# Accelerators For Everything

Bolaji Bankole, Jens Ertman

# QsCores (Quasi-Specific Cores)

# What is a QsCore

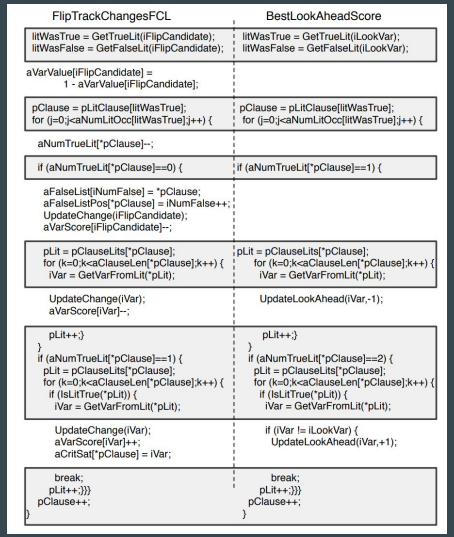
- A hardware accelerator core connected to a CPU
- Composed to accelerate several specific segments of code
- Synthesized hardware determined before the chip is manufactured
- Can be combined with other QsCores to accelerate more at the expense of area
  - And energy but not in the same way (we'll get to it)
- Can be called with arguments in lieu of running on the general purpose CPU

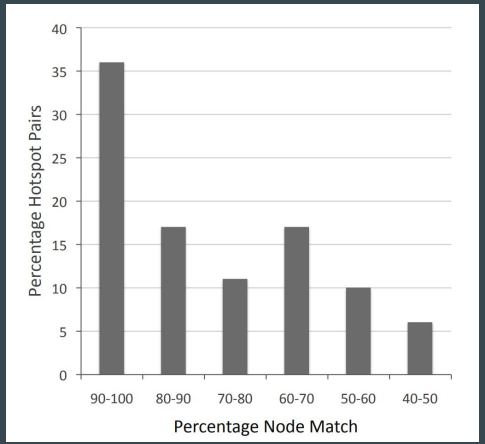
# Motivation

- With advances in transistor technology counts are going up but usable area is going down
- Why not take extra area and make accelerators for common tasks?
  - What if those accelerators focused on energy efficiency?
  - What if those accelerators combined multiple similar “hotspots” of the code to cover more of the runtime?
- More energy efficiency means that more compute can occur on the chip

# Mining for Similar Code Patterns

- Generate a program dependence graph for each hotspot in the code
- Compare these graphs based on the similarity of their nodes and dependencies
- Take the two hotspots and generate a new graph that performs both



# Determining the Set of QsCores

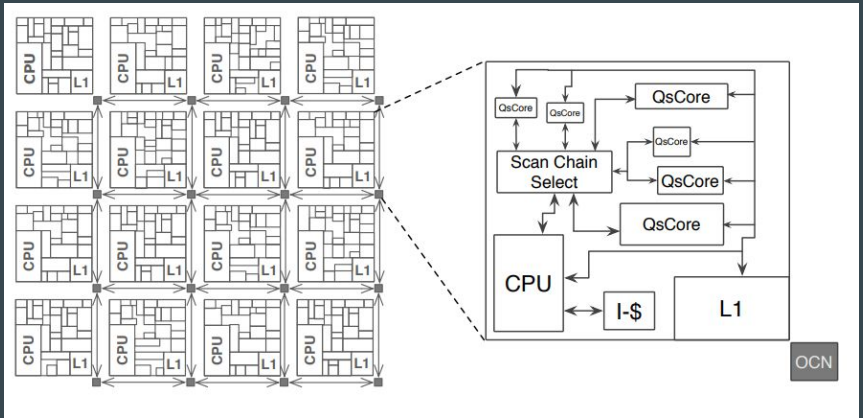
- Generate all pairs in the merge set
- Take the highest quality QsCore merge and replace the previous two in the set with it
- Keep going until either an area constraint is met or there is nothing left to merge

$$Q_b = \frac{C_b S_b}{A_b P_b}$$

**while** $|B| > 1$ **do**
  $(b_1, b_2) = \text{varmax}_{(b_1 \in B, b_2 \in B)} \frac{C_{b_1 \bowtie b_2}}{A^2_{b_1 \bowtie b_2}} - \frac{C_{b_1}}{A^2_{b_1}} - \frac{C_{b_2}}{A^2_{b_2}}$
  $B = B \setminus \{b_1, b_2\}$
  $B = B \cup \{b_1 \bowtie b_2\}$
  Record the merging of $b_1$ and $b_2$ and the resulting values of $Q'_B$ and $\sum_{b \in B} A_b$.
**end while**

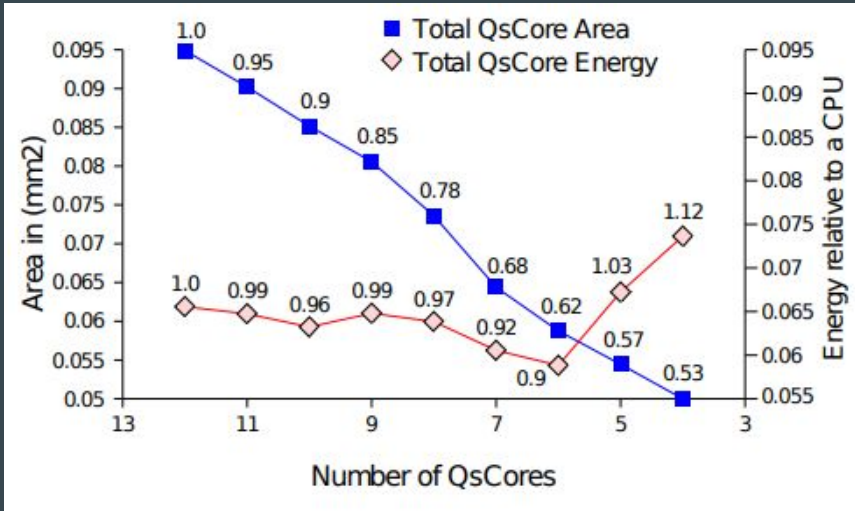
# Physical QsCores

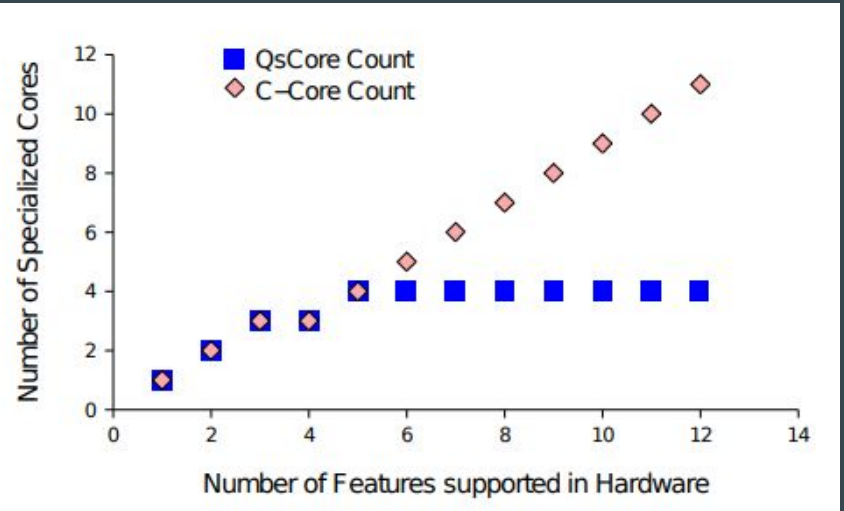
- Generated from the C code to verilog then synthesized
- Cores are then integrated with a CPU with shared D and I cache using scan chains



# Results Core Count

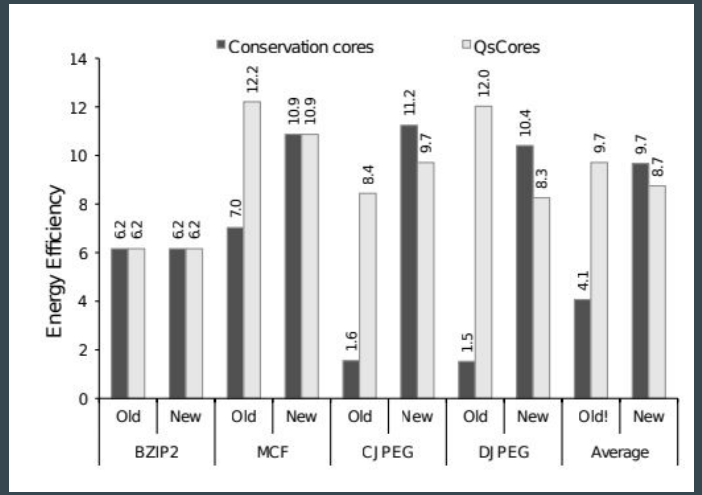
- Energy use increases slower than decreasing area
- Much fewer cores required to cover a larger number of features



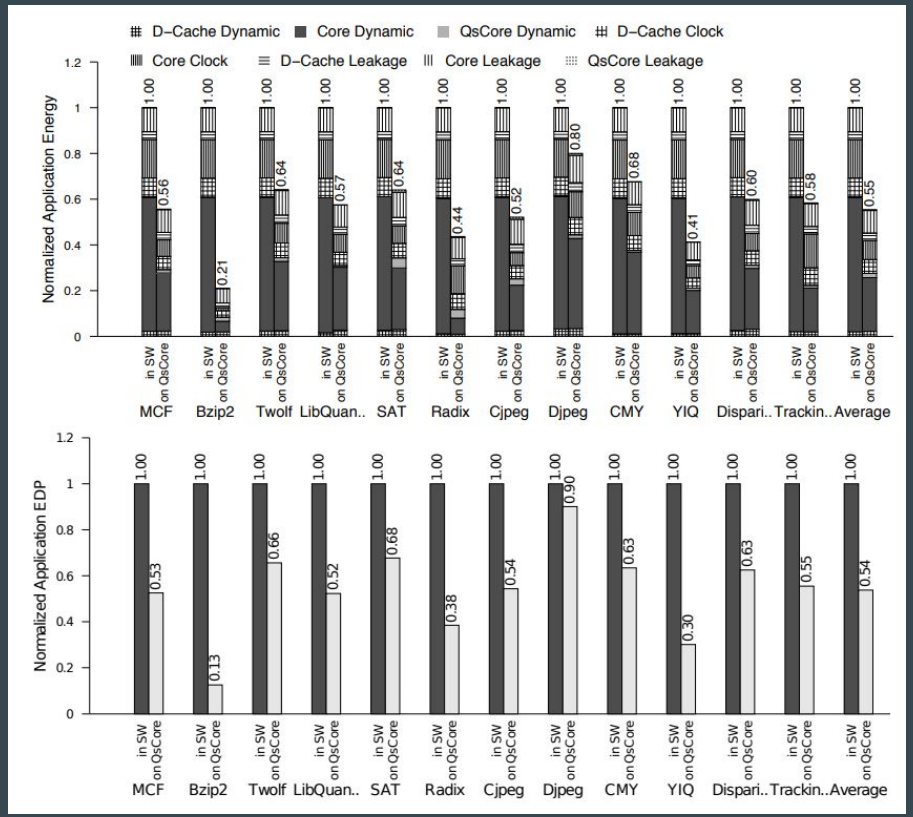
# Quality of QsCores

- In testing the set of QsCores determined by their algorithm it created the best set of QsCores in all cases
- QsCores are backwards compatible if old versions of the code are included in the set of hotspots to be merged



# Final Results of Energy Efficiency

| | |
|---|---|
| Average QsCore power | 6.7 mW |
| Average QsCore energy efficiency (compared to baseline CPU) | 23.23× |
| Average execution time per QsCore invocation | 8951 cycles |
| Average invocation overhead | 316 cycles |
| Average system execution coverage per QsCore | 4.41% |
| Average QsCore area | 0.041 $mm^2$ |



# Conservation Cores

# What

- Accelerators with the goal of energy reduction
  - Less sensitive in this than performance oriented accelerators
- Patchable(?) to add flexibility and longevity
- Communicate with the system through shared caches and scan-chain interface
- Very similar idea to QsCores

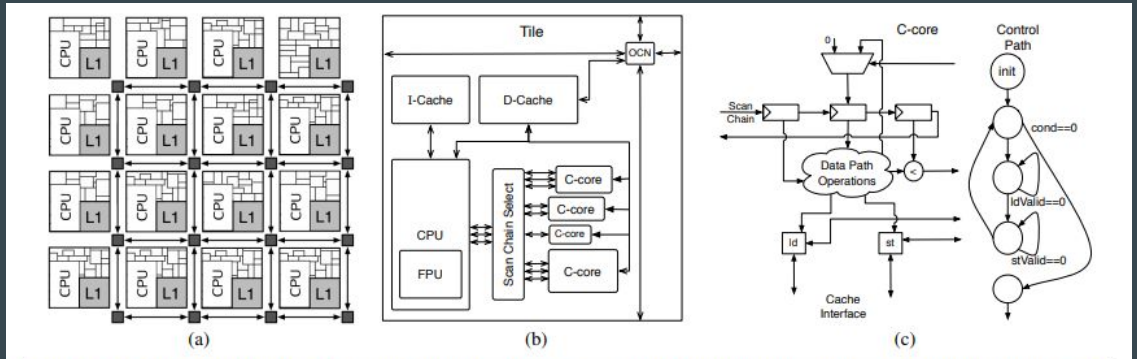



**Figure 1. The high-level structure of a c-core-enabled system** A c-core-enabled system (a) is made up of multiple individual tiles (b), each of which contains multiple c-cores (c). Conservation cores communicate with the rest of the system through a coherent memory system and a simple scan-chain-based interface. Different tiles may contain different c-cores. Not drawn to scale.

# Why

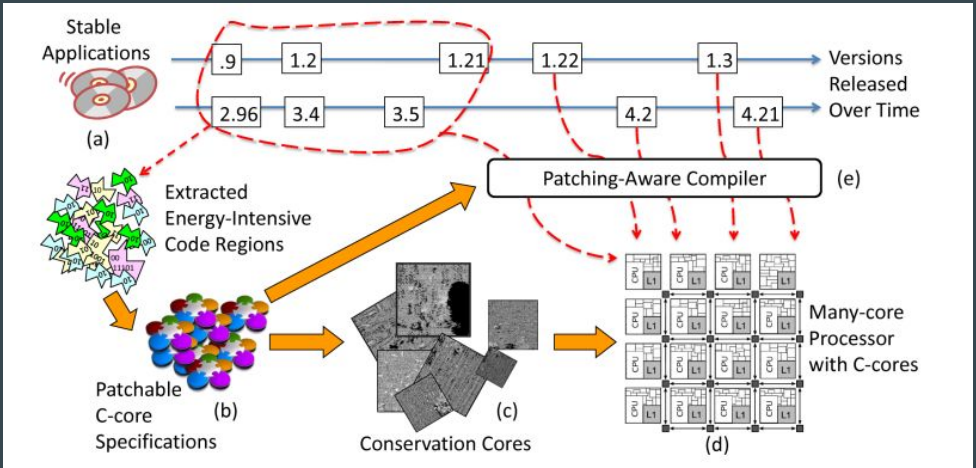
- Breakdown of CMOS scaling means that only so much a of processor can be practically ran at full speed
- Trade area for energy efficiency to get better use of the die area
- Same overall rationale as QsCores

| Process | 90 nm TSMC | 45 nm TSMC | 32 nm ITRS |
|---|---|---|---|
| Frequency (GHz) | 2.1 | 5.2 | 7.3 |
| $mm^2$ Per Op. | .00724 | .00164 | .00082 |
| # Operators | 41k | 180k | 360k |
| Full Chip Watts | 455 | 1225 | 2401 |
| Utilization at 80 W | 17.6% | 6.5% | 3.3% |

**Table 2. Experiments quantifying the utilization wall** Our experiments used Synopsys CAD tools and TSMC standard cell libraries to evaluate the power and utilization of a 300 $mm^2$ chip filled with 64-bit adders, separated by registers, which is used to approximate active logic in a processor.

# How

- Most frequently used code snippets are augmented for reconfigurability and synthesized
- Compiler knows the c-cores in the processor and includes stubs to invoke them, with patches when necessary



# C-Core Function

- State machine closely resembles code structure
  - Helps memory ordering
- Multi cycle loops for complex operations and memory
- Small scan chains for arguments, large ones for patches, other ones for internal state
  - Added instructions to move data to and from scan chains
- At runtime, check for relevant c-core and use it if available

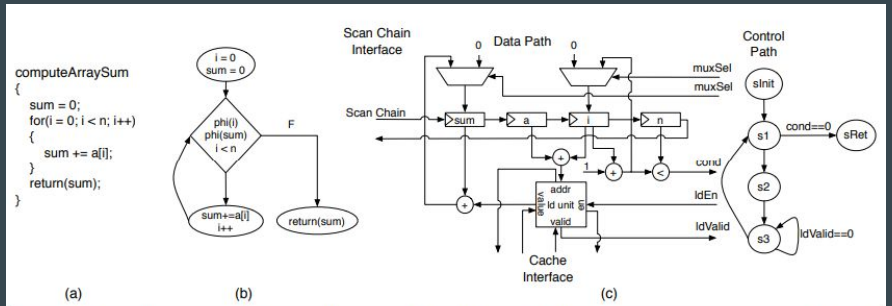



**Figure 3. Conservation core example** An example showing the translation from C code (a), to the compiler's internal representation (b), and finally to hardware (c). The hardware schematic and state machine correspond very closely to the data and control flow graphs of the C code.

# Patching

- Configurable constants
  - Registers to change constants in the program
- Generalized operators
- Control flow changes
  - Raise exceptions for CPU to handle, modify conditionals, etc

| Structure | Area ($\mu m^2$) | Replaced by | Area ($\mu m^2$) |
|---|---|---|---|
| adder<br>subtractor | 270<br>270 | AddSub | 365 |
| comparator (GE) | 133 | Compare6 | 216 |
| bitwise AND, OR<br>bitwise XOR | 34<br>56 | Bitwise | 191 |
| constant value | $\sim 0$ | 32-bit register | 160 |

**Table 5. Area costs of patchability** The AddSub unit can perform addition or subtraction. Similarly, Compare6 replaces any single comparator (e.g., $\geq$) with any of ($=, \neq, \geq, >, \leq, <$). Constant values in non-patchable hardware contribute little or even "negative" area because they can enable many optimizations.

# Results

- Benefits (and costs) of patchability

| C-core | Ver. | Key | LOC | % Exe. | Area ($mm^2$) Non-P. | Area ($mm^2$) Patch. | Freq. (MHz) Non-P. | Freq. (MHz) Patch. |
|---|---|---|---|---|---|---|---|---|
| **bzip2** | | | | | | | | |
| fallbackSort | 1.0.0 | A *i* | 231 | 71.1 | 0.128 | 0.275 | 1345 | 1161 |
| fallbackSort | 1.0.5 | F *i* | 231 | 71.1 | 0.128 | 0.275 | 1345 | 1161 |
| **cjpeg** | | | | | | | | |
| extract_MCUs | v1 | A *i* | 266 | 49.3 | 0.108 | 0.205 | 1556 | 916 |
| get_rgb_ycc_rows | v1 | A *ii* | 39 | 5.1 | 0.020 | 0.044 | 1808 | 1039 |
| subsample | v1 | A *iii* | 40 | 17.7 | 0.023 | 0.039 | 1651 | 1568 |
| extract_MCUs | v2 | B *i* | 277 | 49.5 | 0.108 | 0.205 | 1556 | 916 |
| get_rgb_ycc_rows | v2 | B *ii* | 37 | 5.1 | 0.020 | 0.044 | 1808 | 1039 |
| subsample | v2 | B *iii* | 36 | 17.8 | 0.023 | 0.039 | 1651 | 1568 |
| **djpeg** | | | | | | | | |
| jpeg_idct_islow | v5 | A *i* | 223 | 21.5 | 0.133 | 0.222 | 1336 | 932 |
| ycc_rgb_convert | v5 | A *ii* | 35 | 33.0 | 0.023 | 0.043 | 1663 | 1539 |
| jpeg_idct_islow | v6 | B *i* | 236 | 21.7 | 0.135 | 0.222 | 1390 | 932 |
| ycc_rgb_convert | v6 | B *ii* | 35 | 33.7 | 0.024 | 0.043 | 1676 | 1539 |
| **mcf** | | | | | | | | |
| primal_bea_mpp | 2000 | A *i* | 64 | 35.2 | 0.033 | 0.077 | 1628 | 1412 |
| refresh_potential | 2000 | A *ii* | 44 | 8.8 | 0.017 | 0.033 | 1899 | 1647 |
| primal_bea_mpp | 2006 | B *i* | 64 | 53.3 | 0.032 | 0.077 | 1568 | 1412 |
| refresh_potential | 2006 | B *ii* | 41 | 1.3 | 0.015 | 0.028 | 1871 | 1639 |
| **vpr** | | | | | | | | |
| try_swap | 4.22 | A *i* | 858 | 61.1 | 0.181 | 0.326 | 1199 | 912 |
| try_swap | 4.3 | B *i* | 861 | 27.0 | 0.181 | 0.326 | 1199 | 912 |

**Table 3. Conservation core statistics** The c-cores we generated vary greatly in size and complexity. In the "Key" column, the letters correspond to application versions and the Roman numerals denote specific functions from the application that a c-core targets. "LOC" is lines of C source code, and "% Exe." is the percentage of execution that each function comprises in the application.

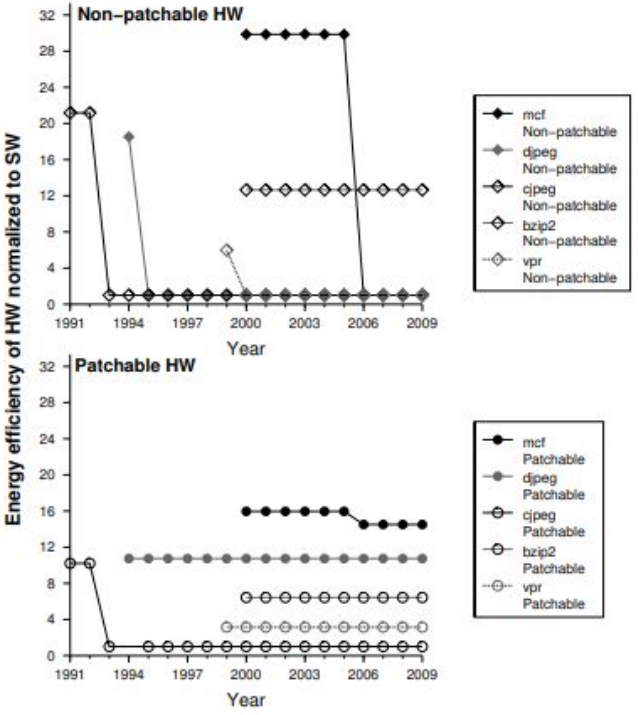
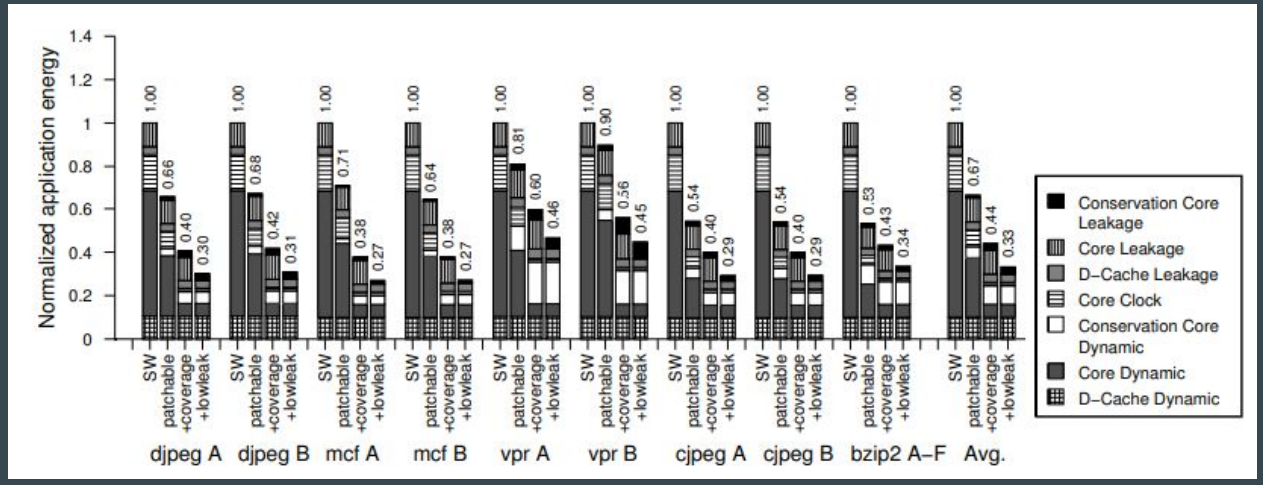
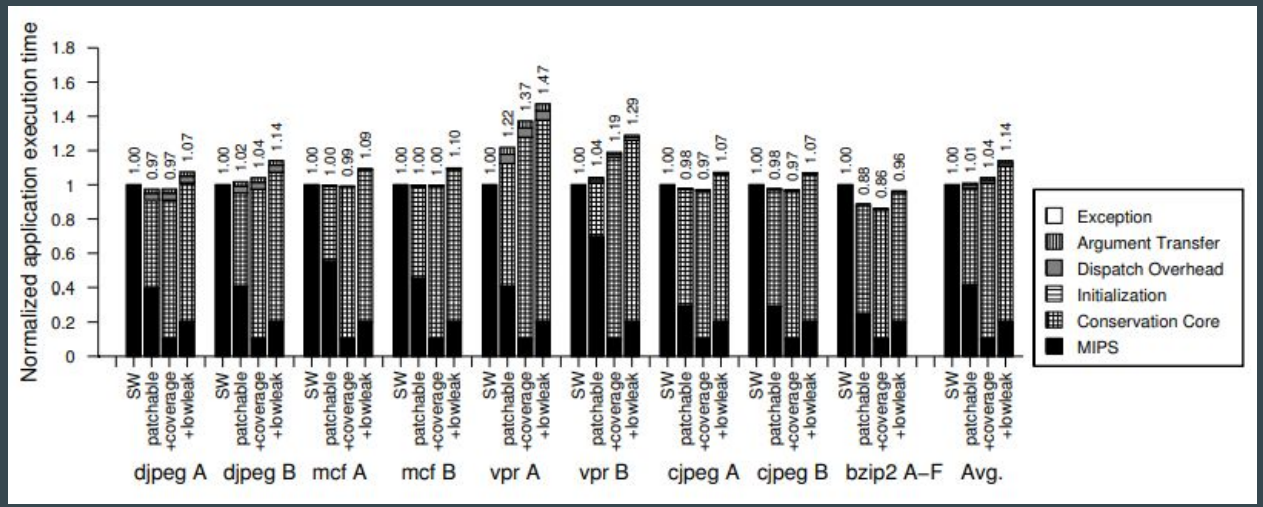



**Figure 7. Conservation core effectiveness over time** Since c-cores target stable applications, they can deliver efficiency gains over a very long period of time.

# Results


Normalized application energy
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1.00 0.66 0.40 0.30
1.00 0.68 0.42 0.31
1.00 0.71 0.38 0.27
1.00 0.64 0.38 0.27
1.00 0.81 0.60 0.46
1.00 0.90 0.56 0.45
1.00 0.54 0.40 0.29
1.00 0.54 0.40 0.29
1.00 0.53 0.43 0.34
1.00 0.67 0.44 0.33
SW
patchable
+coverage
+lowleak
djpeg A
djpeg B
mcf A
mcf B
vpr A
vpr B
cjpeg A
cjpeg B
bzip2 A–F
Avg.
Conservation Core Leakage
Core Leakage
D–Cache Leakage
Core Clock
Conservation Core Dynamic
Core Dynamic
D–Cache Dynamic


Normalized application execution time
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1.00 0.97 0.97 1.07
1.00 1.02 1.04 1.14
1.00 1.00 0.99 1.09
1.00 1.00 1.00 1.10
1.00 1.22 1.37 1.47
1.00 1.04 1.19 1.29
1.00 0.98 0.97 1.07
1.00 0.98 0.97 1.07
1.00 0.88 0.86 0.96
1.00 1.01 1.04 1.14
SW
patchable
+coverage
+lowleak
djpeg A
djpeg B
mcf A
mcf B
vpr A
vpr B
cjpeg A
cjpeg B
bzip2 A–F
Avg.
Exception
Argument Transfer
Dispatch Overhead
Initialization
Conservation Core
MIPS