An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors

Jack Sampson, Manish Arora, Nathan Goulding-Hotta, Ganesh Venkatesh, Jonathan Babb$^+$, Vikram Bhatt, Steven Swanson, and Michael Bedford Taylor
Department of Computer Science and Engineering
University of California, San Diego
$^+$Massachusetts Institute of Technology

***Abstract*— As the complexity of FPGA-based systems scales, the importance of efficiently handling irregular code increases. Recent work has proposed *Irregular Code Energy Reducers* (ICERs), a high-level synthesis approach for FPGAs that offers significant energy reduction for irregular code compared to a soft core processor. ICERs target the hot-spots of programs, and are seamlessly connected via a shared L1 cache with a soft processor that executes the cold code. This paper evaluates the application of the *selective depipelining* (SDP) technique to ICERs, which greatly reduces both the execution time and energy of irregular computations.**

SDP enables irregular computations to be expressed as large, fast, low-power combinational blocks. SDP maintains high memory bandwidth by scheduling the many potentially dependent memory operations within these blocks onto a high-frequency, highly-multiplexed coherent memory while scheduling combinational operations at a much lower frequency. SDP is a key enabler for improving the execution properties of irregular computations that are difficult to parallelize. We show that applying SDP to ICERs reduces energy-delay by 2.62× relative to ICERs. ICERs with SDP are up to 2.38× faster than a soft core processor and reduce energy consumption by up to 15.83× for a variety of irregular applications.

I. Introduction

FPGAs now play host to increasingly large-scale systems with complex and varied behaviors. To manage complexity, many designers are turning toward high-level synthesis (HLS) tools, which allow them to specify system behavior in a high-level language. A common approach incorporates application-specific hardware to execute the highly-parallel regions of code, coupled with a soft processor core to handle the remaining code. Although existing HLS tools can target highly structured, parallel code for acceleration, most are ill-suited for the remaining irregular, difficult-to-parallelize code. However, as systems continue to scale, the execution of the non-parallel regions on soft cores limits the efficiency of the system as a whole. Recent work [26], [20], [1] attained energy savings by converting even irregular code regions into specialized circuits.

One such approach [1] improves the energy efficiency of soft-core-based systems by converting hot regions of large, irregular C programs into a collection of energy-saving application-specialized coprocessors called *Irregular Code Energy Reducers*, or ICERs. Execution jumps between hot code, which runs on the ICERs, and cold code, which runs on the soft core. In both cases, memory operations are performed through a shared L1 cache, which eliminates the need for pointer analysis and enables high code coverage.

Although ICERs were able to achieve up to 9.5× savings in energy for targeted code at approximately the same level of performance, the energy and performance were limited by a fundamental tension. For low energy and high performance on non-memory operations, we would like large, low-power combinational blocks that chain together many operators with few registers and a slow clock. For high performance on memory operations, we would like an ultra-fast clock that can multiplex many memory operations onto a shared, pipelined, L1 cache. The ICER compromise was to pipeline the combinational portions of the circuit to bring its frequency up to the speed of the L1. At the same time, the L1 was run unpipelined in order to reduce complexity for cache miss handling. This led to less-than-optimal energy and performance for three key reasons: First, the interface to the memory system limited the window for exploiting instruction-level parallelism (ILP). Second, ICERs did not directly attack the large fraction of dynamic power consumed by the clock tree. Third, ICERs require many pipeline registers to make the circuit meet timing at the higher frequency: Although FPGA register resources are plentiful, unrestrained usage drives up energy overheads.

This paper addresses the three key limitations of ICERs by applying a technique called *selective depipelining*, or SDP, originally developed for patchable, ASIC-based coprocessors [20]. SDP converts each basic block in a program into a single macro operator and removes many pipeline registers entirely while retaining fast, highly-pipelined access to the L1 for memory operations. Synchronization between the pipelined and unpipelined portions of the circuits is enforced statically through the use of large numbers of automatically-generated multicycle path assertions for the CAD tools. This reduces clock power, increases memory parallelism, and extracts greater ILP. SDP scales to produce macro operators that can be very large: For the applications we examine, the macro operators cover up to 77 sub-operations including 23 memory requests. By applying this technique to ICERs, we can construct application-specific coprocessors that efficiently target large bodies of code with little parallelism and irregular memory behavior.

Compared to a soft core, selectively depipelined ICERs, or *SDP-ICERs*, improve, on average, performance by 2.0× and energy-delay product by 16.46× for targeted functions. When including the program regions that run on the soft processor, average application performance improves by 1.79× and application energy-delay product by 9.79×.

The rest of the paper is organized as follows. Section II provides an overview of ICER-based systems and context for selective depipelining. Section III describes SDP and its application to ICERs. Section IV evaluates the benefits of ICERs enhanced with SDP. Finally, Section V reviews related work, and Section VI concludes.

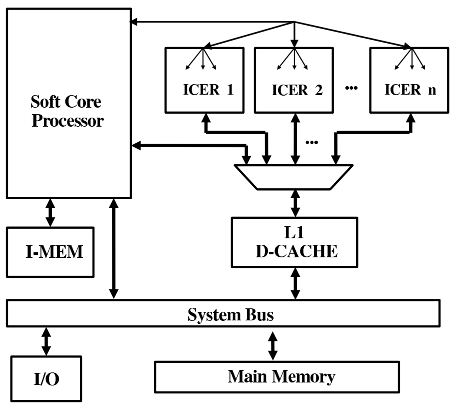


Fig. 1. **ICER-based system architecture** In an ICER-based system, a soft core processor controls one or more ICERs and executes any code not mapped into ICERs. The soft core and ICERs communicate primarily through the shared L1 cache. A narrow tree network carries control signals and provides the soft core access to each ICER's internal state.

II. ICER Architecture overview

To understand the impact of SDP, we first introduce the baseline ICER architecture to which SDP will be applied. ICERs are FPGA-based coprocessors generated by an automated toolchain [1] from the source code for frequently-executed regions of an application. Each ICER comprises a datapath and control unit. To construct the datapath, the toolchain converts each static operator in the input program into a dedicated functional unit. Dependencies between operators are converted into wires between these functional units. Registers distributed throughout the datapath store data needed across clock cycles. The control unit directs execution in the datapath and consists of a set of distributed state machines. These state machines correspond very closely to the control flow graph (CFG) of the original program to allow ICERs to precisely replicate the semantics of the soft core execution of the code. ICERs improve on soft core energy efficiency by eliminating many of the overheads present in conventional processor pipelines, including instruction fetch/decode, register file accesses, and generalized bypass networks.

Figure 1 shows a block diagram of a system built around a soft core and several ICERs. The soft core executes code not mapped into ICERs and handles dispatch to and from the ICERs. When program execution reaches a region covered by an ICER, the soft core transfers execution to the ICER and sleeps, waiting for the ICER to finish. ICERs target difficult-to-parallelize codes with difficult-to-analyze memory accesses. For these computations, coherent caches and a single shared address space are a sensible memory solution. Additionally, the cache serves as the on-chip communication mechanism among ICERs and with the soft core, avoiding the energy costs of off-chip communication through main memory.

Two key aspects of the ICER architecture are its integration with the soft core, and its compatibility with the soft core's view of memory. ICER memory operations complete in original program order, and ICERs stall during cache misses. Since the ICERs and soft core operate coherently on the same in-memory data, the system can fall back to running the code on the soft core at any time. This allows designers to port systems to smaller fabrics and eases debugging efforts. In addition to the shared data cache, the soft core also connects directly to the ICERs via a pipelined tree network. Every internal register in an ICER has an address in the tree. The soft core issues read and write commands to the tree to initialize ICERs and to retrieve state and restart them when handling exceptions.

III. Selective depipelining

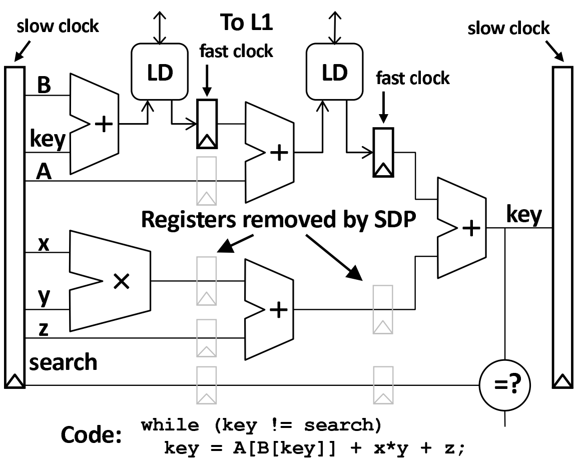
We enhance FPGA-based ICERs by applying *selective depipelining*, a technique originally developed for patchable, ASIC-based coprocessors [20]. SDP increases operator efficiency and reduces memory interface and clocking overheads. SDP is a pipelining scheme that improves performance and reduces datapath energy consumption in non-pipeline-parallel code, such as the irregular computations that ICERs target. SDP selectively eliminates pipeline registers from the datapath in order to reduce clock power and increase opportunities for combinational logic optimization. The technique works for blocks with many, potentially dependent, operations, and maps readily to both the FPGA environment and the ICER architecture. We call ICERs incorporating this technique *selectively depipelined ICERs*, or SDP-ICERs.

Selective depipelining takes advantage of the fact that memory and datapath sub-operations within a macro operation have different needs. As chip resources continue to scale, spatial computation [14] techniques that replicate datapath operators become progressively cheaper to apply. However, irregular computations continue to demand a centralized memory interface. SDP runs memory operations in a pipelined fashion synchronized to the frequency of the nearest level of memory while operating the datapath at a much lower frequency corresponding to one logical clock tick per basic block. Using separate logical clock frequencies effectively replicates the memory interface in time, while the datapath runs at a slower clock rate and without the overheads of pipeline registers present solely to synchronize the datapath and memory system on every clock cycle.

For every basic block in a program's CFG, SDP generates a macro operator. Then, it generates a control unit that operates on the memory clock (*fast clock*) and generates the pulsed datapath clock (*slow clock*) on each basic block transition. For each macro operator, there is one control state, which is further subdivided into *fast states* corresponding to fast-clock cycles. The number of fast states in a given control state is based on the number of memory operations in the macro operator and the latency of the critical combinational path through the sub-operations. This means that different basic blocks operate at different slow clock frequencies.

The execution of a macro operator begins with a slow clock pulse from the control unit. One macro operator executes for each pulse. During a slow clock cycle, only the control path and the currently-executing basic block are active. The pulse latches live-out data values from the previous macro operator and applies them as live-ins to the current macro operator. When the macro operator comprising the basic block completes, it triggers the next pulse of the slow clock and the execution of the next basic block.

During the execution of a macro operator, the control unit passes through fast states in sequence. Some fast states correspond to memory operations. Unlike datapath operators,



```
while (key != search)
  key = A[B[key]] + x*y + z;
```

Fig. 2. **An example of selective depipelining (SDP)** This diagram shows the ICER datapath for an example while loop. Under SDP, the highlighted registers are removed from the datapath, saving energy. Circuit performance is maintained through multicycle constraints.

which are scheduled according to combinational delay, SDP schedules memory accesses with respect to the fast clock for pipelining.

Pipelined loads and stores occur in two phases: request and response. When the datapath generates a new request, it sends it to the memory hierarchy and continues performing other operations in parallel. In each fast cycle, the datapath can both issue and retire operations at the width of the memory interface. SDP handles dependent memory operations by forcing memory operations to complete in order, but multiple outstanding requests can be in flight at any time. In the response phase, the control unit waits as needed for a load value or store confirmation. Fast clock registers save loaded values for use by dependent sub-operations. These are the only registers within a macro operator.

SDP example Figure 2 illustrates SDP over one basic block from an example while loop. The datapath contains arithmetic operators and load/store units for individual operations from the original program. The datapath can take multiple fast cycles to settle during one slow clock cycle, while memory operations proceed according to the fast clock.

As seen in the figure, SDP saves energy and resources by completely eliminating registers. This can be more effective than merely clock gating registers when not in use, because the clock accounts for a significant portion of dynamic power, and SDP completely eliminates some leaves from the clock tree. Spatial computation in the datapath eliminates the need for registers to time multiplex datapath operators and opens up opportunities for more aggressive combinational logic optimizations within the datapath.

By incorporating SDP, SDP-ICERs offer substantial improvements over ICERs. First, the focus of previous ICER work [1] was energy reduction, but SDP allows SDP-ICERs to both reduce energy and provide speedup for irregular code offloaded from the soft core. Second, the original ICER datapaths could contain at most one memory operation per control block, forcing basic blocks with multiple memory operations to be split. SDP-ICERs, on the other hand, can contain macro operators for arbitrary basic blocks, even those with multiple dependent memory operations. Using SDP, our toolchain built SDP-ICERs containing macro operations encompassing up to 77 operators including 23 memory requests. Below, we describe the toolchain and the synthesis process.

A. Building SDP-ICERs using SDP

We have integrated SDP into the ICER toolchain. Given a target code base, the toolchain first profiles the application and selects frequently-executed regions of code to convert into SDP-ICERs. The toolchain applies SDP and generates synthesizeable Verilog for the target regions. It also generates interface code that the application uses to access the new hardware.

Clocking Both our L1 data cache and all of our SDP-ICERs can run at 130 MHz or faster, whereas the soft core can only achieve 80 MHz on our target platform. In contrast to [20], which uses the processor clock as the fast clock, SDP-ICERs use the L1 data cache clock, while the soft core runs on its own clock. For simplicity, we clock both the L1 data cache and the SDP-ICERs at the frequency of the slowest SDP-ICER when any SDP-ICER is in operation, and downclock the L1 data cache to 80 MHz when the soft core is running. Without SDP, the slowest ICERs could only synthesize to 80 MHz, and share the clock with the soft core.

Many paths in a macro operator are purely combinational, and need only to complete within the minimum execution time, in fast cycles, that the toolchain assigns to the basic block. However, sub-operations that access memory are more constrained. The inputs to memory sub-operations must be ready sufficiently in advance of the fast clock boundary to issue the operations synchronously. Our toolchain enumerates all such multicycle constraints and propagates them to synthesis.

Memory SDP provides SDP-ICERs with in-order completion of memory requests that allows SDP-ICERs, like ICERs, to enforce the memory ordering semantics that imperative programming languages require. To reduce complexity and save power, SDP-ICERs issue and receive at most one memory operation per cycle, but they take advantage of the pipelined SDP memory interface to the L1 cache to support memory parallelism and improved performance.

Long-latency operations Some sub-operations in a macro operator, like memory requests, have long or variable latencies. In our implementation, these include integer division and floating point operations. SDP-ICERs handle these operations similarly to memory requests: The macro operator stalls in a specific fast state until it receives a valid signal from the corresponding functional unit.

Scheduling There are three key elements to scheduling for macro operators: estimating the length of the critical path through the entire macro operator, assigning memory operations to particular fast states, and enumerating all of the multicycle paths within the macro operator's datapath. If the constraint estimates closely match actual timing, then the CAD tools can more heavily optimize for power without sacrificing performance.

Non-memory sub-operators in an SDP-ICER chain together, which complicates the estimation of delay through any given sub-operator: Depending on the sequence of sub-operations, bit-level parallelism may allow the incremental cost of a given

Workload	Description	# SDP-ICERs	Coverage (%)	Freq. (MHz)	# Slices	Slices vs. ICERs	Slice Regs. vs. ICERS	Clk. Energy vs. ICERs	DSPs
b-tree [5]	Search tree traversal	1	86	231	1308	-14%	-13%	-36%	0
bzip2 [23]	Data compression algorithm	1	73	143	6172	-23%	-33%	-42%	0
grapht [5]	Sparse graph depth-first traversal	1	98	201	998	-18%	-18%	-28%	3
mcf [23]	Single-depot vehicle scheduling	2	42	193	3239	-11%	-9%	-31%	0
radix [28]	Sorting algorithm	1	94	130	3934	-15%	-24%	-51%	5
viterbi [7]	Convolutional code decoder	1	99	176	4583	-10%	-31%	-52%	0

TABLE I
WORKLOAD DATA ARE SHOWN FOR SEVEN SDP-ICERS REACHING FREQUENCIES OF 130 MHZ OR GREATER FOR SIX IRREGULAR APPLICATIONS.

sub-operation to be much less than its delay in isolation. The operation scheduler approximates delay reductions due to bit-level parallelism through a pre-computed lookup table of all sequences of two dependent sub-operators.

To determine how many fast states a control state will contain, and to schedule memory operations into particular fast states, an operation scheduler first estimates the completion time of each sub-operator under a contention-free resource model to determine criticality. The operation scheduler then assigns memory operations to the earliest fast states in which their inputs will be ready. Since SDP-ICERs issue up to one memory request per fast cycle, multiple memory operations ready in the same fast state are scheduled by criticality. The schedule produced assumes all memory operations will hit in the L1 cache.

IV. EVALUATING THE IMPACT OF SDP

In this section we describe our methodology and workloads, and we evaluate the impact of SDP on SDP-ICER efficiency, performance, and energy-delay product (EDP).

Synthesis, simulation, and power measurement The SDP-ICER toolchain relies on the OpenIMPACT (1.0rc4) [16] and LLVM (2.4) [10] compiler infrastructures. It converts arbitrary C programs into synthesizeable Verilog and modules for our cycle-accurate simulator. The toolchain uses Synopsys Synplify (D-2010.03) for synthesis and the Xilinx toolflow to translate, map, and place and route the SDP-ICER Verilog onto the Virtex 5 family of FPGAs. The specific device used is xc5vlx110t-ff1136-3.

Like previous ICER work, we use an energy-efficient, pipelined MIPS processor derived from the MIT Raw [13] processor as our soft core. The soft core operates within 20% of the dynamic power of a resource-equivalent MicroBlaze, with better instruction throughput. We use the cycle-accurate simulator modules produced by the toolchain to measure SDP-ICER performance compared to the soft core. We use traces from the simulator to measure power by periodically sampling execution, recording all SDP-ICER control and memory interactions. We then generate a Verilog test harness to drive the SDP-ICER in the Synopsys VCS (C-2009.06) logic simulator. This produces an activity file which is used together with the post-place-and-route designs to measure power consumption with the Xilinx XPower tool. A similar process generates power numbers for the soft core using samples from portions of software execution.

Benchmarks We used our toolchain to automatically generate seven SDP-ICERs for six applications. Table I describes each benchmark and shows the following: the fraction of dynamic execution covered by SDP-ICERs, the minimum achievable frequency across all SDP-ICERs that the application uses, and the FPGA resources each set of SDP-ICERs requires. Even the SDP-ICERs with the longest critical paths can run at significantly higher frequencies than the soft core. For all applications, our automated toolchain achieved at least 42% execution coverage, and 82% on average.

Results Table I shows how SDP reduces resource usage compared to ICERs. SDP reduces register counts by up to 33%, eliminating many leaves from the clock tree. SDP improves the average frequency of ICERs by 32%, and all run at 130 MHz or higher. Clock energy is reduced by up to 52%. An additional benefit of removing registers is that it improves logic optimizations across operators, further increasing energy improvements.

Figure 3 shows the EDP improvement (a-b), speedup (c-d), and energy breakdown (e-f) for ICERs and SDP-ICERs, normalized to running the code entirely on the soft core. Figure 3(a) shows that SDP-ICERs improve EDP by up to 34.76× (16.46× on average) for the specific code that they target. Figure 3(b) shows results that include the rest of the application, which runs on the soft core; even here, the SDP-ICERs improve EDP by 9.79× on average. EDP improvement for an application is largely dependent on execution coverage: The more time spent running on the SDP-ICERs, the greater the speedup and lower the energy.

EDP improvement comes from faster execution and reduced energy. Figure 3(c) shows that whereas ICERs maintain performance of the soft core, SDP-ICERs achieve a 2.0× speedup on average. SDP enables a faster clock frequency, greater memory parallelism, and more compact scheduling, producing significant speedups while saving clock energy by removing registers.

Finally, Figures 3(e-f) show a breakdown of energy use in the system. For the soft core, ICERs, and SDP-ICERs alike (Figure 3(f)), clock is the largest contributor to energy. However, with SDP, even though SDP-ICERs run at a faster frequency than ICERs, the savings from register removal and decreased run-time allow SDP-ICERs to reduce clock energy relative to ICERs.

V. RELATED WORK

In this section we discuss trends in other high-level synthesis frameworks and techniques related to selective depipelining. We also review other efforts that augment a general-purpose processor with coprocessors on a reconfigurable fabric.

High-level synthesis High-level synthesis is a very active area featuring a wide variety of commercial tools [6]. Optimization of multi-cycle operations is an ongoing research

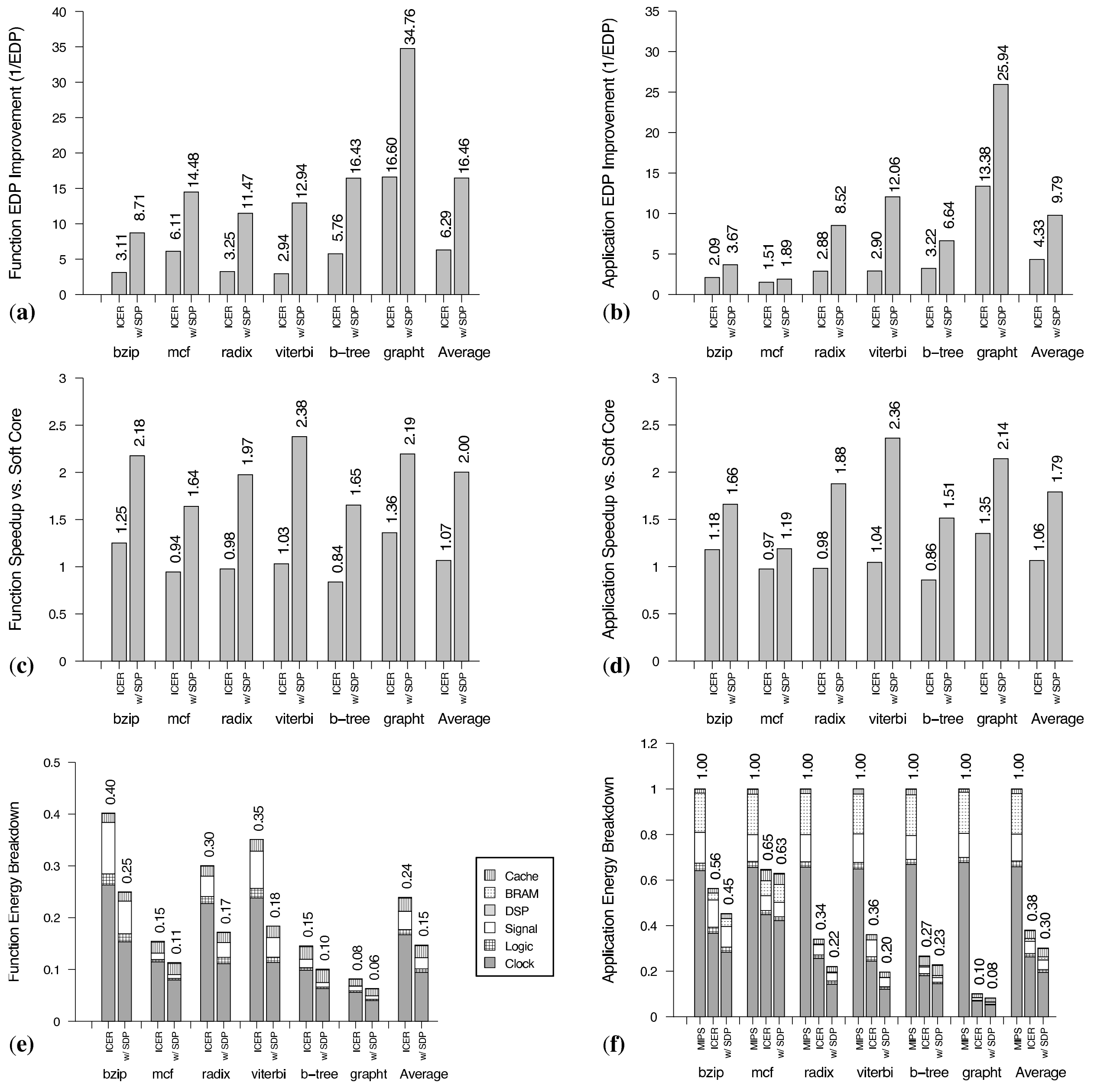


Fig. 3. **Benefits of applying selective depipelining to ICERs** SDP-ICERs offer energy-delay improvement (a-b, higher is better), speedup (c-d, higher is better), and energy reduction (e-f, lower is better) over both ICERs and a soft core for targeted functions (a,c,e) and whole applications (b,d,f). All results are normalized to running entirely on the soft core. Clock power is the single largest energy component, and SDP-ICERs save considerable energy over ICERs by using SDP to eliminate registers.

topic in high-level synthesis, including techniques to reduce resource utilization in special cases, such as [15]; these techniques are synergistic with our approach and can be applied during the synthesis stage.

Frameworks such as AutoESL [30], Impulse C [8], Synopsys Synphony/PICO [19], CHiMPS [18], and Altera C2H [11] build accelerators directly from high-level language source code. Creating high-performance accelerators with minimal effort has been the primary goal for most of these tools. Because they must either infer parallel execution from serial code or force the programmer to rewrite code in a more explicitly-parallel language or dialect [24], they tend to face the same challenges as parallelizing compilers. To overcome these challenges, a common tradeoff in existing tools is to compromise on the classes of codes amenable to automation and backward compatibility with the processor. In contrast, SDP allows coprocessors to achieve both energy savings and acceleration even for codes that are difficult to parallelize. We have applied SDP to an automated toolchain that targets arbitrary code and produces drop-in replacement hardware.

Selective depipelining SDP [20] is one of many techniques that manipulate operators and registers to create macro operators and expose bit-level parallelism between datapath operations. Retiming [12] techniques are common back-end optimizations in FPGA synthesis tools that move registers across combinational logic to optimize timing. SDP differs in both goals and implementation. SDP uses compiler-level information to maximize throughput to a highly-multiplexed memory while minimizing energy in the combinational portions of the circuit. It reduces clock tree energy by completely eliminating many registers. SDP works within the context of existing synthesis tools, and requires only support for multicycle paths. SDP is related to microarchitecture techniques to reduce communication for transient operands, allowing processors to construct complex operations from simple operations [21], as well as macro-op scheduling [9], which transforms a series of instructions into a multi-cycle scheduling unit. Other related techniques reduce register file accesses [17] or perform architectural retiming [22]. Rather than increasing operator size, the work in [27] proposed increasing pipelining to reduce glitching power. However, in our experiments, clock tree power was the dominant component, so removing registers was more profitable than reducing glitch behavior.

In many of these techniques, memory operations are not included in the macro operations, and operators are rescheduled across just one or two cycles. SDP works more aggressively by eliminating many of the pipeline registers between many dependent datapath components, which directly reduces clock energy. SDP can construct macro operators with dozens of sub-operations, including multiple, dependent memory operations. Compared to other techniques, SDP is a particularly good fit for the ICER architecture and an FPGA platform because it can be automatically applied to irregular computations, is compatible with caching, and directly addresses both memory timing and clock energy overheads. Furthermore, SDP leaves memory to run fully pipelined by applying chaining only to arithmetic operators.

Reconfigurable substrates Many previous projects, such as GARP [4] and Chimaera [29], have examined the benefits of augmenting a general-purpose processor with accelerators built in a reconfigurable fabric. In contrast to the application of instruction-set extensions [2] to FPGA soft cores [3], SDP-ICERs move much larger regions of code into low-power, high-performance custom logic while minimizing communication with the soft core itself. The work on Warp [25] examines optimizations for on-the-fly synthesis by performing dynamic translation of binaries into reconfigurable hardware. However, Warp employs an additional soft core to run the high-performance synthesis. The implications of mapping entire programs onto a large hierarchical asynchronous reconfigurable fabric have been examined in Tartan [14]. Recent work on ICERs [1] creates energy-efficient specialized processors for irregular applications. Whereas ICERs offered little performance improvement, SDP allows SDP-ICERs to both reduce energy and provide significant speed-up for irregular codes.

VI. Conclusion

We have evaluated applying the SDP technique to ICERs to improve both the energy efficiency and performance of irregular programs. SDP-ICERs provide a 2.62× EDP improvement over ICERs for targeted code. Relative to a soft core, SDP-ICERs improve the EDP of whole applications by 9.79×.

Acknowledgment

This research was funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

References

[1] M. Arora, J. Sampson, N. Goulding-Hotta, J. Babb, G. Venkatesh, M. B. Taylor, and S. Swanson. Reducing the energy cost of irregular code bases in soft processor systems. In *FCCM*, to appear 2011.

[2] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. of DAC*, pages 256–261, 2003.

[3] P. Biswas, S. Banerjee, P. Ienne, L. Pozzi, and N. Dutt. Performance and energy benefits of instruction set extensions in an FPGA soft core. In *Proc. 19th Int. Conf. VLSID*, pages 651–656, 2006.

[4] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, April 2000.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, second edition, 2001.

[6] P. Coussy and A. Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[7] Embedded Microprocessor Benchmark Consortium. EEMBC benchmark suite. http://www.eembc.org.

[8] Impulse C website. http://www.impulseaccelerated.com/.

[9] I. Kim and M. H. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *Proceedings of the International Symposium on Microarchitecture*, pages 277–288, 2003.

[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, 2006.

[12] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[13] M. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *ISCA*, 2004.

[14] 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.

[15] M. Molina, R. Ruiz-Sautua, J. Mendias, and R. Hermida. Area optimization of multi-cycle operators in high-level synthesis. In *DATE*, 2007.

[16] OpenIMPACT. http://gelato.uiuc.edu/.

[17] Y. Park, H. Park, and S. Mahlke. CGRA express: Accelerating execution using dynamic operation fusion. In *CASES*, 2009.

[18] 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.

[19] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *J. VLSI Signal Processing Systems*, June 2002.

[20] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor. Efficient complex operators for irregular codes. In *HPCA*, pages 491–502, February 2011.

[21] P. G. Sassone, D. S. Wills, and G. H. Loh. Static strands: Safely exposing dependence chains for increasing embedded power efficiency. *ACM Trans. Embed. Comput. Syst.*, 6, September 2007.

[22] M. Sivaraman and S. Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *CASES*, 2002.

[23] SPEC. SPEC CPU 2000 benchmark specifications, 2000.

[24] D. Unnikrishnan, J. Zhao, and R. Tessier. Application specific customization and scalability of soft multiprocessors. In *FCCM*, 2009.

[25] F. Vahid, G. Stitt, and R. Lysecky. Warp Processing: Dynamic Translation of Binaries to FPGA Circuits. *Computer*, July 2008.

[26] 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.

[27] S. Wilton, S.-S. Ang, and W. Luk. The impact of pipelining on energy per operation in field-programmable gate arrays. In *FPL*, 2004.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[29] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA*, pages 225–235, 2000.

[30] 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.