Composable Accelerator-rich Microprocessor Enhanced for Adaptivity and Longevity

Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, Glenn Reinman
UCLA Computer Science Department and Center for Domain-Specific Computing (CDSC)
{cong, ghodrat, mgill, bgrigori, huihuang, reinman}@cs.ucla.edu

Abstract

Accelerator-rich platforms demonstrate orders of magnitude improvement in performance and energy efficiency over software, yet they lack adaptivity to new algorithms and can see low accelerator utilization. To address these issues we propose CAMEL: Composable Accelerator-rich Microprocessor Enhanced for Longevity. CAMEL features programmable fabric (PF) to extend the use of ASIC composable accelerators in supporting algorithms that are beyond the scope of the baseline platform. Using a combination of hardware extensions and compiler support, we demonstrate on average 11.6X performance improvement and 13.9X energy savings across benchmarks that deviate from the original domain for our baseline platform.

Keywords

Hardware Accelerators, Accelerator Composition, Programmable Fabric, Adaptivity, Longevity

I. Introduction

Accelerator-rich designs have become more attractive in recent years, providing power-efficient performance through domain-specific specialization. Despite the dramatic improvement in performance and power-efficiency, accelerator-rich designs lack flexibility and longevity. Accelerators are typically designed for a particular algorithm or domain, and may have limited usefulness when new algorithms emerge within a domain, or when applied to an entirely different domain.

Instruction-based programmable accelerators [1], [2] are one approach to adding flexibility. Rather than optimize the accelerator for a single task, the accelerator executes a program using potentially domain-specific components. For example, while a GPU has been designed for graphics processing, it is flexible enough to be used in other domains that can leverage the stream-level parallelism that it exploits. However, as we will demonstrate, this programmability comes at a cost in power and performance efficiency compared to an application-specific accelerator design. Intuitively, if a design is general enough to run an application, it will be saddled with the overhead of processing these instructions: instruction decoding, architected register movement, etc.

Another alternative is to make use of programmable fabric (PF) – i.e. FPGAs – to implement customized accelerators for different tasks (e.g. [3], [4], [5]). This approach has nearly unlimited potential for longevity and flexibility, as accelerators are fluid: designers have the freedom to pick and choose the accelerators that are instantiated on the fabric, even creating new accelerators to adapt to algorithmic changes in a domain. However, as we will demonstrate, accelerators implemented in PF are considerably larger and slower than ASIC accelerators. It was shown in [6] that on average an FPGA implementation is 40X larger and 3.2X slower, with 12X higher dynamic power consumption. The inefficiency comes from the use of fine-grain programmable elements like LUTs to implement accelerator components and the high overhead of programmable interconnects. The area inefficiency can be particularly costly if one is constrained by a given size for the hardware; in this case, you may not be able to implement as many accelerators, or may resort to increased amounts of dynamic reprogramming, both of which negatively impact performance.

Yet another approach is to compose accelerators out of coarser-grain building blocks [7], [8], [9], [10]. In these designs, the recipe for composition is determined statically (i.e. at compile time), but the resource allocation is dynamic (i.e. at runtime). Since the programmability of the design is done at a coarse granularity (i.e. composing small accelerators), it does not have the same overhead as an instruction-based programmable accelerator, which must interpret an instruction stream. This design provides flexibility in that the building blocks may compose a variety of accelerators, as opposed to a monolithic accelerator design, which is restricted in its reusability. However, the design is still limited to the set of available building blocks that were originally provisioned. While this set can be optimized for a particular domain, the longevity of the design may suffer in the face of new algorithmic innovations or if the design is used for a domain that is different than that for which it was originally intended.

In this paper, we propose a hybrid approach that combines the performance of composable accelerators with the flexibility and longevity of PF-based accelerators. The PF will enable the instantiation of new building blocks, while the performance impact of the fabric will be mitigated by the fact that we still maintain a rich set of building blocks implemented in ASIC for the domain. Thus we will support composition of accelerators that are a mixture of ASIC and PF components. This approach provides a number of benefits. First, the programmable fabric serves as a design catch-all: we need not implement infrequently used building blocks for a domain in ASIC, as these blocks can be covered by the PF. This frees up silicon resources for more critical building blocks. Second, the programmable fabric can help adapt to domain or algorithm variance: we can more efficiently employ our design for domains/algorithms other than those originally intended by instantiating new building blocks in the PF and still making use of any useful building blocks that were already implemented in ASIC. Our contributions are as follows:

- **Compiler and Runtime Framework to Support ASIC and PF Allocation** - Our compilation framework generates a task flow graph of interconnected building blocks for a given kernel; it can also perform platform-aware partitioning of the task flow graph into subgraphs that can be accommodated by on-chip resources; at runtime, our resource manager uses these graphs to compose accelerators by allocating either ASIC- or PF-based building blocks.
- **Slack Analysis and Rate Matching** - Our compiler statically identifies imbalance in the task flow graph, and compensates for the computational slack in shorter paths by allocating extra buffer space; our hardware reduces PF performance overhead through *rate-matching*, where it instantiates multiple PF-based building blocks to collectively match the ASIC design throughput.
- **Design Space Exploration** - We demonstrate the enhanced flexibility from our approach through analysis on four distinct application domains, examining the

978-1-4799-1235-3/13/$31.00 ©2013 IEEE

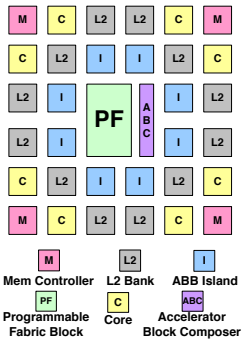

Fig. 1: CAMEL Microarchitecture

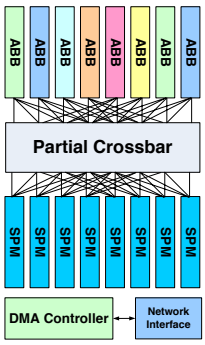

Fig. 2: Internal Structure of an ABB Island

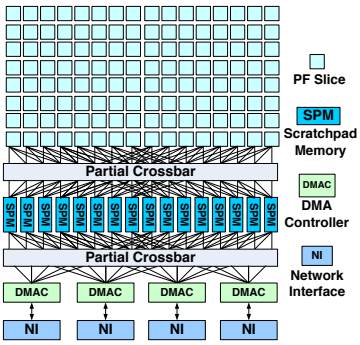

Fig. 3: Programmable Fabric

benefits our approach provides to design extensibility and longevity; while we analyze our results on one candidate architecture for accelerator composition, our techniques are more generally applicable to other composable architectures.

The rest of this paper is organized as follows: Section III introduces the CAMEL architecture, while Sections IV and V outline our evaluation methodology and results. We discuss prior work in Section II and conclude with Section VI.

II. Related Work

Prior art in both academia and industry has evaluated integrating accelerators and processing cores on a single chip. Garp [3], UltraSPARC T2 [11], Intel's Larrabee [12], and QsCores [13] feature designs where accelerators are tightly coupled with processing cores, or groups of cores. On the other hand, ARC [14] and IBM's WSP processor [15] are designed with looser coupling, both in terms of programmability and use. Our work focuses on loosely coupled accelerators that are shared among multiple cores and can be composed to form larger virtual accelerators.

Prior work has also examined the concept of constructing large and complex computational elements out of simpler structures. Examples of this are core fusion [16], core spilling [17], and TRIPS [18]. In those works, the goal of composition is to construct a mostly general-purpose processing element, while our work focuses on composing highly specialized structures that are capable of performance and efficiency beyond the capability of general-purpose cores.

Accelerator virtualization has likewise been studied by works such as VEAL [8], PPA [9], DySER [7], and CHARM [10]. VEAL proposes an architecture template for loop accelerators, along with a hybrid static-dynamic approach to mapping a loop to that structure. PPA features an array of processing elements which are configured to collaborate together. DySER proposes integrating a configurable accelerator into a core's execution engine to allow running programs to dynamically encode program regions into custom instructions. CHARM features hardware-based resource management, load-balancing, and accelerator virtualization via building blocks. These designs all lack support for virtualizing accelerators outside the scope of their available building blocks. While our experimental design driver is based on CHARM, the techniques and contributions of this work could essentially be integrated with any of these composable accelerator architectures to enhance their adaptivity and longevity. Furthermore, while our composable acceleration scheme could be implemented on a programmable SoC, such as Zynq from Xilinx [19], integrating PF into an ASIC-based accelerator architecture allows for added performance benefits from using the customized ASIC accelerators. We also advocate dynamic accelerator composition and management in hardware, which limits core interaction with accelerators, thereby removing it as a performance bottleneck.

III. CAMEL Architecture

The CAMEL architecture uses a combination of software and hardware components to improve flexibility and longevity. The hardware components are responsible for the actual accelerator composition, where the virtual accelerators, or loosely-coupled accelerators (LCAs), are dynamically constructed using either the available accelerator building blocks (ABBs) in ASIC or ABBs that have been instantiated in PF. While our contributions in the CAMEL architecture are generally applicable to composable architectures, in this paper we implement our techniques and analyze results on the CHARM architecture [10]. An overview of the CAMEL microarchitectural components is presented (not to scale) in Fig. 1. This figure consists of a set of cores with private L1 caches, shared L2 cache banks, and the following specialized CAMEL components: (1) ABBs grouped into a series of islands (shown as "I"); (2) Accelerator Block Composer (ABC) responsible for accelerator composition, PF assignment, and CAMEL resource arbitration; and (3) PF (for additional ABBs).

A. ABB Islands

Fig. 2 shows the internal structure of an ABB island; in this sample figure there are 8 ABBs, 8 Scratchpad Memory (SPM) banks, and 1 multi-channel DMA controller (DMAC). Each ABB has access to only 4 of the SPM banks using a partial 8x8 crossbar [20]. These SPMs are in turn connected to the multi-channel DMAC. The numbers and types of the ABBs are determined using software-driven design-space exploration, and the ABBs of a given type are distributed evenly across the islands in a round-robin fashion.

B. Programmable Fabric

The PF is used for hosting the ABBs required by new applications (in new or existing domains). The internal design of the PF in CAMEL is shown in Fig. 3. It consists of PF slices, 16 SPM banks, 4 DMACs, 4 network interfaces (NI), and 2 crossbars: one to connect a selected set of PF slices to SPMs and one to connect SPMs to DMACs. Although a monolithic PF presents challenges in its shared usage (i.e. ports, NoC congestion, etc.), it accommodates ABBs of any size and avoids performance hits due to static partitioning of resources. The main advantage of using a PF is its reusability and run-time reconfigurability. However, ABBs implemented on the PF are less area- and power-efficient, and have lower performance compared to ABBs implemented on ASIC. While the area and power issues are largely technology-dependent, we address energy consumption and performance using hardware techniques that compensate for the mismatch in computation speed.

When a virtual LCA is invoked, software sends to the ABC an encoded task flow graph representing the LCA's

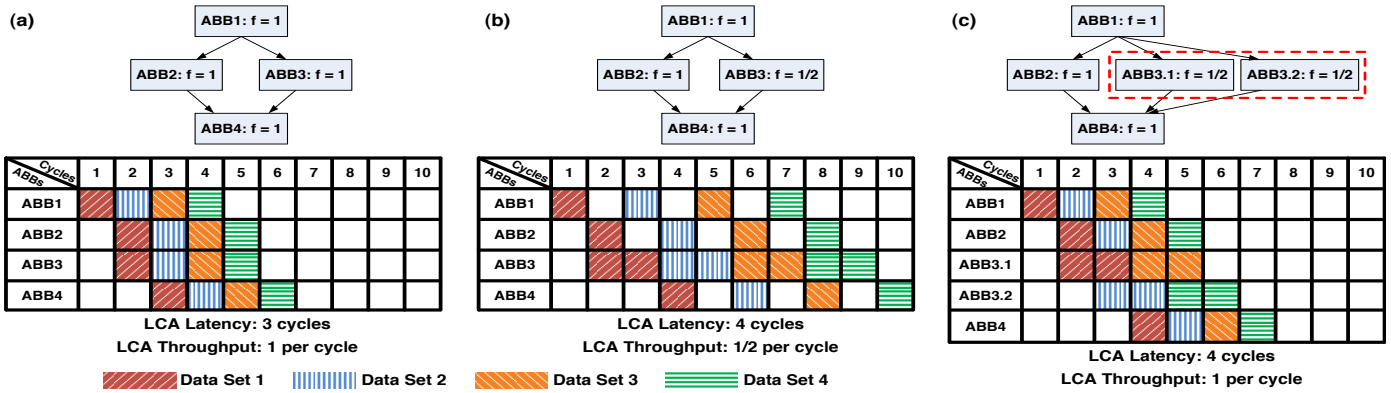

Fig. 4: Motivational Example of Applying Rate-Matching on PF

functionality. Nodes in this graph represent functionalities of individual ABBs, while edges represent data transfers. This functionality is executed in a pipelined fashion, with each ABB in the graph communicating with others by means of bulk transfers from/to its local SPM to/from remote SPMs or memory. If a PF-implemented (presumably less efficient) ABB is on the critical path, it can negatively impact the performance of the entire LCA. Fig. 4 exemplifies this scenario and how rate-matching helps. In this figure, the same task flow graph is instantiated for three different hardware allocation scenarios, and we see how four independent data sets (illustrated by different shading patterns) would flow through the connected ABBs. As Fig. 4-a shows, when all ABBs are operating at the same frequency (e.g. f = 1), the LCA they compose will have that same throughput. However, as shown in Fig. 4-b, if one of the ABBs is slower than the others (e.g. ABB3 has f = 1/2), this ABB becomes a bottleneck and the other ABBs are forced to stall. This results in the LCA as a whole progressing at the rate of this single slow component. Since the ABBs allocated in the PF typically have less throughput than ASIC ones, the inclusion of a PF-based ABB could result in such a bottleneck. To address this, CAMEL allocates multiple copies of the slower ABB to bring the aggregate throughput of the collection of slow ABBs up to match that of the faster ABBs. This is referred to as *rate-matching*, and is shown in Fig. 4-c. Provided there are sufficient PF resources for multiple ABB instantiations, this technique interleaves independent data sets between the duplicated PF-based ABBs and allows for the LCA to make more efficient use of the ASIC-based ABBs. As throughput is increased, the other ABBs and overall system components are left idle for a shorter period of time, thereby reducing static energy consumption. Although dynamic power is slightly increased, dynamic energy remains constant and so overall energy consumption is reduced. Thus, while rate matching not only improves performance and resource utilization, it also improves energy efficiency. The implementation of this technique is described in Section III-C.

C. Runtime PF Allocation

The ABC performs PF-based ABB allocation using the algorithm shown in Fig. 5. It receives information on the available space on the PF, along with the list of available ASIC ABBs and the LCA task flow graph. Using these it determines what ABBs to allocate in PF. To achieve the best allocation, it starts with the minimum configuration as a feasibility test; if the minimum currently cannot fit, it temporarily keeps the task until enough space is available on PF. If the minimum cannot be implemented at all, the ABC informs the requesting core of the failure to implement. If the feasibility check passes, the

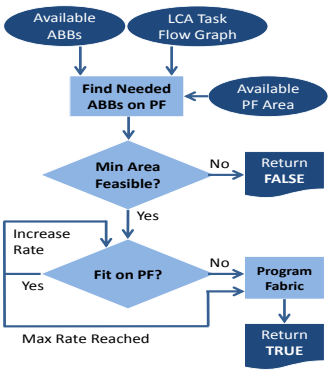

Fig. 5: PF Allocation Algorithm

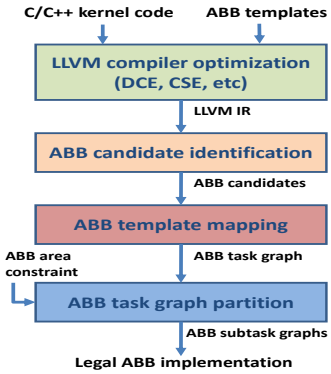

Fig. 6: Compiler Framework

ABC attempts rate-matching: it iteratively increases the PF-based allocation of critical ABBs (i.e. those on the critical path of the task flow graph) until either no space is left on the PF or the best rate-match is achieved.

D. Compiler Support

An overview of the CAMEL compiler framework is shown in Fig. 6. Given information on ABB types to potentially use, the compiler is responsible for mapping a given program kernel to a set of those ABB types, producing a data flow graph (i.e. task flow graph) whose nodes are ABBs and whose edges are data transfers. The algorithm used is similar to that described in [21]. Provided supplemental information on the available ASIC ABBs and PF for a given platform, the compiler can also determine if a kernel being mapped is too large for the total number of ASIC ABBs combined with the total PF. In these cases, the kernel's task flow graph is partitioned into the fewest number of regions such that allocation is possible. Partitioning is done along regions of the graph such as to minimize data transfer between partitions, and temporary storage is allocated to store intermediate data. The partitioned regions become subgraphs that can then be run sequentially. An example of this is shown in Section V-D. After a mapping solution exists, addressing for the local SPM of each ABB is calculated. Part of this calculation is an optimization for graphs that feature multiple paths of different lengths (i.e. slack) between a pair of nodes. Once this slack is identified, computational correctness is ensured by allocating extra buffer space along shorter paths. By avoiding stalls, this method allows for higher ABB utilization and overall throughput along all paths.

IV. Methodology

A. Tool Chain

In order to evaluate this architecture, we extended Simics [22] and GEMS [23] with the cycle-accurate models needed by CAMEL. Table I shows the simulation parameters used. We also implemented a complete tool-chain for generating

TABLE I: Simulation Parameters

Parameter	Value
Main Memory	Latency: 280 cycles, bandwidth: 10 B/cycle per controller
L2 Cache	8MB, 8-way set-associative, 32 banks, latency: 10 cycles
Coherence Protocol	Shared banked L2-cache, L2: MOSI, L1: MSI
Network Topology	4x8 MESH, latency: link 1 cycle & router 5 cycles, bandwidth: 72 B/cycle per link
ABB Islands (Base)	16 islands; 14 ABBs and 14 4KB SPMs per island

TABLE II: Tools for Timing and Power Models

Tool	Purpose
Xilinx Vivado Design Suite [19]	Accelerator high-level synthesis
Synopsys Design Compiler (32nm) [24]	ASIC synthesis (power, performance)
Xilinx ISE [19]	PF synthesis (performance)
Xilinx Virtex 6 XPower Estimator [19]	PF power analysis
CACTI [25]	Cache and scratchpad modeling
Orion [26]	NoC power and area
McPat [27]	Core and cache power analysis

simulator models starting from C-based kernel code. Table II shows the additional tools used for acquiring accurate timing and power values for these models. Furthermore, the compiler framework was implemented in LLVM, and has an average compilation time of 6.1 seconds per kernel for our benchmarks.

B. Domains

In this work, we target the four application domains described below. These four domains not only provide coverage of real-world applications with interesting computational demands, they also represent classes of applications that are algorithmically diverse in nature. Table III shows the numbers and types of ABBs used for accelerating each domain using one set of accelerators. Note that by *one set of accelerators* we mean as many ABBs as it would take to instantiate one of each virtual LCA in the domain. In our experiments, we have used four sets of accelerators.

1) Medical Imaging (Med): Medical imaging is an important tool for diagnosis and treatment. Because of the high volumes of data and high computational demands, the algorithms cannot be easily used in real-time clinical diagnosis, making them excellent candidates for acceleration. The medical imaging pipeline includes denoising, debluring, fluid registration, image segmentation, and compressive sensing for reconstruction. These algorithms and their acceleration strategies are described further in [28].

2) Commercial (Com): We have selected three applications from the PARSEC [29] suite to represent the commercial domain: BlackScholes, Streamcluster, and Swaptions. These applications solve partial differential equations, online clustering problems, and probability distribution estimations.

3) Vision (Vis): Computer vision is a compute-intensive domain with inherent parallelism that makes it ideal for streaming-data style of acceleration. Two main categories of applications in this domainare feature extraction, for which we include implementations of SURF from OpenCV [30] and LPCIP from MRPT [31], and image processing, for which we include the Texture Synthesis application from SD-VBS [32]. These applications provide a variety of computation including complex matrix-based, trigonometric, log-polar, and gradient histogram computations, with fluctuating memory usage.

4) Navigation (Nav): Navigation is a compute-intensive, AI-related domain that aims to achieve high levels of situational awareness. We include EKF-SLAM from MRPT [31], along with Robot Localization and Disparity Map from SD-VBS [32]. These applications provide diverse computation in the form of partial derivatives, covariance, spherical coordinates, probabilistic models, particle filters, search for minimal sum of absolute differences, etc.

C. ABB Characterization

The ASIC ABBs for our system have all been synthesized with a frequency of 1GHz and an initiation interval (II) of 1.

TABLE III: ABB Types, PF Synthesis, Domain Numbers, and Func.

ABB Type	FPGA Slices	Power (mW)	Freq (GHz)	ABBs per Domain				Functionality Description
				Med	Com	Vis	Nav	
poly	3536	571	1/4	47	95	143	167	16 I/O Polynomial (floating pt.)
sqrtf	672	176	1/3	2	1	1	3	Square root (floating pt.)
divf	255	84	1/3	6	5	6	7	Divide (floating pt.)
powf	672	176	1/3	1	3	1	0	Power function (floating pt.)
logf	672	176	1/3	0	3	0	0	Log base e (floating pt.)
rr1D	25	2	1/2	0	2	0	0	Random read in 1 dimension
rr2D	90	70	1/2	0	0	2	0	Random read in 2 dimension
rr3D	145	91	1/2	0	0	73	0	Random read in 3 dimension
rw1D	25	2	1/2	0	1	0	0	Random write in 1 dimension
selff	58	54	1/2	0	4	50	0	MUX (float inputs, float select)
selfi	57	54	1/2	0	3	4	0	MUX (float inputs, int select)
selif	27	84	1	0	4	8	0	MUX (int inputs, float select)
selii	30	85	1	0	1	0	0	MUX (int inputs, int select)
sum	134	77	1/2	0	1	8	1	Accumulate a vector
castfi	94	32	1/4	0	0	43	0	Cast float to integer
castif	108	35	1/4	0	0	1	0	Cast integer to float
mod	255	84	1/3	0	0	2	0	Modulo
min	65	54	1/2	0	0	3	0	Find minimum value in vector

TABLE IV: Power and Area for CAMEL Base Platform

Unit Type	Num. Units	Power per Unit (mW)	Area per Unit (um^2)
ABC	1	66.00	8383
poly ABB	188	6.65	362570
sqrtf ABB	8	9.49	368819
divf ABB	24	0.52	15117
powf ABB	4	9.49	368819
SPM	240	17.60	40773
DMAC	20	0.59	10071
L2 Bank *	32	148.97	881990
Core *	1	686.46	9868400
NoC *	1	4923.52	557978

* Power varies with execution; average value is shown.

Although the PF ABBs also have II's of 1, they have different operating frequencies depending on their type. Table III details the results of synthesizing the various ABB types for a Xilinx Virtex6 FPGA, along with the numbers of ABBs needed by the four domains and the functionalities of the ABBs. Note that the ABB granularities and functionalities have been determined according to a domain-space optimization primarily for Med, which is the base domain of CAMEL in our case studies (see Section IV-D), with additional ABB types added as needed.

D. Case Studies

For the purposes of this paper, we consider the cases of running single benchmarks, where accelerator needs are known. As such, PF reconfiguration can be done statically, and so reconfiguration time is excluded from all results. In our experiments we have considered the following cases, each representing a different class of accelerator-based architectures:

- **GPU** is a Tesla M2075; performance measurements consider computation only (not data transfer time).
- **LCA-ASIC** is based on an accelerator-rich platform where all LCAs are monolithic and ASIC-based [14].
- **LCA-FPGA** is based on an accelerator-rich platform where all LCAs are monolithic and FPGA-based [14].
- **CHARM** is based on a composable accelerator-rich platform with Med base domain and no PF [10].
- **CAMEL-x%** is the CAMEL architecture with Med base domain and "x" percent of the total ABB area substituted (by removing "x" percent of ABBs of each type, maintaining even ABB distribution across islands) for equivalent area of PF; x ranges 0%-50%.

The power and area values modeled for the CAMEL-0% base platform can be found in Table IV, where the total area of the chip is 122 mm^2. To determine the number of PF slices that can fit in CAMEL-x%, we have used the die area size of Virtex6 (measured by taking X-ray photos) and have estimated 2955 um^2 for each slice in 32nm. Table V shows numbers of PF slices and remaining ASIC-based ABBs for each CAMEL-x% case. Note that ABB types vary in both area and quantity – the distribution shown corresponds specifically to our platform. As PF slices are linearly increased for the CAMEL-x% cases,

TABLE V: Number of ABBs and PF Slices in CAMEL-x%

	CAMEL-0%	CAMEL-10%	CAMEL-20%	CAMEL-30%	CAMEL-40%	CAMEL-50%
# ABBs	224	192	168	148	128	108
# PF Slices	0	2466	4935	7404	9873	12342

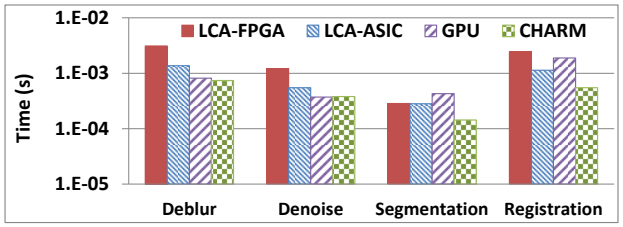

Fig. 7: Performance Comparison between Acceleration Schemes

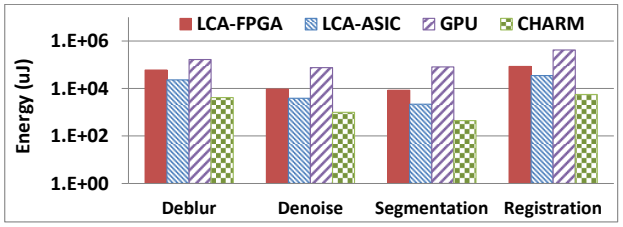

Fig. 8: Energy Usage Comparison between Acceleration Schemes

different numbers of various types of ABBs are removed to make room for the PF area, so the total number of remaining ABBs may not decrease linearly.

V. Results

In this section we present and discuss our simulated results. Although our Simics+GEMS framework simulates an Ultra-SPARC-III-i 1GHz processor (running Solaris 10), we conservatively measure our performance gains in terms of a wall-time-based comparison to fully parallelized runs on a 4-core 2GHz Intel Xeon E5405 processor. When there are insufficient accelerator resources to run a benchmark, we fall back to running on the CPU, and thus exhibit no benefit.

A. Comparison Between Acceleration Schemes

Fig. 7 and Figure 8 compare four accelerator-based architectures running benchmarks from the Med domain. As it features domain-specific acceleration, CHARM (i.e. CAMEL-0%) outperforms by 2.1X and saves energy by 93X compared to the power-hungry GPU. Furthermore, with its ability to load-balance and dynamically virtualize LCAs, CHARM on average outperforms LCA-FPGA by 3.5X and LCA-ASIC by 1.8X, resulting in energy savings of 14.5X and 5.1X, respectively. For an optimal design, we would want the performance and energy usage of CHARM with the adaptivity of GPUs and FPGAs. We show next how CHARM is made adaptive for greater performance and energy savings across domains.

B. Effect on Domain-Span

To evaluate CAMEL support of domain-span, we have used the Med base domain (for ASIC ABBs) and chosen three other target domains: Com, Vis, and Nav (as mentioned in Section IV). In all of these experiments, we have kept the overall area constant by removing 0%-50% of the ASIC ABB area in increments of 10% (maintaining even distributions of ABB types across islands) and adding PF slices equivalent to the removed area.

Fig. 9 shows the aggregate speedup and energy savings for all four domains, while Fig. 10 shows the average speedup and energy savings of each domain vs. software-only versions of the implementations. Since most of these new applications are unable to run on the base without the PF (i.e. exhibit 1X as they fall back to running on the CPU), the aggregate speedup of CAMEL-0% (i.e. CHARM) across all benchmarks is relatively low. As seen in Fig. 10-a, the Med applications, for which this

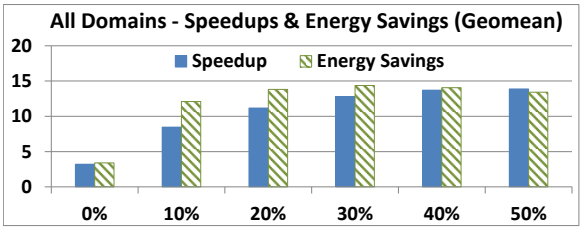

Fig. 9: Geometric Mean of All Speedups and Energy Savings

base was originally optimized, see performance improvement with the addition of a small amount of PF, followed by a decrease in performance as more PF is added. This is intuitively correct, as the platform considered was provisioned with the ASIC-based ABBs designed specifically for accelerating Med applications. A small amount of PF (10%-20%) provides adaptivity for higher load balancing and resource utilization for each individual benchmark, while larger amounts of PF begin to starve the system of the improved performance efficiency of the ASIC ABBs. However, even the small performance improvement initially seen with the addition of PF is not enough to counterbalance the reduction in power-efficiency as ASIC ABBs are replaced by PF. As a result, we see an initially small decrease in energy savings for CAMEL-10% and CAMEL-20%, followed by a larger decrease for CAMEL-30% and onward.

For Com (Fig. 10-b), no applications can be implemented without PF because they all require a variety of new ABB types that do not appear on the base platform (refer to Table III). As PF is added, these ABBs can be instantiated and rate-matched, resulting in large performance gains and energy savings. With Vis (Fig. 10-c), we see behavior similar to that of the Com applications. For Nav (Fig. 10-d), we see an initial speedup even without the PF because this domain shares a lot of the same ABBs as Med, allowing some benchmarks to be minimally implemented on the base platform. As we initially increase PF, we are able to instantiate the missing ABBs and run all benchmarks, resulting in increased average gains in both performance and energy. However, similar to the trends we see with CAMEL-10% and -20% for Med, as more ASIC is replaced by PF for Nav, performance continues improving slightly, yet energy savings begin dropping (e.g. CAMEL-30% and onward).

In summary, as ASIC ABBs are removed and replaced by PF, more useful ABBs become available and rate-matching takes effect. This translates into better adaptivity, and often times higher performance and energy savings for new domains. While these trends depend on the specific workload you are considering, as intuitively suspected, the less similar a workload is to the base domain of the platform, the more useful the PF. As with the law of diminishing returns, however, increasing the PF past a certain point starts reducing the improvements because the system is now removing too many of the useful ASIC ABBs and replacing them with their equivalent PF-based ones. We see this turning point with ∼30% PF for domains similar to the base (e.g. Nav) and ∼50% PF for other domains (e.g. Com and Vis).

C. Effect on Domain Longevity

In order to evaluate the longevity of the base domain, we have added a new application to Med: Compressive Sensing Magnetic Resonance (CS_MR) [33]. This application needs

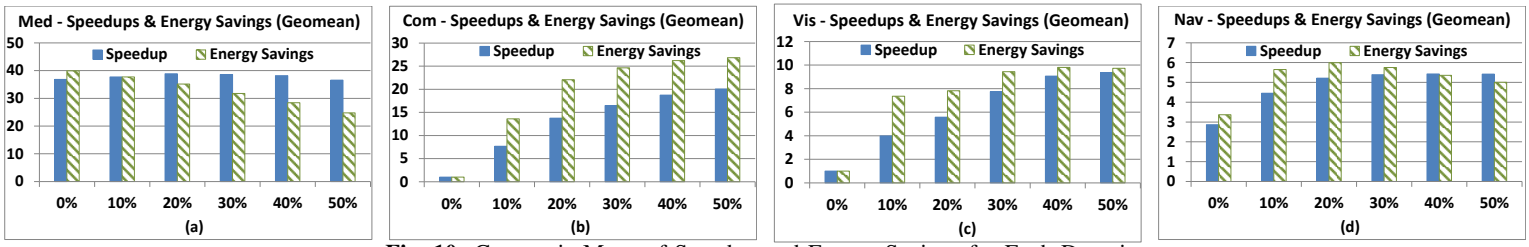

Fig. 10: Geometric Mean of Speedup and Energy Savings for Each Domain

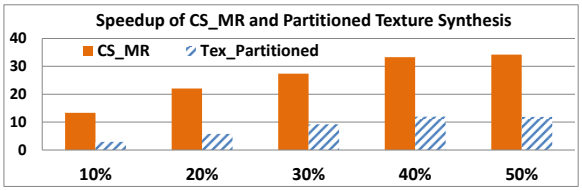

Fig. 11: Domain Longevity and Graph Partitioning

one additional ABB, namely the "sum" ABB, which is not found on the Med base domain of CAMEL. This "sum" ABB is one that accumulates the values of a given vector, and is used to implement the internal FFT engine of CS_MR. The speedup result for CS_MR is shown in Fig. 11. CS_MR does not need many of the ASIC-based ABBs on CAMEL, so as more PF slices are provided, it can use them to implement more "sum" ABBs, allowing it to instantiate more of its virtual LCAs and achieve more speedup.

D. Graph Partitioning for Lower-Capacity Hardware

As described in Section III-D, it is sometimes the case that a benchmark demands a massive LCA for a large kernel and requires more resources than are available on CAMEL, even with PF. Benchmarks like Texture Synthesis, Swaptions, Stream Clusters, and SURF contain kernels that can never be implemented in their original form. To overcome this, our compiler partitions the task flow graph of each of these kernels into a number of subgraphs that can each fit on CAMEL-x% (e.g. Texture Synthesis requires 6 partitions for CAMEL-50%). Fig. 11 shows the result of accelerating Texture Synthesis as an example after applying this graph partitioning technique, where we are able to achieve up to 11.96X speedup.

VI. Conclusion

In this work we have proposed CAMEL, a coordinated hardware-software approach for customized, adaptable acceleration. By incorporating PF into an ASIC composable accelerator platform, we add a new dimension of flexibility while leveraging existing ASIC performance benefits. The key characteristics of CAMEL include task graph partitioning, slack compensation, and automated rate-matching. With this approach we have shown that for three new domains, replacing 50% of the ASIC ABBs with PF achieves on average 11.6X performance improvement and 13.9X energy savings over using composable accelerators without PF.

VII. Acknowledgements

This research is supported by the CDSC funded by the NSF Expedition in Computing Award CCF-0926127, as well as the NSF Graduate Research Fellowship Grant # DGE-0707424. It is also supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] *Nvidia*, http://www.nvidia.com.

[2] *AMD APUs*, http://www.amd.com/us/products/technologies/apu.

[3] J. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *FCCM '97*, pp. 12–21.

[4] J. Cong *et al.*, "FPGA-based hardware acceleration of lithographic aerial image simulation," *ACM Trans. Reconf. Tech. Sys. '09*, pp. 17:1–17:29.

[5] ——, "Accelerating vision and navigation applications on a customizable platform," in *ASAP '11*, pp. 25–32.

[6] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *FPGA '06*, pp. 21–30.

[7] V. Govindaraju *et al.*, "Dynamically specialized datapaths for energy efficient computing," in *HPCA '11*, pp. 503–514.

[8] N. Clark *et al.*, "Veal: Virtualized execution accelerator for loops," in *ISCA '08*.

[9] H. Park *et al.*, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia application," in *MICRO '09*, pp. 370–380.

[10] J. Cong *et al.*, "Charm: a composable heterogeneous accelerator-rich microprocessor," in *ISLPED '12*, pp. 379–384.

[11] T. Johnson and U. Nawathe, "An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2)," in *ISPD '07*, pp. 2–2.

[12] L. Seiler *et al.*, "Larrabee: A many-core x86 architecture for visual computing," *IEEE Micro*, vol. 29, pp. 10–21, 2009.

[13] G. Venkatesh *et al.*, "QsCores: trading dark silicon for scalable energy efficiency with quasi-specific cores," in *MICRO '11*, pp. 163–174.

[14] J. Cong *et al.*, "Architecture support for accelerator-rich cmps," in *DAC '12*, pp. 843–849.

[15] H. Franke *et al.*, "Introduction to the wire-speed processor and architecture," *IBM J. of Research and Development*, pp. 3:1–3:11, 2010.

[16] E. Ipek *et al.*, "Core fusion: accommodating software diversity in chip multiprocessors," in *ISCA '07*, pp. 186–197.

[17] J. Cong *et al.*, "Accelerating sequential applications on cmps using core spilling," *IEEE Trans. on Par. and Dis. Systems*, pp. 1094–1107, 2007.

[18] M. Gebhart *et al.*, "An evaluation of the trips computer system," in *ASPLOS '09*.

[19] *Xilinx*, http://www.xilinx.com.

[20] J. Cong and B. Xiao, "Optimization of interconnects between accelerators and shared memories in dark silicon," in *ICCAD '13*, (To Appear).

[21] A. Hormati *et al.*, "Exploiting Narrow Accelerators with Data-Centric Subgraph Mapping," in *CGO*, 2007, pp. 341–353.

[22] P. S. Magnusson *et al.*, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, 2002.

[23] M. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," in *Computer Architecture New, Sep 2005*.

[24] *Synopsys Design Compiler*, http://www.synopsys.com/Tools/ Implementation/RTLSynthesis/Pages/default.aspx.

[25] *CACTI 5.3*, http://quid.hpl.hp.com:9081/cacti.

[26] H. Wang, X. Zhu, L.-S. Peh, and S. Malik, "Orion: A power-performance simulator for interconnection networks," in *MICRO '02*.

[27] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO '09*.

[28] A. Bui *et al.*, "Customizable domain-specific computing," *Design and Test of Computers, IEEE*, 2011.

[29] C. Bienia *et al.*, "Parsec 2.0: A new benchmark suite for chip multiprocessors," in *Workshop on Modeling, Benchmarking and Sim. '09*.

[30] *Open Source Computer Vision*, http://opencv.willowgarage.com/.

[31] *The Mobile Robot Programming Toolkit*, http://mrpt.org/.

[32] S. K. Venkata *et al.*, "SD-VBS: The san diego vision benchmark suite," in *IISWC '09*, pp. 55–64.

[33] M. Lustig *et al.*, "Sparse MRI: The application of compressed sensing for rapid MRI," *Magnetic Resonance in Med. '07*, pp. 1182–1195.