Efficient Complex Operators for Irregular Codes

Jack Sampson Ganesh Venkatesh Nathan Goulding-Hotta Saturnino Garcia
Steven Swanson Michael Bedford Taylor
Department of Computer Science & Engineering
University of California at San Diego
{jsampson,gvenkatesh,ngouldin,sat,swanson,mbtaylor}@cs.ucsd.edu

Abstract

Complex "fat operators" are important contributors to the efficiency of specialized hardware. This paper introduces two new techniques for constructing efficient fat operators featuring up to dozens of operations with arbitrary and irregular data and memory dependencies. These techniques focus on minimizing critical path length and load-use delay, which are key concerns for irregular computations. Selective Depipelining*(SDP) is a pipelining technique that allows fat operators containing several, possibly dependent, memory operations. SDP allows memory requests to operate at a faster clock rate than the datapath, saving power in the datapath and improving memory performance.* Cachelets *are small, customized, distributed L0 caches embedded in the datapath to reduce load-use latency.*

We apply these techniques to C*onservation Cores(c-cores) to produce coprocessors that accelerate irregular code regions while still providing superior energy efficiency. On average, these enhanced c-cores reduce EDP by 2× and area by 35% relative to c-cores. They are up to* $2.5\times$ *faster than a general-purpose processor and reduce energy consumption by up to* $8\times$ *for a variety of irregular applications including several SPECINT benchmarks.*

1 Introduction

Power limitations prevent modern processors from fully utilizing the large number of transistors that modern process technologies provide. Recent work [27, 5] has shown that the percentage of a silicon chip that can be switched at full frequency is dropping exponentially with each process generation, and will continue to drop with 3-D integration. This *utilization wall* forces designers to ensure that at any point in time, large fractions of their chips are effectively *dark* or *dim silicon* – that is, not actively used for computation, or significantly underclocked.

Simply reducing die area to avoid the creation of dark silicon has undesirable consequences. Doing so would freeze transistor budgets, effectively ending Moore's Law trend of increasing integration, thus stifling opportunities for innovation and increasing parallelism. In contrast, heterogeneity and specialization are effective responses to the utilization wall and the dark silicon problem. The utilization wall means that the opportunity cost of building specialized processors is falling: The silicon area that they consume would otherwise go unused.

Specialization is especially profitable in extremely power-constrained designs such as the mobile application processors that power the world's emerging computing platforms: cell phones, e-book readers, media players, and other portable devices. Mobile application processors differ from conventional laptop or desktop processors in that they have vastly lower power budgets and in that usage is heavily concentrated around a core collection of applications.

Mobile designers reduce power consumption, in part, by leveraging customized low-power hardware implementations of common functions such as audio and video decoders and 3G/4G radio processing. These computations are highly parallel, regularly structured, and very well-suited to traditional accelerator or custom ASIC implementations. The remaining code (user interface elements, application logic, operating system, etc.) resembles traditional desktop code and is ill-suited to conventional, parallelism-centric accelerator architectures. This code has traditionally been of limited importance, but the rising popularity of sophisticated mobile applications suggests this code will become more prominent and consume larger fractions of device power budgets. As a result, applying hardware specialization to frequently-executed irregular code regions will become a profitable system-level optimization.

Recent work [27] examined one approach to improving the energy efficiency of these codes by converting dark silicon into a collection of energy-saving application-specialized cores called *Conservation Cores*, or c-cores. That approach emphasized energy savings while matching the performance of a conventional processor, but three factors limited its performance and energy efficiency gains. First, synchronization with the memory system restricted

ILP, required large numbers of pipeline registers, and increased power consumption. Second, L1 cache accesses consumed significant energy and limited performance. Third, the mechanisms for adapting to software changes increased energy and area use significantly.

This paper introduces two techniques which, unlike many conventional architectural features, simultaneously improve both energy efficiency and performance. The first is a new pipeline design technique called *selective depipelining*(SDP), to reduce clock power, increase memory parallelism, and extract ILP by converting each basic block into a "fat operator". For the applications examined, these fat operators could be very complex, covering up to 103 sub-operations including 17 memory requests.

Second, we incorporate specialized energy-efficient, per-instruction data caches called *cachelets*, which allow for sub-cycle cache-coherent memory accesses. By applying these techniques to c-cores, we can construct application-specific coprocessors that efficiently target codes with little parallelism and irregular memory behavior. These techniques are fundamental to the design of the coprocessors in this paper, but can also apply to any architecture that uses fat operators, such as the "magic" instructions discussed in [15].

Additionally, we use workload profiling to reduce the costs of the reconfigurability mechanisms that allow c-cores to adapt to changes in the software they target. These techniques reduce EDP by 2× and area by 35% relative to prior work. Compared to an efficient, in-order MIPS processor, these enhanced c-cores improve, on average, performance by 1.5×for the function the c-core targets, application performance by 1.3×, targeted function energy-delay-product by 7.1×, and application energy-delay-product by 2.9×.

The rest of the paper proceeds as follows. Section 2 gives an overview of c-core-based architectures, and provides context for selective depipelining and cachelets. Sections 3 and 4 examine the techniques in detail. Section 5 reviews related work, and Section 6 concludes.

2 Architecture overview

In this section, we provide an overview of the Conservation Core [27] architectural model that we extend. Architectures based on specialized hardware must address three issues: 1) how a coprocessor's memory interfaces with host system memory, 2) how the system withstands changes to the software that it targets, and 3) how the coprocessor integrates with the host system's general-purpose processor(s). We address each question in turn.

2.1 C-cores

Figure 1(a) shows c-cores integrated into a tiled multicore architecture that connects multiple tiles and external memory via a point-to-point interconnection network. Each tile (see Figure 1(b)) contains a general purpose processor (the CPU) that is tightly coupled with multiple c-cores. The c-cores and the CPU share the tile's resources, including the coherent L1 data cache, the on-chip network interface (OCN), and the combined FPU/Multiplier unit.

The c-core toolchain automatically partitions a program between the CPU and the specialized hardware according to a cost model that estimates the benefit of running the code in dedicated silicon. It generates a hardware specification that it then synthesizes, places, and routes in 45 nm technology. Each c-core targets a frequently executed, or "hot", region of an application. They achieve energy and power savings by using specialized hardware datapaths that eliminate much of the overhead in conventional processor pipelines, including instruction fetch, register file accesses, and bypassing. Each c-core encompasses many basic blocks and executes one basic block at a time. C-cores clock-gate hardware not needed by the currently executing block. Control flow edges between blocks in a c-core are fixed, but can be arbitrary. A set of distributed state machines that closely mirror the control flow graph of the source program controls these datapaths, as shown in Figure 1(c). This mirroring allows for precise replication of the semantics of the CPU execution of the code. These datapaths communicate with the CPU and other tiles via connections to the shared L1 cache, also shown in Figure 1(d). The CPU executes code that is not mapped to a c-core. This includes parts of the application that occur infrequently or that post-date manufacturing of the chip. Execution shifts between the CPU and the c-cores as an application enters and exits the code regions that the c-cores support. Finally, c-cores also support patching, a form of reconfiguration, via a scan chain interface.

C-cores map readily to domain-specialized chips, but are also useful within general-purpose systems. The utilization wall leaves many transistors idle. Rather than underclock processor cores, or abandon increasing integration, we envision allocating the transistors to c-cores and other specialized hardware. Any commonly used applications or libraries (e.g. windowing systems, GUIs, codecs) are viable targets.

2.2 Selective depipelining and cachelets

We enhance c-cores by applying techniques that increase operator the efficiency. We call c-cores incorporating these techniques *Efficient Complex Operation Cores* (ECOcores). ECOcores are a significant extension of c-cores. C-core datapaths can contain at most one memory operation, whereas ECOcores can contain fat operators for arbitrary basic blocks, even those including several dependent memory operations. ECOcores also have a different focus than previous c-core work [27]: While both approaches reduce

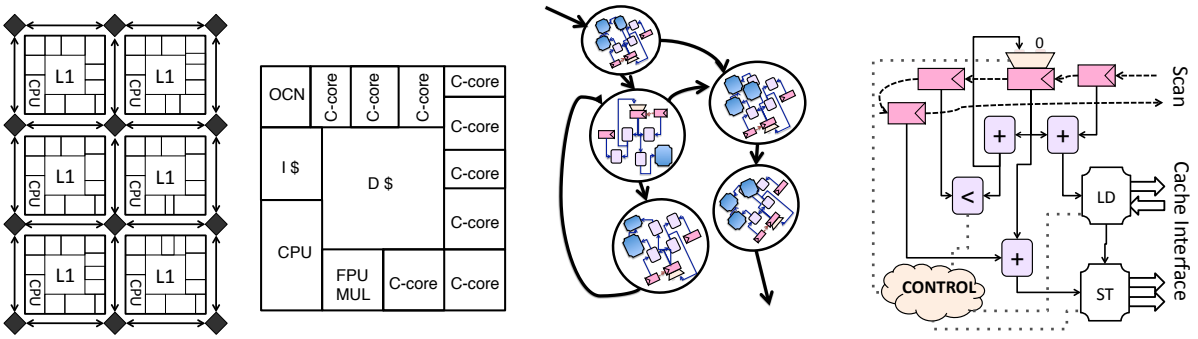

Figure 1. *A c-core-enabled, tiled architecture* **A tiled c-core-based system (a) includes several tiles (b) that each include a general-purpose, in-order processor and several c-cores (c) that target hot regions of code. Internally, c-cores will implement each basic block (d) using SDP (see Section 3).**

energy, ECOcores also aim to accelerate irregular codes, whereas c-cores offer minimal speedup.

ECOcores improve energy efficiency and performance over other systems designed to execute irregular code by leveraging two architectural techniques. The first technique, *selective depipelining*, is a novel pipelining scheme that significantly improves performance and reduces energy consumption by eliminating two important sources of waste in the generation of complex operators. It reduces both unnecessary clock power and time wasted due to poor alignment of operators within cycles. We show that we can pack dozens of operations, including multiple, dependent memory operations, into a single, efficient, logical clock cycle. We show that this technique works for blocks with many, potentially dependent, operations, with high performance, and without requiring asynchronous logic. The second technique, *cachelets*, is a new type of small, distributed, coherent L0 data cache that specializes individual loads and stores to reduce common case memory latency. In the following sections, we describe our two techniques in detail and highlight the unique challenges of irregular codes.

3 Selective depipelining

Selective depipelining, or SDP, takes advantage of the fact that memory and datapath sub-operations within a composite fat operation have different needs. Datapath operators are inexpensive to replicate, whereas the memory interface is inherently centralized. SDP bridges the gap between these disparate requirements. SDP allows memory to run at a much higher clock frequency than the datapath. The fast clock effectively replicates the memory interface *in time* (by exploiting pipeline parallelism), while the datapath runs at a slower clock rate, saving power and leveraging ILP by replicating computation resources *in space*. Using SDP, we have been able to efficiently construct fat operations encompassing up to 103 operators including 17 memory requests.

With SDP, ECOcores execute faster and consume less energy than a general-purpose processor, or even other special-purpose hardware such as [27]. SDP improves performance by enabling memory pipelining and exploiting ILP in the datapath. SDP reduces static and dynamic power because the datapath requires fewer pipeline registers and synthesis can use smaller, slower gates.

Datapath organization Under SDP, we organize datapath operators according to the basic blocks in a program's CFG, and one basic block executes for each pulse of the slow clock. During a slow clock cycle, only the control path and the currently executing basic block are active. The execution of a basic block begins with a slow clock pulse from the control unit. The pulse latches live-out data values from the previous basic block and applies them as live-ins to the current block. The next pulse of the slow clock, which will trigger the execution of the next basic block, will not occur until the entire basic block is complete.

For each basic block, there is one control state, and each state contains multiple substates called *fast states*. The number of fast states in a given control state is based on the number of memory operations in the block and the latency of the datapath operators. This means that different basic blocks operate at different slow clocks. During the execution of the basic block, the control unit passes through fast states in order. Some fast states correspond to memory operations. For these, the ECOcore sends out a load or store request to the memory hierarchy. The ECOcore also includes a register to receive the result of loads. Unlike the registers between basic blocks, these registers latch values on fast clock edges. These are the only registers within a basic block. The ECOcore remains in the fast state receiving from memory until the memory operation completes.

While most operations are scheduled at the basic block level, memory accesses and other long-latency operations

are scheduled with respect to the fast clock for pipelining.

Pipelined memory operations ECOcores enforce the memory ordering semantics that imperative programming languages require. ECOcores require in-order completion of memory requests to reduce complexity and save power, but they also pipeline the interface to support memory parallelism and improved performance.

Every load and store occurs in two steps: request and response. A request consists of an address and, for stores, the value to be stored. When the datapath generates a new request, the ECOcore sends it to the memory hierarchy and continues performing other operations in parallel.

In the response step, the control unit waits if necessary for the load value or store confirmation. Fast-clock registers save load values for use by dependent operators in the datapath. By splitting memory accesses into two phases, an ECOcore can initiate up to one memory request (load or store) and receive up to one memory response (load value or store confirmation) on every cycle. Memory operations complete in order, but multiple outstanding requests can be in flight at any time.

Long-latency operations In addition to memory operations, some non-memory operations (such as integer division and floating point operations) also have a long and/or variable latency. ECOcores handle these long-latency operations just like memory requests: They wait in a specific fast state for a valid signal from the corresponding functional unit.

SDP example Figure 2 illustrates SDP over one basic block. C source is shown at right, alongside a timing diagram, control flow graph (CFG) and datapath for the implementation of that code. The datapath contains arithmetic operators and load/store units for individual operations from the original program. The timing diagram shows how the datapath logic can take multiple fast cycles to settle while the datapath makes multiple memory requests.

The figure demonstrates how SDP saves energy and improves performance. In a traditional pipeline, the registers at fast clock boundaries would latch all the live values in the basic block. SDP is more effective than merely clock gating because it eliminates registers altogether, reducing latency, area and energy. It also eliminates many leaves from the clock tree, reducing clock tree area, capacitance and leakage. Eliminating registers also allows for more flexible scheduling of operations and removes the set-up, hold-time, and propagation delays that registers introduce. Also, having a very slow "slow clock" and only having one basic block active at a time enables an extremely aggressive clock-gating approach: In addition to leaf-level gating, we can gate all branches of the tree going to other basic blocks, and within the active basic block, each register will only be active once per dynamic execution.

3.1 Implementation

Since ECOcore-based chips will contain tens to hundreds of ECOcores, it is infeasible to select and design each ECOcore by hand. Instead, a toolchain automatically selects and synthesizes placed-and-routed ECOcores from a target code base. This section describes the toolchain and the synthesis process.

SDP implementation SDP relies on a fast clock for memory and a separate slow clock for the datapath of each basic block. The fast clock operates at the system frequency of 1.5 GHz. The slow clock signals come from the ECOcore's control unit, which tracks the flow execution through the ECOcore at basic block granularity.

Many signals in the basic block can safely take the entire minimum execution time to propagate through the block. However, the inputs to memory operations need to propagate more quickly because they must be ready on the fast clock boundary where the operation issues to memory. For instance, in Figure 2, the path from input `i` through the increment and compare can take up to eight fast clock cycles, while the path from `B` to the first load must complete in a single cycle. Similarly, the result of the third load has just 2 cycles to propagate to the store in fast state 1.8. Our toolchain generates these multi-cycle constraints and passes them to the synthesis toolchain.

Scheduling To generate multi-cycle constraints, an operation scheduler estimates the number of fast states each register-register, register-memory, and memory-register path within the basic block requires. If the scheduler is too conservative, the ECOcore will waste time in unnecessary fast cycles, resulting in slower performance. If the scheduler is too aggressive, the back-end CAD tools will not be able to meet timing requirements, causing the ECOcore to run at a slower clock frequency. Thus, the benefits of SDP are sensitive to the accuracy of the multi-cycle constraints.

To determine how many fast states a control state will contain, the operation scheduler calculates a minimum execution time for the block, in terms of fast clock cycles. This number is the maximum of the number of memory operations in the block and the critical path through the block divided by the fast clock period. For this calculation, the tool chain assumes that all memory operations will hit in the L1 cache. To achieve maximum performance, the ECOcore scheduler must accurately estimate the number of fast states required for the critical path through each basic block and assign memory operations to the earliest fast states in which their inputs will be ready.

The ECOcore approach to scheduling accounts for both widely varying operation latencies (from 10 ps for a NAND to over 1 ns for a multiply) and the degree to which bit-level parallelism in back-to-back operations can reduce the latency of a sequence of operations. For example, consider

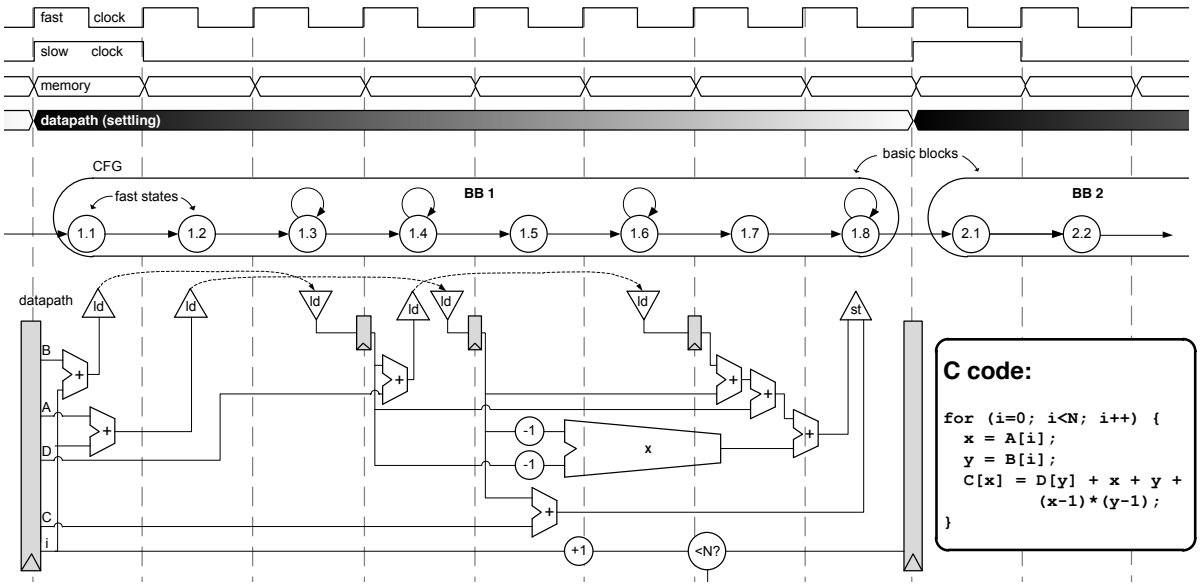

Figure 2. *Example datapath and timing diagram demonstrating SDP within one control state* **Under SDP, non-memory datapath operators chain freely within a basic block, while memory operators and associated receive registers align to fast clock boundaries.**

a multiply followed by an add. At 45 nm, a single 32-bit add takes approximately 0.31 ns, and a single 32-bit multiply takes 1.12 ns, resulting in a naïve estimate of 1.43 ns for the combined operation. However, after CAD tool optimizations the chained multiply-plus-add operation takes only 1.14 ns (a savings of 20%). A pre-computed lookup table of all sequences of two back-to-back operators approximates the effects of bit-level parallelism.

Patching ECOcores, like c-cores, are patchable. Analyzing the programs in Table 1 shows an opportunity to reduce the patching overheads present in [27]: In our workloads, 87% of all compile-time constants can be represented by 8 or fewer bits. Thus, we can use smaller configurable registers to represent constants with little risk of reducing generality. This allows us to reduce patching area and energy overheads in ECOcores by 43% and 70%, respectively, without significantly impacting our ability to adapt to software changes.

Synthesizing ECOcores The ECOcore toolchain extends the c-core [27] toolchain, and uses the OpenIMPACT (1.0rc4) [20], CodeSurfer (2.1p1) [7], and LLVM (2.4) [17] compiler infrastructures. It can process arbitrary C programs and automatically selects parts that are a good match for conversion into hardware.

Our toolchain generates synthesizeable Verilog and automatically processes the design in the Synopsys CAD tool flow, starting with netlist generation and continuing through placement, clock tree synthesis, routing, and post-route optimizations. For synthesis, we target a TSMC 45 nm GS process using Synopsys Design Compiler (C-2009.06-SP2) and IC Compiler (C-2009.06-SP2). We configure the tools to optimize for speed and power.

Simulation and power measurement We use a cycle-accurate simulator to measure ECOcore performance compared to a general-purpose MIPS processor without ECOcores. The toolchain automatically generates simulator models for ECOcores. The simulator measures power by periodically sampling execution, tracing the ECOcore's inputs and outputs. Traces drive the Synopsys VCS (C-2009.06) logic simulator and Synopsys PrimeTime (C-2009.06-SP2). PrimeTime computes static and dynamic power for each sampling period.

We derive processor and clock power values for other system components from specifications for a MIPS 24KE processor in a TSMC 45 nm process [18] and component ratios for Raw reported in [16]. We assume a MIPS core frequency of 1.5 GHz with 0.10 mW/MHz for CPU operation. We use CACTI 5.3 [29] for I- and D-cache power.

Modeling memory performance To quickly explore a wide range of memory architectures, we have developed an energy, performance, and area model for the ECOcore memory hierarchy. For large (>2KB) cache arrays, we use data from CACTI [29] for all three metrics. We also include extra wire delay for reaching the arrays based on our place-and-routed ECOcore designs. In Section 4, we explore the use of very small caches. We model these as arrays of latches and use values from measurements of arrays synthesized in our ASIC tool flow.

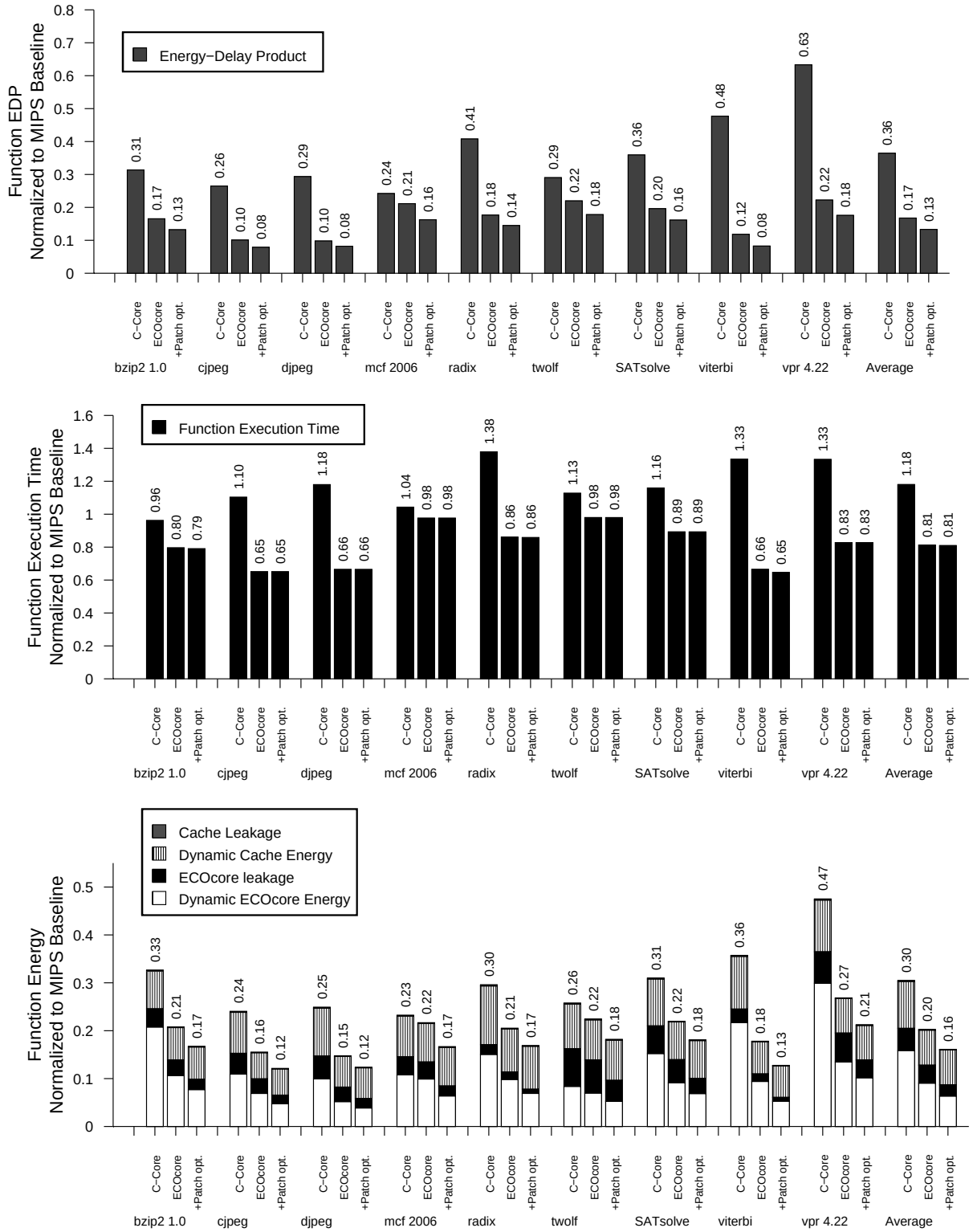

Figure 3. *ECOcore performance and efficiency* **Baseline ECOcores provide significantly better energy-delay (top), using SDP to achieve lower latency (middle), and energy usage (bottom) compared to c-cores.**

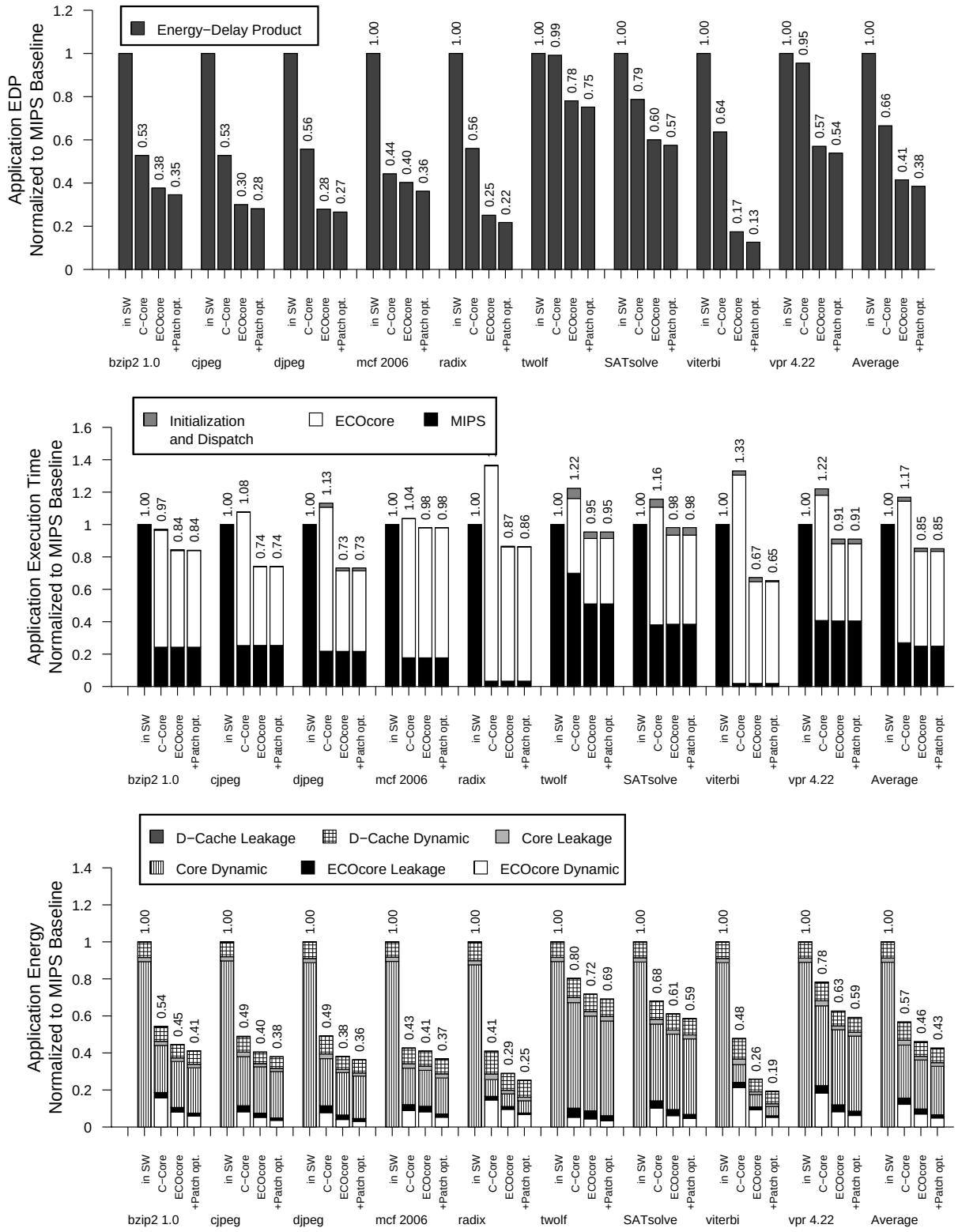

Figure 4. *Application performance and efficiency with ECOcores* **Energy-delay (top), application latency (middle), and energy (bottom) improvements from using ECOcores can be large, with latency reductions of up to 35% and average EDP reductions of more than 2×. The benefits of ECOcores increase with higher application coverage.**

Workload	Description	# ECOcores	Coverage %	Avg. Slow clock MHz	ECOcores Area mm^2	ECOcores +Patch Opt. Area mm^2
bzip2 [26]	Data compression algorithm	1	76	366.74	0.27	0.18
cjpeg [14]	JPEG image compression	3	75	116.73	0.31	0.18
djpeg [14]	JPEG image decompression	3	77	85.32	0.33	0.21
mcf [26]	Single-depot vehicle scheduling	3	82	302.41	0.28	0.17
radix [30]	Sorting algorithm	1	94	120.38	0.17	0.10
sat solver [**?**]	Stochastic local search SAT solver	2	66	215.20	0.30	0.20
twolf [26]	Placement & connection of transistors	4	49	252.20	0.20	0.13
viterbi [11]	Convolutional code decoder	1	98	259.07	0.22	0.12
vpr [26]	Place and route algorithm	1	61	684.93	0.37	0.23

Table 1. *ECOcore Workloads* **We built 19 ECOcores running at 1.5GHz for 9 irregular applications, covering the majority of execution. Patching optimizations significantly reduce area.**

3.2 Evaluating SDP

In this section we describe our workloads and evaluate the impact of SDP on ECOcore efficiency, performance, and energy-delay product.

Table 1 describes the nine applications for which we have created ECOcores. For each, our toolchain uses execution profiles to identify the most time-consuming functions and loop bodies in the application. The toolchain then applies aggressive function inlining and loop body outlining to isolate these portions of the program for conversion into ECOcores.

We evaluate SDP and its associated scheduling and logic optimizations compared to c-core and software implementations of our workload. Figure 3 shows energy-delay product (EDP), and its two components (execution time and energy), for the portions of the applications executed on ECOcores. We normalize results to the baseline single-issue low-power MIPS processor executing the same function. In addition to the baseline ECOcore design, we also present numbers for c-cores and an ECOcore with reduced patchability overheads ("+Patch Opt."). Since the ECOcore execution model is basic block based, benchmarks with larger basic blocks show greater improvements. We do not currently perform loop unrolling, but these results indicate it may be a fruitful optimization for ECOcores, at the expense of some additional area. The ECOcores not only outperform both the MIPS baseline and c-cores, but they are substantially more energy-efficient than c-cores. On average, the ECOcore baseline has a speedup of 1.27 relative to MIPS and 1.47 relative to c-cores. The baseline ECOcore reduces energy for covered execution by 80% over MIPS and by 33% over c-cores.

Figure 4 shows how these performance and efficiency gains are translated to the application level, where ECOcores offer an average EDP improvement of 59%.

4 Cachelets

Our measurements (see Figure 6) show that load-use latency, and equivalently, L1 hit time, in an ECOcore is a limiting factor for its performance. On average, L1 cache hits account for 30.8% of total time on the critical execution path for an ECOcore. Thus, reducing the load-use penalty should significantly improve ECOcore performance.

In conventional processors, all loads and stores go to a single cache since all load and store instructions execute on a small set of load/store functional units, but ECOcores can optimize load and store operations in isolation. ECOcores use small, very fast, distributed L0 caches called cachelets to reduce average memory latency. Cachelets provide sub-cycle load-use latency, 6× faster than the L1. Cachelets contain one to four cache lines and are tightly integrated into the ECOcore data path. Each ECOcore may have several cachelets. Each cachelet serves a fixed subset of these static operations, all of whose accesses go through the cachelet. Cachelets are fully coherent, and an inclusive L1 backs all lines in cachelets. Operations that have not been statically mapped to a cachelet communicate directly with the L1.

Both the MIPS and ECOcore baselines have a 3-cycle load-use latency to the L1. The small size and datapath integration of cachelets combine to offer hit times of half a cycle (based on synthesis results), reducing common case memory latency by 83%. Figure 5 shows how an ECOcore with cachelets communicates with the L1 cache and shows the internal structure of a cachelet. In the figure, two communicating memory operations share a single, one-line cachelet, while a third accesses the L1. Internally, cachelets share many similarities with small full-scale caches, such as tags, comparators, and word select muxes, but they use latches rather than SRAMs to store data.

Below, we present a simple coherence protocol for cachelets, explore alternatives for deciding what types of cachelets to instantiate, and evaluate their impact on performance and EDP.

Coherence The ECOcore execution model requires a coherent memory system, so the coherence protocol must extend to cachelets. In order to provide such low latency, cachelets must be distributed: Synthesis experiments showed that, for a single shared L0, multiplexing across

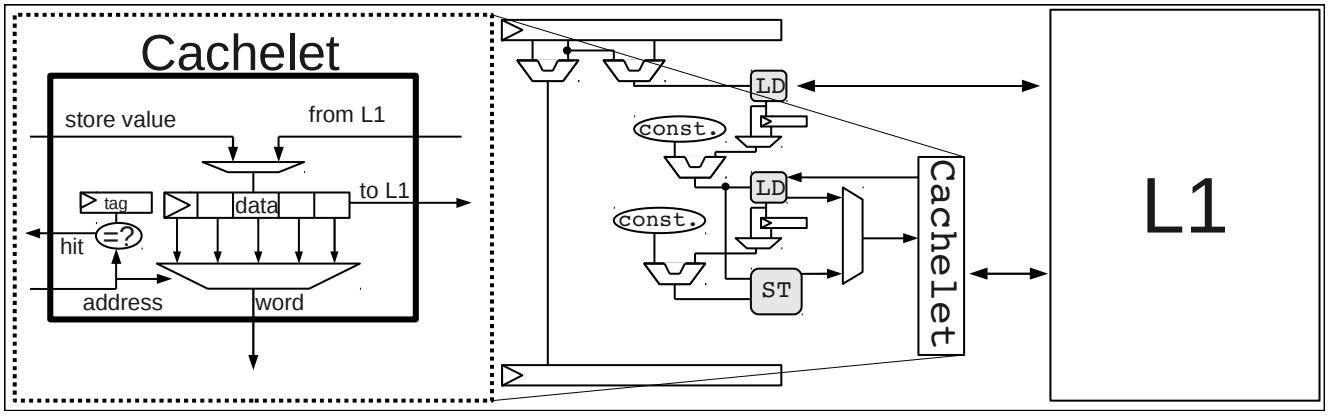

Figure 5. *Cachelet architecture* **With cachelets, memory operations with good locality will be mapped to local, low-latency memories while other operations continue to interface directly to the L1.**

all memory operations in an ECOcore would have higher latency than a cachelet access. Likewise, making each cachelet a full-fledged cache from the protocol's perspective is not practical because the coherence controller and state machines for the cachelet would be much larger than the cachelet itself. This, and the distributed nature of the cachelets, differentiate them from an L0 cache.

To provide cachelet coherence at minimal cost, we allow cachelets to "check out" cache lines from the shared L1 cache. To check out a cache line, the cachelet issues a fill request to the L1 cache. The L1 acquires exclusive access to the line and returns its contents to the cachelet. The cachelet now has exclusive access to the line. If another cachelet, the general-purpose core, or another processor in the system attempts to access that line, the L1 detects this and forcibly reclaims the line from the cachelet.

To perform a reclamation, the L1 freezes the ECOcore to prevent concurrent updates to the cachelet, copies the cachelet's contents back into the L1, invalidates the line in the cachelet, and completes the coherence request. The ECOcore can then continue execution, potentially reacquiring the line if it needs it again.

Since it requires halting ECOcore execution, eviction is a heavy-weight operation. We minimize costs through profiling and careful assignment of cachelets to memory operations (described below). Additionally, when an ECOcore finishes executing, the ECOcore implements a cachelet flush mechanism that writes back the contents of all dirty cachelets in the ECOcore and invalidates all lines in cachelets.

Cachelet selection Judicious assignment of cachelets to static memory operations is essential for good performance. Including too many cachelets increases ECOcore area requirements without significantly improving performance, whereas including too few limits performance gains. Likewise, we must avoid operation-to-cachelet mappings that would result in poor hit rates or frequent coherence traffic.

We have developed two strategies for selecting which cachelets to instantiate. The first strategy, called *private* performs an LRU-stack-based [4] cache simulation in which every memory operation has a dedicated cache. The simulation reveals how many lines the cachelet needs in order to significantly reduce the miss rate for that operation. The simulation includes coherence misses, so operations that share data with other memory operations are unlikely to receive a cachelet. The private strategy includes a cachelet if it would require fewer than 4 lines, and would have a hit rate of at least 66%.

The second strategy, called *shared*, analyzes the communication patterns and assigns a shared cachelet to communicating sets of memory operations. It forms transitive closures of communication operations within an ECOcore, partitioning operations into sets such that, during an invocation of an ECOcore no operation in one set accesses any line of memory that any operation in another set accesses. It uses the same LRU-stack analysis as in the private strategy to determine whether to include a cachelet and how big it should be.

Cachelet evaluation We measured the impact of adding cachelets to ECOcores using both strategies. On average, the private scheme produces 8.4 cachelets per ECOcore and shared produces 6.2. In the shared case, each cachelet served an average of 10.3 memory operations. No single ECOcore utilized more than 28 total lines of cache across its cachelets, and on average used fewer than 16 total lines. Area overheads for private and shared are 13.4% and 16.8%, respectively.

Figure 6 shows the impact of cachelets on ECOcore performance (top), application performance (middle), and application EDP (bottom). The first bar in each series depicts a baseline ECOcore without cachelets (the "+Patch Opt." bar from Figures 3 and 4), and the second and third bars

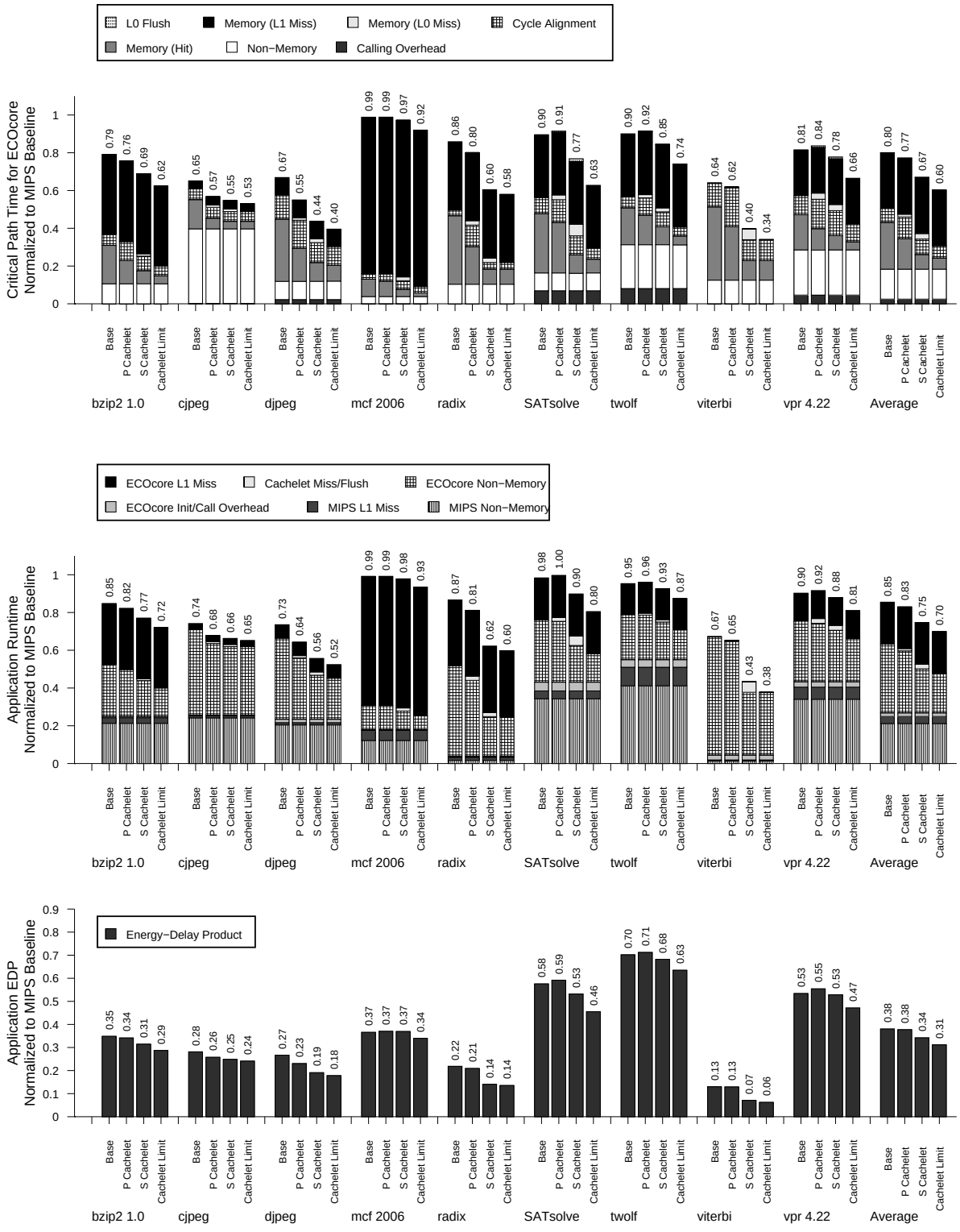

Figure 6. *Cachelet performance and efficiency* **The addition of cachelets greatly reduces latency and further improves EDP.**

present the private and shared strategies, respectively. The fourth bar shows results for a limit study for cachelet benefits assuming a 0.5-cycle, 32-KB L1. Both the private and shared cachelet approaches offer performance benefits, but the private strategy covers fewer critical memory operations, due to frequent communication between memory operations. The shared strategy realizes 66% of the performance potential seen in the limit study.

Adding cachelets to SDP reduces ECOcore latency by 13%, application latency by 10%, and application EDP by 4%. In total, the benefits of ECOcores with SDP, patching optimizations and cachelets provide average improvements for covered code of 7.1× in EDP and a speedup of 1.5×. At the application level, this translates to an average speedup of 1.33× and an average application EDP reduction of 66%.

5 Related work

Specialized coprocessors are a subject of increasing interest. Recent work has targeted accelerators for computations such as cryptography [31], signal processing [10, 13], vector processing [2, 8], physical simulation [1], and computer graphics [19, 3, 21]. Many of the ASIC-like accelerators [6, 12, 32] have focused on using modulo scheduling to exploit regular loop bodies that have ample loop parallelism and easy-to-analyze memory access patterns. Among these, the work in [12] and [32] design circuits with limited flexibility by incorporating limited programmability, or by merging multiple circuits into one, respectively. ECOcores differ in that they target the more general class of irregular, hard-to-parallelize computations that are not well-suited to modulo scheduling.

Conservation cores [27] are automatically-generated, application-specific hardware designed to improve application energy efficiency. While c-cores are very energy-efficient and offer a patching-based model for preserving longevity, previous work did not focus on performance, and offered minimal speedup. In contrast, ECOcores focus on both energy efficiency and performance, which both SDP and cachelets provide. ECOcores also improve upon the patching-based model for longevity, using bitwidth analysis on compile-time constants to reduce patching overheads.

Several designs have leveraged the bit-level parallelism that SDP exposes between datapath operations. The approach presented in [25] schedules multiple dependent operators back-to-back in the same cycle to help physical synthesis meet frequency targets. The approach in [22] uses the technique to reduce register file accesses for sequential code regions. Finally, the work in [9] moves datapath operators across pipeline registers to prevent short path-related false positive timing errors. These techniques reschedule operators across just one or two cycles. SDP applies this technique more aggressively, eliminating most pipeline registers between datapath components and can incorporate dozens of operations, including many memory operations, into a single fat operation spanning a single slow-clock cycle. Furthermore, SDP applies chaining only to arithmetic operators, leaving memory to run fully pipelined.

ECOcores provide a higher-performing and more-efficient memory system, with pipelined access and integrated cachelets. The CHiMPS multi-cache architecture [23] uses several application-specific caches and enforces coherence via flushing, but the purpose, sizing, and implementation of CHiMPS multi-cache differs from the cachelet approach. CHiMPS aggregates 4-KB block RAMs on an FPGA into caches backing different regions of memory in order to provide memory parallelism and to simplify the memory interface for a C-like programming model. In contrast, cachelets utilize small caches with between one and four lines that reduce the average hit time and access energy by eliding accesses to the L1.

Both the cachelet and SDP techniques apply broadly. SDP allows accelerators to greatly reduce clock energy and improve performance by implementing complex operators that include cache accesses. This approach can be used, for example, to generate the "magic" instructions discussed in [15]. Cachelets reduce the average cost of cache accesses to a fraction of L1 latency. Both custom datapath architectures that support caching, such as [28], and more conventional processors with static instruction based cluster-steering [24] can apply the cachelet technique.

6 Conclusion

We have presented ECOcores, an extension of c-cores that improve the performance and energy efficiency of irregular programs. ECOcores use two techniques to reduce energy consumption and improve performance compared to both a general purpose processor and existing work on similar specialized hardware. First, ECOcores use SDP to efficiently construct and clock complex operators capable of containing dependent memory references. Second, cachelets reduce L1 hit times while maintaining a coherent memory interface. Together, these techniques speed up the code they target by 1.5×, improve EDP by 7.1× and speed up the whole application by 1.33× on average, while reducing application energy-delay by 66%.

Acknowledgements

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] Ageia Technologies. PhysX by Ageia. http://www.ageia.com/pdf/ds_product_overview.pdf.

[2] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the Imagine Stream Architecture. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 14–25. IEEE Computer Society, 2004.

[3] ATI website. http://www.ati.com.

[4] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, pages 353–357, July 1975.

[5] K. Chakraborty. *Over-provisioned Multicore System*. PhD thesis, University of Wisconsin-Madison, 2008.

[6] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, Washington, DC, USA, 2008. IEEE Computer Society.

[7] CodeSurfer by GrammaTech, Inc. http://www.grammatech.com/products/codesurfer/.

[8] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. IEEE Computer Society, 2003.

[9] G. Dasika, S. Das, K. Fan, S. Mahlke, and D. Bull. Dvfs in loop accelerators using blades. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 894–897, New York, NY, USA, 2008. ACM.

[10] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - reconfigurable pipelined datapath. In *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135. Springer-Verlag, 1996.

[11] Embedded Microprocessor Benchmark Consortium. Eembc benchmark suite. http://www.eembc.org.

[12] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *HPCA: High Performance Computer Architecture.*, pages 313–322, Feb. 2009.

[13] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *ISCA '99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39. IEEE Computer Society, 1999.

[14] I. J. Group. Library for jpeg image compression. http://www.ijg.org/.

[15] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 37–47, New York, NY, USA, 2010. ACM.

[16] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *International Symposium on Low Power Electronics and Design*, San Diego, CA, USA, August 2003.

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75. IEEE Computer Society, 2004.

[18] MIPS Technologies. MIPS Technologies product page. http://www.mips.com/products/cores/32-64-bit-cores/mips32-24ke, 2010.

[19] nVidia website. http://www.nvidia.com.

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

[21] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, , and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[22] Y. Park, H. Park, and S. Mahlke. Cgra express: accelerating execution using dynamic operation fusion. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 271–280, New York, NY, USA, 2009. ACM.

[23] 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 '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 395–405, New York, NY, USA, 2009. ACM.

[24] S. S. Sastry, S. Palacharla, and J. E. Smith. Exploiting idle floating-point resources for integer execution. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 118–129, New York, NY, USA, 1998. ACM.

[25] M. Sivaraman and S. Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 35–42, New York, NY, USA, 2002. ACM.

[26] SPEC. SPEC CPU 2000 benchmark specifications, 2000. SPEC2000 Benchmark Release.

[27] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS 2010: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 205–218, New York, NY, USA, 2010. ACM.

[28] F.-J. Veredas, M. Scheppler, W. Moffat, and B. Mei. Custom implementation of the coarse-grained reconfigurable adres architecture for multimedia purposes. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 106 – 111, 24-26 2005.

[29] N. Wilton, S.J.E.; Jouppi. Cacti: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, May 1996.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[31] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 110–119. ACM Press, 2001.

[32] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *HPCA 15: High Performance Computer Architecture*, pages 277–288, Feb. 2009.