
DENVER: NVIDIA'S FIRST 64-BIT ARM PROCESSOR

NVIDIA'S FIRST 64-BIT ARM PROCESSOR, CODE-NAMED DENVER, LEVERAGES A HOST OF NEW TECHNOLOGIES, SUCH AS DYNAMIC CODE OPTIMIZATION, TO ENABLE HIGH-PERFORMANCE MOBILE COMPUTING. IMPLEMENTED IN A 28-NM PROCESS, THE DENVER CPU CAN ATTAIN CLOCK SPEEDS OF UP TO 2.5 GHz. THIS ARTICLE OUTLINES THE DENVER ARCHITECTURE, DESCRIBES ITS TECHNOLOGICAL INNOVATIONS, AND PROVIDES RELEVANT COMPARISONS AGAINST COMPETING MOBILE PROCESSORS.

.....Nvidia has created its first internally designed microprocessor, a 64-bit ARMv8-A CPU built to incorporate into systems on chip (SoCs) and address the performance segment of the mobile market. Code-named Denver, the Tegra K1-64 CPU capitalizes on Nvidia's processor methodologies and innovation to bring PC-class performance to mobile platforms with energy efficiency comparable to high-end mobile ARM and x86 Celeron processors. Denver will extend the reach of the ARM processor family beyond its current scope and capability to enable high-performance computing, content creation, visual computing, and gaming on mobile devices.

Implemented in a Taiwan Semiconductor Manufacturing Company 28-nm high-performance mobile (HPM) process, the Denver CPU can reach clock speeds of up to 2.5 GHz, providing considerable performance headroom. Denver introduces a radically new architecture that leverages and benefits from dynamic code optimization (DCO).

In this article, we present details of the Denver CPU microarchitecture with relevant comparisons to the Tegra K1-32 processor.

We discuss DCO and its inner workings in greater detail, including a real-world code optimization example. We also provide compelling performance data at various power envelopes to enable comparisons against competing mobile processors.

Microarchitecture

The Tegra K1-64 SoC includes a fully compatible ARMv8 dual-core Denver CPU that supports both AArch32 and AArch64 instruction sets. Figure 1 shows a block diagram of Denver, a seven-wide superscalar architecture that delivers out-of-order class performance without the power cost and complexity associated with traditional out-of-order designs. Table 1 highlights microarchitecture information for the Tegra K1-64 SoC powered by Denver cores, compared to the Cortex-A15-based Tegra K1-32 processor.

The Denver CPU supports two modes of operation: the ARM hardware-decoder mode and the optimized microcode mode. In the hardware-decoder mode, the front-end fetches ARM instructions from the level-1 (L1)

Darrell Boggs
Gary Brown
Nathan Tuck
K S Venkatraman
Nvidia

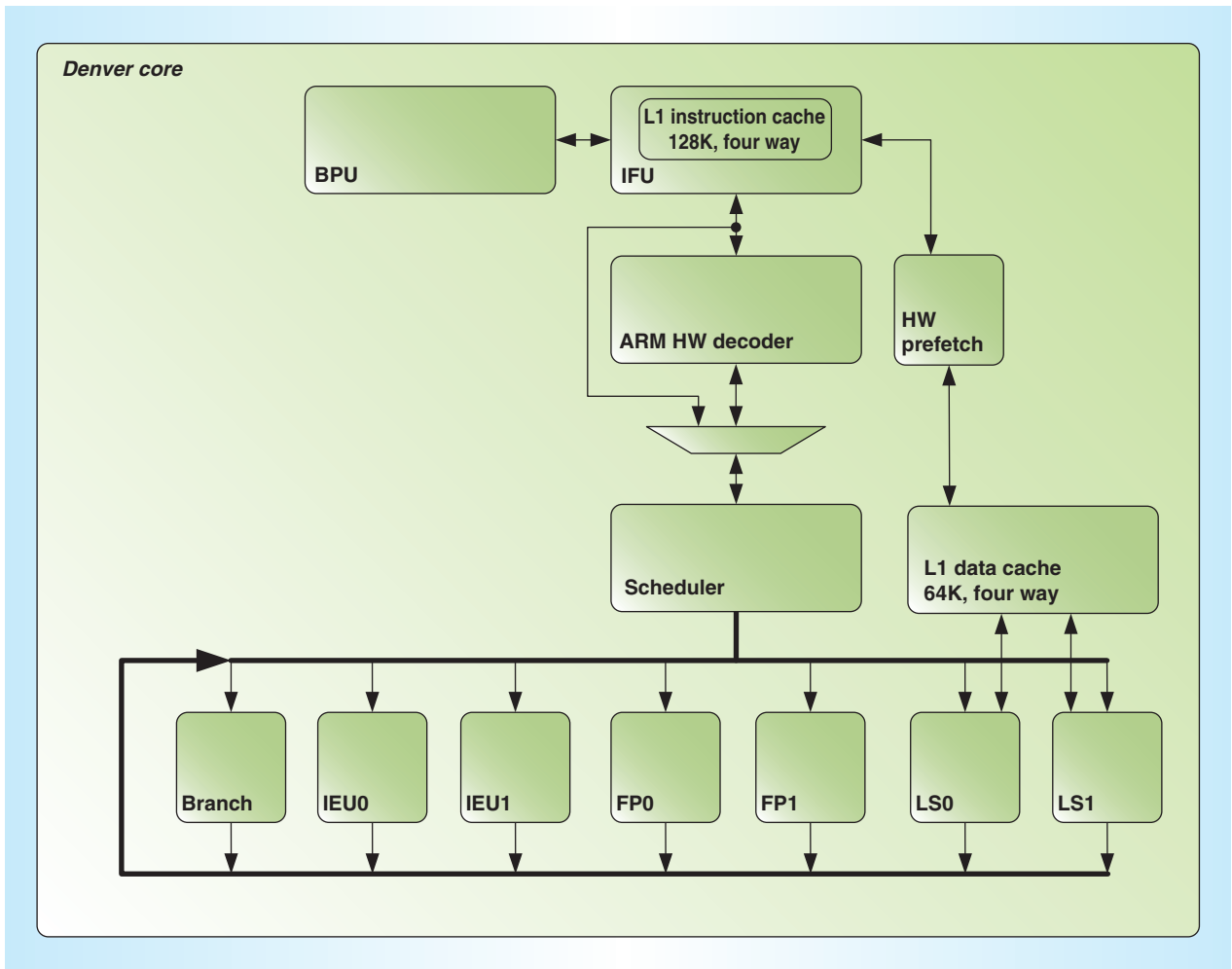


Figure 1. Block diagram of the Denver CPU. The diagram shows the important functional units and the seven-wide superscalar architecture.

instruction cache. Two instructions are decoded per cycle and, if possible, fused to form a bundle of micro-operations. The bundles are dispatched by the scheduler to the execution units. The peak throughput of the Denver CPU in this mode is two ARM instructions per cycle. The hardware-decoder mode sustains an acceptable level of baseline performance and efficiency for code that has not been observed and optimized by DCO.

Figure 1 also depicts a bypass path around the ARM hardware decoder, which is active in the optimized mode. In steady state, the Denver CPU executes predominantly in this mode of operation. Optimized bundles created by DCO are stored in memory, cached in the L1 instruction cache, fetched by the front-end hardware, and fed to the scheduler.

The in-order scheduler doesn't distinguish between the two modes of operation. Once operands are ready, it dispatches bundles to the execution stack. Dependency stalls are more common in the ARM hardware-decoder mode because of the underlying in-order execution hardware. DCO extracts parallelism from the instruction stream, creating densely packed micro-operation bundles and effectively reducing dependency stalls between them. DCO can also eliminate redundant instructions, which can yield a peak throughput of greater than seven ARM instructions per cycle.

The execution stack in Figure 1 comprises seven execution units: two integer, two floating-point (FP)/Neon, two load/store, and one branch execution unit. The two load/

Table 1. Comparison of cores in Tegra K1-32 and Tegra K1-64 systems on chip (SoCs). Information is per core unless stated otherwise.

Parameter	Tegra K1-32	Tegra K1-64
Maximum frequency	2.3 GHz	2.5 GHz
Process	28-nm high-performance mobile	28-nm high-performance mobile
CPU	4x Cortex-A15	2x Denver
Peak instructions per cycle	3	7+
Integer units	2	4
Floating-point datapath	2 × 64 bits	2 × 128 bits
Memory	1 load + 1 store	2 load/2 store/1 load + 1 store
Instruction level-1 cache	32 Kbytes	128 Kbytes
Data level-1 cache	32 Kbytes	64 Kbytes
Level-2 cache	2 Mbytes shared	2 Mbytes shared
Instruction translation look-aside buffer (TLB)	32 entries	128 entries
Data TLB	32 entries	256 entries
L2 TLB	512 entries	2,048 entries

store units can also execute integer operations, enabling DCO to utilize and execute up to four 64-bit integer operations in a bundle when no memory operations must be performed. Each FP/Neon execution unit employs a full 128-bit datapath, with one unit capable of floating-point multiply-accumulate operations. The peak throughput is 12 single-precision FP operations per cycle or six double-precision FP operations per cycle. The Denver CPU achieves three times greater double-precision floating-point horsepower than the Cortex-A15 CPU on a per-core basis.

The Denver CPU's memory subsystem includes a 64-Kbyte, four-way L1 data cache, backed by a 2-Mbyte, 16-way L2 cache that includes the L1 instruction and data caches. Larger L1 cache sizes allow for code and data expansion during DCO's operation to efficiently create optimized microcode sequences. The load-to-load latency is three cycles for the L1 data cache and 18 cycles for the 2-Mbyte L2 cache. A 256 entry, eight-way data translation look-aside buffer (DTLB) is backed by an accelerator cache for intermediate page table entries. The memory controller on the Tegra K1-64 SoC supports a single 64-bit channel of DDR3-1866 and provides a peak bandwidth of 14.9 Gbytes per second.

Reducing the effects of long cache-miss penalties has been a major focus of the micro-architecture, using techniques like prefetching and run-ahead. An aggressive hardware prefetcher implementation detects L2 cache requests and tracks up to 32 streams, each with complex stride patterns.

Run-ahead uses the idle time that a CPU spends waiting on a long latency operation to discover cache and DTLB misses further down the instruction stream and generates prefetch requests for these misses.¹ These prefetch requests warm up the data cache and DTLB well before the actual execution of the instructions that require the data. Run-ahead complements the hardware prefetcher because it's better at prefetching nonstrided streams, and it trains the hardware prefetcher faster than normal execution to yield a combined benefit of 13 percent on SPECint2000 and up to 60 percent on SPECfp2000.

Figure 2a shows the pipeline for the Denver CPU with a 13-cycle mispredict penalty, compared to 15 cycles for the Cortex-A15 core. On a combination of integer, floating-point, and JavaScript benchmarks, the Denver CPU achieves up to 37 percent lower misprediction rates than the Cortex-A15 CPU.

A key feature of the Denver microarchitecture is the skewed aspect of the Denver

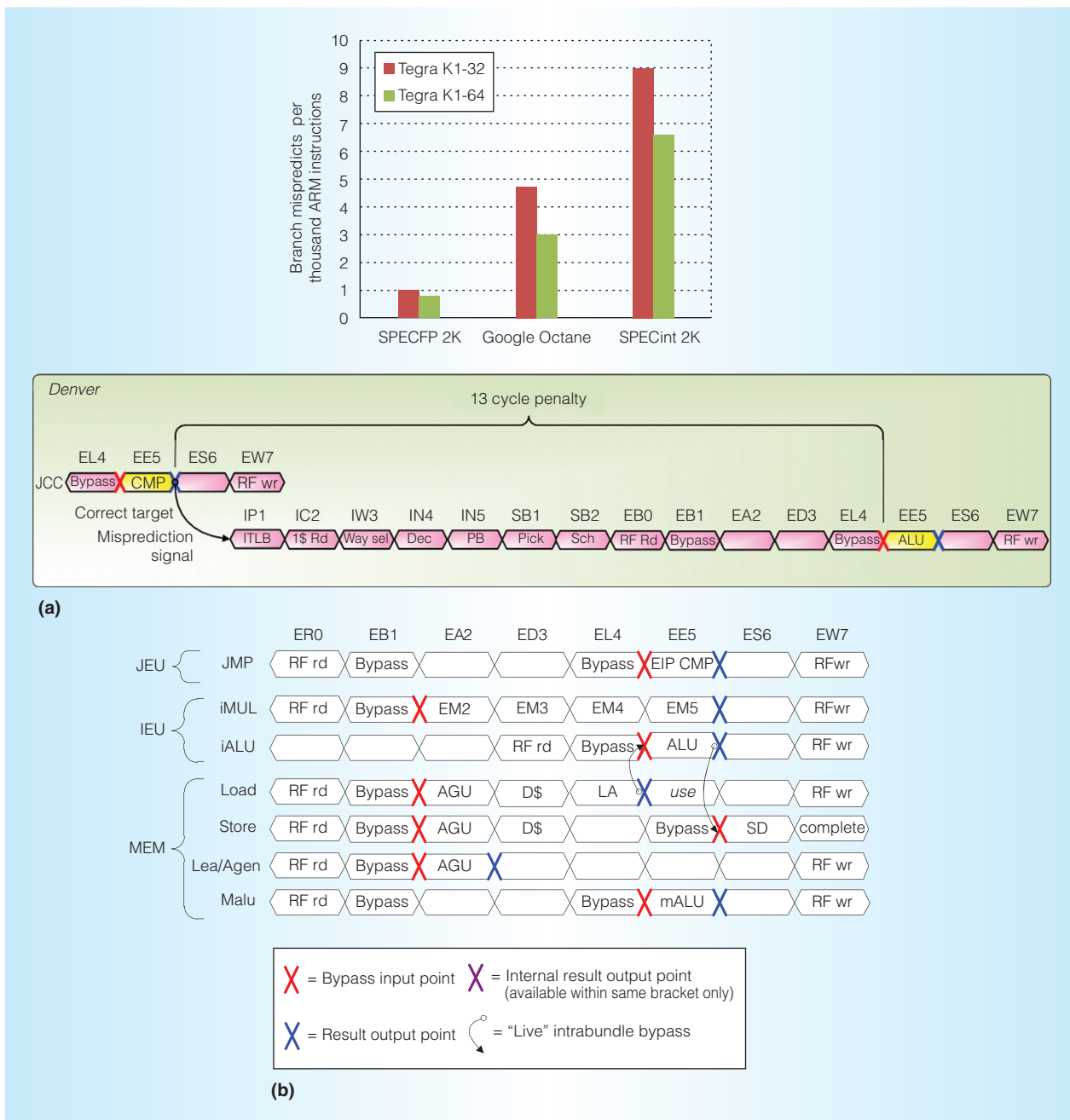


Figure 2. Denver CPU pipeline. (a) Branch predictor performance and mispredict penalty. (b) Delayed integer execution and intrabundle bypass. The register-file read for an integer arithmetic logic unit (ALU) operation is delayed by three cycles to line up with the L1 data cache read, allowing for a load, a dependent integer ALU operation, and a store to be packed and executed in a single bundle.

CPU pipeline. As Figure 2b shows, the register-file read for an integer arithmetic logic unit (ALU) operation is delayed by three cycles to line up with the L1 data cache read. The data for a load is bypassed to the integer ALU. Furthermore, the data from an

integer operation can be forwarded to a store. This allows for a load feeding a dependent integer ALU operation, which in turn feeds a store to be packed and executed within a single bundle, benefiting read-modify-write operations. Delaying integer ALU execution

forces branch mispredicts to be resolved later in the pipeline.

Dynamic code optimization

The powerful and efficient CPU microarchitecture described in the previous section provides a strong foundation for Nvidia's dynamic code optimization software to yield performance leadership and energy efficiency. Specifically, the hardware decoder provides good performance from a cold start not available in previous binary translation systems, while optimized regions can more than double the performance of hot code.

Design and validation

The DCO system employed in the Denver CPU is codesigned software that extends ideas from prior system-level binary translators.² The primary function is to execute the user's code. The secondary function is to profile execution, create, optimize, and manage regions of tens to thousands of ARM instructions to form equivalent microcode-optimized regions that execute efficiently on the underlying microarchitecture.

Latency and overhead are key metrics for acceptance of such a system, and have traditionally been seen as critical problems for binary translated machines. DCO provides twice the performance of the hardware-decoder mode overall, whereas the hardware-decoder mode reduces latency impacts and overall overhead compared to prior binary translation systems.

The DCO system is written in a mixture of C/C++ and assembly, compiled into a binary package and securely loaded. The DCO system launches microcode threads for ARM architectural execution, hot code region formation, optimization, and optimization cache management. All ARM code execution is done within a microcode thread and can be a mix of hardware decoder, static microcode, and optimized microcode sequences. After initializing the processor and the ARM state of the thread, the thread launches itself by invoking the hardware decoder at the ARM architectural reset vector. ARM execution starts out of the hardware decoder, periodically being interrupted to profile or to run other threads.

The Denver hardware decoder provides a mechanism for periodically profiling recently taken branches. This branch history is moved into a shared buffer that can be processed from other cores, thereby minimizing the latency of the interruption. The DCO system will then run a thread that uses this profile to evaluate the dynamic properties of code executing and to assemble a picture of which code regions are hottest across all the processors. On finding sufficiently hot code, the DCO system will begin an optimization process to turn this input ARM code into a microcode execution region. The optimization process uses well-known traditional³ and more speculative compiler techniques to reduce work and increase efficiency of execution on the underlying skewed pipeline. To keep the latency of interruptions to a minimum, the optimizer thread is time-sliced with ARM execution (if any) and runs in a mode that can be quickly interrupted.

Optimized regions emitted by DCO are patched and installed into a carve-out execution memory area shared between all the system's cores. This area is large enough to cover the working set of most mobile applications. Received optimized regions are checked for consistency with memory and are protected by hardware structures against completing execution if the source ARM code changes because of coherent I/O or CPU traffic. Optimized regions can reach other regions either by direct microcode jumps (chaining) or via indirect branches and returns that use an ARM instruction pointer. The hardware has associative lookup tables and an ARM return stack, which allow branches and returns to logical ARM addresses that vector to optimized regions.

Optimized regions in Denver are transactional in nature, and the optimizer can exploit various speculative techniques to extract additional parallelism without needing to generate difficult recovery code. Poisoning loads and alias-detection hardware are aggressively used to allow loads to be hoisted above control within transactions and for stores to be sunk below their normal execution points. Redundant or dead work can be eliminated. In the rare case that an optimized region cannot be executed as is owing to overspeculation, the hardware decoder or

even the DCO interpreter can execute that region of code. If the optimized region is frequently unable to be executed or experiences poor performance, the region may be annotated and reoptimized. Frequently successfully executed regions are fed back to the optimizer for further analysis. Optimization cache management uses a mark-and-sweep algorithm to manage the available optimization execution memory.

The DCO software can operate in the background when an ARM core appears to be in a low-power state. If both cores are active and DCO needs to generate optimizations, one of the cores can be preempted for a short period of time. Background operations can be a latency, performance, and energy-efficiency win in a symmetric multiprocessing system with less-utilized cores. However, this means that the DCO system itself must be responsible for some aspects of power management. The overall process is simplified and hidden from ARM software by defining low-power states to have architectural and microarchitectural components. The core provides simplified power state entry sequences for ARM architectural software and then performs the microarchitectural work to enter a low-power state when sufficient background optimization work completes. Likewise, loaded cores in the system can request that another core leave a microarchitectural low-power state to perform background operations such as creating optimized microcode sequences. The operating system can restrict this background activity to comply with power delivery system specifications.

Codesigning a hardware processor with a DCO software system creates both additional validation exposure and benefits. The DCO system can be upgraded in the field to address functionality, performance, or security issues.

Software is difficult to validate completely, and even minor changes can have unintended consequences. We mitigated this risk with extensive testing of the software stack starting long before emulation. We used the usual gamut of unit tests, directed random tests that understand optimization constructs, performance tests, and ARM's architectural validation test suite in combination with cosimulation of independently developed models. Future releases of the Denver DCO

system are considering more use of formal methods and increasingly powerful random tests to further close the validation loop.

Because the DCO system is implemented as an integral part of the hardware design, hardware bugs found late can be worked around via changes to the DCO system itself rather than burdening traditional ARM software with a complex legacy. This was a major time-to-market advantage in the Tegra K1-64 program and allowed Nvidia to start production within nine months of tape-out.

Example

Figure 3a shows an example loop from the `inflate_stored()` routine in `164.zip` from SPECint2000. Figure 3b shows the 21 ARM instructions that form the commonly executed portion of the loop. The profile collected indicates that the loop has a very high trip count, and that branches out of the loop are rarely taken. The DCO system unrolls the loop four times, hoists a loop invariant load out of the loop, further reorders loads and stores, performs if-conversion, uses non-architectural registers and predicates, and turns a never-taken compare-and-branch sequence into a microarchitectural compare and fault. The resulting inner loop schedule, shown in Figure 4, has 11 bundles that execute four iterations and yields a scheduled efficiency of 6.5 instructions per cycle. This number exceeds the overall integer width of the machine, owing to work reduction. Throughout the entire `gzip` benchmark, including cleanup code, this optimized region delivers 5.7 ARM instructions per cycle.

Performance and power

As we noted earlier, the Denver CPU provides out-of-order PC-class performance in a mobile form-factor suitable for premium tablets and fanless clamshell devices.

Table 2 highlights the Denver CPU's superior efficiency on Dhrystone. The Qualcomm APQ8084 SoC and the Intel Baytrail Z3745 run out of steam at 2 W and can't yield higher performance even when allocated higher power budgets. Beyond 2 W, the Denver CPU's efficiency scales well, yielding 87 percent higher performance than Qualcomm's APQ8084 and 3.5 times greater


```

/* read and output
the compressed data */
while (n--)
{
    NEEDBITS(8)
    slide[w++] = (uch)b;
    if (w == WSIZE)
    {
        flush_output(w);
        w = 0;
    }
    DUMPBITS(8)
}

```

(a)

```

BLOCK 3 ip=0xdc38 succ=4:100% pred=6,1
cmp r4, #7
add r3, r6, #1
bhi 0xdcba

BLOCK 4 ip=0xdc40 succ=2:0%,5:100% pred=3
ldr r0, [r10 + #0]
ldr r2, [r12 + #0]
add r14, r0, #1
cmp r2, r0
bls 0xdca2

BLOCK 5 ip=0xdc50 succ=2:0%,6:100% pred=4
ldrb r0, [r1 + r0]
cmp r3, 0x8000
mov r0, r0, r4
orr r5, r5, r0, #0
str r14, [r10 + #0]
strb r5, [r11 + r6]
add r4, r4, #8
beq 0xdcd6

BLOCK 6 ip=0xdc6c succ=3:100%,2:0% pred=5
subs r7, r7, #1
mov r6, r3
mov r5, r5, #8
sub r4, r4, #8
bne 0xdc38

```

(b)

Figure 3. Example code snippet. (a) Source code. (b) ARM assembly code instructions from the commonly executed portion of the `inflate_stored()` routine in `164.gzip` from SPECint2000.

performance than Intel's Baytrail mobile processor.

The mobile industry has moved away from Dhrystone because it is a simple L1-cache resident integer benchmark. The SPEC CPU2000 suite comprises benchmark components that are peer reviewed and inspired from a selection of integer- and floating-point-intensive applications. Many CPU2000 components have working sets that do not fit in the last-level cache and are sensitive to DRAM latency, where the effects of latency-hiding techniques like run-ahead and pre-fetcher substantially benefit the Denver CPU.

Figure 5 outlines SPECint2000 performance at different power envelopes. All data points are relative to scores achieved by the Tegra K1-32 SoC at a power budget of 2 W. An additional comparison point includes Intel's 22-nm Haswell Celeron processor, which is shipping in Chromebooks and has a thermal design power specification of 15 W. Even at the lowest operating frequency and voltage, the power consumption of the Haswell Celeron CPU was found to be above 2 W.

Figure 6 highlights the peak unconstrained performance benefit of Denver compared to other mobile processors. Workloads built from source for these comparisons use “-O3” levels of optimization with a compiler best suited for the target platform. For example, Baytrail and Haswell binaries were generated using the Intel compiler, A7 binaries use the LLVM compiler in Xcode, and Denver and S800 binaries use the gcc compiler. In addition to compelling performance at 2 W, Denver races ahead of its competitors when supplied with a higher power budget.

The Denver CPU introduces a new power management state called CC4, or the *core cluster retention state*. In this state, the core cluster supply voltage is lowered below the active minimum voltage to a retention voltage where registers and caches retain state.⁴ Before the core cluster can perform any processing, the supply voltage must be raised to the minimum active voltage. ARM software can program a model-specific register to enable transitions into and out of this new state.

Operationally, such a state is similar to a fully power-gated state, because the core can no longer respond to snoops. However, the

```

entrypoint:
{
    opttrans.cmit;
    mov_copy.64 r16 = rzero, r7;
}
{
    ld.32 r2 = [r12];
    mov_copy.64 r15 = rzero, rzero;
}

loop:
{
    opttrans.cmit;
    lea_add.32 r17 = r11, r6;
    cmp.32.le p1 = r16, 4;
}
{
    cmp.32.ls p0 = r2, r18;
    lea_add.32 r3 = r6, 1;
    njpp p0 cleanup;
}
{
    lea_add.32 r20 = r18, 2;
    cmp.32.eq p0 = r3, 0x8000;
}
{
    ld.8 r21 = [r1+r18];
    shl.32 r0 = r21, r4;
}
{
    ld.8 r21 = [r1+r18+1];
    shl.32 r22 = r21, r4;
    njpp p0 cleanup;
}
{
    ld.8 r21 = [r1+r18+2];
    shr.32 r5 = r5, 8;
    njpp p0 cleanup;
}
{
    cmp.32.eq p0 = r19, 0x8000;
    shl.32 r21 = r21, r4;
}
{
    ld.32 rsink = [r10];
    lea_add.32 lr = r18, 3;
    njpp p0 cleanup;
}
{
    cmp.32.ls p0 = r2, r20;
    or.32 r23 = r22, r21;
    njpp p0 cleanup;
}
{
    cmp.32.eq p0 = r3, 0x8000;
    st.32 [r10] = lr;
    njpp p0 cleanup;
}
{
    mov_copy.64 r6 = rzero, r3;
    shr.32 r5 = r23, 8;
    njmp loop;
}
cleanup:
    <omitted for clarity>

```

Figure 4. 164.gzip optimized microcode for the inner loop schedule of the inflate_stored() routine that yields 6.5 instructions per cycle and consists of 11 bundles.

Table 2. Dhrystone performance comparisons at various power envelopes.

Power envelope	Tegra K1-32 DMIPS	TegraK1-64 DMIPS	APQ8084 DMIPS	Baytrail Z3745 DMIPS
2 W	7,351	7,331	8,000	4,243
4 W	8,000	11,728	8,000	4,243
6 W	8,000	14,300	8,000	4,243
8 W	8,000	15,000	8,000	4,243

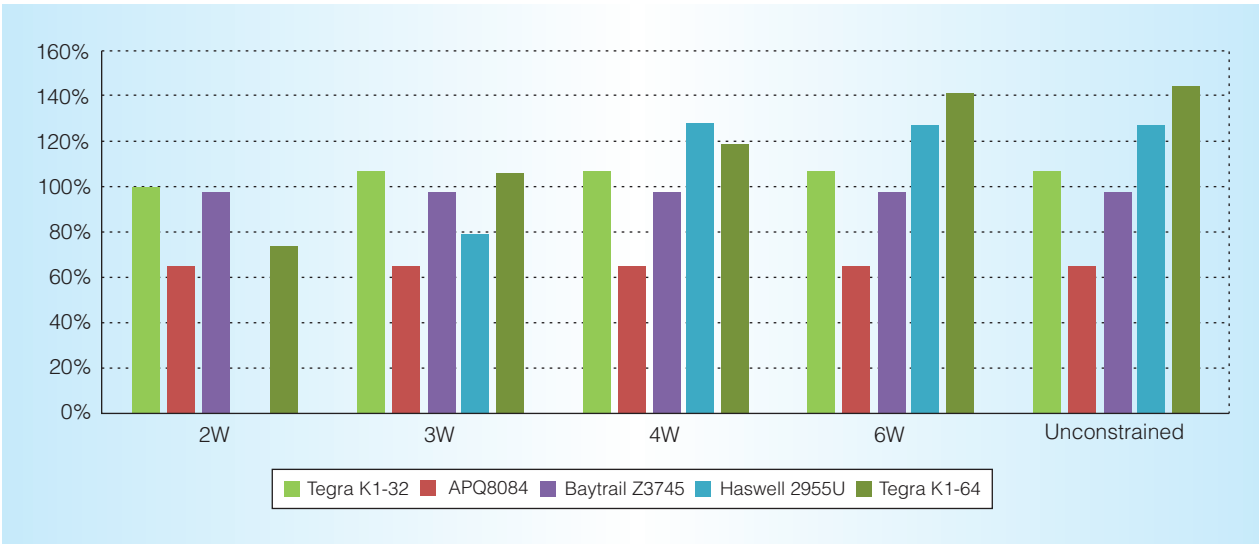


Figure 5. Denver performance at watt comparisons against mobile SoCs relative to Tegra K1-32 at 2 W. The Denver-based Tegra K1-64 processor outperforms the competition at premium tablet and clamshell power envelopes.

entry and exit latency is lower than that for a fully power-gated state, because the retention voltage is closer to the minimum active voltage, and a subset of the processor state is retained that does not need restoring. Lower latencies allow CC4 to be used more frequently than a fully power-gated state.

Figure 7 highlights the benefits of the core cluster retention state. The data was measured on Denver Silicon with a synthetic power test and displays average power versus idle duration. For workloads with short periods of idle episodes less than 100 ms, the retention state is far more efficient compared to a fully power-gated state, or traditional clock gating. As the idle episodes increase in duration, moving to a fully power-gated state is beneficial. The Denver CPU can efficiently transition between clock-gated, retention, and power-gated states depending on the workload.

The Denver CPU is the first of a modern family of ARM processors where a powerful and efficient core is paired with a modern DCO system to yield performance leadership and energy efficiency. Going forward, Nvidia anticipates DCO being utilized as an effective technology that provides higher performance and efficiency. DCO can be targeted toward high-reliability systems and can address challenges posed by future process technology because of its greater hardware and software synergy. Additionally, CPU-based DCO systems can provide benefits in cooperation with other devices in the system.

MICRO

Acknowledgments

We thank the entire Nvidia Denver CPU team for their tireless efforts toward making Denver a reality. It was a huge team effort

and the realization of a dream for all of us. Special thanks to Giang Hoang for performance and power data and to Anurag Negi for reviewing and formatting help with this article.

References

1. O. Mutlu, H. Kim, and Y.N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," *Proc. 32nd Ann. Int'l Symp. Computer Architecture*, 2005, pp. 370–381.
2. J.C. Dehnert et al., "The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges," *Proc. Int'l Symp. Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2003, pp. 15–24.
3. D.I. August et al., "Integrated Predicated and Speculative Execution in the Impact Epic Architecture," *Proc. 25th Ann. Int'l Symp. Computer Architecture*, 1998, pp. 227–237.
4. A. Naveh et al., "Power Management Architecture of the 2nd Generation Intel Core Microarchitecture, Formerly Codenamed Sandy Bridge," *Hot Chips 23*, 2011.

Darrell Boggs is a senior distinguished engineer and senior director of CPU architecture at Nvidia. His interests include computer architectures and low-power designs. Boggs has an MS in electrical engineering from Brigham Young University. Contact him at dboggs@nvidia.com.

Gary Brown is the vice president of CPU engineering at Nvidia. His interests include efficient microarchitectures and CPU-GPU synergy. Brown has an MS in electrical engineering from Brigham Young University. Contact him at garyb@nvidia.com.

Nathan Tuck is a principal engineer and software manager at Nvidia. His interests include software optimization techniques and CPU architectures. Tuck has a BS in computer science from Harvey Mudd College. Contact him at ntuck@nvidia.com.

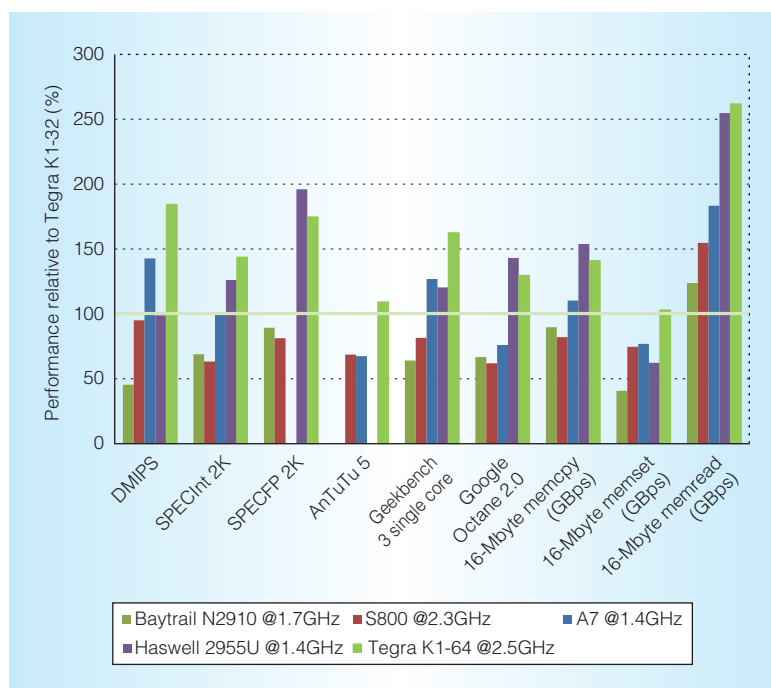


Figure 6. Denver performance comparisons against competing mobile processors highlighting its peak performance benefits across a varied set of relevant mobile workloads.

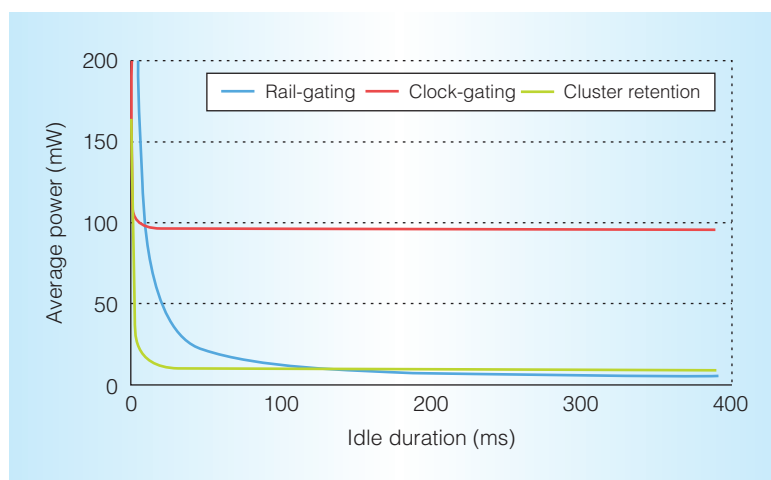


Figure 7. Core cluster retention state. The benefits of power savings for low idle durations are shown.

K S Venkatraman is a principal engineer and senior architecture manager at Nvidia. His interests include architecture, hardware design, verification, performance, and competitive analysis. Venkatraman has an MS in electrical and computer engineering from Villanova University. Contact him at ksv@nvidia.com.