Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems

Manish Arora, Jack Sampson, Nathan Goulding-Hotta, Jonathan Babb[+], Ganesh Venkatesh,
Michael Bedford Taylor, Steven Swanson
Department of Computer Science and Engineering
University of California, San Diego
[+]*Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology*

***Abstract*— This paper describes an architecture and FPGA synthesis toolchain for building specialized, energy-saving coprocessors called *Irregular Code Energy Reducers* (ICERs) for a wide range of unmodified C programs. FPGAs are increasingly used to build large-scale systems, and many large software systems contain relatively little code that is amenable to automatic, semi-automatic, or even manual parallelization. Whereas accelerator approaches have traditionally achieved energy benefits as a side effect from increasing performance via parallel execution, ICERs aim to achieve energy gains even on code with little exploitable parallelism.**

Traditional approaches to automatically generating accelerators from existing software rely on inferring parallel execution from serial code, so they face the same code analysis challenges as parallelizing compilers. In contrast, because the ICER approach targets energy rather than performance, it easily scales to large, irregular applications that are poor candidates for traditional acceleration. Our results show that, compared to a baseline system with soft processor cores, ICERs can reduce energy consumption by up to 9.5× for the code they target and 2.8× for whole applications.

***Keywords*-Accelerator architectures; Reconfigurable architectures; Energy efficiency; High level synthesis**

I. Introduction

As reconfigurable fabrics scale in capacity and capability, the systems they implement will similarly expand in scope and complexity. Designers will increasingly use large bodies of existing code in a high-level language like C to specify the behavior of these systems. By moving from a pure software implementation to a hybrid architecture, designers hope to improve performance and, increasingly, reduce energy consumption.

Although traditional high-level synthesis (HLS) tools make it easier to create coprocessors that increase performance by several orders of magnitude, these approaches have their limits. HLS tools must infer parallel execution from serial code, so they face the same challenges as parallelizing compilers: Analyzing pointers in free-form code is difficult, memory parallelism is often scarce, and it is often difficult to extract and formulate efficient parallel schedules for critical loops. Frequently, parallelization is only possible after significant human refactoring of the underlying algorithms (e.g., [1], [8], [9]).

Because current HLS tools focus on performance above all else, these tools can only save energy on code they can accelerate. However, power and energy concerns are becoming increasingly dominant constraints for all executed code, and HLS techniques should be able to significantly reduce energy consumption even when they cannot improve performance.

This paper describes a new approach to HLS that focuses on building custom coprocessors that increase energy efficiency for unmodified C code, regardless of whether acceleration is possible. We call these coprocessors *Irregular Code Energy Reducers*, or *ICERs*. The ICER toolchain does not rely on parallelization techniques to build efficient hardware. As a result, a design can incorporate ICERs for *any* code in which it spends a large fraction of execution time, regardless of whether the code contains extractable parallelism. We envision ICERs working along with conventional accelerators to maximize both performance and efficiency for complex FPGA-based systems.

We evaluate ICERs using a collection of large, hard-to-parallelize, irregular programs such as a graph flow solver, search, and a B-tree implementation. Our results show that, relative to a baseline system with soft processor cores, ICERs can increase energy efficiency of individual functions by up to 9.5×. For whole applications, ICERs reduce energy consumption by 2.8×. ICER performance is almost identical to soft core performance, on average.

II. Architecture overview

The ICER toolchain automatically converts application source code into a hardware-software partitioned system consisting of one or more ICERs integrated with a soft core. It profiles input applications and selects regions of code for conversion into hardware based on dynamic execution coverage. Unlike conventional C-to-FPGA design flows, our toolchain's primary goal is energy efficiency rather than performance. This shift in focus allows us to support a wider range of programming constructs than conventional accelerator design flows, such as arbitrary pointers and recursion.

In this section, we first describe a system architecture integrating ICERs with a soft core processor and its memory hierarchy. Then, we overview the automatic selection and generation of ICERs.

A. System Architecture

Figure 1 shows a block diagram of an ICER-enabled system. The CPU controls the ICERs and executes code that no ICER covers. The CPU and ICERs share the L1 data cache. Below, we describe the soft core and ICER components, their interfaces, and the execution and memory model for the system.

Execution model ICERs are drop-in replacements for the code they implement. This backward compatibility allows the soft core+ICER system to gracefully degrade if ported

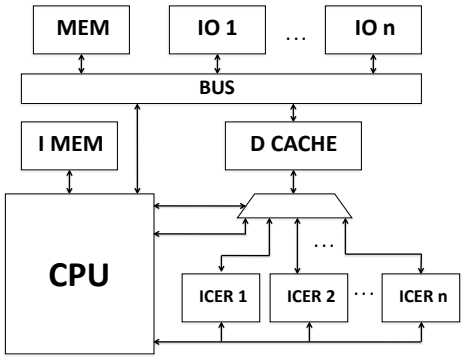

Figure 1. **ICER system architecture** An ICER-based system features a soft core that controls one or more ICERs. The shared L1 cache acts as the primary interface among the ICERs and between the soft core and ICERs. A narrow secondary network allows the processor to transfer arguments to the ICERs and initiate execution.

to a smaller fabric. When the profiler selects a function to convert into hardware, the compiler will insert stubs that enable the runtime to select between using the ICER or executing the code on the soft core.

When an ICER finishes execution or raises an exception, control transfers back to the soft core. The soft core extracts the cause from the ICER exception register and executes an appropriate software handler. The soft core has access to all internal state in the ICER via a secondary interface and can re-initialize the ICER to resume execution starting in an arbitrary control state. As a result, the toolchain supports code that contains non-inlineable function calls, such as dynamically linked library and system calls.

Memory As Figure 1 shows, the ICERs and the soft core share the coherent L1 data cache and they use the same address space. The ICERs ensure compatibility by splitting basic blocks into control blocks containing at most one memory operation and activating only one control block at a time, guaranteeing that memory operations execute in the correct order. If a memory operation takes multiple cycles, execution of the current block stalls until the memory request completes.

Soft core We use an energy-efficient, pipelined MIPS processor as our soft core processor. The MIPS processor core derives from the MIT Raw [17] processor and has an eight stage, in-order, single-issue pipeline. The core includes an L1 data cache, instruction memory, and the Raw network-on-chip router for one Raw tile. Microbenchmarks show that the Raw processor soft core operates within ∼20% of the dynamic power of a similarly configured MicroBlaze, with comparable or better instruction throughput.

B. ICER Interface

Fast invocation makes it profitable to build ICERs even for small functions. The toolchain inserts wrappers around each selected region. These invoke the ICERs, passing any global variables by reference as additional input arguments. The coherent memory interface makes marshalling costs similar to function invocations. To transfer control to an ICER, the soft core passes up to eight arguments over the secondary network, starts the ICER, and goes to sleep in a clock-gated state.

Benchmark	Description	Suite	LOC
bzip2 [16]	Data compression algorithm	SPEC 2000	7625
cjpeg [6]	JPEG image compression	EEMBC	13272
mcf [16]	Single-depot vehicle scheduling	SPEC 2000	2478
radix [21]	Sorting algorithm	SPLASH–2	895
viterbi [5]	Viterbi decoder	EEMBC	11154
b-tree [3]	Range traversal on a b-tree	IBS	222

Benchmark	# ICERs	Coverage (%)	Freq. (MHz)	Slice Regs.	LUTs	DSP48Es
bzip2	1	68.2	81	8661	22675	1
cjpeg	3	70.9	135	10591	22354	52
mcf	2	44.5	97	4649	9588	0
radix	1	94.0	81	6172	14305	17
viterbi	1	98.6	78	7097	18716	18
b-tree	1	70.3	123	1968	3733	0

Table I. **ICER Statistics** We used our toolchain to automatically generate nine ICERs.

C. ICER Generation

The ICER toolchain makes use of the OpenIMPACT (1.0rc4) [13] and LLVM (2.4) [10] compiler infrastructures to select and transform code regions into ICERs. It accepts all C programs that the above tools accept, including programs with arbitrary pointer references, gotos, switch statements, and loops with complex conditions. The ICER toolchain uses inlining to remove function call overhead where possible.

By design, the ICER datapath and control closely resemble the data and control flow graphs of the original C code as expressed in OpenIMPACT's Lcode intermediate representation, although our toolchain splits basic blocks containing multiple memory operations. This allows for simple semantics when transferring control between an ICER and the soft core during the ICER's execution. Every static operator in the intermediate representation becomes a dedicated functional unit, and every basic block live-out value becomes a register. Memory operations within an ICER share a single time-multiplexed cache interface.

The toolchain also constructs a control unit alongside the datapath that follows the control flow of the software computation. This control unit activates one basic block per cycle, tracking the transitions between basic blocks via branch outcomes. For multi-cycle and variable-latency operations, the control unit remains in the current active basic block until the operation completes.

III. Results

In this section, we discuss the benchmarks we use to evaluate ICERs, describe our experimental methodology, and analyze the impact of using ICERs on performance and efficiency for both the targeted function and whole application.

A. Benchmarks

Table I describes the six irregular applications we use. They come from SPEC 2000, EEMBC, SPLASH–

2, and IBS (an Irregular Benchmark Suite). The benchmarks perform irregular, non-parallelizable computation, including data compression, sorting, and data-dependent graph traversals. The average size of an input benchmark program is 5,941 lines of code (excluding headers).

We used the ICER toolchain to automatically generate nine ICERs. Table I shows statistics for the resulting hardware. The coverage column measures the percentage of execution time spent in the code regions converted into ICERs. Clock frequencies for the ICERs range from ∼80 to 149 MHz, matching or exceeding the soft core synthesis frequency of 80 MHz. FPGA resource usage (slice registers, LUTs, DSP48Es, etc.) varies from roughly one-third to two times that of a single soft core.

B. Methodology

We synthesized ICERs using the Xilinx toolflow for the Virtex 5 family of FPGAs. The specific device targeted was an xc5vlx110t-ff1136-3. We synthesized the soft core using Synopsys Synplify followed by mapping, placement, routing, and optimizations using Xilinx tools.

We use a cycle-accurate simulation infrastructure based on *btl* [17]. When the toolchain generates ICERs it also generates models of the new hardware for the cycle-accurate system simulator.

To measure ICER power usage, our simulator traces all ICER inputs and outputs for sample periods of 10,000 cycles. From each trace, we create a testbench to drive a post-place and route model using the Synopsys VCS (C-2009.06) logic simulator. This generates VCD activity files, which we use as inputs to Xilinx XPower. A similar process generates power numbers for the soft core using samples from equivalent portions of software execution.

C. ICER Performance and Efficiency

Figure 2 shows the energy-delay product (EDP) improvement, speedup, and energy of ICER-enabled systems and ICERs, normalized to the MIPS soft core. ICERs use up to 9.5× less energy (5.3× on average) for the regions of code that they target. They do this while maintaining comparable or better levels of performance, resulting in an average EDP improvement of 5.1×.

Performance trends for the entire applications are very similar to those for the targeted functions. Excepting b-tree, ICER-based system performance is comparable to or better than soft core performance. For both b-tree and mcf, a lack of memory pipelining in ICERs limits performance. At the application level, energy gains are highly correlated with application coverage, because of the soft core's high clock tree energy and greater BRAM energy. Across all benchmarks, ICERs use 2.27× less energy, improving EDP by 2.32×. Code regions with poor memory performance show the largest energy improvements, with mcf achieving a 9.5× improvement for covered code.

Figure 2 (bottom) shows the breakdown of component energy (block RAMs, DSP, wires, logic, and clock) across the workload for the soft core MIPS processor, combined

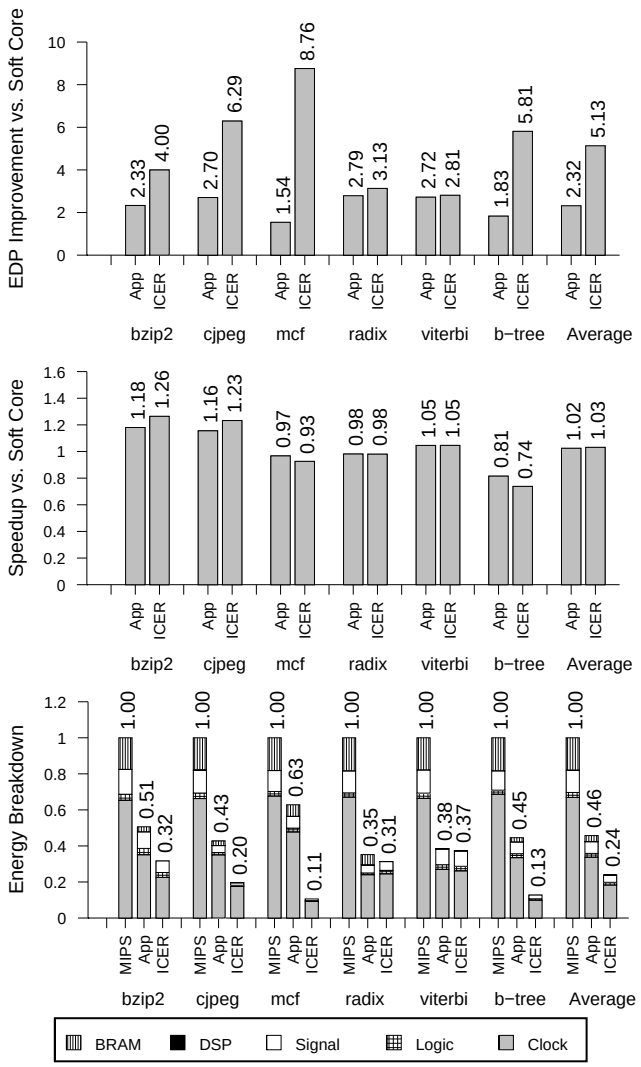

Figure 2. **ICER EDP improvement, speedup, and energy breakdown** ICERs significantly improve energy-delay product (top, higher is better), maintaining performance (middle, higher is better), while greatly reducing energy (bottom, lower is better) compared to a soft core MIPS processor. Bars labeled 'App' report values for the whole benchmark, and bars labeled 'ICER' correspond to code covered by the ICER. Energy and EDP improvements are closely correlated with application coverage.

system, and ICERs in isolation. In every case, the largest component is clock tree energy. ICERs greatly reduce clock energy and all but eliminate block RAM energy for the code that they target. Even at the application level, ICERs reduce clock energy by half. ICERs provide great savings here, but the clock still accounts for a large fraction, highlighting the importance of clock gating the soft core and inactive ICERs. It also showcases how the ICER execution model is an excellent fit for FPGAs: Since only one basic block in an ICER is active at a time, the synthesis tools can clock-gate ICERs more aggressively to take advantage of their very low duty cycles.

IV. Related Work

This section compares ICERs with previous efforts in high-level synthesis, custom coprocessor design, and other FPGA-based accelerator platforms.

High-level synthesis High-level synthesis research has been going on for several decades leading to a variety of commercial tools, as detailed in a recent book [4]. The primary goal of C-to-silicon synthesis frameworks such as AutoESL [23], Impulse C [7], Synopsys Synphony/PICO [15], CHiMPS [14], and Altera C2H [11] is to reduce the effort that creating accelerators requires, by building them directly from a high-level language. To accelerate execution, these tools must either infer parallel execution from serial code or force the programmer to rewrite their code in a more explicitly parallel language or dialect [18]. Because of this, they face the same challenges as parallelizing compilers. In addition, acceleration typically requires a parallel memory system that is difficult to integrate with existing serial soft cores. Because of the difficulty of these challenges, existing tools tend to compromise on automation and backward compatibility. In contrast, ICERs focus on energy first and performance second. This allows the approach to be completely automated, achieve high execution coverage, retain backward compatibility, and save energy on arbitrary code.

Reconfigurable substrates Several related efforts examine the benefits of coupling non-commodity reconfigurable fabrics with a processor core for program acceleration. GARP [2] and Chimaera [22] were early works that proposed automated approaches for offloading execution to reconfigurable fabrics integrated with a hard core. Tartan [12] examined the implications of mapping entire programs onto a hierarchical coarse-grained asynchronous fabric. Warp [19] performs dynamic translation of binaries to a specialized FPGA substrate optimized for on-the-fly synthesis, but employs an additional soft core to run the high-performance synthesis infrastructure. Conservation cores [20] have recently been proposed to create energy-efficient ASICs for irregular applications, but have limited reconfigurability.

V. Conclusion

We have presented ICERs, customized logic circuits that reduce the dynamic power of FPGA system components traditionally run on soft cores. Tight coupling with a soft processor, including sharing of the L1 data cache and support for arbitrary control transitions between the soft core and ICER allow ICERs to be drop-in replacements for the code they implement. This greatly eases system-level design and testing complexity, and allows for full automation of both ICER construction and system integration with no programmer intervention. ICERs retain the performance of the soft cores they replace, but reduce compute energy by 5.3× and improve EDP by 5.1×.

Acknowledgment

This research was funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794. We would also like to thank Adrian Caulfield for help with the b-tree benchmark.

References

[1] D. Abramson, P. Logothetis, A. Postula, and M. Randall. FPGA Based Custom Computing Machines for Irregular Problems. In *HPCA*, 1998.
[2] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, April 2000.
[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, second edition, 2001.
[4] P. Coussy and A. Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.
[5] Embedded Microprocessor Benchmark Consortium. EEMBC benchmark suite. http://www.eembc.org.
[6] I. J. Group. Library for JPEG image compression. http://www.ijg.org/.
[7] Impulse C website. http://www.impulseaccelerated.com/.
[8] N. Kapre and A. DeHon. Accelerating SPICE Model-Evaluation using FPGAs. *FCCM*, 2009.
[9] M. Karkooti, P. Radosavljevic, and J. R. Cavallaro. Configurable LDPC decoder architectures for regular and irregular codes. *Journal of Signal Processing Systems*, 2008.
[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
[11] D. Lau, O. Pritchard, and P. Molson. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. In *FCCM*, pages 45–56, Washington, DC, USA, 2006. IEEE Computer Society.
[12] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. Tartan: Evaluating spatial computation for whole program execution. *ASPLOS*, 2006.
[13] OpenIMPACT. http://gelato.uiuc.edu/.
[14] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig. Performance and power of cache-based reconfigurable computing. In *ISCA*, pages 395–405, 2009.
[15] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal VLSI Signal Processing Systems*, June 2002.
[16] SPEC. SPEC CPU 2000 benchmark specifications, 2000.
[17] M. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA*, 2004.
[18] D. Unnikrishnan, J. Zhao, and R. Tessier. Application specific customization and scalability of soft multiprocessors. In *FCCM*, 2009.
[19] F. Vahid, G. Stitt, and R. Lysecky. Warp Processing: Dynamic Translation of Binaries to FPGA Circuits. *Computer*, July 2008.
[20] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.
[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995.
[22] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA*, 2000.
[23] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.