The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future

Nathan Goulding-Hotta
Jack Sampson
Ganesh Venkatesh
Saturnino Garcia
Joe Auricchio
Po-Chao Huang
Manish Arora
Siddhartha Nath
Vikram Bhatt
Jonathan Babb
Steven Swanson
Michael Bedford Taylor
University of California, San Diego

Dark silicon has emerged as the fundamental limiter in modern processor design. The GreenDroid mobile application processor demonstrates an approach that uses dark silicon to execute general-purpose smartphone applications with 11 times less energy than today's most energy-efficient designs.

• • • • • • The GreenDroid mobile application processor is a 45-nm multicore research prototype that targets the Android mobile-phone software stack and can execute general-purpose mobile programs with 11 times less energy than today's most energy-efficient designs, at similar or better performance levels. It does this through the use of a hundred or so automatically generated, highly specialized, energy-reducing cores, called *conservation cores*.

Our research attacks a key technological problem for microprocessor architects, which we call the *utilization wall*.[1] The utilization wall says that, with each process generation, the percentage of transistors that a chip design can switch at full frequency drops exponentially because of power constraints. A direct consequence of this is *dark silicon*—large swaths of a chip's silicon area that must remain mostly passive to stay within the chip's power budget. Currently, only about 1 percent of a modest-sized 32-nm mobile chip can switch at full frequency within a 3-W power budget.

With each process generation, dark silicon gets exponentially cheaper, whereas the power budget is becoming exponentially more valuable. Our research leverages two key insights. First, it makes sense to find architectural techniques that trade this cheap resource, dark silicon, for the more valuable resource, energy efficiency. Second, specialized logic can attain 10× to 1,000× better energy efficiency over general-purpose processors. Our approach is to fill a chip's dark silicon area with specialized cores to save energy on common applications. These cores are automatically generated from the code base that the processor is intended to run—that is, the Android mobile-phone software stack. The cores feature a focused reconfigurability so that they can remain useful even as the code they target evolves.

Published by the IEEE Computer Society

0272-1732/11/$26.00 © 2011 IEEE

Understanding the Origins of the Utilization Wall

We demonstrate the origin of the utilization wall in two ways. First, we extend Dennard's CMOS scaling theory to include limits on per-device power scaling because of leakage-related limits on threshold voltage scaling.[1] Second, we demonstrate the utilization wall using our own experimental results.

Table A. Classical vs. leakage-limited scaling. Under the leakage-limited regime, the total chip utilization for a fixed-power budget drops by a factor of S^2 with each process generation.

Transistor property	Classical scaling	Leakage-limited scaling
ΔV_t (threshold voltage)	$1/S$	1
ΔV_{DD} (supply voltage) $\approx V_t \times 3$	$1/S$	1
Δ quantity	S^2	S^2
Δ frequency	S	S
Δ capacitance	$1/S$	$1/S$
$\Rightarrow \Delta$ power $= \Delta(QFCV_{DD}^2)$	1	S^2
$\Rightarrow \Delta$ utilization $=$ 1/power	1	$1/S^2$

CMOS scaling

Table A shows how transistor properties change with each process generation, where S is the scaling factor. For instance, when moving from a 45-nm to a 32-nm process generation, S would be $45/32 = 1.4$. The "classical scaling" column shows how transistor properties changed before 2005, when it was possible to scale the threshold voltage and the supply voltage together. The "leakage-limited scaling" column shows how chip properties changed once we could no longer easily lower threshold or supply voltage without causing either exponential increases in leakage or transistor delay.

In both cases, the quantity of transistors increases by a multiplicative factor of S^2, their native operating frequency increases by S, and their capacitance decreases by $1/S$. However, the two cases differ in supply voltage (V_{DD}) scaling: Under classical scaling, V_{DD} goes down by $1/S$, but in the leakage-limited regime, V_{DD} remains fixed because the threshold voltage (V_t) cannot be scaled. When scaling down to the next process generation, the change in a design's power consumption is the product of all of these terms, with additional squaring for the V_{DD} term.

As Table A shows, although classical scaling resulted in constant power between process generations, power is now increasing as S^2. Because our power budget is constant, the utilization of the silicon resources is actually dropping by $1/S^2$, or a factor of 2 with every process generation.

Empirical results

To confirm the leakage-limited CMOS scaling theory's predictions, we conducted some experiments on current fabrication processes and by using projections from the 2009 *International Technology Roadmap for Semiconductors* (*ITRS*). We replicated a small data path (an arithmetic logic unit and two registers) across a 40-mm^2 chip in a 90-nm Taiwan Semiconductor Manufacturing Company (TSMC) process. We found that only 5 percent of the chip could run at full speed within a 3-W power budget. At 45 nm, this number dropped to 1.8 percent, a factor of 2.8×. *ITRS* projections put the utilization of the same design ported to 32 nm at a paltry 0.9 percent.

Reference

1. G. Venkatesh et al., "Conservation Cores: Reducing the Energy of Mature Computations," *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 2010, pp. 205-218.

The utilization wall

Although Moore's law continues to offer exponential increases in transistor count—especially with the promise of 3D integration—CMOS scaling has broken down. We refer to CMOS scaling as the scaling of transistor properties as set down by Dennard in his 1974 paper.[2] It is this breakdown of CMOS scaling that led to the industrial shift from single-threaded to multicore processors around 2005. (The "Understanding the Origins of the Utilization Wall" sidebar explains this breakdown in greater detail.) Although a fixed-size chip's computing capabilities continue to increase exponentially at 2.8× per process generation owing to both increases in maximum transistor count (2.0×) and improved transistor frequencies (1.4×), the underlying energy efficiency of the transistors is only improving at a rate of about 1.4×. Because they must adhere to a fixed power budget, chip designers can only exploit these improved capabilities to the extent they are matched by an equivalent improvement in energy efficiency. The shortfall of 2× per generation is the cause of the utilization wall, and leads to the exponentially worsening problem of dark silicon.

The utilization wall problem is already apparent indirectly through the product lines of major processor manufacturers. Processor frequencies haven't increased for almost half a decade, and the number of cores on a chip hasn't been scaling at the same rate as the increase in the number of transistors. An increasing percentage of each chip is being dedicated to cache or low-activity logic such as memory controllers and portions of the processor's chipset. Recently, Intel's Nehalem architecture has featured a Turbo Boost mode that runs some cores faster if the others are switched off.

All of these observations show that the utilization wall is strongly shaping the evolution of processor designs. CMOS scaling theory indicates that things are going to get exponentially worse. Future architectures that try to maximize the benefit due to new process generations will need to be consciously designed to leverage many, many transistors, in a way that uses only a tiny fraction of them at a time. GreenDroid's conservation cores have these exact properties and can be used to relax the utilization wall's extreme power constraints.

The GreenDroid architecture

The GreenDroid architecture uses specialized, energy-efficient processors, called conservation cores, or c-cores,[1,3] to execute frequently used portions of the application code. Collectively, the c-cores span approximately 95 percent of the execution time of our test Android-based workload.

Figure 1 shows the high-level architecture of a GreenDroid system. The system comprises an array of tiles (Figure 1a). Each tile uses a standard template (Figure 1b) of an energy-efficient in-order processor, a 32-Kbyte banked Level 1 (L1) data cache, and a point-to-point mesh interconnect (on-chip network, or OCN). The OCN is used for memory traffic and synchronization, similar to the Raw scalable tiled architecture.[4] Each tile is unique and is configured with an array of 8 to 15 c-cores, which are tightly coupled to the host CPU via the L1 data cache and a specialized interface, shown in Figure 1c. This interface lets the host CPU pass arguments to the c-core, perform context switches, and reconfigure the hardware to adapt to changes in the application code.

To create GreenDroid, we profiled the target workload to determine the execution hot spots—the regions of code where the processor spends most of its time. Using our fully automated toolchain, we automatically transform these hot spots into specialized circuits, which are attached to a nearby host CPU via the shared L1 cache. The cold code—that is, the less frequently executed code—runs on the host CPU, whereas the c-cores handle the hot code. Because the c-cores access data primarily through the shared L1 cache, execution can jump back and forth between a c-core and the CPU as it moves from hot code to cold code and back. The specialized circuits that comprise the c-cores are generated in a stylized way that maintains a correspondence with the original program code. They contain extra logic that allows patching—that is, modification of the c-core's behavior as the code that generated the c-core evolves with new software releases. This logic also lets the CPU inspect the code's interior variables during c-core execution. The c-cores' existence is largely transparent to the programmer; a specialized compiler is responsible for recognizing regions of code that align well with the c-cores and generating CPU code and c-core patches, and a runtime system manages the allocation of c-cores to programs according to availability.

The c-cores average 18× less energy per instruction for the code that's translated into specialized circuits. With such high savings, we must pay attention to Amdahl's law-style effects, which say that overall system energy savings are negatively impacted by three things: the energy for running cold code on the less-efficient host CPU, the energy spent in the L1 cache, and the energy spent in leakage and for clock power. We reduce the first effect by attaining high execution coverage on the c-cores, targeting regions that cover even less than 1 percent of total execution coverage. We've attacked the last two through novel memory system optimizations, power gating, and clock power reduction techniques.

Implementation details

Each tile's CPU is a full-featured 32-bit, seven-stage, in-order pipeline, and features a single-precision floating-point unit (FPU),

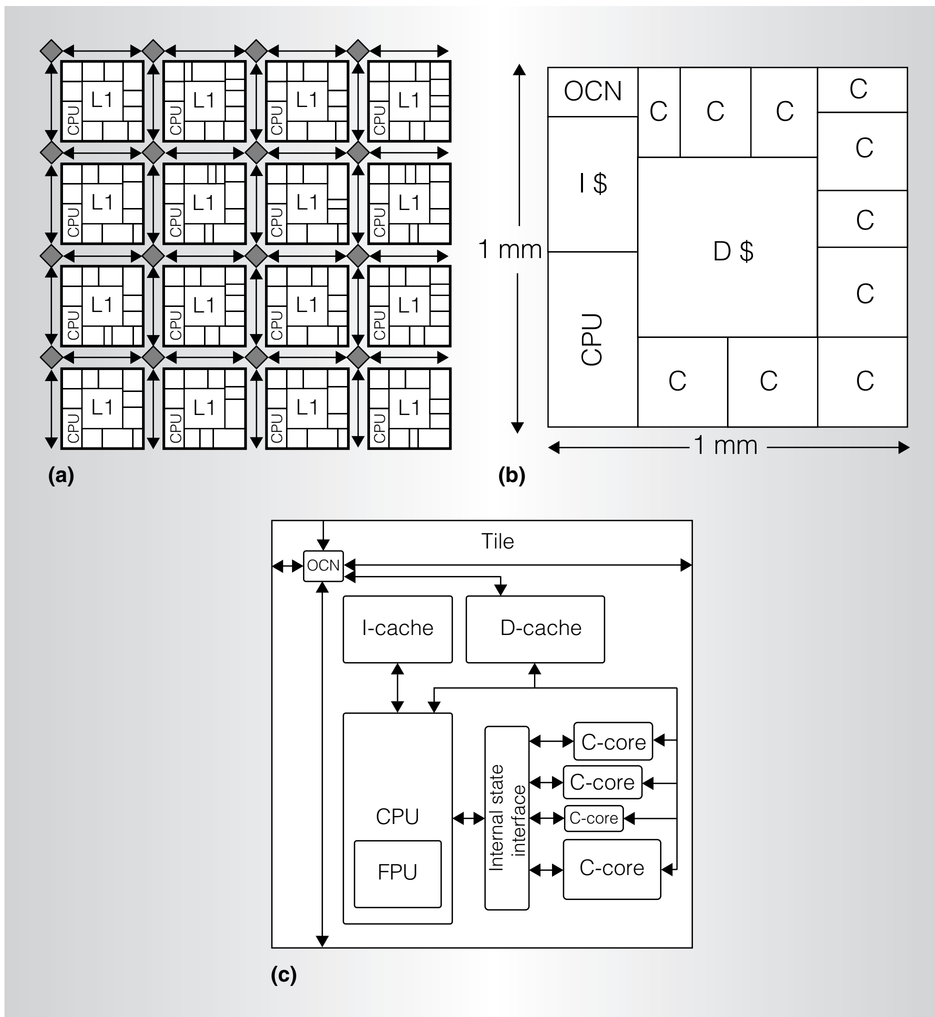

Figure 1. The GreenDroid architecture. The system comprises 16 nonidentical tiles (a). Each tile holds components common to every tile—the CPU, on-chip network (OCN), and shared Level 1 (L1) data cache—and provides space for multiple conservation cores (c-cores) of various sizes (b). The c-cores are tightly coupled to the host CPU via the L1 data cache and a specialized interface (c). (C: c-core; D $: data cache; I $: instruction cache; FPU: floating-point unit.)

a multiplier, a 16-Kbyte instruction cache, a translation look-aside buffer (TLB), and a 32-Kbyte banked L1 data cache. Our frequency target of 1.5 GHz is set by the cache access time, and is a reasonably aggressive frequency for a 45-nm design. The tiles' L1 data caches are used to collectively provide a large L2 for the system. Cache coherence between cores is provided by lightweight L2 directories residing at the DRAM interfaces (on the side of the array of tiles; not pictured in Figure 1), which use the L1 caches of all the cores as a victim cache. In addition to sharing the data cache, the c-cores optionally share the FPU and multiplier with the CPU, depending on the code's execution requirements. Collectively, the tiles in the GreenDroid system exceed the system's power budget. As a result, most of the c-cores and tiles are usually power gated to reduce energy consumption.

Execution model

At design time, the tool clusters c-cores on the basis of profiling of Android workloads, examining both control flow and data movement between code regions. It places related c-cores on the same or

nearby tiles, and in some cases, replicates them. At runtime, an application starts on one of the general-purpose CPUs, and whenever the CPU enters a hot-code region, transfers execution to the appropriate c-core. Execution moves from tile to tile on the basis of the applications that are currently active and the c-cores they use. Coherent caches let data be automatically pulled to where it's needed, but data associated with a given c-core will generally stay in that c-core's L1 cache. We use aggressive power and clock gating to reduce static power dissipation.

Targeting the Android mobile software stack

Android is an open-source mobile software stack developed by Google that features a Linux kernel, a set of application libraries, and a virtual machine called Dalvik. User applications, such as those available in the application store, run on top of the Dalvik virtual machine.

We found that Android is well-suited for c-cores for several reasons. First, although many applications are available for download, Android has a core set of commonly used applications, such as a Web browser, an e-mail client, and media players. Typically, hot code is concentrated in the application libraries, the Dalvik virtual machine, and a few locations in the kernel. Because the hot code is well concentrated, targeting all these components with c-cores lets us attain high coverage over the source base and a significant impact on overall energy usage. Although c-cores support patching, which reduces the impact of post-silicon source base modification, we are also aided by smartphones' short replacement cycle (typically every 2 to 3 years), which lets smartphone chip designers deploy new c-cores to target new applications. The c-cores interface lets Android phone designers remove c-cores from their designs without impacting code compatibility.

In our experiments with Android-based workloads—which included the Web browser, Mail, Maps, Video Player, Pandora, and many other applications—we could cover 95 percent of the Android system using just 43,000 static instructions—about 7 mm^2 of c-cores in a 45-nm process. Of this 95 percent, approximately 72 percent of the code was library or Dalvik code shared between multiple applications within the workload.

Creating GreenDroid's conservation cores

An individual c-core comprises a data path and control state machine derived directly from the code it targets. By design, these data path and control components mimic the structure of the C source code. The data path contains functional units (such as adders and shifters) to execute instructions, multiplexers to implement control decisions, and registers to hold program values across clock cycles. The control unit implements a state machine that mirrors the code's control flow graph (CFG). It also tracks branch outcomes (computed in the data path) to determine which hardware basic block will be active during each cycle.

C-cores enforce memory ordering constraints by issuing at most one memory operation per cycle to a pipelined, in-order cache interface. Both the c-core and the cache block on misses. The load/store units connect to a coherent data cache that ensures that all loads and stores are visible to the rest of the system regardless of which addresses the c-core accesses.

Most of the communication between c-cores and the CPU occurs via the shared L1 cache. A coherent, shared memory interface lets us construct c-cores for applications with unpredictable access patterns. Conventional accelerators can't speed up these applications because they can't extract enough memory parallelism. C-cores, however, can attain energy savings even in the absence of memory parallelism.

Figure 2 shows the translation from C code (Figure 2a) to hardware schematic and state machine (Figure 2c). The hardware corresponds closely to the internal compiler representation of the sample code (Figure 2b). It has a multiplexer for the variable *i*—defined in two basic blocks—corresponding to the CFG's *phi* operator, which sets a value based on a branch outcome. Also, the c-core's state machine is almost identical to the CFG, but with additional self-looping substates for memory and other multicycle operations. The data path has a load unit and a store unit to access

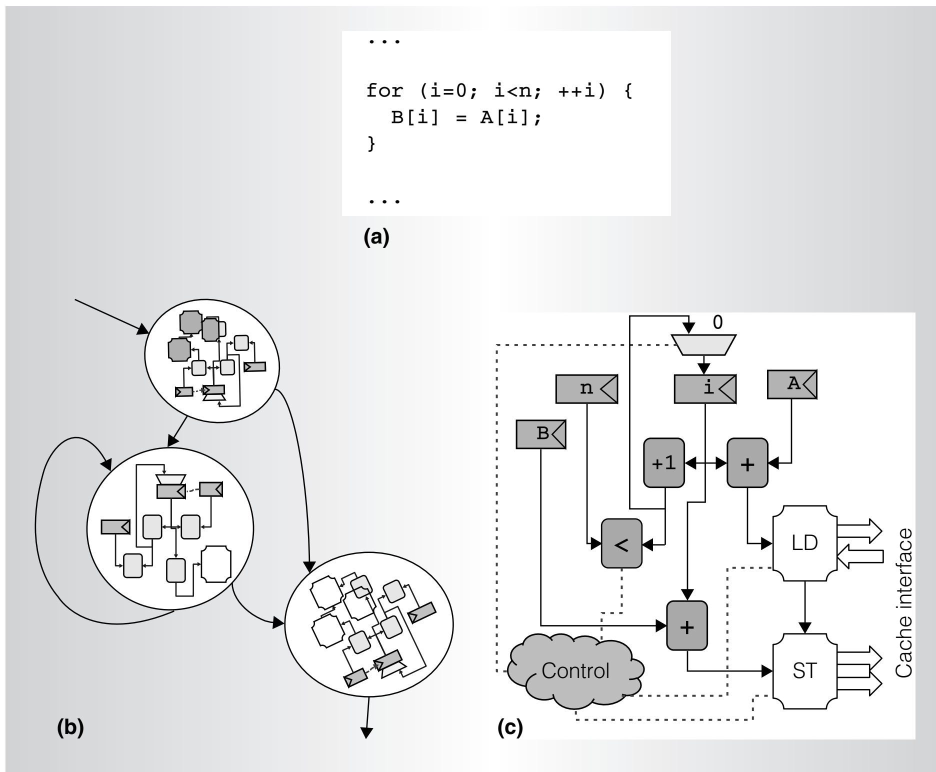

Figure 2. An example showing the translation from C code (a) to the compiler's internal representation (b), and finally to hardware for each basic block (c). The hardware data path and state machine correspond closely to the C code's data flow and control flow graphs.

the memory hierarchy in order to read from the array A and write to the array B.

Patching support

To remain useful as new versions of the Android platform emerge, GreenDroid's c-cores must adapt. To support this, c-cores include targeted reconfigurability that lets them maintain perfect fidelity to source code, even as the source code changes. The adaptation mechanisms include redefining compile-time constants in hardware and a general exception mechanism that lets c-cores transfer control back and forth to the general-purpose core during any control flow transition.

Adding this reconfigurability increases the energy and area needs for c-cores, but significantly improves the span of years over which c-cores can provide energy savings. For the open source codes we used in our experiments, patchable c-cores remained useful for more than a decade of updates and bug fixes, far greater than the typical mobile phone's lifespan.

Synthesizing c-cores

A GreenDroid processor will contain many different c-cores, each targeting a different portion of the Android system. Designing each c-core by hand isn't tractable, especially because software release cycles can be short. Instead, we've built a C/C++-to-Verilog toolchain that converts arbitrary regions of code into c-core hardware.[1] (See the "Research Related to GreenDroid" sidebar to understand this work's relationship to accelerators and high-level synthesis.)

The toolchain first identifies the key functions and loops in the target workload and extracts them by outlining loops and inlining functions. A compiler parses the resulting C code and generates a static single-assignment-based internal representation of the CFG and data flow graph. The compiler then generates Verilog code for the control unit and data path that closely mimics those representations. The compiler also generates function stubs that applications can

Research Related to GreenDroid

GreenDroid is most closely related to prior work in two key areas: construction of specialized accelerators and development of high-level synthesis (HLS) tools.

Accelerators

Specialized accelerators have been getting increasingly more attention lately because they let designers trade customized silicon area for performance and, often, energy efficiency. At the heart of most accelerators' performance is the fact that designers have figured out how to attain parallel execution of the underlying algorithm, realized efficiently in hardware.

One extreme example of such an accelerator is Anton, which attains 100 times or more improvement in molecular-dynamics simulation versus general-purpose supercomputer machines.[1] At a smaller level, we see accelerators throughout the computing space, whether for baseband processing, 3D graphics, or video decoding or encoding.

The challenge in creating accelerators is in reorganizing the algorithm to achieve parallel execution. Being able to do this effectively depends on the availability of exploitable parallelism in the algorithm and the ability to expose this parallelism in the form of an accelerator circuit without errors or excessive effort, complexity, or cost. In particular, creating accelerators for irregular code that's difficult to analyze or lacks parallelism is often challenging, if not impossible.

The conservation cores that comprise the GreenDroid system have different, but related, underlying goals compared to accelerators. Fundamentally, conservation cores (c-cores) focus on reducing the energy of executing code, even if they only modestly improve the resulting execution time. As a result, c-cores don't rely on parallelization technology for a successful outcome; they can target any code in which sufficient execution time is spent. Because of this, a far greater percentage of code can be turned into c-cores than can be transformed into accelerators. We expect that in a commercialized version of GreenDroid, code that can be accelerated (such as video encoding or baseband processing) would be expressed as accelerators and generated using either handcrafted accelerators or high-level synthesis tools. The remaining unacceleratable, irregular code (millions and millions of lines of it) creates an Amdahl's-law-like limit on the system's maximum energy efficiency. Conservation cores provide an automatic way to reduce the energy of this remaining class of difficult code. This distinction drives some of the key differences between GreenDroid's c-cores generation and conventional high-level synthesis technology, which focuses on generating accelerators from parallelizable code.

High-level synthesis tools

HLS tools seek to raise the abstraction level required to create accelerators that improve the execution performance of critical algorithms used in systems on chips (SoCs). HLS research has a long and rich history, which has culminated in the availability of several commercial tools, including AutoESL's AutoPilot, Cadence's C-to-Silicon Compiler, Mentor Graphics' Catapult C, and Synopsys' Synphony. (Coussy and Morawiec survey recent advances in this area.[2])

Because these tools seek to infer parallel execution from serial code, they have many of the same limitations that parallelizing compilers suffer from—namely, the difficulties of analyzing pointers in free-form code, extracting memory parallelism, and extracting and formulating efficient parallel schedules for the operations in critical loops. These are difficult tasks that are generally NP-complete or worse in complexity. However, without successful parallelization, the code is unlikely to run much faster in specialized silicon than it would on a processor core implemented in the SoC.

To address the parallelization challenges, HLS tools have adapted by either limiting the input language (for example, no pointers, no dynamic memory allocation, or no go-tos) or relying on user-transformed code or pragmas for guiding the tool in generating output with good results. Consequently, these tools' expected usage model is that the user will shepherd code through the tools, transforming the code and performing trial-and-error transformations to attain the expected quality of results. Typically, operating-system code and I/O code are considered unsynthesizable, and either the HLS tool ignores this code or the user must comment it out or refactor it to a different part of the system.

Because GreenDroid targets a system with millions of lines of difficult-to-parallelize code, including the Linux kernel, its focus is different: the toolchain must automatically and successfully reduce the energy of large bodies of nonparallelizable code without user intervention. The code base is too large to afford manual intervention, and it's constantly evolving and maintained by a third party (Google and the Linux kernel maintainers). Our toolchain doesn't need user pragmas for effective transformation or require any source code modification to remove unsupported constructs. It supports code that has I/O and system calls—even parts of the operating system can be translated. Additionally, the patching mechanism lets us support changes in the underlying source base, which lets the c-cores evolve as the targeted software changes. The c-cores' memory model ensures compatibility even with code that isn't written in a type-safe fashion or that's expected to communicate in a multiprocessor environment through shared memory, and it allows compatibility with virtual memory. Finally, c-cores and the main processor share the Level 1 (L1) cache, ensuring that execution can migrate quickly between the two, which improves resilience to software changes and maximizes the percentage of execution that can be realized in hardware.

References

1. D.E. Shaw et al., ''Anton: A Special-Purpose Machine for Molecular Dynamics Simulation,'' *Proc. 34th Ann. Int'l Symp. Computer Architecture* (ISCA 07), IEEE CS Press, 2007, doi:10.1145/1250662.1250664.
2. P. Coussy and A. Morawiec, *High-Level Synthesis: from Algorithm to Digital Circuit,* Springer, 2008.

call in place of the original functions to invoke the hardware.

Finally, the compiler generates a description of the c-core that provides the basis for generating patches that will let the c-core run new versions of the same functions. The close mapping between the compiler's intermediate representation and the hardware is essential here: small, patchable changes in the source code correspond to small, patchable changes in the hardware.

Because c-cores focus on reducing energy and power consumption rather than exploiting high levels of parallelism, they can profitably target a much wider range of C constructs. Although conventional accelerators struggle to speed up applications with irregular control and limited memory parallelism, c-cores can significantly reduce the energy and power costs of such codes.

Examining one GreenDroid tile

Figure 3 shows the placed-and-routed floorplan for one of GreenDroid's tiles. In addition to the standard components, the tile contains nine c-cores targeting important functions from the Android code base. Of these nine targeted functions, seven come from *libskia*, a 2D graphics library that provides compositing, rendering, and geometry calculations for most Android applications. The other two come from a JPEG decompression library and a fast Fourier transform (FFT).

Table 1 lists each c-core's description, function name, and vital statistics. On average, each c-core occupies 0.064 mm^2 and runs at 1,568 MHz. Together, all the c-cores occupy 58 percent of the tile's area. The code they execute accounts for a total of 10.6 percent of execution for our benchmarks. The entire GreenDroid chip will contain 16 of these tiles, each with a different set of c-cores.

Figure 4 shows the projected energy savings in GreenDroid and the origin of these savings. The savings come from two sources. First, c-cores don't require instruction fetch, instruction decode, a conventional register file, or any of the associated structures. Removing these reduces energy consumption by 56 percent. The second source of savings (35 percent of energy) comes from the specialization of the c-cores' data path. The result is that average per-instruction energy drops from 91 pJ per instruction to just 8 pJ per instruction.

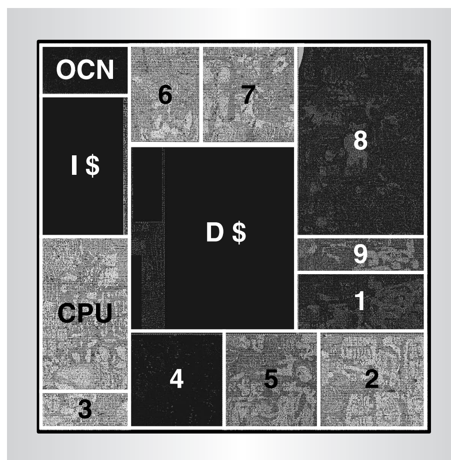

Figure 3. One tile of the GreenDroid processor containing the standard tile components (CPU, I-cache, D-cache, and OCN) in addition to nine c-cores (numbered 1 through 9) that target Android's 2D graphics library, JPEG decompression, and fast Fourier transform (FFT) functions.

Over the next five to 10 years, the breakdown of conventional silicon scaling and the resulting utilization wall will exponentially increase the amount of dark silicon in both desktop and mobile processors. The GreenDroid prototype demonstrates that c-cores offer a new technique to convert dark silicon into energy savings and increased parallel execution under strict power budgets. We estimate that the prototype will reduce processor energy consumption by 91 percent for the code that c-cores target, and result in an overall savings of 7.4×.

The GreenDroid processor design effort is steadily marching toward completion: Our toolchain automatically generates placed-and-routed c-core tiles, given the source code and information about execution properties. Our cycle- and energy-accurate simulation tools have confirmed the energy savings provided by c-cores. We're currently working on more detailed full-system Android emulation to improve our workload modeling so that we can finalize the selection of c-cores that will populate GreenDroid's dark silicon. In parallel with this effort, we're working on timing closure and physical design. MICRO

Table 1. C-cores in a sample GreenDroid tile. The tile in Figure 3 contains c-cores for these nine functions, many of which are from libskia, Android's 2D graphics library. The c-cores range from 0.024 mm^2 to 0.168 mm^2 and cover 10.6 percent of execution. Other tiles will have c-cores for other Android functions.

No.	Description	Android function	Dynamic execution coverage (%)	No. of static instructions	Size (mm^2)
1	Dithering function	S32A_D565_Opaque_Dither	2.78	80	0.052
2	Downsampling	S32_opaque_D32_filter_DXDY	2.20	86	0.070
3	Bit-block image transfer subroutine	S32A_Opaque_BlitRow32	1.15	96	0.024
4	Render with overlay	Sprite_D16_S4444_Opaque::blitRect	1.11	96	0.059
5	Saturating downsampling	Clamp_S16_D16_filter_DX_shaderproc	0.80	97	0.063
6	Fast Fourier transform (FFT)	fft_rx4_long	0.76	138	0.066
7	Image conversion algorithm	ycc_rgba_8888_convert	0.61	92	0.046
8	Lempel-Ziv decompression routine for GIF files	DGifDecompressLine	0.59	334	0.168
9	Image format conversion for dithering	Sample_Index_D565_D	0.57	67	0.032

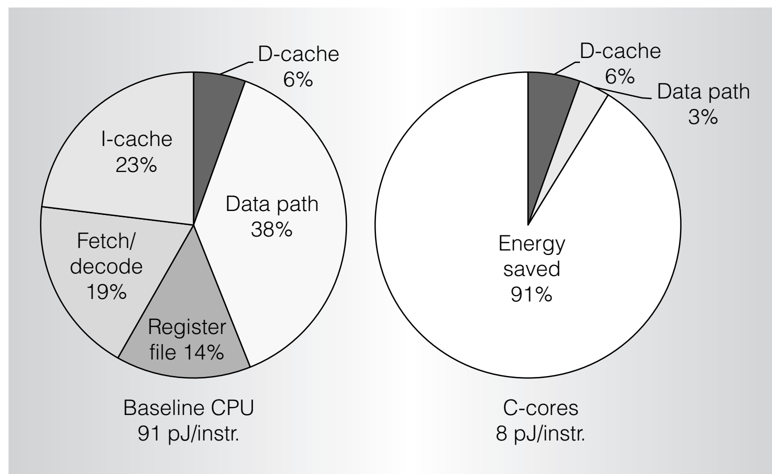

Figure 4. Energy savings in c-cores. Removing hardware structures responsible for fetching and decoding instructions as well as generalized data path components reduces per-instruction energy by 91 percent.

Acknowledgments

This research was funded by the US National Science Foundation under Career Awards 06483880 and 0846152 and under Computing and Communication Foundations Award 0811794.

References

1. G. Venkatesh et al., ''Conservation Cores: Reducing the Energy of Mature Computations,'' *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 2010, pp. 205-218.
2. R. Dennard et al., ''Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions,'' *IEEE J. Solid-State Circuits,* vol. 9, no. 5, 1974, pp. 256-268.
3. J. Sampson et al., ''Efficient Complex Operators for Irregular Codes,'' *Proc. 17th IEEE Int'l Symp. High Performance Computer Architecture,* IEEE Press, 2011, pp. 491-502.
4. M. Taylor et al., ''The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs,'' *IEEE Micro,* vol. 22, no. 2, 2002, pp. 25-35.

Nathan Goulding-Hotta is a PhD student in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include building low-power chips incorporating heterogeneous architectures. He has a BS in electrical engineering and a BS in computer science, both from the New Mexico Institute of Mining and Technology.

Jack Sampson is a postdoctoral scholar in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include heterogeneous architectures, automatically generated

coprocessors, and scalable memory systems. Sampson has a PhD in computer engineering from the University of California, San Diego.

Ganesh Venkatesh is a PhD candidate in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include core architecture, program analysis for designing specialized coprocessors, and algorithmic optimizations to address system bottlenecks. Venkatesh has an MS in computer science from the University of California, San Diego.

Saturnino Garcia is a PhD candidate in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include program analysis, multicore architectures, and software engineering for parallel systems. Garcia has an MS in computer science from the University of California, San Diego.

Joe Auricchio is a graduate student in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include computer architecture, embedded systems, networking, security, and programmable logic. Auricchio has a BS in computer science from the University of California, San Diego.

Po-Chao Huang is a graduate student in the Department of Electrical and Computer Engineering at the University of California, San Diego. His research interests include computer architecture and VLSI digital-signal processing. Huang has a BS in electronics engineering from National Chiao Tung University, Taiwan.

Manish Arora is a PhD student in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include low-power computer architecture for irregular computations, heterogeneous architecture, and parallel systems. Arora has an MS in computer engineering from the University of Texas at Austin.

Siddhartha Nath is a PhD student in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include low-power reconfigurable and heterogeneous architectures. Nath has a BE in electrical engineering from the Birla Institute of Technology and Science, Pilani, India.

Vikram Bhatt is a graduate student in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include physical design automation and low-power design techniques. Bhatt has a BE in electronics and communications engineering from Sri Jayachamarajendra College of Engineering, India.

Jonathan Babb is a postdoctoral researcher at the Massachusetts Institute of Technology. His research interests include extending the life of silicon technology and creating the next generation of carbon-based computing platforms in the emerging fields of synthetic biology and bio-CAD. Babb has a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology.

Steven Swanson is an assistant professor in the Department of Computer Science and Engineering at the University of California, San Diego, and he jointly leads the GreenDroid project. His research interests include specialized architectures for low-power computing and system-level applications for nonvolatile, solid-state memories. Swanson has a PhD in computer science and engineering from the University of Washington.

Michael Bedford Taylor is an assistant professor in the Department of Computer Science and Engineering at the University of California, San Diego, and he jointly leads the GreenDroid project. His research interests include tiled multicore processor design, specialized architectures for smartphones, and software engineering tools that simplify parallelization. Taylor has a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology.

Direct questions and comments about this article to Nathan Goulding-Hotta, Computer Science and Eng. Dept., Univ. of California, San Diego, 9500 Gilman Dr. (MC 0404), La Jolla, CA 92093; ngouldin@cs.ucsd.edu.