GreenDroid: Exploring the Next Evolution in Smartphone Application Processors

Steven Swanson and Michael Bedford Taylor, University of California

ABSTRACT

Mobile application processors are soon to replace desktop processors as the focus of innovation in microprocessor technology. Already, these processors have largely caught up to their more power hungry cousins, supporting out-of-order execution and multicore processing. In the near future, the exponentially worsening problem of dark silicon is going to be the primary force that dictates the evolution of these designs. In recent work, we have argued that the natural evolution of mobile application processors is to use this dark silicon to create hundreds of automatically generated energy-saving cores, called conservation cores, which can reduce energy consumption by an order of magnitude. This article describes GreenDroid, a research prototype that demonstrates the use of such cores to save energy broadly across the hotspots in the Android mobile phone software stack.

INTRODUCTION

Mobile devices have recently emerged as the most exciting and fast-changing segment of computing platforms. A typical high-end smartphone or tablet contains a panoply of processors, including a *mobile application processor* for running the Android or iPhone software environments and user applications and games, a *graphics processor* for rendering on the user's screen, and a *cellular baseband processor* for communicating with the cellular networks. In addition to these flexible processors, there are more specialized circuits that implement Wi-Fi, Bluetooth, and GPS connectivity as well as accelerator circuits for playing and recording video and sound.

As a larger percentage of cellular network traffic becomes data rather than voice, the capabilities of the mobile application processor that generates this data have become exponentially more important. In recent years, we have seen a corresponding exponential improvement in the capabilities of mobile application processors, so these processors are now approaching similar levels of sophistication to those in desktop machines. In fact, this process parallels similar progress that happened when desktop processors mirrored the development of earlier mainframe computers. Figure 1 shows the deployment of architectural features in mainframes, desktop machines, and mobile application processors. As of 2010, mobile application processors have already integrated the most significant innovations of processor architectures of the last 50 years, integrating multiple out-of-order, superscalar, pipelined cores in a single die.

As Moore's Law and complementary metal oxide semiconductor (CMOS) scaling provide improving energy efficiencies and transistor counts, cheaper processors eventually are able to incorporate the features from their older relatives; first, pipelined execution, then superscalar execution, then out-of-order execution, and finally, multicore. Today, because sales quantities are higher, processor features tend to move from desktop processor designs to mainframe designs rather than in the opposite direction. As mobile application processor sales surpass those of desktops, it is likely that smartphone processors will become the new nexus of advancement in processor design.

THE UTILIZATION WALL

Our research at the University of California (UC) San Diego focuses on understanding the technological forces that will shape the development of these future processors and proposing architectural approaches that match these forces. To reduce our proposals to practice, we are designing and implementing a novel mobile application processor called *GreenDroid* [1]. GreenDroid will serve as a prototype for mobile application processors in the next five to ten years. In particular, GreenDroid attacks one of the most important realities of Moore's Law as it continues from today into the future, which we refer to as the *utilization wall* [2].

The utilization wall dictates that due to poor CMOS scaling, improvements in processor performance are determined not by improvements in transistor frequency or transistor count, but rather by the degree to which each process shrink reduces the switching energy of the underlying transistors. Because transistor counts are growing much faster than the underlying energy efficiency is improving, a direct consequence of this is the phenomenon of *dark silicon* — that is, large swaths of a chip's silicon area that must remain mostly passive in order to stay within the chip's power budget. As we show later in this article, only 1 percent or so of a modest sized 32 nm mobile chip can switch at full frequency within a 3 W power budget. The dark silicon problem is directly responsible for the desktop processor industry's decision to stop scaling clock frequencies and instead build multicore processors. It will play an equally pivotal role in shaping the future of mobile processors as well.

With each process generation, dark silicon is a resource that gets exponentially cheaper, while the power budget becomes exponentially more valuable in comparison.

Our Approach

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 10–1000 times better energy efficiency over general-purpose processors. Our approach is to fill the dark silicon area of a chip with specialized cores in order to save energy on common applications. These cores are automatically generated from the codebase the processor is intended to run. In our case, this codebase is the Android mobile phone software stack, but our approach could also be applied to the iPhone OS as well. We believe that incorporating many automatically generated, specialized cores for the express purpose of saving energy is the next evolution in application processors after multicore.

GreenDroid 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 levels of performance. It does this through the use of 100 or so automatically generated, highly specialized, energy-reducing cores, called *conservation cores*, or c-cores [2, 3]. Our work is novel relative to earlier work on accelerators and high-level synthesis [4, 5] because it adapts these techniques to work in the context of large systems (like Android) for which parallelism is limited, and shows that they are a key tool in attacking the utilization wall CMOS systems face today. In particular, of note is our use of techniques that allow the generation of c-cores to be completely automatic, our introduction of patching mechanisms and mechanisms for hiding the existence of the c-cores, and our results on both attainable coverage and potential energy savings.

This article continues as follows. First, we explore the factors that lead to the utilization wall. We continue by examining how the architecture of c-cores allows them to take advantage of the utilization wall. Then we examine trends in mobile application processors, and show how the Android operating system lends itself to the use of c-cores. Finally, we conclude the article by examining software depipeling, a key microarchitectural technique that helps save power in c-cores.

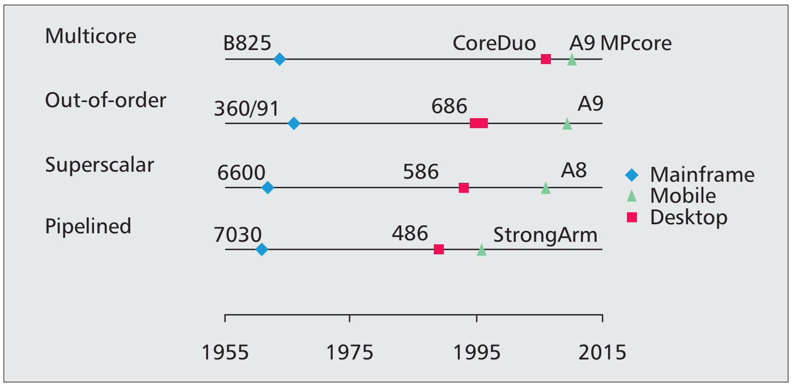


Figure 1. *Evolution of features in mainframe, desktop, and mobile application processors The evolution of features of mobile application processors (e.g., ARM StrongARM, Cortex A8, A9, and A9 MPcore) has mirrored that of desktop processors (e.g., the Intel 486, 586, 686, and Core Duo), and desktop processors in turn have mirrored mainframe processors from the 1960s (e.g., the IBM 7030, CDC 6600, IBM 360/91, and Burroughs 825). As of the current date, mobile application processors have largely caught up, and are soon to enter undiscovered territory.*

Understanding the Utilization Wall

Historically, Moore's Law has been the engine that drives growth in the underlying computing capability of computing devices. Although we continue to have exponential improvements in the number of transistors we can pack into a single chip, this is not enough to maintain historic growth in processor performance. Dennard's 1974 paper [6] detailed a roadmap for scaling CMOS devices, which, since the middle of the last decade, has broken down. This breakdown has fundamentally changed the way that all high-performance digital devices are designed today. One consequence of this breakdown was the industry-wide transition from single-core processors to multicore processors. The consequences are likely to be even more far-reaching going into the future.

In this subsection, we outline a simple argument [2] that shows the difference between historical CMOS scaling and today's CMOS scaling. The overall consequence is that, although transistors continue to get exponentially more numerous and exponentially faster, overall system performance of current architectures is largely unaffected by these factors. Instead, system performance is driven by the degree to which transistors get more energy efficient with each process generation — approximately at the same rate at which the capacitance of those transistors drops as they shrink.

Each transistor transition imparts an energy cost, and the sum of all of these transitions must stay within the active power budget of the system. This power budget is set by either thermal limitations (e.g., the discomfort of placing a 100 W device next to your face) or battery limitations (e.g., a 6 Wh battery that must last for 8 hours of active use can only average 750 mW over that time period.) As we see shortly, in current systems, it is easy to exceed this budget with only a small percentage of the total transistors on a chip.

CMOS scaling theory predicts exponential decreases in the amount of non-dark silicon with each process generation. To adapt, we need to create architectures that can leverage many, many transistors without actually actively switching them all.

Transistor property	Classical	Leakage-limited
Δ Quantity	S^2	S^2
Δ Frequency	S	S
Δ Capacitance	$1/S$	$1/S$
ΔV^2_{dd}	$1/S^2$	1
$\Rightarrow \Delta$ Power $= \Delta QFCV^2$	1	S^2
$\Rightarrow \Delta$ Utilization $=$ 1/Power	1	$1/S^2$

Table 1. *Classical vs. leakage-limited scaling. In contrast to the classical regime proposed by Denard, 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.*

This argument is summarized in Table 1, which takes as input variable a scaling factor S, which describes the ratio between the feature sizes of two processes (e.g., $S = 45/32 = 1.4x$ between 45 nm and 32 nm technology). In "classical" (i.e., pre-2005) scaling proposed by Denard, we are able to scale the threshold voltage and operating voltage together. Currently, we are in a "leakage-limited" regime where we cannot decrease lower threshold and operating voltages without exponentially increasing either transistor delay or leakage.

In both regimes, full-chip transistor counts increase by S^2, the native switching frequency of transistors increases by S, and capacitance decreases by $1/S$. However, the two cases differ in operating voltage (V_{dd}) scaling: with classical scaling, V_{dd} decreases by $1/S$, but with leakage-limited scaling, V_{dd} stays fixed. When transitioning to another process generation, the change in power consumption is the product of these terms and an additional factor of V_{dd}.

Thus, currently, the only factor decreasing power consumption as we move to a new process generation is the reduction of capacitance per transistor, at a rate of $1/S$, while the other factors are increasing it by S^3.

As shown in Table 1, in classical scaling, using all of the chip area for transistors running at maximum frequency would result in constant power between process generations, and we retain the ability to utilize all of the chip resources. Today, doing the same would increase power consumption by S^2. Since power budgets are constrained in real systems, we must instead reduce utilization of chip resources by $1/S^2$ (i.e., 2× with each process generation). Effectively, a greater and greater fraction of the silicon chip will have to be dark silicon.

Experimental Verification

To validate these scaling theory predictions, we performed several experiments targeting current-day fabrication processes. A small datapath — an arithmetic logic unit (ALU) sandwiched between two registers — was replicated across a 40-mm^2 chip in a 90 nm Taiwan Semiconductor Manufacturing Corporation (TSMC) generation. We found that a 3 W power budget would allow only 5 percent of the chip to run at full speed. In a 45 nm TSMC process, this percentage drops to 1.8 percent, a factor of 2.8×. Applying the International Technology Roadmap for Semiconductors (ITRS) for 32 nm suggests utilization would drop to 0.9 percent. These measurements confirm that the trend is upon us, although it has been mitigated slightly by one-off improvements to process technology (e.g., strained silicon).

Real World Observations

The real world also provides direct evidence of the utilization wall. Desktop and laptop processor frequencies have increased very slowly for the better part of a decade, and chip core counts have scaled much more slowly than the increase in transistor count. Increasing fractions of the chips are used for cache or low-activity "uncore" logic like memory controllers and chipsets. Recently, Intel and AMD have advertised a "turbo mode" that runs some cores faster if the others are switched off. We can expect similar trends for the future of mobile processors as well.

Designing New Architectures for the Utilization Wall

These observations show that the utilization wall is a fundamental first order constraint for processor design. CMOS scaling theory predicts exponential decreases in the amount of non-dark silicon with each process generation. To adapt, we need to create architectures that can leverage many, many transistors without actually actively switching them all. In the following section, we describe GreenDroid's design, and show how c-cores have these exact qualities and can employ otherwise unused dark silicon to mitigate the extreme power constraints that the utilization wall imposes.

The GreenDroid Architecture

A GreenDroid processor combines general-purpose processors with application-specific coprocessors that are very energy efficient. These conservation cores, or c-cores [2], execute most of an application's code and will account for well over 90 percent of execution time. GreenDroid is a heterogeneous tiled architecture. Figure 2a illustrates how it uses a grid-based organization to connect multiple tiles. Figure 2b show the floor plan for one of the tiles. It contains an energy-efficient 32-bit 7-stage in-order pipeline that runs at 1.5 GHz in a 45 nm process technology. It includes a single-precision floating point unit (FPU), multiplier, 16-kbyte I-cache, translation lookaside buffer (TLB), and 32-kbyte banked L1 data cache. The architecture also includes a mesh-based on-chip network (OCN). The OCN carries memory traffic and supports fast synchronization primitives, similar to the Raw scalable tiled

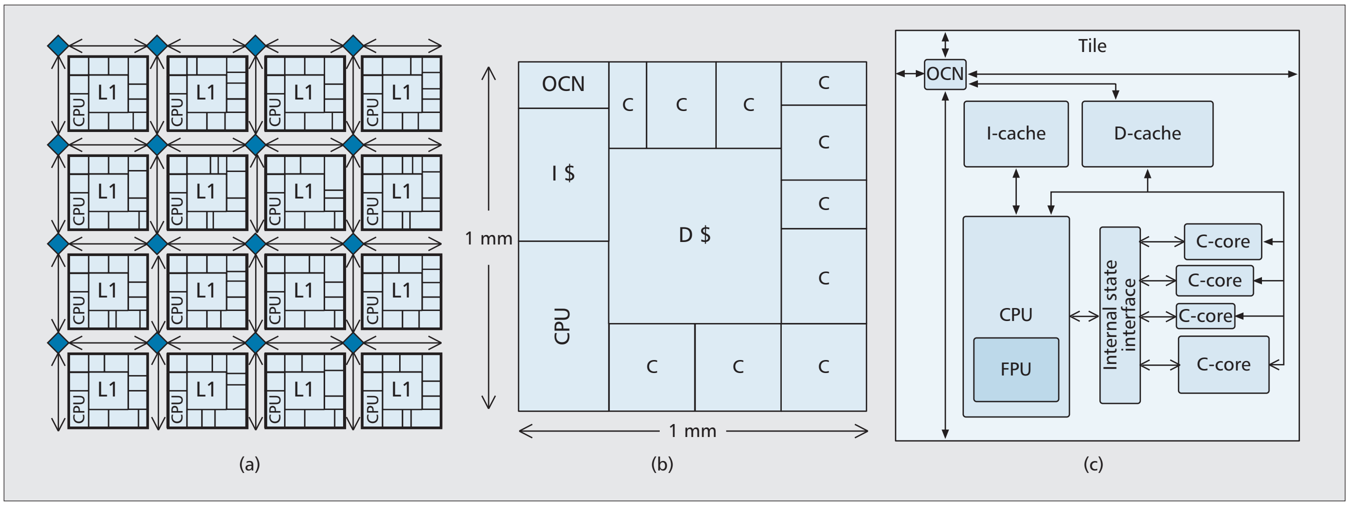


Figure 2. *The GreenDroid architecture. The GreenDroid mobile application processor (a) is made up of 16 non-identical tiles. Each tile (b) holds components common to every tile — the CPU, on-chip network (OCN), and shared L1 data cache — and provides space for multiple c-cores (labeled C) of various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

architecture [7]. The tiles' caches are kept coherent through a simple cache coherence scheme that allows the level 1 (L1) caches of inactive tiles to be collectively used as a level 2 (L2) cache.

Unlike Raw, however, GreenDroid tiles are not uniform. Each of them contains a unique collection of 8–15 c-cores. The c-cores communicate with the general-purpose processor via an L1 data cache and specialized register-mapped interface (Fig. 2c). Together, these two interfaces support argument passing and context switches for the c-cores. The register-mapped interface also supports a specialized form of reconfiguration called *patching* that allows c-cores to adapt to small changes in application code.

C-cores are most useful when they target code that executes frequently. This means we can let the Android code base determine which portions of Android should be converted into GreenDroid c-cores. We use a system-wide profiling mechanism to gather an execution time breakdown for common Android applications (e.g., web browsing, media players, and games). The c-core tool chain transforms the most frequently executed code into c-core hardware. Subject to area constraints, the tools then assign the c-cores to tiles to minimize data movement and computation migration costs. To support this last step, the profiler collects information about both control flow and data movement between code regions. Related c-cores end up on the same or nearby tiles, and cache blocks can migrate automatically between them via cache coherence. In some cases, c-cores are replicated to avoid hotspotting. If a given c-core is oversubscribed, the c-cores have the option of running the software equivalent version of the code.

Internally, c-cores mimic the structure of the code on which they are based. There is one arithmetic unit for each instruction in the target code, and the wires between them mirror data dependence arcs in the source program. The close correspondence between the structure of the c-core and the source code is useful for three reasons. First, it makes it possible to automatically generate c-cores from arbitrary code without complex compiler analyses. Second, it allows us to integrate a limited degree of reconfigurability that allows one c-core to execute multiple versions of the same source code (e.g., as might appear during a system upgrade): small changes in the source code will correspond naturally to small changes in hardware. Full details of c-cores' patching facilities are available in [2].

The final advantage of the correspondence is that it allows c-cores to function as drop-in replacements for the code they target, even in complex multithreaded applications. The correspondence guarantees that the c-core will execute exactly the same loads and stores as the general-purpose processor while executing the target code. Furthermore, the c-core will execute them in the same order. This means that the existence of the c-cores is transparent to the programmer. A specialized compiler recognizes regions of code that align well with the c-cores, and generates "stub" functions and a patching configuration that allows the c-cores to replicate a function's behavior.

On average, c-cores reduce energy consumption by 94 percent compared to the general-purpose processor running the code that the c-cores target. Overall system energy savings are smaller because of three effects:

- Cold code still runs on the (relatively inefficient) CPU.
- C-core optimizations do not reduce the energy costs of the L1 cache.
- Significant energy still goes toward leakage and the clock.

The first effect we reduce by attaining high execution coverage by the c-cores, targeting regions that cover as little as 1 percent of total execution coverage. The last two we have attacked through novel memory system optimizations, power gating, and clock power reduction techniques.

The specialized nature of c-cores allows them to overcome the utilization wall. Collectively, the tiles in the GreenDroid system exceed the power budget of the system. As a result, most of the

We have profiled a diverse set of Android applications including the web browser, Mail, Maps, Video Player, Pandora, and many other applications. We found that applications spend 95 percent of their time executing just 43,000 static instructions.

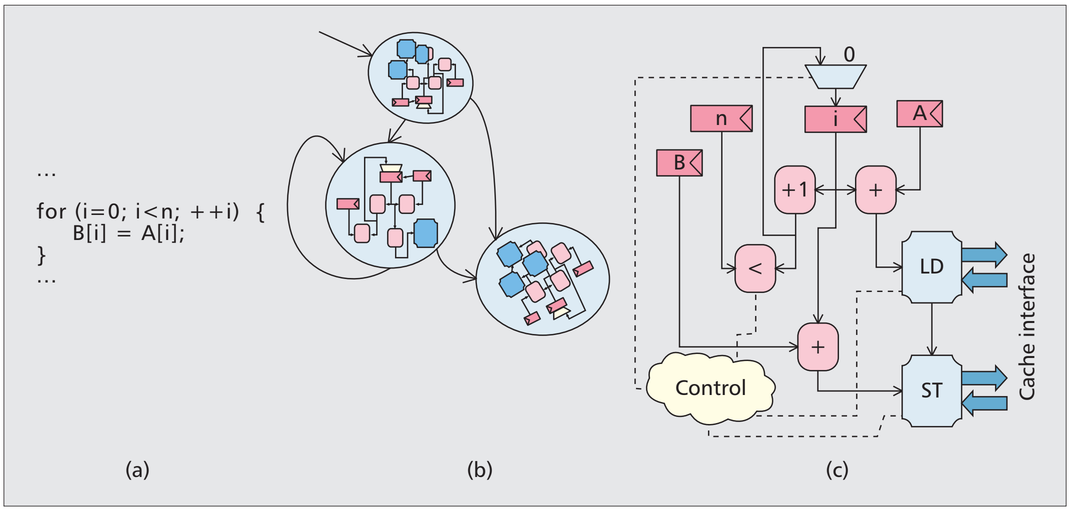


Figure 3. *Conservation core example: an example showing the translation from C code (a), to the compiler's internal representation (b), and finally to hardware for each basic block (c). The hardware datapath and state machine correspond very closely to the data and control flow graphs of the C code.*

time, most of the c-cores and tiles are idle (and power gated), so they consume very little power. Execution moves from tile to tile, activating only the c-cores the application needs.

ANDROID: GREENDROID'S TARGET WORKLOAD

Android is an excellent target for a c-core-enabled, GreenDroid-style architecture. Android comprises three main components: a version of the Linux kernel, a collection of native libraries (written in C and C++), and the Dalvik virtual machine (VM). Android's libraries include functions that target frequently executed, computationally intensive (or "hot") portions of many applications (e.g., 2D compositing, media decoding, garbage collection). The remaining "cold" code runs on the Dalvik VM, and, as a result, the Dalvik VM is also "hot." Therefore, a small number of c-cores that target the Android libraries and Dalvik VM should be able to achieve very high coverage for Android applications.

We have profiled a diverse set of Android applications including the web browser, Mail, Maps, Video Player, Pandora, and many other applications. We found that this workload spends 95 percent of its time executing just 43,000 static instructions. Our experience building c-cores suggests that just 7 mm^2 of conservation cores in a 45 nm process could replace these key instructions. Even more encouraging, approximately 72 percent of this 95 percent was library or Dalvik code that multiple applications used.

Android's usage model also reduces the need for the patching support c-cores provide. Since cell phones have a very short replacement cycle (typically two to three years), it is less important that a c-core be able to adapt to new software versions as they emerge. Furthermore, handset manufacturers can be slow to push out new versions. In contrast, desktop machines have an expected lifetime of between five and ten years, and the updates are more frequent.

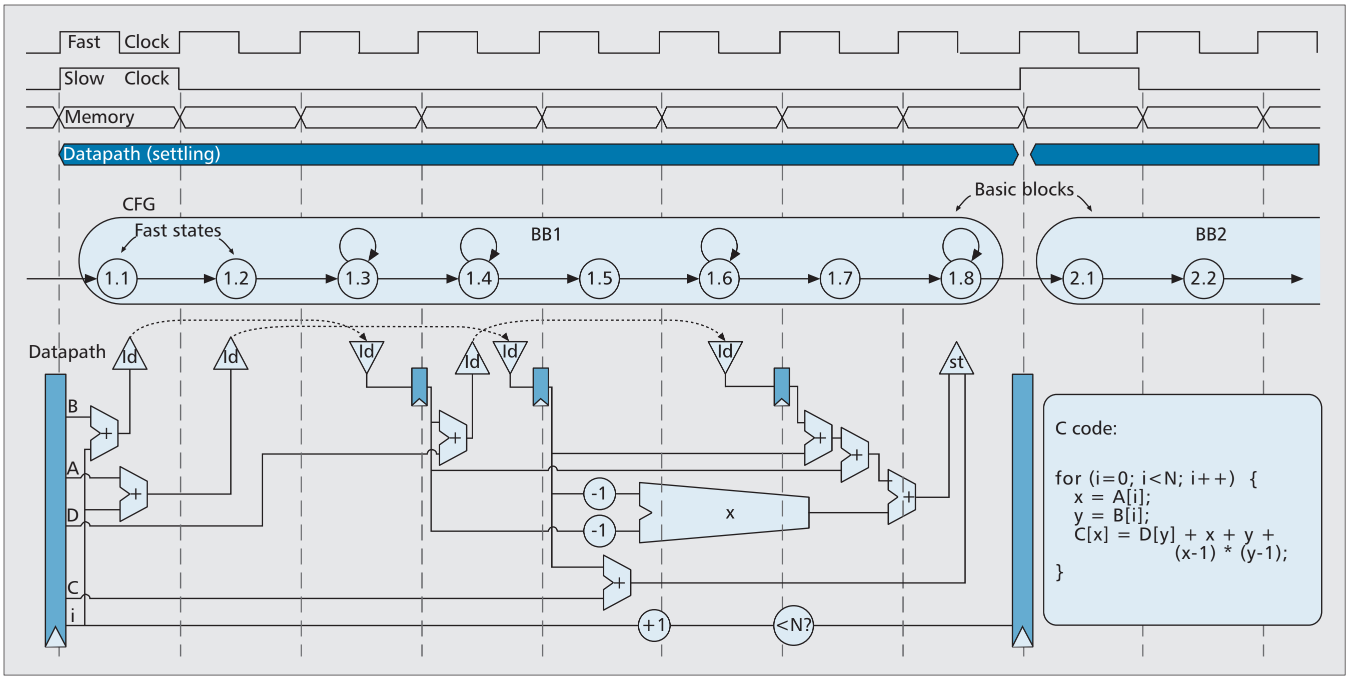
SYNTHESIZING C-CORES FOR GREENDROID

A single GreenDroid processor will contain tens or hundreds of different c-cores that each implement a different key function in Android. Designing this many c-cores by hand is not practical. Fortunately, the c-core toolchain makes this unnecessary since it can convert arbitrary C functions into c-core hardware.

The c-core toolchain differs from conventional C-to-Verilog systems, since it focuses on saving energy rather than attaining large improvements in performance. This means it can omit the complex analyses necessary to extract the parallelism accelerators must exploit. Consequently, our toolchain can target a much wider range of C constructs, and can build energy-saving c-cores for functions that are poor targets for acceleration.

The toolchain's profiling pass identifies "hot" functions and loops in the target workload, and isolates them by inlining functions and outlining loops. Our C-to-Verilog compiler divides the c-core for each hot region into control and data paths. The data path mirrors the single static assignment program representation the compiler uses internally and groups instructions together by basic block. The control path tracks execution through the function with a state machine that closely matches the function's control flow graph (CFG). The compiler also generates function stubs to replace the original functions by invoking the hardware.

Figure 3a illustrates this process on a simple loop. Figure 3b shows the corresponding data and CFG, and Fig. 3c shows the resulting hardware. The hardware contains a mux for *i*, since it is defined in two basic blocks. The c-core's state machine is very similar to CFG, but the c-core adds self-loops for multicycle operations (e.g., memory accesses). The data path includes simple functional units to implement each instruction in Fig. 3b, including a load and store unit to access arrays A and B.



C code:

```
for (i=0; i<N; i++)  {
  x = A[i];
  y = B[i];
  C[x] = D[y] + x + y +
             (x-1) * (y-1);
}
```

Figure 4. *Example SDP datapath and timing diagram In SDP circuits, non-memory datapath operators chain combinationally within a basic block and are attached to the slow clock. Memory operators and associated receive registers align to the fast clock.*

As new versions of the Android platform emerge, GreenDroid's c-cores may need to change to remain useful. C-cores support this by providing targeted reconfigurability that lets them maintain perfect fidelity to changing source code. C-cores provide built-in support for changes to compile-time constants as well as a general mechanism for transferring control back to the general-purpose core to execute individual basic blocks. Whether to include patching support for a particular function depends on how mature the function is. For functions that change very little, leaving out patching support may make sense since it can save area and increase energy efficiency.

Conservation Core Microarchitecture

Historically, logic design techniques in processor architecture have emphasized pipelining to equalize critical path lengths and increase performance by increasing the clock rate. The utilization wall means that this can be a suboptimal approach. Adding registers increases switched capacitance, increasing per-op energy and delay. Furthermore, these registers increase the capacitance of the clock tree, a problem compounded by the increased switching frequency rising clock rates require.

For computations that have pipeline parallelism, pipelining can increase performance by overlapping the execution of multiple iterations. This improves energy-delay product despite the increase in energy. However, most irregular code does not have pipeline parallelism, and as a result, pipelining is a net loss in terms of both energy and delay. In fact, the ideal case is to have extremely long combinational paths fed by a slow clock. This minimizes both the energy costs and the performance impact of pipeline registers.

The use of long combinational paths carries two challenges. First, different basic blocks have different combinational critical path lengths, which would require the distribution of many different clocks. Second, there is no way to multiplex centralized or area-intensive resources such as memory and FPUs into these combinational paths.

Selective Depipelining

In order to generate efficient hardware, we developed a technique that integrates the best aspects of pipelined and unpipelined approaches. The technique is called selective depipelining, or SDP [5], and it integrates the memory and data path suboperations from a basic block into a composite fat operation. SDP replicates low-cost data path operators into long combinational data path chains, driven by a slow clock. At the same time, memory accesses present in these fat operations are scheduled onto an L1 cache that operates at a much higher frequency than the data path. In effect, SDP replicates the memory interface in time and the data path operators in space, saving power and exploiting ILP. To generate these circuits, the c-core flow makes heavy use of multicycle timing assertions between the data path chains and the multiplexers that connect to the L1 caches. Except for registers to capture the outputs of the multiplexed memory, sequential elements are only stored at the end of the execution of a fat operator.

Using our automatic SDP algorithm, we have observed efficient fat operations encompassing up to 103 operators and including 17 memory requests. The data path runs at a tiny fraction of the memory clock speed, reducing overall clock

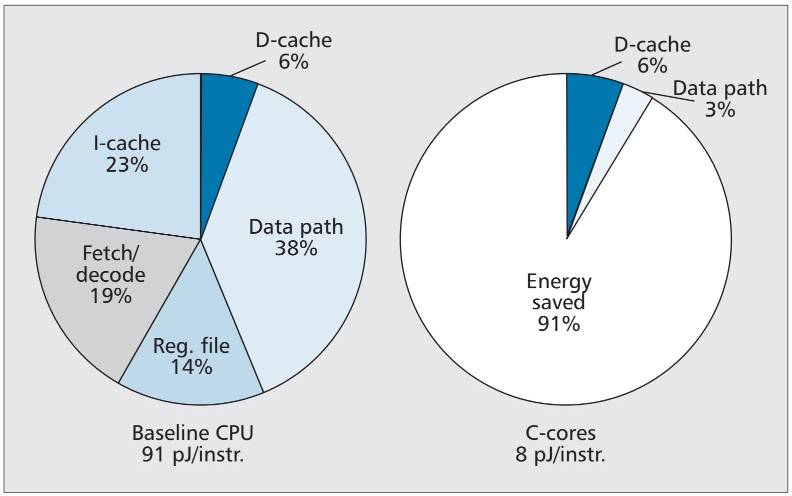


Figure 5. *Energy savings in c-cores Eliminating instruction fetch and decoding as well as overheads such as register files, bypass paths, and ALU muxes drops per-instruction energy by 91 percent.*

energy and improving performance. With SDP, c-cores execute faster and consume less energy than either general-purpose processors or c-core designs without SDP. SDP improves performance by enabling memory pipelining and exploiting ILP in the data path while it minimizes both static and dynamic power because fewer pipeline registers are required. In addition, synthesis can employ smaller, slower gates, since many paths easily fit within the constraints of the slow clock cycle time.

Slow States and Fast States

With SDP, one basic block from the program's control flow graph (CFG) executes for each pulse of the slow clock. The execution of a basic block begins with a slow clock pulse from the control unit. The pulse latches live-out data values from the previous basic block and applies them as live-ins to the current block. The next pulse of the slow clock, which will trigger the execution of the next basic block, will not occur until the entire basic block is complete.

For each basic block, there is a single control state, which contains multiple substates called *fast states*. The number of fast states in a given control state is based on the number of memory operations in the block and the latency of the datapath operators. This means that different basic blocks operate at different slow clocks, however they are always a multiple of the fast clock. During a basic block's execution, the control unit passes through fast states in order. Some fast states correspond to memory operations, for which the c-core sends out a split-phase load or store request to the memory hierarchy, which returns a few cycles later and is received by a register connected to the fast clock. The register holds the value steady for the rest of the slow clock cycle. In the meantime, memory accesses and other operations can be performed. To allow for variable latency due to caches, the c-core will stall until the memory operation completes. While most operations are scheduled at some combinational delay relative to the slow clock edge, memory accesses and other multiplexed operations such as FPU operations are scheduled relative to the fast clock. Conservation cores employ in-order completion of memory requests because this minimize complexity and power and simplifies multithreaded memory semantics.

SDP Example

Figure 4 shows an example of an SDP implementation of a basic block. Source code for the block is on the right. The timing diagram, CFG, and data path show the execution of the code as time flows from left to right. The data path contains arithmetic operators and load/store units that correspond to each of the original program's operations. As shown in the timing diagram, the data path logic can take multiple fast cycles to settle while the data path makes multiple memory requests.

Figure 4 shows how SDP improves energy and performance. In conventional high-frequency multicycle designs, all live values in the basic block would be registered at fast clock boundaries (i.e., at each vertical dotted line). Instead, SDP eliminates registers altogether, reducing latency, area, and energy. It also eliminates many clock-tree leaf nodes, reducing clock tree area, capacitance, and leakage. Removing registers also enables more flexible gate-level optimizations.

Sources of Energy Savings for Conservation Cores

GreenDroid's projected energy savings are shown in Fig. 5. Two sources are primarily responsible for savings: First, c-cores do not require instruction fetch, instruction decode, a conventional register file, or any of the associated structures. Elimination of these elements reduces energy consumption by 56 percent. Specialization of the c-core's datapath is responsible for the remaining 35 percent of energy. Average per-instruction energy drops from 91 pJ/inst to just 8 pJ/inst.

Conclusion

The utilization wall will exponentially worsen the problem of dark silicon in both desktop and mobile processors. The GreenDroid prototype is a demonstration vehicle that shows the widespread application of c-cores to a large code base: Android. Conservation cores will enable the conversion of dark silicon into energy savings and allow increased parallel execution under strict power budgets. The prototype uses c-cores to reduce energy consumption for key regions of the Android system, even if those regions have irregular control and unpredictable dependent memory accesses. Conservation cores make use of the selective depipelining technique to reduce the overhead of executing highly irregular code by minimizing registers and clock transitions. 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×.

Acknowledgments

We thank all of the members of the GreenDroid and Conservation Core teams, including Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Jonathan Babb, Manish Arora, Siddhartha Nath, Vikram Bhatt, Slavik Bryksin, Po-Chao Huang, Joe Auricchio, David Curran, Scott Ricketts, and Jose Lugo-Martinez. The conservation core research is partially funded by the National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

References

[1] J. Babb *et al.*, "Parallelizing Applications into Silicon," *Proc. 7th Annual IEEE Symp. Field-Programmable Custom Computing Machines*, 1999.

[2] R. Dennard *et al.*, "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions," *IEEE J. Solid-State Circuits*, Oct. 1974.

[3] N. Goulding *et al.*, "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon," *HotChips*, 2010.

[4] V. Kathail *et al.*, "Pico: Automatically Designing Custom Computers," *Computer*, vol. 35, Sept. 2002, pp. 39–47.

[5] J. Sampson *et al.*, "Efficient Complex Operators for Irregular Codes," *Proc. Symp. High Perf. Computer Architecture*, Feb. 2011.

[6] M. Taylor *et al.*, "The Raw Processor: A Scalable 32-bit Fabric for General Purpose and Embedded Computing," *HotChips*, 2001.

[7] G. Venkatesh *et al.*, "Conservation Cores: Reducing the Energy of Mature Computations," *ASPLOS XV: Proc. 15th Int'l. Conf. Architectural Support for Prog. Languages and Op. Sys.*, Mar. 2010.

Biographies

STEVEN SWANSON is an assistant professor at the University of California, San Diego, and jointly leads the GreenDroid project. His areas of interest include specialized architectures for low-power computing and system-level applications for non-volatile solid state memories. He received his Ph.D. from the University of Washington in 2006.

MICHAEL BEDFORD TAYLOR is an assistant professor at the University of California, San Diego, and jointly leads the GreenDroid project. He was lead architect of the 16-core MIT Raw microprocessor, and co-author of Connectix Corp's Virtual PC, version 1.0. He received the Intel Foundation Ph.D. Fellowship in 2003 and the NSF CAREER Award in 2009. He holds an A.B. from Dartmouth College and a Ph.D. from MIT.