GreenDroid: An Architecture for the Dark Silicon Age

Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Joe Auricchio,
Steven Swanson and Michael Bedford Taylor
http://greendroid.org
Department of Computer Science and Engineering
University of California, San Diego

Abstract— The Dark Silicon Age kicked off with the transition to multicore and will be characterized by a wild chase for seemingly ever-more insane architectural designs. At the heart of this transformation is the Utilization Wall, which states that, with each new process generation, the percentage of transistors that a chip can switch at full frequency is dropping exponentially due to power constraints. This has led to increasingly larger and larger fractions of a chip's silicon area that must remain passive, or dark.

Since Dark Silicon is an exponentially-worsening phenomenon, getting worse at the same rate that Moore's Law is ostensibly making process technology better, we need to seek out fundamentally new approaches to designing processors for the Dark Silicon Age. Simply tweaking existing designs is not enough. Our research attacks the Dark Silicon problem directly through a set of energy-saving accelerators, called Conservation Cores, or c-cores. C-cores are a post-multicore approach that constructively uses dark silicon to reduce the energy consumption of an application by 10× or more. To examine the utility of c-cores, we are developing GreenDroid, a multicore chip that targets the Android mobile software stack. Our mobile application processor prototype targets a 32-nm process and is comprised of hundreds of automatically generated, specialized, patchable c-cores. These cores target specific Android hotspots, including the kernel. Our preliminary results suggest that we can attain up to 11× improvement in energy efficiency using a modest amount of silicon.

I. Introduction

Over the last five years, the phenomenon known as Dark Silicon has emerged as the most fundamental factor that constrains our ability to exploit the exponentially increasing resources that Moore's Law provides. Dark Silicon refers to the exponentially increasing fraction of a chip's transistors that must remain passive, or "dark" in order to stay within a chip's power budget. Although Dark Silicon has shaped processor design since the cancellation of the Pentium 4, it was not well understood why Dark Silicon was happening, nor how bad the Dark Silicon problem would get.

In this paper, we begin by identifying the source of the Dark Silicon problem, and we characterize how bad the problem will get. (In short, it will be very bad; exponentially bad, in fact.) We continue the paper by describing our approach, called *Conservation Cores*, or *c-cores* [3, 5], which is a way to take Dark Silicon and use it to make computation much more energy efficient, effectively using Dark Silicon to combat the Utilization Wall. Our approach is to use Dark Silicon to build a large collection of specialized cores, each of which can save 11× more energy for targeted code, compared to an energy-efficient general-purpose processor. We demonstrate the Conservation Cores concept by applying the technique to the Android mobile software stack in order to build a mobile application processor that runs applications with a fraction of the energy consumption. We also examine the key scalability properties that allow Conservation Cores to target much broader bodies of code than today's custom-built accelerators.

II. Origins of Dark Silicon

To understand the Dark Silicon phenomenon better, we introduce the concept of the *Utilization Wall*:

> **Utilization Wall:** With each successive process generation, the percentage of a chip that can switch at full frequency drops exponentially due to power constraints.

In this section, we will show three sources of evidence that we've hit the Utilization Wall [3], drawing from 1) CMOS scaling theory, 2) experiments performed in our lab, and 3) observations in the wild.

A. Scaling Theory

Moore Scaling The most elementary CMOS scaling theory is derived directly from Moore's Law. If we examine two process generations, with feature widths of say 65 nm and 32 nm, it is useful for us to employ a value S, the *scaling factor*, which is the ratio of the feature widths of two process generations; in this case, $S = 65/32 = 2$. For typical process shrinks, $S = 1.4\times$. From elementary scaling theory, we know that available transistors scales as S^2, or 2× per process generation. Prior to 2005, the number of cores in early multicore processors more or less matched the availability of transistors, growing by 2× per process generation. For instance, the MIT Raw Processor had 16 cores in 180-nm, while the Tilera TILE64 version of the chip had 64 cores in 90-nm, resulting in 4× as many

cores for a scaling factor of 2×. More recently, however, the rate has slowed to just S, or 1.4×, for reasons that we shall see shortly.

Dennardian Scaling However, the computing capabilities of silicon are not summarized simply by the number of transistors that we can integrate into a chip. To more fully understand the picture, we need to also know how the properties of transistors change as they are scaled down. To understand this better, we need to turn to Robert Dennard, who besides being the inventor of DRAM, wrote a seminal 1974 paper which set down transistor scaling [1]. Dennard's paper says that while transistor count scales by S^2, the native frequency of those transistors improves by S, resulting in a net S^3 improvement in computational potential of a fixed-area silicon die. Thus, for typical scaling factors of 1.4×, we can expect to have a factor of 2.8× improvement in compute capabilities per process generation.

However, within this rosy picture lies a potential problem – if transistor energy efficiency does not also scale as S^3, we will end up having chips that have exponentially rising energy consumption, because we are switching S^3 more transistors per unit time. Fortunately, Dennard outlined a solution to this exponential problem. First, the switching capacitance of transistors drops by a factor of S with scaling, and if we scale the supply voltage, V_{dd}, by S, then we reduce the energy consumption by an additional S^2. As a result, the energy consumption of a transistor transition drops by S^3, exactly matching the improvements in transistor transitions per unit time. In short, with standard V_{dd} scaling, we were able to have our transistors, *AND* switch them at full speed.

Post-Dennardian Scaling Starting in 2005, Dennardian scaling started to break down. The root of the problem was that scaling V_{dd} requires a commensurate reduction in V_t, the threshold voltage of the transistor, in order to maintain transistor performance[1]. Unfortunately V_t reduction causes leakage to increase exponentially at a rate determined by the processes' sub-threshold slope, typically 90 mV per decade; e.g., 10× increase in leakage for every 90 mV reduction in threshold voltage.[2] At this point in time, this leakage energy became so large that it could not be reasonably increased. As a result, V_t values could not be scaled, and therefore neither could V_{dd}.

The end result is that we have lost V_{dd} scaling as an effective way to offset the increase in the computing potential of the underlying silicon. As a result, with each process generation, we gain only $S = 1.4\times$ improvement in energy efficiency, which means that, under fixed power budgets, our utilization of the silicon will drop by $S^3/S = S^2 = 2\times$ per process generation. This is what we mean by the *Utilization Wall*. Exponentially growing numbers of transistors must be left underclocked to stay within the power budget, resulting in Dim or Dark Silicon.

B. Experiments in Our Lab

To confirm the Utilization Wall, we performed a series of experiments in our lab using a TSMC process and a standard Synopsys IC Compiler flow. Using 90-nm and 45-nm technology files, we synthesized two 40 mm^2 chips filled with ALUs – 32-bit adders sandwiched between two flip-flops. Running the 90-nm chip at the native operating frequency of these ALUs, we found that only 5% of the chip could be run at full frequency in a 3-W power budget typical of mobile devices. In 45-nm, this fraction dropped to 1.8%, a factor of 2.8×. Using ITRS projections, a 32-nm chip would drop to 0.9%. We obtained similar results for desktop-like platforms with 200 mm^2 of area and an 80-W power budget.

These numbers often seem suspiciously low – after all, 90-nm designs were only just beginning to experience power issues. The explanation is that RAMs typically have 1/10 the utilization per unit area compared to datapath logic. However, the point is not so much what the exact percentage is for any process node, but rather, that once the Utilization Wall starts to become a problem, it will become *exponentially worse* from then on. This exponential worsening means that the onset of the problem is very quick, and is in part responsible for why industry was taken by surprise by the power problem in 2005.

C. Industrial Designs as Evidence of the Utilization Wall

The Utilization Wall is also written all over the commercial endeavors of many microprocessor companies. One salient example of a trend that reflects the Utilization Wall is the flat frequency curve of processors from 2005 onward. The underlying transistors have in fact gotten much faster, but frequencies have been held flat. Another example is the emergence of Intel and AMD's turbo boost feature, which allows a single core to run faster if the other cores are not in use. We are also observing an increased fraction of chips dedicated to lower frequency and lower activity-factor logic such as L3 cache and so-called uncore logic – i.e., memory controllers and other support logic.

The industrial switch to multicore is also a consequence of the Utilization Wall. Ironically, multicore itself is not a direct solution to the Utilization Wall problem. Originally, when multicore was proposed as a new direction, it was postulated that the number of cores would double with each process generation, increasing with transistor count. However, this is in violation of the Utilization Wall, which says that computing capabilities can only increase at the same rate as energy efficiency improves, i.e., at a rate of S. Looking at Intel 65-W desktop processors, with two cores in 65-nm, and four cores in 32-nm, we can compute S (= 2×); and also increase in core count (= 2×), and increase

[1]This is because V_{dd} *overdrive* (= V_{dd}/V_t) values less than 2.5× cause massive drops in transistor performance.

[2]Small sub-threshold slope values are better, because they result in lower required threshold voltages to reduce leakage to a given level. Sub-threshold slope of MOSFETs cannot be lower than 60 mV per decade at room temperature, as it is set by thermionic emission of electrons across a potential well. Gradually, as we shrink transistors down, they become less ideal, and the sub-threshold slope worsens. Innovations such as Intel's Tri-Gate serve to reduce this non-ideal behavior with a one-time improvement. If we are to find devices without leakage limitations to combat Dark Silicon, we must look for non-MOSFET devices!

in frequency (roughly constant at ∼3 GHz), and see that scaling has occurred consistent with the Utilization Wall, and not with earlier predictions.

One interesting observation is that the Utilization Wall says that there is a spectrum of other design points that could have been done trading off processor frequency and core count, with the extreme end being to, with each process generation, increase frequency instead of core count. This would result in, for the previous example, two-core 32-nm processors running at ∼6 GHz. Conventional wisdom says that this higher frequency design would have better uni-processor performance and be more preferable because it applies to all computations, not just parallel computations. The jury is still out on this. However, for throughput-oriented computations, the higher frequency design is still worse. The reason is that the cost of a cache miss to DRAM, as measured in ALU ops lost, is lower for lower-clocked multicore chips, so in the face of cache misses and given sufficient throughput, higher core count is more performant than higher frequency.

III. Conservation Cores

Now that we know that Dark Silicon is an inevitable and exponentially worsening problem, what do we do with Dark Silicon? Our group has developed a set of techniques that allow us to leverage Dark Silicon to fight the Utilization Wall. For our research, we draw from two insights. First, *power is now more expensive than area.* Thus, if we can find architectural ways to trade area for power, this is a good architectural trade-off. In fact, area is a resource that is becoming exponentially cheaper with each process generation, while power efficiency is something that requires massive engineering effort and offers diminishing returns with conventional optimization approaches. The second insight is that specialized logic has been shown as a promising way to improve energy efficiency by 10–1000×.

As a result of these insights, we have developed an approach that fills Dark Silicon with specialized energy-saving coprocessors that save energy on commonly executed applications. The idea is that you only turn on the coprocessors as you need them, and execution jumps from core to core according to the needs of the computation. The rest of the cores are power-gated. As a result of the specialized coprocessors, we execute the hotspots of the computation with vastly more energy efficiency. In effect, we are recouping the S^2 energy efficiency lost by the lack of V_{dd} scaling by using Dark Silicon to exploit specialization.

A. Related Work: Accelerators

Accelerators are another class of specialized core that has been used to improve energy efficiency and has found widespread use in smartphones and other systems. Conservation Cores overcome some of the key limitations of accelerators, which include:

- **Speedup Fixation** Accelerators fixate on speedup of target code, while energy savings is a secondary goal. In contrast, Conservation Cores target energy savings as their primary goal; performance is a secondary goal. With Dark Silicon, energy efficiency eclipses performance as a concern. Since, as we shall see, attaining speedup versus a processor is a fundamentally harder problem than attaining energy savings, it makes sense to re-prioritize on saving energy.
- **Regular Computations** Accelerators generally rely upon exploitation of program structure for improvements in performance and energy efficiency. We refer to the code targeted by accelerators as being *regular*, i.e., possessing properties that make it relatively amenable to parallelization. These properties include moderate or high levels of parallelism, predictable memory accesses and branch directions, and small numbers of lines of code. Even with this structure, accelerators tend to require human guidance, such as `#pragmas`, or manual transformation, in order to attain success.
- **Parallelization Required** Because the transformations required to generate accelerators [2] generally correspond to the same transformations that parallelizing compilers perform (e.g., pointer analysis and deep code transformations), accelerator generation is seldom automated or scalable, inheriting the very same problems that have inhibited widespread industrial use of parallelizing compilers. Instead, accelerator creation tends to be successful only when multi-man-year efforts are applied, or in cases where the underlying algorithm in mind has been explicitly designed for hardware.
- **Static Code Bases** Accelerators tend to target relatively static code bases that do not evolve. In many cases, the evolution of the target code base is intentionally limited through the use of standards (e.g., JPEG), or internal specification documents.

Accelerators are thus limited in their applicability. Amdahl's Law tells us that the achievable benefits attainable by an optimization are limited by the fraction of the workload the optimization targets. Thus, in order to get widespread energy-savings, we need to broaden the applicability of coprocessors to all code, including code that changes frequently, code that is irregular, and code that is not parallelizable. This is the goal of Conservation Cores.

B. Conservation Core Architecture

Conservation Cores, or c-cores, are a class of specialized coprocessors that targets the reduction of energy across all code, including irregular code. C-cores are always paired with an energy-efficient general-purpose host CPU, and perform all of their memory operations through the same L1 data cache as the host core. Frequently-executed hot code regions are implemented using the c-cores, while the cold code regions are executed on the host CPU. Because the data cache is shared, the memory system is coherent between the c-core and host CPU, and, unlike GPUs, the

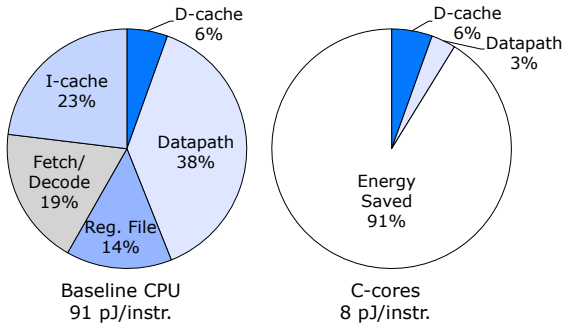


Fig. 1. **Energy savings in c-cores** Removing hardware structures responsible for fetching and decoding instructions as well as generalized datapath components, reduces per-instruction energy by 91%.

App.	C-cores	Area (mm^2)	Freq. (GHz)	Energy Improvement
bzip2	1	0.18	1.24	10.89×
twolf	1	0.12	1.26	10.67×
vpr	1	0.24	1.10	7.72×
mcf	3	0.17	1.41	12.74×
radix	1	0.10	1.36	13.57×
cjpeg	3	0.18	1.45	16.51×
sat	2	0.20	1.28	15.10×
viterbi	1	0.12	1.26	17.61×
djpeg	3	0.21	1.46	18.28×
Average		0.17	1.31	13.7×

TABLE I
C-CORE AREA AND FREQUENCIES FOR A SELECTION OF IRREGULAR CODES IN A 45-NM PROCESS.

migration of execution between c-core and host CPU can be performed in approximately the same overhead as a function call.

In order to address the issue of software change, c-cores have support for patching in hardware; that is, a bitstream can be downloaded into the c-core to allow the c-core to track the changes in the software that was used to generate the hardware in the first place.

C-cores do not rely upon deep compiler analysis; in fact, the hardware generated maps nearly one-to-one to the original control and dataflow graphs of the original source code. Because of this c-cores can be generated from any kind of code – whether from the Linux kernel, or from Java. They do not rely on brittle compiler analyses and transformations such as pointer analyses and affine loop transformations.

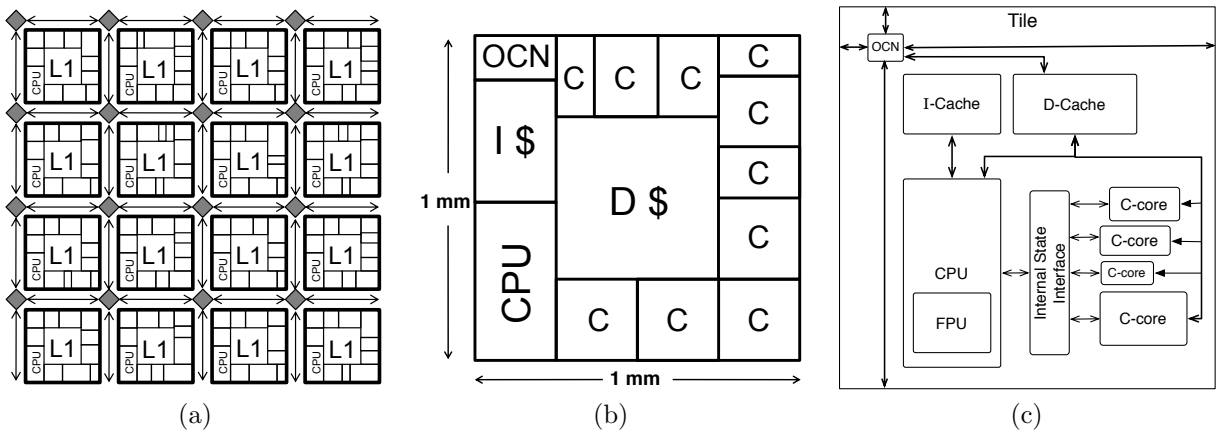
Because they do not leverage parallelization technology, Conservation Cores generally have only slight improvements in performance versus general-purpose cores. However, they can have up to 18× improvements in energy efficiency.

Life Cycle of a C-core The life cycle of a c-core begins with a set of relatively stable versions of applications – i.e., at least version 1.0 of the targeted software. These versions are profiled in order to identify a set of energy-intensive code regions. From these regions, we generate a set of c-core specifications. The specifications are then used to generate Verilog, which is then used to generate the c-core-enabled hardware that links the c-core through the data cache of the processor to an array of host CPUs. At the same time, the specifications are used as input to a patching-aware compiler. The patching-aware compiler maintains a library of the available c-cores, and then uses code recognition algorithms to map the original code onto the c-cores. In cases where the source code is a variant of the original code, the compiler is able to generate a bit stream that configures the behavior of the c-core to track the changes in the software. One notable benefit of this system is that the existence of the c-cores is hidden from the programmer. As a result, we do not need to worry about maintaining backward compatibility, and can retire c-cores in future chips as the underlying programs evolve.

Generation of a C-core The generation of c-cores a relatively straightforward process. It is parallelism- and regularity-agnostic, and employs a function-call, rather than trace-based or hyperblock-based interface. The first step in our compiler pass is what we call *code reconstitution* – this is essentially a set of reshaping transformations that better exposes the hot regions of code, and removes non-hot regions. Typically, this process involves outlining non-essential code regions, and inlining the essential ones. C-core generation supports complex control flow, arbitrary memory access patterns, data structures, etc. No parallelizing compiler passes are employed. After the code has been reshaped, we build a control flow graph (CFG) for the target region, and a data flow graph for each basic block. Each instruction in the basic block is directly converted into a hardware operator; e.g., an add turns into a hardware adder, and a move operation turns into a wire. In a few cases, we multiplex large-area operators, such as floating point operators and integer multipliers. Loads and stores generate small adapters that feed into a large mux at the input to the L1 cache. The live-ins of the data flow graphs are turned into registers. In addition to the datapath logic, we also create control logic that sequences the multiplexed operators, and stalls the sequencing logic appropriately on a cache miss. This logic is all represented using Verilog, which is then run through a Synopsys IC Compiler flow to perform synthesis, placement, clock tree generation, and routing.

Experimental C-core Data In our earlier research, we automatically built 21 c-cores for 9 "hard" applications, drawn from SPECINT and other irregular applications, using a 45-nm TSMC process. The results are shown in Table I. Overall, the area requirements are relatively modest, averaging 0.17 mm^2, while energy improvements for operators converted into c-core logic average 13.7×. With D-cache included, savings average 11×.

Source of Energy Savings The primary source of en-



(a) (b) (c)

Fig. 2. **The GreenDroid architecture** The GreenDroid system (a) is made up of 16 non-identical tiles. Each tile (b) holds components common to every tile—the CPU, on-chip network (OCN), and shared L1 data cache—and provides space for multiple c-cores of various sizes. (c) shows connections among these components and the c-cores.

ergy savings for c-cores can be ascertained from Figure 1. A baseline energy-efficient in-order MIPS host CPU consumes 91 pJ/instruction, most of which is spent on various overheads due to aspects of instruction interpretation, including I-cache (23%), Fetch/Decode(19%), Register File (14%), and Datapath (38%). C-cores, on the other hand, eliminate 91% of the energy used by the host CPU, inheriting only the D-cache power (6%) and the portion of the original datapath energy that is involved in performing the actual operations (3%). Thus, c-cores reduce instruction energy from those costs incurred by an instruction marching down a pipeline to those costs incurred by two operators such as adders placed nearby by wires. The end result is an average of 11× reduction in energy per instruction.

Supporting Software Changes Since large bodies of source code are likely to change due to bug fixes and feature enhancements, robust coprocessors must be able to continue to function even if the underlying source code changes. Conservation Cores offer three levels of protection against software change. First, if the changed software is not contained in the hotspots of the code, then they do not affect c-core operation. Second, c-cores support a general-purpose patching exception mechanism that allows arbitrary insertion of code into the code targeted by the c-core. Each edge in the CFG is annotated with an exception bit, which allows the c-core state machine to transition control over to the host CPU. The host CPU can then access the c-core's state via a state tree, a fast mechanism that exposes all of the c-core's state to be read and written. The host CPU will read the required state to execute the changed code, and then write it back into the c-core and resume execution. The third level of protection optimizes for common-case changes that we discovered by examining a 10-year span of versions of the software we targeted. Here, we add in configurable registers that allow constants such as field offsets and `#define`'s to change; we found this is very important because simply adding a field to the middle of a struct will instantly change the offsets in the code, invalidating a c-core without this support. We also generalize operators slightly to allow them to be changed via configuration state, such as turning less-than compares into less-than-or-equals, and changing adders into add/subtractors.

IV. The GreenDroid Mobile Application Processor

In order to validate Conservation Cores, our team at UC San Diego is building the GreenDroid mobile application processor [7, 4] that uses c-cores to drastically reduce the energy consumption of the Android-based mobile software stack. Mobile application processors are an exciting new centroid of microprocessor evolution, taking over the excitement and center of gravity from desktop processors, just as those desktop processors took over from mini- and main-frame processors that preceded them. Now that these processors support dual and quad-core configurations, there is little left to borrow from their more power-hungry ancestors. We believe that Conservation Cores are the next logical development after multicore, brought on by the emergence of Dark Silicon.

Desirable Characteristics of the Android Mobile Software Stack Although we believe that Conservation Cores will apply generally even for desktop computations, they are an especially good fit for the Android mobile stack, because user time is concentrated among a relatively small number of apps. This means that building a relatively small number of Conservation Cores can potentially have a very large impact on the energy efficiency of the mobile application processing. According to a recent 2011 study by Neilsen Smartphone Analytics, 33% of user time is spent in the web browser; and another 40% of user time is spent in the top 50 apps after the web browser. Fine-grained extrapolation of the data suggests that as much as 83% of application time is spent in the top 100 Android applications. This means that the mobile workload is highly concentrated, and that a relatively

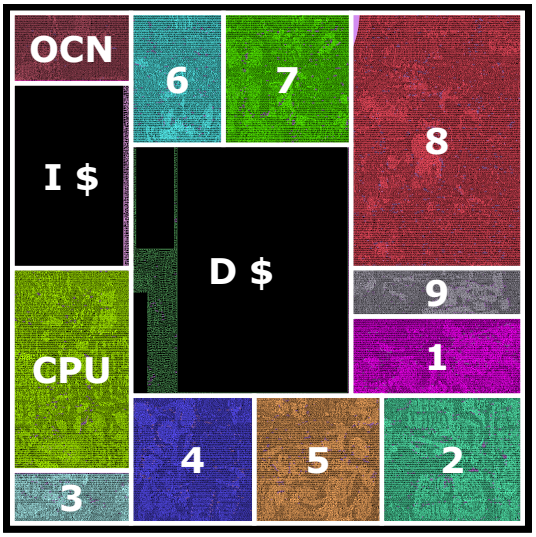


Fig. 3. **One tile of the GreenDroid processor** This tile contains the standard tile components (CPU, I- and D-caches, and on-chip network (OCN)) in addition to nine c-cores (numbered 1–9) that target Android's 2D graphics library, JPEG decompression, and FFT functions.

small amount of area (our estimate is 17 mm^2 in a 22-nm process) is sufficient to cover it. (Of course, some games like Angry Birds may pass too quickly for us to capture in silicon!)

Addressing Scalability in GreenDroid We anticipate that GreenDroid systems will have hundreds or even thousands of Conservation Cores in order to support large numbers of applications. Since c-cores connect to the data cache, there are scalability limits that occur as a result of lengthening wires as we attach more and more c-cores: Both energy and delay worsen. To address this problem, we envision a tiled system that has multiple tiles attached by a low-power interconnect. Each tile consists of 10-15 c-cores, a single host CPU, their shared L1 data cache, and a network router that connects to L2 cache and main memory. Figure 2 shows the GreenDroid architecture.

32-nm GreenDroid Prototype A prototyping effort is well underway. We are targeting a 32-nm Global Foundries process; our first prototype is a 2-mm^2 four-tile chip with approximately 60 c-cores. Each tile has a 32-KB data cache, a host CPU including 16-KB instruction cache, four routers, and a selection of different c-cores. Figure 3 shows one of the four tiles of the first GreenDroid prototype chip.

Bringup and Emulation System In order to perform system bringup, we have a Xilinx ML-605 board which hosts an FMC daughtercard that will have a GreenDroid chip and an FPGA attached. All I/O and DRAM access will be performed over the FMC connector to an FPGA-based chipset that processes off-chip messages into requests to the ML-605's DDR-3 DRAM and I/O devices.

For verification purposes, the system is already running in emulation on the ML-605 board; instead of having a daughterboard with FPGA, we merely map the GreenDroid RTL to the ML-605's FPGA, and validated c-cores with much larger input sizes than will run on our C and RTL simulation infrastructure.

Our design is undergoing iterative tuning, validation, and improved calibration with our Android infrastructure. After we are satisfied with the design in emulation, we will move the RTL and verification suite over from our internal TSMC flow to the finalized Global Foundries flow for final tapeout. Our RTL is designed to be highly portable and process-independent; we have already ported it between IBM, Xilinx, and TSMC flows with little trouble.

In our subsequent prototypes, we plan to increase the number of tiles, improve our support for extreme power-gating, and iterate on a more mature memory system and interconnect design.

V. Conclusions

The Utilization Wall leads to an exponential worsening of the Dark Silicon problem, and will transform how we implement computation. The severity of the problem argues for developing new architectural tradeoffs that trade Dark Silicon, an exponentially cheapening resource, for energy, which is the true limiter of performance today. Our work on Conservation Cores and GreenDroid offers one potential way to attack the Dark Silicon problem; we believe that Dark Silicon is an exciting new area which will open up new opportunities to rethink the computation stack.

Acknowledgments

This research was funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

References

[1] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions", IEEE Journal of Solid-State Circuits, October 1974.

[2] P. Coussy, and A. Morawiec, "High-Level Synthesis: From Algorithm to Digital Circuit", Springer Publishing Company, 2008.

[3] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations", ASPLOS, 2010.

[4] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. B. Taylor, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future", IEEE Micro, March 2011, pp. 86-95.

[5] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor, "Efficient Complex Operators for Irregular Codes", HPCA, 2011.

[6] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. Kota Venkata, M. B. Taylor, and S. Swanson. "QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores", MICRO 2011.

[7] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. B. Taylor, and S. Swanson, "GreenDroid: A Mobile Application Processor: A Mobile Application Processor for a Future of Dark Future", HOTCHIPS 2010.