a chip's transistors can be fully utilized, silicon area becomes less expensive relative to power and energy consumption. This shift calls for new architectural techniques that trade dark silicon area for energy efficiency. One such technique is the use of specialized coprocessors. Specialized coprocessors are becoming commonplace across smartphone, tablet, and desktop chips. These chips now include diverse functions such as H.264 accelerators, Viterbi baseband processing blocks, and cascade-based face detection pipelines. This trend will continue to accelerate as energy efficiency continues to drive processor design.

Dark silicon is a plentiful resource now and will become more so. As a result, chip designers can include many of these coprocessors, each one specializing to an even greater degree for a smaller fraction of the workload. This specialization can target both energy savings [Venkatesh et al. 2010] and/or performance [Clark et al. 2008]. Recent work [Goulding et al. 2010; Hardavellas et al. 2011] has proposed using dark silicon to implement a host of specialized coprocessors, each of which is a factor of ten or more energy efficient than a general-purpose processor. Although prior work has explored the effectiveness of coprocessor-enabled systems for single applications or small, targeted workloads, to be generally useful these coprocesor-enabled systems must realize savings across broad and diverse workloads, which means scaling to workloads featuring dozens or even hundreds of applications.

As the number of coprocessors scales up, these designs will transform from coprocessor-enabled systems to *Coprocessor-Dominated Architectures* (*CoDAs*). In CoDAs execution hops among coprocessors and general-purpose cores depending on which is most efficient for the current task, while unused components enter deep low-power modes. Area budgets at the 22nm node and beyond will provide sufficient transistor resources to build CoDAs that contain hundreds or thousands of coprocessors, enabling designers to target higher coverage over ever-larger workloads. The larger the fraction of the workload that the specialized coprocessors can cover, the larger the potential increase in overall efficiency that CoDAs can provide.

However, designing scalable CoDAs will raise numerous architectural challenges. Energy consumption from integration overheads grows as CoDAs scale, eroding potential savings. Although each coprocessor can improve performance and/or efficiency in isolation, assembling many coprocessors into a single architecture causes expansion of the on-chip interconnect and increases the complexity of the memory system. So much of the chip is idle (i.e., dark) at any moment that leakage energy from idle components is a much larger problem for dark silicon systems than for conventional designs. Increasing coprocessor counts can increase the frequency of migration between them, adding migration overheads and impacting cache performance. If designers are not careful, these inefficiencies can overshadow the benefits that the coprocessors provide.

CoDAs also raise questions with respect to application coverage and the usage model for coprocessors. Traditional coprocessors target a few, key applications (e.g., video decoding). However, in a CoDA, almost all applications will be using coprocessors, and multithreaded applications may use several at once. If applications compete for a particular coprocessor, then either performance or efficiency will suffer, as the losing thread either waits for access to the coprocessor or falls back to executing on a general-purpose core. For multithreaded workloads, these conflicts can dramatically reduce the efficiency of CoDAs.

This article systematically explores the design space for CoDA systems to observe how CoDA efficiency scales with larger and highly multithreaded workloads. We survey CoDA designs to understand the impact of both high-level architectural decisions (e.g., cache sizes and the number of coprocessors) and low-level implementation choices (e.g., the type of transistors to use and how to manage power gating). Then, we measure the impact of running concurrent threads on a CoDA, and explore methods for reducing the impact of competition for contended coprocessors.

This article shows that the main limiter on the efficiency of larger CoDAs is the efficiency of the other on-chip components and the leakage through dark silicon. In particular, we show the following.

—*Without aggressive power management, leakage precludes efficiency benefits from large CoDAs, and we show that, even with aggressive power management, leakage is still a sizable fraction of CoDA energy that grows with coprocessor count.* In a CoDA, the leakage of the inactive components can be higher than the dynamic power of the active components. Despite this, we also show that CoDAs can still scale to workloads requiring hundreds of coprocessors while retaining 3.5× efficiency gains.

—*CoDAs must have efficient power management, networks, and memory systems in order to retain high overall efficiency as they scale.* Our results provide a roadmap for how improvements in power management, network, and memory system efficiency would improve CoDA efficiency. In particular, the results provide strong motivation for mechanisms to render dark silicon truly dark. For multithreaded workloads, the results suggest that the impact of threads competing for coprocessors can be mitigated with only a modest increase in area.

—*A scalable CoDA design approach can continue to deliver superior efficiency even for large workloads.* The study suggests that a CoDA design approach that can deliver 5.3× improvements in energy efficiency and 5.0× improvements in energy-delay product for small workloads could continue to yield improvements of 3.7× in energy and 3.5× in energy delay for designs covering over 100 applications.

The rest of this article proceeds as follows. Section 2 describes the CoDA and coprocessor architectures we use in this work and the workload we target. Section 3 describes the model we use to evaluate potential CoDA designs. Section 4 explores the design space of energy consumption in CoDA designs, and Section 5 addresses issues related to multithreading. Finally, Section 6 reviews related work, and Section 7 concludes.

2. CODA ARCHITECTURE AND WORKLOAD

A scalable approach to building CoDAs needs to accommodate designs with hundreds or thousands of coprocessors that target a wide range of applications. In this section we describe the type of coprocessors we will target in this work and introduce a class of heterogeneous, tile-based architectures that will allow designers to build (and us to evaluate) CoDA designs covering a wide range of sizes. Although our approach to evaluating CoDAs is independent of the internal architecture of the coprocessors, for simplicity, we will focus on a single style of coprocessor design. This section also describes the workload we use to guide the design of the CoDAs we evaluate in later sections.

2.1. Architecture

Figure 1 provides a high-level view of the CoDAs this work examines. These CoDAs are heterogeneous, tiled designs. Each tile contains one general-purpose host processor, coherent L1 instruction and data caches, a dynamic routing network switch, and many

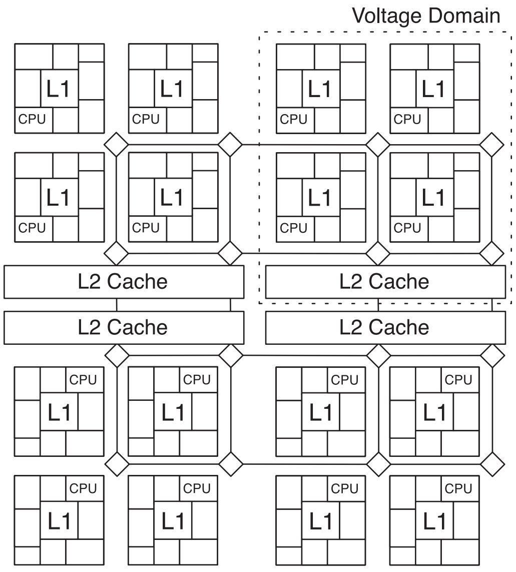

Fig. 1. *Prototypical CoDA*. The prototypical CoDA comprises a set of tiles, each of which contains a host CPU, on-chip network interface, multiple coprocessors, and a shared, coherent L1 data cache. At coarser granularity, the CoDA comprises several voltage domains, each containing one or more tiles. One L2 is present for each voltage domain.

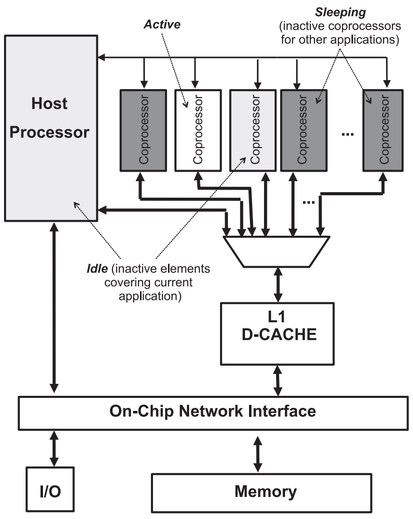

Fig. 2. *Tightly coupled coprocessor integration.* Coprocessors in CoDAs share the data cache and use the same memory model as the general-purpose host processors. The memory and CPU-to-coprocessor networks are circuit switched and allow only one active coprocessor at a time.

specialized coprocessors. The chip also contains one or more shared L2 caches. The tiles communicate with each other and the L2 caches via a point-to-point, wormhole-routed mesh network that uses physical rather than virtual channels.

Figure 2 shows the connections among the components within a single tile. Only one processing element on a tile, either the host processor or one of the coprocessors, can be active at one time, so we can use a scalable, circuit-switched tree-based interconnect between the cache and coprocessors. The other coprocessors will either be idle, if they are associated with a currently running application, or power gated if they are not associated with any scheduled application. The thread associated with a tile can make use of any of the coprocessors on that tile. To utilize other coprocessors, the thread must migrate to the tile containing those coprocessors.

L1 access latency is a function of both the number of coprocessors in a tile and the distance between a given coprocessor and the L1. L1 access latency is critical to performance, so this can limit the performance scalability of larger tiles. In practice, not all coprocessors on a tile or tiles in a CoDA will place an equal demand on the memory interface or have equivalent sensitivity to memory latency. Rather than use a multiplexing solution that provides uniform latency to memory, CoDAs use profiling to organize coprocessors with higher traffic and more latency-sensitive coprocessors closer to the L1 data cache. This saves wire and muxing energy and minimizes performance degradation due to wire delay.

The host processor on each tile is a compact, energy-efficient in-order processor optimized for efficiency and fast wakeup from deep sleep. By design, CoDAs only execute code on the host processor infrequently. Thus, its performance is less critical than its simplicity because there will be as many host processors as there are tiles. The processor in our design is based on the MIPS-like processor in Taylor et al. [2004]. The general-purpose host processor controls the coprocessors via a tree-based interconnect that also provides access to the coprocessors' internal state.

Our architecture breaks the array of tiles into multiple voltage domains (the dotted boxes in Figure 1). Each domain contains several tiles and a shared L2 cache. Each domain has its own power rail controlled by an off-chip voltage regulator. This allows domains to completely power off when they are not in use, but it means that threads may not benefit from the L2 resources of other domains. For the purposes of this article, we assume it takes hundreds of μs to flush caches and power down or power up a domain, so changing which domains are active would only occur at OS scheduling timescales.

CoDAs can employ deep-sleep power gating at OS scheduling timescales, reconfiguring the powered regions of coprocessors at application granularity. Efficiently managing numerous inactive elements requires that they are in a deeply power-gated sleep state by default, and that the OS configures shared resources proportional to concurrency and not to connectivity. Since the coprocessors that an application may request are highly predictable and highly specific to that application, this can be a low-frequency event and is therefore compatible with the timescales of both current [Jotwani et al. 2010] and more aggressive proposed [Henry and Nazhandali 2010; Henry et al. 2011; Dadgour and Banerjee 2007] power-gating techniques.

2.2. Executing in CoDAs

A program executing in a CoDA system migrates between coprocessors and general-purpose processors. To orchestrate transitions, a CoDA-aware compiler replaces functions that a coprocessor implements with a "stub" that will invoke the specialized hardware if it is available or execute the original function in software if it is not. From the perspective of the rest of the program, the stub behaves exactly like the original function. This similarity is intentional and fundamental to the vision of the CoDA design paradigm. The particular hardware in a given CoDA is compiler visible, but not programmer visible, and the dynamic check for hardware allows CoDAs to deal with contention, defective components, and legacy code or hardware.

When multiple programs or threads are running concurrently in a CoDA they can compete for coprocessors. To manage this contention, the stub function checks if the desired coprocessor is available and reserves it before transitioning to it. If the coprocessor is not available, the stub invokes the original version of the function that runs on a general-purpose processor. If the coprocessors used in the CoDA support context switching (like conservation cores [Venkatesh et al. 2010; Sampson et al. 2011], described shortly), then the CoDA compiler will generate compensation code so that an execution begun in a coprocessor may finish in software if necessary.

2.3. Coprocessor Selection and Coverage

There are many types of coprocessors that a designer could choose to include in a CoDA, offering a wide variety of design trade-offs in terms of performance, energy savings, and coverage potential. Since engineering effort is a primary barrier to the creation of coprocessors, CoDAs using automatically generated coprocessors will have greater scalability across many diverse codebases. In order to examine the scalability limits of CoDAs, we conservatively restrict ourselves to only those types of coprocessors that we can generate from arbitrary code. We therefore focus this study on automatically generated coprocessors that attain substantial energy-delay product improvements without the use of complex pointer analysis and code transformations. That said, our framework remains applicable to many different kinds of coprocessors, so long as they are tightly coupled to the host processor and to each other through a shared memory as shown in Figure 2.

Conservation cores (*c-cores*) [Venkatesh et al. 2010] are a class of automatically generated, energy-reducing coprocessors that meet the aforementioned requirements. Since it is possible to automatically generate a c-core for almost any function, it is possible to leverage a large number of them, and we use c-cores as the model for coprocessors in this study. Previous work [Sampson et al. 2011; Goulding et al. 2010; Venkatesh et al. 2011] has shown that c-cores can offer up to 10× reduction in energy (30× more efficient for nonmemory operations [Goulding et al. 2010], and 10× overall) and 23× improvement in energy delay compared to the same code running on an in-order general-purpose processor. C-cores focus on saving energy rather than directly improving application speed. However, by lowering energy per instruction, c-cores do allow greater concurrency within the same power budget.

Each c-core covers a specific portion of a target program, and an automated toolchain generates the c-cores directly from program source code. The general-purpose processors handle remaining cold code regions. During execution, threads migrate between the general-purpose core and c-cores to minimize Energy-Delay Product (EDP). Since c-cores are individually far more efficient than a CPU, the energy of optimized regions all but disappears, and system energy savings are more-or-less proportional to the coverage attained, in accordance with Amdahl's Law. In an idealized CoDA system, as the area dedicated to c-cores increases to provide increasing coverage of the workload, the energy per operation would correspondingly improve.

The c-core toolchain generates c-cores as follows. First, the toolchain performs profiling to locate hot code regions. Then, each hot code region is decomposed into a collection of basic blocks or hyperblocks (for recognized switch statements). C-cores use a spatial computation approach, and create dedicated functional units for each operator within each block. At the same time, the toolchain also creates the c-core control logic, which is a state machine that sequences the blocks. Collectively, the assorted datapaths for the hot region and the associated control logic comprise a single c-core, which often corresponds to an outer loop or function. As an optimization, however, c-core compilation will outline cold code within hot loops and functions as exceptional cases to be handled by the host processor. Since the c-cores do not algorithmically change the target region, executing all or part of the original region in software on the host processor instead of the c-core is still possible.

Table I. Applications

Workload	Description	22 nm C-core Area (mm^2)
astar [Standard Performance Evaluation Corporation 2006]	pathfinding	0.044
bzip2 [Standard Performance Evaluation Corporation 2000]	data compression	0.329
cjpeg [Independent JPEG Group 2002]	jpeg encoding	0.076
crafty [Standard Performance Evaluation Corporation 2000]	chess	0.580
djpeg [Independent JPEG Group 2002]	jpeg decoding	0.118
gzip [Standard Performance Evaluation Corporation 2000]	compression/decompression	0.190
mcf [Standard Performance Evaluation Corporation 2000]	multi-commodity flow	0.056
viterbi [Embedded Microprocessor Benchmark Consortium 2002]	convolutional decoding	0.039

The workload generator modifies the properties of these eight seed applications to generate workloads of up to 128 cores. The c-core toolchain created 22 placed-and-routed c-cores for the seed applications in order to provide accurate characterization.

C-cores use the same memory model as the host processor, and share the L1 data cache with both the host processor and the other c-cores, as seen in Figure 2. To minimize communication costs across c-core to c-core boundaries, we profile memory communication among application hotspots hierarchically and provide this as an input to both c-core selection and CoDA placement. C-cores use techniques including selective depipelining and cachelets [Sampson et al. 2011] to minimize area and energy costs while maximizing performance.

2.4. Applications

The selection of coprocessors in a CoDA depends on the set of applications it targets. Our goal is to understand how CoDAs scale from designs with a handful of coprocessors to designs featuring hundreds of coprocessors, so we need a correspondingly broad set of applications to target.

Our workload generator employs a set of "seed" applications from SPEC 2006 [Standard Performance Evaluation Corporation 2006], SPEC 2000 [Standard Performance Evaluation Corporation 2000], and EEMBC [Embedded Microprocessor Benchmark Consortium 2002] and modifies their properties to model a greater span of program characteristics. Table I lists the applications and properties of the c-cores that target them. This set of seed applications was characterized via 22 automatically generated c-cores that were run all the way to placed-and-routed netlists and simulated at the gate level with detailed parasitics.

To generate larger workload sizes and model the potential c-cores to cover them, the workload generator replicates each application 2, 4, 8, and 16 times and adjusts the area for each of the resulting c-cores by up to 50% to provide variability in hotspot code density. This produces a generated workload of 16, 32, 64, and 128 applications in addition to the original 8-application workload. The largest of these workloads requires 352 c-cores to achieve more than 97% coverage. The CoDA containing these 352 c-cores would require 74 mm^2 in a 22nm process technology.

3. MODELING CODAS

Among the aims of this article is to develop an understanding of the CoDA approach in sufficient depth to derive insights about the energy scalability of CoDAs. However, there are many possible CoDAs, and fully synthesizing and simulating all of the CoDA designs we will consider is intractable, so this work employs an analytical model for CoDA performance, area, and energy efficiency.

3.1. Methodology

Our analytical model is driven by three primary components. First, we use the properties of fully synthesized CoDA subcomponents as inputs to our model. Then, we perform trace-driven analysis of our workloads to develop our model of dynamic program behavior, including migration and cache coherence effects. Finally, we use the data gathered from the preceding studies across entire workloads, scale to a 22nm process, and provide breakdowns for the energy, area, and performance properties of the resulting CoDA.

We use Synopsys Design Compiler, IC Compiler, and PrimeTime as our tools to measure the fully synthesized, placed-and-routed c-cores for a subset of the hot regions in the applications in Table I. We build these coprocessor components individually and use the fully synthesized placed-and-routed netlist for the general-purpose host processor to inform our model. We synthesize these designs using TSMC 45nmG and 40nmLP technology nodes, and then scale to derive their properties at other design points.

To collect data about program behavior, we use LLVM [Lattner and Adve 2004] to annotate an executable, in order to map different parts of execution to c-cores and general-purpose processors. Each execution simulates a particular mapping of regions of code to particular tiles on a CoDA and either the host or a coprocessor on that tile. The annotated executables provide a detailed trace of memory operations, including coherence messages, cache-to-cache transfers, and NoC segments traversed, as well as transitions between c-core and non-c-core execution. Aside from these overheads, we model the progress of execution at one cycle per instruction. The annotated binaries output summaries of these key statistics as inputs for our analytical model.

3.2. Model Parameters

$$Wire\_length = 2\sum_{i=1}^{n}\sqrt{Component\_Area_i},\ i \in every\ component\ along\ the\ path. \tag{1}$$

Several parameters in our analytical model come directly from existing literature or are scaled from actual 40/45nm measurements to 22nm. Table II lists the key parameters. To calculate wire energy, we multiply the wire length of each traversed segment of Manhattan-distance routing and the wire energy per mm, as seen in Eq. (1). To calculate the wire length from coprocessor to mux, we sort the coprocessors by the memory access demand rate, and then place the coprocessors with higher access rates higher up in the mux tree.

We set the transition cost between software and hardware on the same tile at 30 cycles, based on microbenchmarks exercising a fully synthesized model of the c-core host interface. Transitions between active tiles take 300 cycles including interrupt handling and context transfers over the on-chip network, modeled on context-switch overheads in RAW [Taylor et al. 2004].

$$Area_{new} = Area_{old} * (\lambda_{new}/\lambda_{old})^2. \tag{2}$$

$$\begin{aligned} Leakage\_energy\_per\_square\_mm_{new} = (&Leakage\_energy\_per\_square\_mm_{old} \\ &* 3D\_Factor * (\lambda_{old}/\lambda_{new})^2). \end{aligned} \tag{3}$$

$$Dynamic\_energy_{new} = Dynamic\_energy_{old} * (\lambda_{new}/\lambda_{old}). \tag{4}$$

Eqs. (2), (3), and (4) are used to scale the area, leakage energy, and dynamic energy, respectively. λ_{old} and λ_{new} represent the feature size of the old and new process

Table II. Model Parameter Values

Model Parameter	Source of Values
Wire energy per mm	Bill Dally's 2009 DAC keynote
Host processor energy	Host energy per instruction from [Goulding et al. 2010], scaled to 22 nm using Equations 3 and 4
Coprocessor energy	C-core energy/instruction from [Goulding et al. 2010], scaled to 22 nm using Equations 3 and 4
NoC router energy	Modeled as equivalent to one coprocessor instruction per routing decision
Cache leakage energy, area, and access time	CACTI [Thoziyoor et al. 2008], scaled to 22 nm
Main memory bandwidth	We assume LPDDR2 in our system, with 3.2 GB/s bandwidth
Transistor speed	Relative latency of HP and LSTP from synthesized circuits, also could derive similar value from [Semiconductor Industries Association 2012]. LP interpolated: HP latency = 1, LP latency = 1.25 and LSTP latency = 2.5
Transistor leakage/dynamic energy	From circuit evaluations, scaled to 22 nm
Software and hardware transitions	30 cycles, measured from microbenchmarks
Execution migration between active tiles	300 cycles, measured from microbenchmarks

The sources for our model parameters involve a mix of measurements scaled to 22nm, and simplifying assumptions about average efficiencies.

technology. Eq. (2) is straightforward: transistor density will continue to increase with process feature size for compute-constrained designs. Eq. (3) is slightly more complex. Transistor leakage is a parameter that designers have quite a bit of control over, by setting the threshold and supply voltages. However, limiting leakage has brought us to a post-Dennardian scaling regime for dynamic energy at the cost of holding per-transistor leakage at bay. Thus, for the first part of Eq. (3) we hold innate leakage constant per transistor. To account for the move from planar to 3D transistors at the 32nm to 22nm transition, we employ an additional scaling factor to account for FinFET-specific efficiencies. The 3D_Factor we use is 0.7, derived from Intel literature on their 22nm process [Bohr and Mistry 2011]. The third part is the transistor density scale factor. The reason for choosing the leakage energy per square mm as our parameter is that we also need to model the inactive leakage energy, which is easy to calculate by area and this parameter. We compute separate leakage/unit area numbers for each of the three transistor types we consider. Finally, the dynamic energy model (Eq. (4)) is more direct. In a post-Dennardian scaling scenario, we elide energy savings from voltage reduction, and only credit reductions in capacitance as the sources of improved dynamic energy. This appears as the second factor in Eq. (4).

The memory trace drives a cache simulator modeling the L1 and L2 caches. We use this simulator and parameters from CACTI [Thoziyoor et al. 2008] to model area, dynamic and static energy for our caches. We model the cache in CACTI at 32nm, and then scale down to 22nm. Area, leakage, and dynamic energy are scaled according to Eqs. (2), (3), and (4), respectively.

Each L2 is noninclusive and serves only the L1s in the corresponding voltage domain. Evictions from the L1 cause allocations in L2, and hits in the L2 transfer the line to the requesting L1 and invalidate it in the L2. On an L1 miss, the L1 accesses the L2 before consulting the coherence directory for that domain. If the missed line is present in another L1, the directory will initiate a cache-to-cache transfer between the two L1s.

$$Performance \begin{cases} Software\ execution\ cycles\ on\ host\ processor \\ Hardware\ execution\ cycles\ on\ coprocessors \\ Software\ and\ hardware\ transition\ cycles \\ Tile\ transition\ cycles\ including\ interrupt\ handling\ and\ context\ transfers \\ NoC\ route\ cycles \\ L1\ cache\ access\ cycles \\ L2\ cache\ access\ cycles \\ Main\ memory\ access\ cycles \\ Cache-to-cache\ transfer\ cycles\ including\ all\ coherence \\ Cycles\ per\ instruction\ (CPI) \end{cases}$$

Fig. 3. *Hierarchical view of performance model.* We use the output from the annotated binaries for our workload to calculate all components of execution time.

Dynamic and static instruction counts for the annotated regions let us model c-core area (for those c-cores not already run through place-and-route) and execution coverage. The individual c-cores range in area from 0.0015 to 0.28 mm^2. We estimate area for c-cores that we have not yet built based on a simple regression model using static counts of each operator type (add, multiply, load, shift, FMAD, etc.) in the annotated region that we calibrated against previously published areas for the placed-and-routed c-cores in Sampson et al. [2011] and those fully placed-and-routed for the workload in this article. Similarly, we use data from previously published work [Sampson et al. 2011] to model increased execution time due to increased L1 access latencies as c-cores move further away from the L1 in larger tile designs.

Transistor leakage and performance can vary dramatically depending on technology library. We derive leakage and area values from post-place-and-route synthesis of c-cores in a 40nm low-power library and a 45nm general-purpose library, both from TSMC. From these experiments, we model c-core and other noncache logic leakage scaled to 22nm at 0.25 mW/mm^2, 1.02 mW/mm^2, and 29.28 mW/mm^2 for leakage-optimized, low-power, and performance-oriented designs, respectively. The scaling process follows Eq. (3). To model the effect of transistor choice on performance, we scale the nonmemory portion of compute time for CoDAs by 2.0, 1.0, and 0.8 for the leakage-optimized, low-power, and performance-oriented designs, respectively.

We compare against our energy-efficient in-order baseline processor running at 3 GHz. Based on the c-core energy component breakdown in Goulding et al. [2010], we model dynamic energy per instruction for nonmemory computation in our c-cores as 30× less than that for our host processor, which uses 43 pJ/instruction in 22nm. Prior work [Sampson et al. 2011] shows that c-cores execute 27% faster than the host processor on average.

3.3. Components of Performance, Area, and Anergy Models

The analytical model has three primary outputs: performance, area, and energy. We give more details about each submodel in this section. Figure 3 shows the performance components we modeled. All performance values except CPI come from our trace simulator. CPI is done separately, because it can be a function of internal tile placement due to additional applications' c-cores receiving higher priority in the mux tree to the L1, and the mux tree is modeled at a higher level. In our analytical model, we use these execution cycles for all processor components to calculate the effective CPI, which is later used to calculate the energy per instruction.

$$Area \begin{cases} Area\ of\ the\ tiles \begin{cases} Host\ processor\ area \\ L1\ cache\ area \\ NoC\ router\ area \\ Coprocessors\ area \\ MUX\ area \end{cases} \\ \\ L2\ cache\ area \end{cases}$$

Fig. 4. *Hierarchical view of area model.* We model each of these separately and then use formulas to combine them.

Figure 4 lists the main contributors to chip area. The total area of L2 caches depends on both the size of each L2 cache and the number of voltage domains that it is duplicated across.

We model the energy usage by different components at several levels of detail. For normalization across benchmarks, and between hardware and software, we normalize our energy model in terms of energy per equivalent instruction in software. Figure 5 demonstrates all the components and their hierarchical relationship in the energy model. Additional modeling in Section 5 concerning concurrent execution uses statistical models of c-core occupancy to determine the impacts of contention on energy per instruction.

In this article, we focus on the energy consumed by the on-chip portion of CoDA-based systems. In real systems, on-chip energy is clearly not the only contributor to energy consumption, and other researchers are actively optimizing these other components. For instance, promising research on low-energy DRAM systems is advancing the use of through-silicon vias, package-on-package, and low-energy off-chip signaling. Similarly, passive display technologies such as Qualcomm's Mirasol are on the horizon. Although these efforts would easily compose with our work, the final properties of these technologies is still uncertain. Rather than add noise to our results by attempting to incorporate these components, we defer to other research (e.g., [IMOD Technology Overview 2008; Lee et al. 2009]).

4. THE CODA DESIGN SPACE

The basic architecture in Figure 1 still allows great flexibility in a CoDA's configuration, and the efficiency of a particular CoDA design will vary with the number of tiles, their size, the selection of caches, etc. Furthermore, under different design constraints (e.g., varying area budgets) the ideal values vary.

To understand how the optimal design decisions for CoDAs vary, this section carries out a systematic survey of the CoDA design space. We use the workloads described earlier to drive the design of CoDAs ranging in size from a single small tile and a handful of c-cores to very large devices with hundreds of c-cores.

4.1. Design Parameters

There are many potential CoDA designs. To understand the trade-offs among them, we systematically survey the space of possible configurations targeting the workload.

Table III describes the space of CoDA designs that we consider. The design space includes designs ranging from a conventional, general-purpose processor with large caches and a handful of coprocessors to large arrays of tiles with small caches and many coprocessors. To limit the size of the design space, we did not consider tiles with heterogeneous cache resources. In total, we evaluated our workloads on 7200 CoDA configurations.

$$
Energy \begin{cases}
Host\ processor\ energy \begin{cases}
Processor\ dynamic\ energy \begin{cases} Software\ execution\ energy \\ Coprocessor\ conflict\ energy \\ Thread\ conflict\ ratio \end{cases} \\
Processor\ active\ leakage\ energy \\
Processor\ inactive\ leakage\ energy \\
Each\ processor\ power\ off\ and\ power\ gate\ ratio \\
Host\ processor\ coverage
\end{cases} \\
Coprocessor\ energy \begin{cases}
Coprocessor\ dynamic\ energy \\
Coprocessor\ active\ leakage\ energy \\
Coprocessor\ inactive\ leakage\ energy \\
Each\ coprocessor\ power\ off\ and\ power\ gate\ ratio \\
Coprocessor\ coverage
\end{cases} \\
L1\ cache\ energy \begin{cases}
L1\ cache\ dynamic\ energy \\
L1\ cache\ active\ leakage\ energy \\
L1\ cache\ inactive\ leakage\ energy \\
Each\ L1\ cache\ power\ off\ and\ power\ gate\ ratio
\end{cases} \\
L2\ cache\ energy \begin{cases}
L2\ cache\ dynamic\ energy \\
L2\ cache\ leakage\ energy
\end{cases} \\
Communication \begin{cases}
Wire\ energy \begin{cases}
From\ each\ coprocessor\ to\ MUX \\
From\ host\ processor\ to\ MUX \\
In\ MUX \\
From\ MUX\ to\ L1\ cache \\
From\ L1\ cache\ to\ L2\ cache \\
From\ L2\ cache\ to\ main\ memory
\end{cases} \\
NoC\ energy \begin{cases}
Router\ dynamic\ energy \\
Router\ leakage\ energy \\
Energy\ used\ by\ tree-based\ interconnection
\end{cases} \\
MUX\ energy \begin{cases}
MUX\ leakage\ energy\ in\ active\ tiles \\
MUX\ leakage\ energy\ in\ inactive\ tiles \\
Each\ MUX\ power\ off\ and\ power\ gate\ ratio
\end{cases}
\end{cases}
\end{cases}
$$

Fig. 5. *Hierarchical view of energy-per-instruction model.* We model energy components for all the major parts in the system. For clarity of presentation, we collapse these down to a manageable set that groups lower levels of the hierarchy appropriately.

Table IV provides an overview of how each of the parameters interacts with the models described in Section 3. Several of the parameters are straightforward in nature. For each workload size we consider, we keep coprocessor coverage constant at nearly 98% and vary the number of c-cores needed to cover the workload. The design space includes several possible L1 and L2 configurations. The *tile area* parameter describes

Table III. CoDA Design Space Parameters

Parameter	Values
Workload size (applications)	8, 16, 32, 64, 128
L2 cache size (KB)	512, 2048, 8192
Per-tile L1 cache size (KB)	8, 16, 32, 64
Maximum tile area (mm^2)	0.5, 2, 8, 32, unlimited
Number of voltage domains	1, 4
Power-gating efficiency	0, 90%, 95%, 98%
Transistor library	LSTP, LP, HP

We considered 7200 CoDA designs, one for each possible combination of the preceding parameters.

Table IV. Overview of Parameter Impects on the System Energy, Area, and Performance Models

Parameter	Direct Impact	Impact on Final Results
Workload size	Number of coprocessors	Chip area, leakage power, communication distances
Cache size	Area of the chip	Active leakage, inactive leakage and dynamic power
	Wire length	Energy used by wires
	Cache miss rate and main memory accesses rate	CPI and execution time
	Hit latency	CPI and execution time
Maximum tile area	Area of the chip	Active leakage, inactive leakage and dynamic power
	Wire length	Energy used by wires
	Number of tiles	L1 cache number, thread conflict ratio
Number of voltage domains	Number of L2 caches	Chip area, leakage energy
	On/off-chip wire lengths	Energy used by wires, NoC
Power-gating efficiency	Inactive leakage power	Energy used by CoDA, cache, host processor, MUX, etc.
Transistor library	Dynamic and static energy/instruction	Energy used by all components
	CPI	Total non-memory execution time

While each parameter influences several of the model components described in Section 3, this table summarizes their most immediate and largest overall impacts.

the maximum area of a single tile in our tiled design. Larger tiles can contain more c-cores and reduce inter-tile hopcounts, but c-cores within the tile may have longer intra-tile communication paths.

The final three parameters are somewhat more complex. Among these are the two power management parameters: the number of independent voltage domains on the chip and the efficiency of the power-gating circuits that cut off power to idle chip components within an active voltage domain. An off-chip voltage regulator can cut power to its domain, effectively reducing its leakage to zero. “Power-gating efficiency” determines the effectiveness of the power-gating circuits that cut power to inactive tiles in voltage domains that are powered. The designer has some control over this parameter (e.g., by implementing state-of-the-art power-gating circuits [Jotwani et al. 2010]) but it is also a function of manufacturing technology. The final parameter determines the standard cell library used to implement the design. The available options are low static power (LSTP), low power (LP), and high performance (HP). Section 3.1 described the leakage and performance properties associated with each of these values.

4.2. Pareto Results

Figure 6 shows the results of the design space study, respectively focusing on designs with a single voltage domain and on designs with up to four voltage domains. Figure 6(a) plots all single-voltage domain designs we considered, for all workloads, according to the area they consume and their Energy-Delay Product (EDP) relative to a general-purpose processor without c-cores and equipped with a 32KB L1 and a 512KB L2. Figure 6(b) plots the same, but for up to four voltage domains. Then, Figures 6(c) through (f) plot the Pareto frontier along the energy and delay axes for each of our workload sizes for different power-gating efficiency/voltage domain combinations, denoted as (*X*% PGE, *Y* VD) respectively. The Pareto frontiers consist of the design points for which there are no designs that are both faster (lower delay) and more efficient (lower energy per instruction).

In Figures 6(c) through (f) we can see clearly that the impact of power management increases as CoDAs grow. Without aggressive dynamic leakage control, larger designs without power gating must transition to low-leakage transistors to obtain increasing energy savings, sacrificing performance. Similarly, the cost of using high-performance, high-leakage transistors to buy performance, even with power gating, rapidly increases as the workload size increases. This is due to the corresponding increase in leakage from the additional c-core area the CoDA needs to employ to cover the larger workload.

Even with multiple voltage domains, for larger workloads, tile granularity or finer power gating becomes critical. Additional voltage domains ease the critical dependence on the efficiency of the power-gating implementation, albeit at the cost of duplicating L2 cache resources. This duplication cost is apparent in the spreading of area values in Figure 6(a) compared to Figure 6(b). Adding voltage domains also allows on-chip power gating to operate at coarser granularities, further reducing complexity. As seen in Figure 6(e) and Figure 6(f), the impact is profound for designs without effective power gating.

Figures 6(c) through (f) show that, while changing the number of power domains has limited effect on the bottom right side of the Pareto curve—designs already embracing both low-power transistors and efficient power gating—the differences are clearer for higher-performance designs at the upper left. Overall efficiency for the multiple-domain designs is greater, and the designs spend more area to achieve this effect. In an era with increasing quantities of dark silicon, this may be a reasonable trade off. The differences between power gating techniques are similarly muted by the presence of additional voltage domains, and designs without fine-grained power gating retain more than 3$\times$ efficiency for workloads up to 64 applications, whereas Figure 6(c) these designs have already diverged. As with the single-voltage domain case, however, for higher efficiency at larger workload sizes, these CoDAs must still feature aggressive dynamic power management, or coprocessor leakage will excessively degrade energy efficiency. While the L2 duplication approach is not indefinitely scalable, these trends indicate that, for even larger workloads than those examined here, partitioning the design into more domains will improve results until duplication costs run up against the area budget.

Table V lists the parameters for each of the EDP-optimal configurations from Figures 6(c) through (f), from smallest workload to largest. The EDP-optimal points are similar for most of the workload sizes, but the largest design uses a larger tile size to reduce the number of hops involved in inter-tile communication.

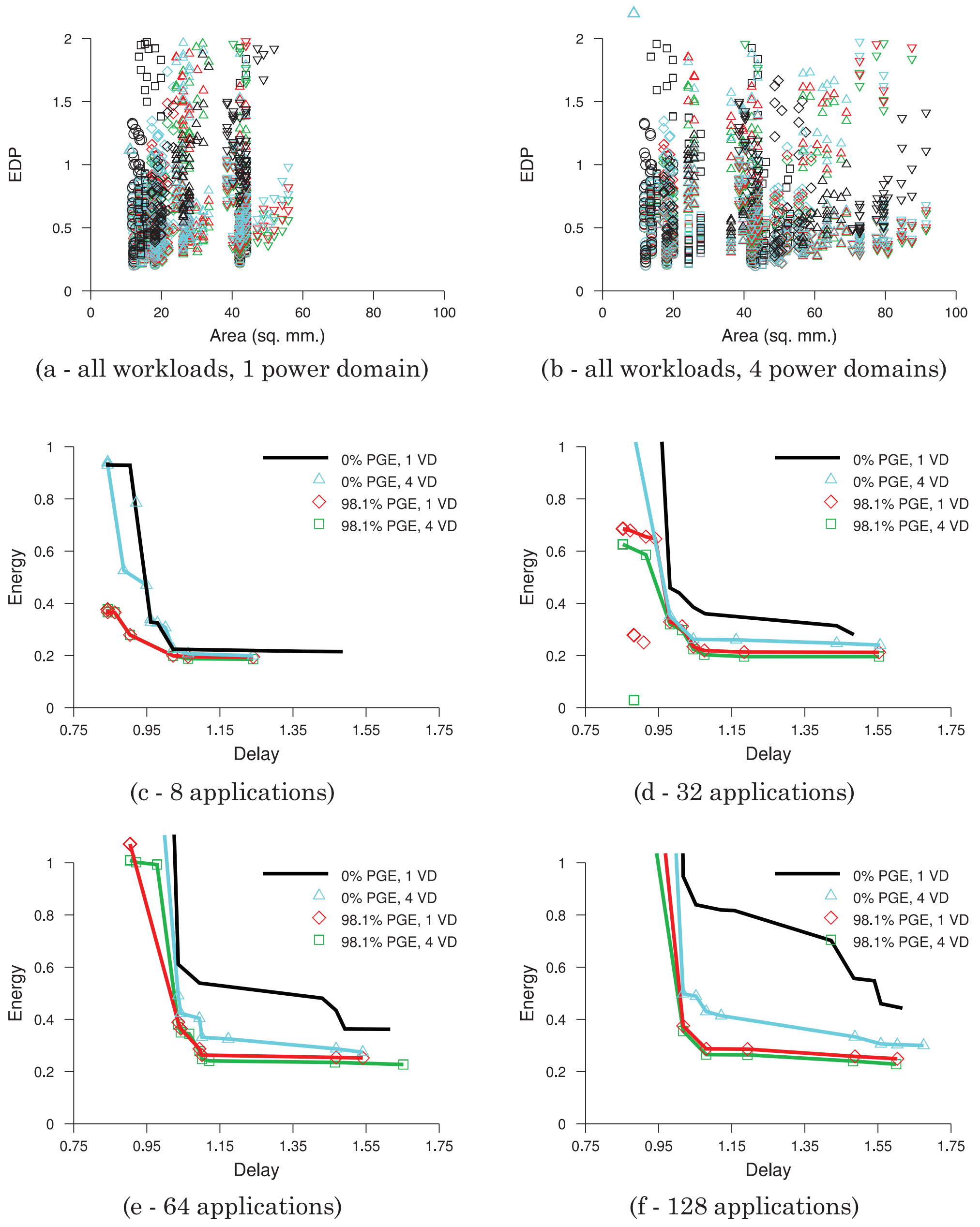

Fig. 6. *EDP vs. area and energy vs. delay over design space for each assumed power-gating efficiency and voltage domain count.* In (a) and (b) we plot EDP versus area over our entire design space for 1 and 4 voltage domains, respectively. Each point marks an averaged evaluation over one workload size for one design. We note different assumptions for power-gating efficiency with different shades and different workload sizes with different shapes. In (c) through (f) we plot the energy vs. delay Pareto curve for the four extreme points of our power-gating efficiency and voltage domain spaces for increasingly large workloads (8, 32, 64, and 128 applications, respectively). As the workload (and CoDA) grow larger, the power-gated and nonpower-gated Pareto lines diverge drastically.

Table V. EDP-Optimal Designs

Apps in Workload	L2 Cache (KB)	L1 Cache (KB)	Max. Tile Area (mm^2)	# Tiles	# C-cores	EDP vs. SW	Area (mm^2)	pJ/inst.	Speedup vs. SW	Xtr. Type	Tiles per VD
Baseline	512	32	-	1	0	1.0	10.29	51.05	1.0	HP	1
8	512	32	0.5	5	22	0.200	43.20	10.40	0.941	LP	1-2
16	512	16	0.5	9	44	0.206	45.20	9.97	0.857	LP	2-3
32	512	32	0.5	20	88	0.218	50.70	11.50	0.930	LP	5
64	512	32	0.5	40	176	0.270	60.70	13.39	0.892	LP	10
128	512	32	2.0	16	352	0.286	72.70	14.80	0.926	LP	4

The parameters for the EDP-optimal CoDA design for each workload size.

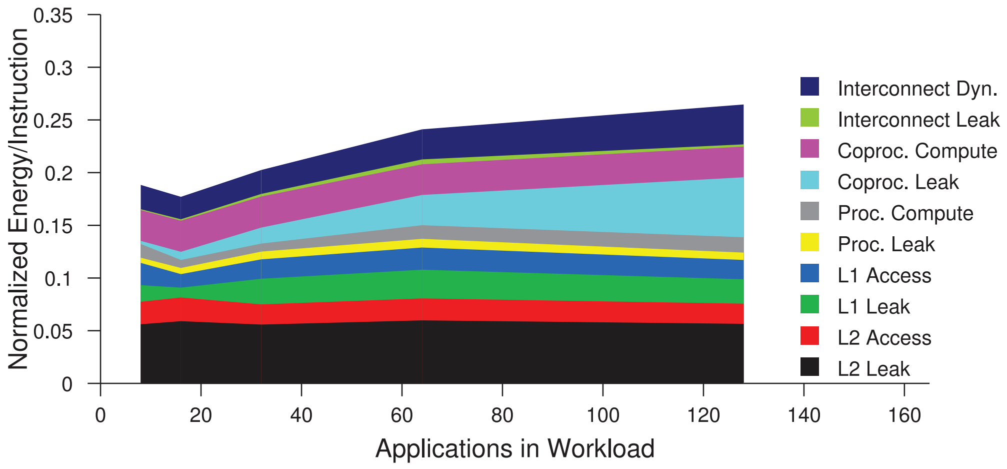

Fig. 7. *Energy components of EDP-optimal designs.* We show the primary contributing energy components for the most efficient points in our design space as we scale the workload size, keeping coverage constant. For larger designs (covering larger workloads), leakage and interface overheads play a larger role, while compute energy progressively decreases due to transitioning to energy-efficient coprocessors.

4.3. Overhead Growth as a Function of Workload Size

Figure 7 shows how the components of overall energy consumption in the Pareto-optimal CoDA designs change as the number of applications increases. It shows that, although a CoDA with sufficient c-cores to cover its workload can reduce compute energy by 94%, that compute energy becomes a tiny fraction of total energy. The culprit is leaking dark silicon (and Amdahl's Law). Even with voltage domains and 98% efficient power gating [Jotwani et al. 2010], the leakage accounts for nearly half of the energy consumption. The costs of traversing local and global interconnect and of accessing the L2 cache also contribute significant energy costs. In the case of the interconnect, the wires, rather than the logic, expend most of the energy.

There are two key takeaways from Figure 7. While it is obvious that managing leakage in a large chip like a CoDA would be important, it was not at all clear that, even with multiple power domains and 98% effective power gating, the leakage of *inactive* components would *remain* a sizeable overhead in total energy per instruction. Fortunately, the other key takeaway from Figure 7 is more positive: Despite the clear growth of overheads as CoDAs scale, they do scale. Even with its overheads, a CoDA covering 128 applications remains several times more efficient than running the software on a general-purpose processor. For many domains, such as Android platforms, where 128 applications may be enough to cover over 80% of the total execution time [Goulding-Hotta et al. 2012], this means that CoDAs scale sufficiently well to be practical.

4.4. Area Scaling and Defect Tolerance

The potential for defect rates to increase as CoDAs grow is an interesting consideration. Since the original application can run unmodified on any tile's host processor without using any particular c-core, the CoDA design approach is highly resistant to defects in most on-chip components; known defective components will simply not be used. Indeed, the same mechanism that checks if a c-core is available in the face of contention can be used to seamlessly avoid execution on any components discovered to be defective. Thus, CoDAs gracefully decay with progressively higher defect rates, or as workloads shift away from their target: CoDAs retain functionality at all times and degrade in power and performance efficiency as a function of unusable c-cores. Because c-core usage in a CoDA is transparent to the programmer, this does not represent a usability or virtualization hurdle at the programming layer. Thus, while CoDAs, being larger, may have lower defect-free yields than smaller SoCs, even defective CoDAs are likely still highly usable. We envision that for CoDAs, rather than speed-binning, a key sorting policy may be coverage-binning based on the expected fraction of the workload that can successfully execute on c-cores.

5. CODAS AND CONCURRENCY

Concurrent execution is now ubiquitous in computing platforms ranging from cell phones to data centers, so understanding the impact of multithreaded and multiprogram workloads on CoDAs (and vice versa) is essential. This section identifies the positive and negative impacts that multithreading has on CoDAs and describes several techniques to address the problems that can arise.

On the positive side, running multiple threads on a CoDA increases overall energy efficiency because it amortizes fixed energy costs, including those due to leakage, across the work from multiple threads. At the same time, however, concurrent threads raise the possibility of competition for c-cores. This can occur when two applications want access to a c-core that targets part of a shared library (e.g., glibc), or when two threads in the same application are executing the same function. In these cases, the "losing" thread will either execute on a general-purpose core, sacrificing efficiency, or wait, sacrificing performance. We assume that the scheduler always schedules the contending thread on a general-purpose core. More aggressive schedulers may use more complicated heuristics to dynamically decide whether to sacrifice energy or performance.

The amount of contention for c-cores depends on the number of instances of that c-core present in the CoDA and the number of threads that need access to it. The profiling process that identifies the "hot" functions to target with c-cores can also determine how many copies of each c-core are necessary to avoid contention.

Another concern for scaling up the number of threads on a CoDA is the utilization of available bandwidth and contention for communication resources. However, while the number of c-cores may rise rapidly, the maximum number of concurrent threads on a CoDA is limited to the number of tiles, which is more modest, the design in Table V with the lowest single-thread EDP uses 16 tiles.

Given that the memory system of each tile is blocking and in-order, the maximum number of outstanding misses at a time for this CoDA is 16, several of which may be resolved in the L2. RAW [Taylor et al. 2004], also with 16 tiles, an equivalent NoC, and a similar memory system (although it lacked L2 caches), saw less than 7% average performance degradation when executing SPEC workloads independently across all 16 of its tiles. In practice, we find that, because of the blocking nature of the L1, we do

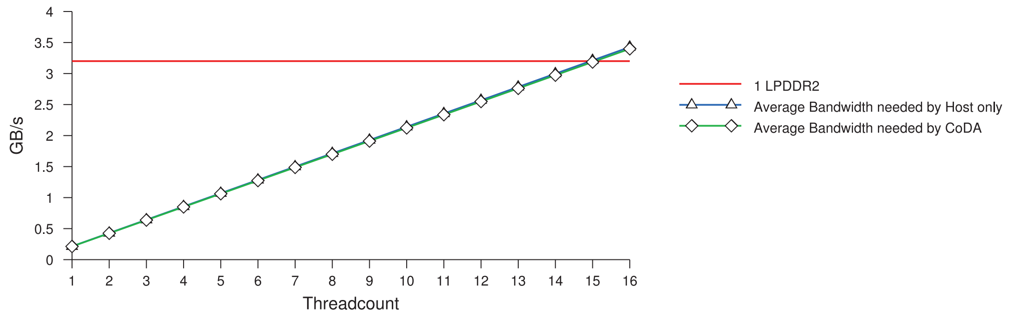

Fig. 8. *Off-chip memory bandwidth usage.* We calculate the average off-chip memory bandwidth needed by all the benchmarks, and calculate how it scales when we run more and more threads, assuming a uniform random distribution among our benchmarks. Between the in-order, blocking nature of the L1 caches that limits the rate of misses and the filtering effects of the L2 caches, the total off-chip bandwidth could be readily served by a small number of LPDDR2 channels.

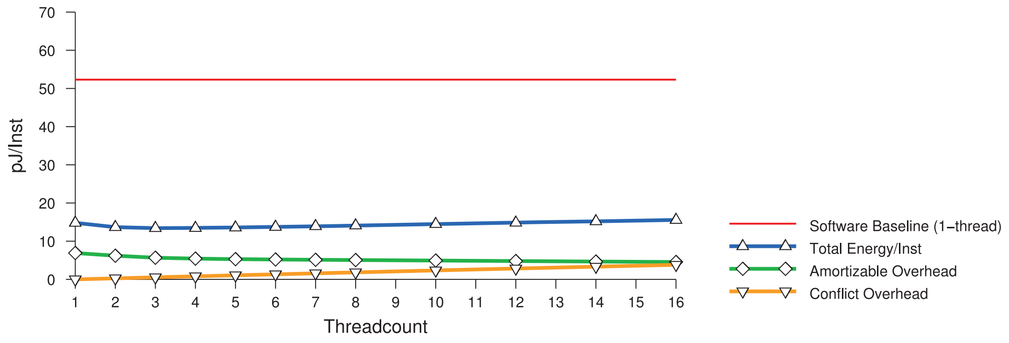

Fig. 9. *The benefits of multithreading.* Total energy per instruction is the sum of the per-thread energy, shared energy overheads, and the energy from execution on a general-purpose core rather than a c-core because of contention. Overall, if the running workload is a good match for the CoDA, energy per instruction drops because multiple threads can amortize the leakage energy of idle, but still powered, components.

not significantly speed up the total rate of memory accesses leaving the L1 compared to the original software-only execution. Our own experiments indicate that in the average case for our workload, the average off chip bandwidth required is quite modest, as shown in Figure 8. Thus, contention due to workload mismatch is likely to dominate multithreading effects for this workload. We acknowledge that, for alternative workloads such as, for instance, running 16 copies of the MCF benchmark, bandwidth contention would be a primary determining factor in performance, since such a workload would require more than 3× the off-chip bandwidth of the one we consider.

5.1. Target Workload Sensitivity

If there is a mismatch between the profile measurements and the workload's needs "in the field," the benefits of amortizing fixed costs across threads will be lost. In our multithreading experiments, we consider two workload distribution scenarios. The first distribution scenario, *Uniform*, describes the case where all applications account for an equal share of execution time. In the second workload distribution scenario, *Nonuniform*, 10% of the applications account for 90% of execution time.

Figures 9 and 10 demonstrate the impact of contention and workload mismatch on energy efficiency. We begin by selecting a particular CoDA design: We use the CoDA that provided the best EDP from our design space for the uniform workload

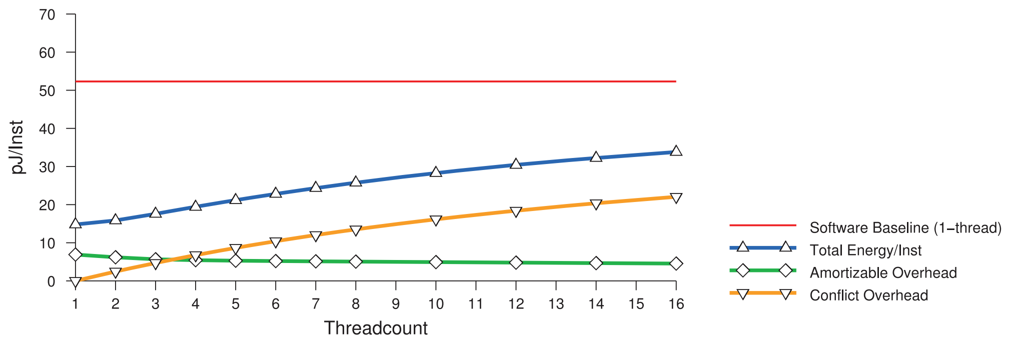

Fig. 10. *The cost of contention.* In this example, the running workload is a poor fit for the CoDA, resulting in high contention for a small number of c-cores. In this case, the conflict energy rises continuously, swamping the gains from amortizing shared energy overheads.

distribution scenario (i.e., the designer expected there to be very little contention) over the 128-program workload. The parameters for this design can be seen as the last row in Table V. As the CoDA has 16 tiles, it can support up to 16 simultaneous threads.

Figure 9 shows how energy per instruction for a fixed CoDA design changes as the number of concurrent threads increases for a workload matching the target distribution (i.e., there actually is very little contention). The graph shows total energy per instruction and two of its subcomponents: *amortizeable overhead* (i.e., fixed leakage costs that multiple threads can amortize), and *conflict overhead* (e.g., extra energy required to execute a thread on a general-purpose core rather than a c-core). For context, a constant line at the top of the graph depicts the energy per instruction for a single-threaded software execution.

The data show that adding three threads can reduce total energy per instruction by 9%. Beyond four threads, the rise in conflict energy overpowers the reduction in shared overhead. Running 16 threads will increase per-instruction energy by 5% over the single-threaded case.

Figure 10 shows energy efficiency for the same CoDA, designed for the uniform distribution scenario, but running a workload which follows the nonuniform distribution scenario, which creates a severe workload mismatch. The result is significantly higher contention and much lower energy efficiency. In this case, there is a long rise in energy per instruction due to growing conflict overhead. At 16 threads the energy per instruction has more than doubled. The conflict overhead rises much more rapidly in this case because the workload distribution differs so greatly from the training set.

5.2. Mitigating Contention via C-Core Merging

In most cases perfect profiling is impossible, and contention for c-cores is inevitable. However, we can take measures to reduce its impact. The simplest way to reduce the cost of contention is to replicate c-cores to provide "spares" that can absorb unexpected increases in demand. However, naively providing spare c-cores nearly doubles CoDA area and would decrease single-threaded efficiency by 23.4% due to increased leakage and interconnect overheads.

To reduce the area cost of replication we can exploit the fact that, in most cases, applications will need the "spare" c-cores infrequently. To exploit this observation, we can merge multiple spare c-cores, allowing a single spare to reduce the impact of contention for many different c-cores. Previous work [Venkatesh et al. 2011] describes how to merge c-cores. That approach automatically identifies target functions that are similar to one another such that generating a single coprocessor that can execute

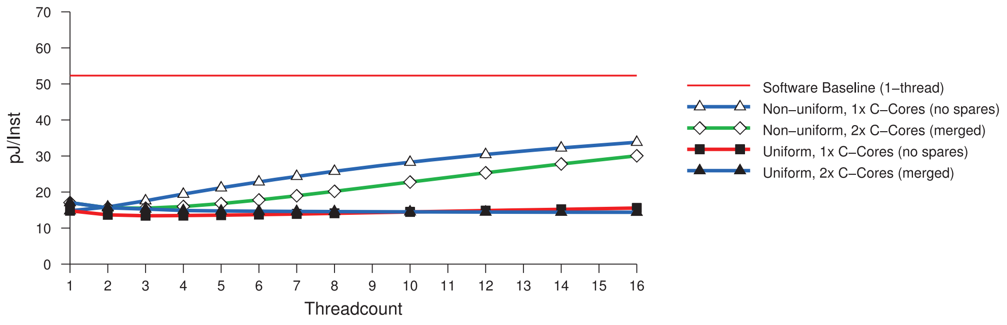

Fig. 11. *The benefits of spare c-cores.* Adding spare c-cores to CoDAs reduces the impact of contention and therefore energy per instruction. Merging the spare c-cores preserves most of the energy savings while reducing the area overhead for the spares.

either function would only be slightly larger and slightly less efficient than a dedicated c-core for each piece of code. That work shows that merging c-cores can reduce the area required to cover a given set of functions by 23% while reducing the energy efficiency of the specialized logic by 27%. Since the dynamic energy of the specialized logic represents a modest fraction of the total energy, this trade-off will often be beneficial for CoDAs targeting multithreaded workloads.

To quantify the benefits of merging, we created a CoDA that uses merged spares to provide twice as many of each type of c-core, at a cost of 41% additional area and a 15% reduction in single-threaded efficiency. Figure 11 plots the total energy per instruction of a CoDA with merged spares compared to the CoDA from Figures 9 and 10. Merging provides benefits for both the uniform (bottom two lines) and nonuniform (middle two lines) workload distributions. For the uniform case, providing spare c-cores improves energy efficiency by 7.4% at 16 threads. In the nonuniform case, where the workload is mismatched, the merged c-core CoDA improves energy efficiency by up to 22.1% (at 7 threads) and continues to provide a gain of 11.1% energy efficiency over a CoDA without merged spares at 16 threads.

6. RELATED WORK

As the dark silicon problem grows, designers are increasingly integrating specialized coprocessors into general-purpose architectures. GPUs are an especially common addition, and the latest offerings from Intel and AMD directly integrate GPUs and processors on-chip. Many recent efforts [Luk et al. 2009; Owens et al. 2005; Wang et al. 2007] attempt to harness these heterogeneous platforms with language extensions like CUDA [Nickolls et al. 2008] and streaming frameworks such as Brook [Buck et al. 2004], but they focus primarily on highly parallel code and loosely coupled execution models.

Even flexible heterogeneous processing frameworks such as Intel's EXOCHI [Wang et al. 2007] face challenges in using 1000s of distinct coprocessors in one design: EXOCHI's uniform abstraction for sequencing execution across heterogeneous execution engines requires specialized compilers for each piece of target hardware. Recent efforts have focused on automating the production and use of specialized coprocessors [Venkatesh et al. 2010; Sampson et al. 2011]. These automatically generated coprocessors do not achieve the performance of hand-crafted accelerators, but they are very energy efficient and can target nearly arbitrary code, including irregular code that is difficult to parallelize.

Previous efforts to execute the majority of applications in hardware relied on reconfigurable fabrics rather than dedicated coprocessors. Tartan [Mishra et al. 2006] mapped entire programs onto a hierarchical coarse-grained asynchronous reconfigurable fabric. Reconfigurable logic allows for greater flexibility, but estimates in Mishra et al. [2006] showed fabric virtualization is necessary to map entire programs, and that adds performance and energy overheads.

Tiled architectures, such as Raw [Taylor et al. 2004], TRIPS [Sankaralingam et al. 2003], and WaveScalar [Swanson et al. 2007], are a common approach to improving scalability because they reduce wire delay. Scalable CoDA systems also use a tiled architecture for this reason and to distribute coprocessors among multiple memory and host interfaces. The authors of GreenDroid [Goulding et al. 2010; Goulding-Hotta et al. 2011] suggested tiling as a means of organizing a coprocessor-enabled system but did not investigate the scalability problems that this work identifies and addresses.

Hannig et. al [2011] describe a model for dynamically mapping computations to a heterogeneous MPSoC via an invasive computing paradigm. While CoDA systems could potentially benefit from such an exploration of parallel resources, the current CoDA approach focuses on reducing energy for primarily serial applications. Moreover, the CoDA approach is intentionally designed to work with completely unmodified legacy code, requiring only a mapping between the functions present in a program and the functions covered in hardware, allowing complete programmer transparency. The best approach to designing new programs written with CoDA systems in mind remains a topic of future research.

Like CoDAs, previous work [Allred et al. 2012] has also proposed a methodology to design multicore systems for dark silicon. While that work operates mainly on a architecturally identical core but individually optimized for different voltage-frequency domains, and only discuess the energy efficiency of the processing cores, CoDAs operate at much finer granularity and far greater scale over diverse processing elements. In this article, we not only discuss the processing cores, but also the cache system and the interconnections.

Previous work, such as Vuletic et al. [2006], that examined interactions between multithreading and coprocessors focused heavily on device virtualization and managing the local memories within accelerators. In contrast, the c-cores in a CoDA are coherent by default and do not have large private memories. C-cores can also use merging [Venkatesh et al. 2011] to mitigate resource contention by increasing the number of c-cores capable of running a given task without increasing the number of c-cores, rather than add full-fledged virtualization.

Previous works that sought to offload the majority of execution to coprocessors [Sampson et al. 2011; Venkatesh et al. 2010] utilized clock gating, but not power gating. As Figure 6 shows, power gating is critical to the efficiency of CoDAs targeting large workloads because so much silicon will sit idle and power gated almost all of the time. This requires designers to assume, from the outset, that all processing elements in CoDAs are in the deepest-sleep state possible by default. While sensor motes [Seok et al. 2008] and other energy-critical systems have long operated with such a model, it is not the traditional model for general-purpose processors.

7. CONCLUSION

This work has examined the scalability challenges that arise with the integration of hundreds of specialized coprocessors into general-purpose architectures. Our systematic survey of the CoDA design space showed that scalable designs that cover over 100 applications can provide 3.7× improvements in energy and 3.5× improvements in energy delay across the entire workload. We found that the key limiters of the efficiency in scalable CoDA designs are leakage in dark silicon and overheads in the network and memory system that arise in large, tile-based designs.

The results suggest that contention among threads for shared coprocessors can limit efficiency gains, but that CoDAs can provide area-efficient “spare” coprocessors to provide up to 3.8× improvements in energy per instruction relative to a single-threaded workload by amortizing fixed leakage, interconnect, and memory system costs.

REFERENCES

Jason Allred, Sanghamitra Roy, and Koushik Chakraborty. 2012. Designing for dark silicon: A methodological perspective on energy efficient systems. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED’12).* ACM Press, New York, 255–260.

Mark Bohr and Kaizad Mistry. 2011. intel’s Revolutionary 22 nm Transistor Technology. http://download.intel.com/newsroom/kits/22nm/pdfs/22nm-Details_Presentation.pdf.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Graph.* 23, 3, 777–786.

Nathan Clark, Amir Hormati, and Scott Mahlke. 2008. VEAL: Virtualized execution accelerator for loops. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA’08).* IEEE Computer Society, 389–400.

Hamed F. Dadgour and Kaustav Banerjee. 2007. Design and analysis of hybrid nems-cmos circuits for ultra low-power applications. In *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC’07).* 306–311.

Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. 1974. Design of ion-implanted mosfet’s with very small physical dimensions. *IEEE J. Solid-State Circ.* 9, 5, 256–268.

Embedded Microprocessor Benchmark Consortium. 2002. Eembc benchmark suite. http://www.eembc.org.

Hadi Esmaeilzadeh, Emily Blem, Renee S. 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).* IEEE, 365–376.

Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael B. Taylor, and Steven Swanson. 2010. GreenDroid: A mobile application processor for a future of dark silicon. http://www.academia.edu/2384482/GreenDroid_A_mobile_application_processor_for_a_future_of_dark_silicon

Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael B. Taylor. 2011. The greendroid mobile application processor: An architecture for silicon’s dark future. *IEEE Micro* 31, 2, 86–95.

Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Joe Auricchio, Steven Swanson, and Michael B. Taylor. 2012. GreenDroid: An architecture for the dark silicon age. In *Proceedings of the 17th Asia and South Pacific Conference on Design Automation (ASP-DAC’12).* IEEE, 100–105.

Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro* 33, 5, 38–51.

Frank Hannig, Sascha Roloff, Gregor Snelting, Jurgen Teich, and Andreas Zwinkau. 2011. Resource-aware programming and simulation of mpsoc architectures through extension of ×10. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems (SCOPES’11).* ACM Press, New York, 48–55.

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward dark silicon in servers. *IEEE Micro* 31, 4, 6–15.

Michael B. Henry, Robert Lyerly, Leyla Nazhandali, Adam Fruehling, and Dimitrios Peroulis. 2011. MEMS-based power gating for highly scalable periodic and event-driven processing. In *Proceedings of the 24th International Conference on VLSI Design (VLSIDesign’11).* 286–291.

Michael B. Henry and Leyla Nazhandali. 2010. From transistors to mems: Throughput-aware power gating in cmos circuits. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE’10).* 130–135.

IMOD Technology Overview. 2008. IMOD technology overview. http://www.qualcomm.com/common/documents/white papers/QMT_Technology_Overview_12-07.pdf.

Independent Jpeg Group. 2002. Library for jpeg image compression. http://www.ijg.org/.

Ravi Jotwani, Sriram Sundaram, Stephen Kosonocky, Alex Schaefer, Victor Andrade, Greg Constant, Amy Novak, and Samuel Naffziger. 2010. An ×86-64 core implemented in 32nm soi cmos. In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'10)*. 106–107.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society, 75–86.

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer rchitecture (ISCA'09)*. ACM Press, New York, 2–13.

Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro'09)*. ACM Press, New York, 45–55.

Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating spatial computation for whole program execution. *SIGOPS Oper. Syst. Rev.* 40, 5, 163–174.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with cuda. In *Proceedings of the ACM SIGGRAPH Classes (SIGGRAPH'08)*. ACM Press, New York, 1–14.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. 2005. A survey of general-purpose computation on graphics hardware. In *Proceedings of the Eurographics State of the Art Reports*. 21–51.

Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael B. Taylor. 2011. Efficient Complex Operators for irregular codes. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*. 491–502.

Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM Press, News York, 422–433.

Semiconductor Industries Association. 2012. International technology roadmap for semiconductors. http://www.itrs.net/Links/2012ITRS/Home2012.htm.

Mingoo Seok, S. Hanson, Yu-Shiang Lin, Zhiyoong Foo, Daeyeon Kim, Yoonmyung Lee, Nurrachman Liu, D. Sylvester, and D. Blaauw. 2008. The phoenix processor: A 30pw platform for sensor applications. In *Proceedings of the IEEE Symposium on VLSI Circuits*. 188–189.

Standard Performance Evaluation Corporation. 2000. SPEC CPU 2000 benchmark specifications. SPEC2000 Benchmark Release. http://www.spec.org/.

Standard Performance Evaluation Corporation. 2006. SPEC CPU 2006 benchmark specifications. SPEC2006 Benchmark Release. http://www.spec.org/.

Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. 2007. The wavescalar architecture. *ACM Trans. Comput. Syst.* 25, 2, 4.

Michael B. Taylor. 2012. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th ACM/IEEE Design Automation Conference (DAC'12)*. ACM Press, New York, 1131–1136.

Michael B. Taylor. 2013. A landscape of the new dark silicon design regime. *IEEE Micro* 33, 5, 8–19.

Michael B. Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. 2004. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. IEEE Computer Society, 2–13.

Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. 2008. CACTI 5.1. Tech. rep. HPL-2008-20. HP Labs, Palo Alto, CA. http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html.

Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael B. Taylor. 2010. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the 15th International Conference Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM Press, New York, 205–218.

Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi K. Venkata, Michael B. Taylor, and Steven Swanson. 2011. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (Micro'11).* 163–174.

Miljan Vuletic, Paolo Ienne, Christopher Claus, and Walter Stechele. 2006. Multithreaded virtual-memory-enabled reconfigurable hardware accelerators. In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT'06).* 197–204.

Perry H. Wang, Jamison D. Collins, Gautham M. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. 2007. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07).* ACM Press, New York, 156–166.

Received January 2013; revised June 2013; accepted September 2013