Quasi-ASICs: Trading Area for Energy by Exploiting Similarity in Synthesized Cores for Irregular Code

Ganesh Venkatesh Jack Sampson Nathan Goulding
Steven Swanson Michael Bedford Taylor

Department of Computer Science & Engineering
University of California, San Diego
{gvenkatesh,jsampson,ngouldin,swanson,mbtaylor}@cs.ucsd.edu

Abstract

The transistor density continues to increase exponentially, but the power dissipation per transistor improves only slightly with each generation of Moore's law. Given the constant chip-level power budgets, this exponentially decreases the fraction of the transistors that can be active simultaneously with each technology generation. Hence, while the area budget continues to increase exponentially, the power budget has become a first-order design constraint in current processors. In this regime, utilizing transistors to design specialized cores that optimize energy-per-computation becomes an effective approach to improve the system performance.

To trade transistors for energy efficiency in a scalable manner, we propose *quasi application-specific integrated circuits,* or QASICs, specialized processors capable of executing multiple *general-purpose* applications while providing an order-of-magnitude more energy efficiency than a general-purpose processor. The QASIC design flow is based on the insight that similar code-patterns exist across applications. Our approach seeks to exploit these similar code patterns to design specialized cores that can support many of the widely used computations.

Our results demonstrate that designing relatively few QASICs can support operator functions of multiple commonly used data structures and these QASICs provide 13.5× energy savings over a general-purpose processor. On a more diverse workload consisting of twelve applications selected from different application domains (including SPECINT, Sat Solver, Vision, EEMBC, among others), our results show that QASICs reduce the required number of application-specific circuits by over 50% and the area requirement by 23% compared to the fully-specialized logic while providing energy-efficiency within 1.27X of that of fully-specialized logic. Also, at system level, our approach reduces the application energy-delay metric by 46% compared to conventional processors.

1. Introduction

Transistor density continues to scale but nearly constant per-transistor power and fixed chip-level power budget places tight constraints on how much of a chip can be active at full frequency at one time. Hence, as the transistor density increases with each generation, so does the fraction of the chip that is under-clocked or underused − referred to as dark silicon − due to the power concerns. This *dark silicon* [3, 21] phenomenon is forcing designers to find new solutions to convert transistors into performance.

Recent work [4, 10, 21] has shown that specialization is an effective solution to this problem of utilizing the increasing transistor budget to extend performance scaling. The specialization-based approaches seeks to trade the cheaper resource, the dark silicon, for the more valuable resource, energy efficiency, to scale system performance. Specialized processors improve the system performance by optimizing per-computation power requirements and hence, allowing more computations to execute within a given power budget.

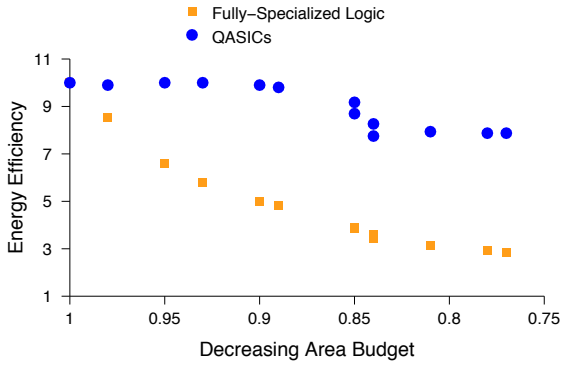

Figure 1. Trade offs between area and energy efficiency The x-axis measures area compared to implementing our workloads (Table 1) in fully-specialized logic. The y-axis measures energy consumption relative to an in-order MIPS processor. As area budgets decrease, QASICs energy efficiency declines much more slowly than it does for fully-specialized logic, because QASICs can save area by increasing the QASIC's computational power rather than removing functionality.

Conventional accelerators reduce power for regular applications containing ample data level parallelism, but recent work [21] has shown that specialized hardware can reduce power for irregular codes as well. These specialized computing elements can support relatively short running computations as well because they are placed close to the general-purpose processor. This increases the fraction of program execution that could execute on specialized, efficient hardware, but it also raises questions about area efficiency and re-usability, since each specialized core targets a very specific computation. In addition, supporting a large number of tasks in hardware would necessitate designing a large number of specialized cores, which in turn would make it difficult to place all of them close to the general-purpose processor. To effectively target a general-purpose system's workload, these specialized processors must be able to support multiple computations, and a relatively small number of these together should be able to support large fractions of execution of an application domain.

To trade area for specialization in a scalable manner, we propose a new class of specialized processors, *Quasi-ASICs*, that unlike the traditional ASICs which target one specific task, can sup-

port multiple general-purpose computations. These QASICs allow us to trade between area and energy efficiency in a fine-grained manner. The QASIC design flow accomplishes this by varying the required number of QASICs as well as their computational power based on the relative importance of the applications and the area budget available to optimize these applications. While the increase in the QASIC's computational power comes with marginal decrease in their energy efficiency, these QASICs are still an order of magnitude more energy-efficient than general-purpose processors. In this manner, our approach can significantly reduce the number of specialized processors as well the area budget required compared to that of fully-specialized logic, without compromising on the fraction of system execution that the specialized logic supports.

Figure 1 demonstrates that compared to the recently-proposed conservation core [21] approach, QASICs give up very little efficiency in return for substantial area savings. As the area budget decreases (left to right on the X-axis), our toolchain designs QASICs with greater computational power to ensure that the QASICs continue to cover all the application hotspots. This increase in the generality of QASICs enable them to provide significant energy efficiency even as the area budget decreases, unlike the fully-specialized logic that sees a 4× decrease in their energy efficiency.

In this paper, we address many of the challenges involved in designing a QASIC-enabled system. The first challenge lies in identifying similar code patterns across a wide range of general-purpose applications. The hotspots of a typical general-purpose application tend to have many hundreds of instructions, complex control-flow and irregular memory-access patterns, making the task of finding similarity both algorithmically challenging and computationally intensive. The second challenge lies in exploiting the similar code patterns to reduce hardware redundancy by designing generalized code-structures that can execute these code patterns. The third challenge involves making the area-energy tradeoffs to ensure that these co-processors fit within the area budget. Addressing this challenge entails finding efficient heuristics that approximate an exhaustive search of the design space but avoid the exponential cost of that search. The final challenge involves modifying the application code/binary appropriately to enable the applications to offload computations on to the QASICs at runtime.

We evaluate our toolchain by designing QASICs for the `find, insert, delete` operations of the commonly used data structures, namely link-list, binary tree, AA tree, and hash table. Our results show that designing just four QASICs can support all these data structure operations and can provide 13.5× energy savings over a general-purpose processor. On a more diverse general-purpose workload consisting of twelve applications selected from different application domains (including SPECINT, Sat Solver, Vision, EEMBC, among others), our results show that QASICs reduce the required number of application-specific circuits by over 50% and the area requirement by 23% compared to the fully-specialized logic while providing energy-efficiency within 1.27X of that of fully-specialized logic. Also, at system level, our approach reduces our workload's energy-delay metric by 46% compared to conventional processors.

The rest of this paper is organized as follows. Section 2 motivates our work in the context of other proposals. Section 3, 4, and 5 describes our QASIC design and hardware generation flow. Section 6 describes our methodology for evaluating the energy and performance efficiency of QASICs. Section 7 presents the results, Section 8 presents the related work and Section 9 concludes.

2. Motivation

In this section, we present some of the recent proposals on designing specialized co-processors and motivate the need for QASICs.

Benchmark Type	Application	HotSpots
Spec CPU 2000-2006	Twolf	new_dbox, new_dbox_a, newpos_a, newpos_b
	Mcf	refresh_potential primal_bea_mpp
	Bzip2	fullGtU
	LibQuantum	cnot, toffoli
Image Compression	CJPEG	ycc_rgb extractMCU
	DJPEG	jpeg_idct, rgb_ycc
Sat Solver	UBC Sat	BestLookAheadScore FlipTrackChangesFCL
Splash	Radix	slave_sort
SD VBS	Image pre-processing	calc_dX, calc_dY imageBlur
	Disparity	finalSAD, findDisparity integralImage2D
EEMBC Consumer	RGB/CMYK	CMYfunction
	RGB/YIQ	YIQfunction

Table 1. Diverse Application Set The table lists applications we use to evaluate the effectiveness of QASICs.

Also, we examine the similarity present across applications and describe how this similarity is exploited by our design methodology.

Specialized Processor Design Yehia et al. demonstrate techniques for automatically designing compound circuits that can support multiple regular loops in [24]. Their technique focussed on streaming loops, used an exploration algorithm to map one loop circuit onto another, and presented evolutionary techniques to enable merging of many loop circuits to form one compound circuit. However, their approach is not directly applicable to the general-purpose domain because the hotspots tend to be much larger, contains complex control constructs, and have irregular memory patterns. The hotspots typically consist of many hundreds of assembly instructions, significantly increasing the exploration space, and the irregular memory pattern is not suited for their streaming memory model.

Venkatesh et al. present techniques for generating application-specific hardware of the general class of irregular, hard-to-parallelize applications in [21]. However, for many applications in a system's target workload, it is not scalable to trade silicon for co-processors that support only one application. Ideally, to make this area-energy tradeoff more favorable, the co-processors should support multiple applications with similar control/data dependence.

To effectively utilize the increasing transistor density, we propose a design methodology that varies the amount of hardware generalization based on the area budget. Our design methodology provides increased generalization by exploiting similar control and data dependences present across applications.

Examining Application Similarity Our technique for reducing hardware redundancy is based on the insight that similar code patterns exist across applications. In this section, we quantify the available similarity and use this to motivate the QASIC design methodology.

To begin with, we examine the hotspots in the UBC Sat Solver [18] to give some insight on the kinds of similarity available. Figure 2 shows the source code for the two hotspots and highlights the similar code segments. The example shows that while the two hotspots as a whole do not have similar control-flow, there are similar code patterns present across them. Our design methodology seeks to exploit these similar code patterns to effectively tradeoff between energy efficiency and the area budget (Figure 1).

To quantify the similarity across applications, we examine a diverse set of applications from SPEC 2000 [15], Splash [23], EEMBC-consumer [7], UBC Sat [18], and SD-VBS [20] benchmark suites (described in Table 1). First, we profile the applica-

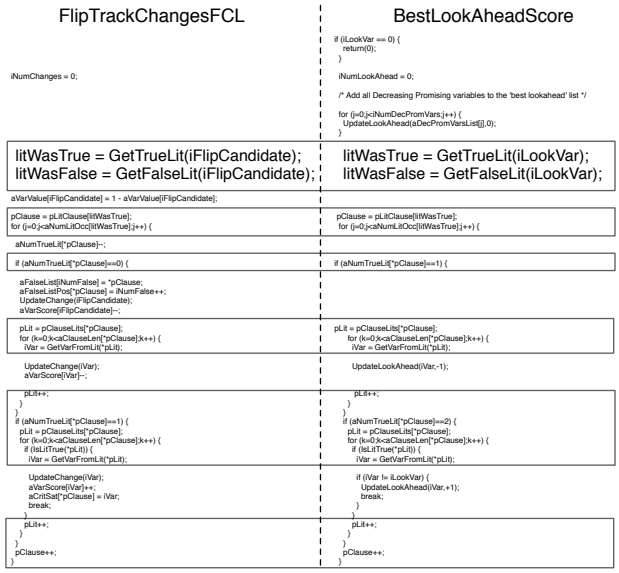

Figure 2. Sat Solver hotspot code comparison Figure highlights similar code patterns across two functions in the UBC Sat Solver tool [18].

tions to find the "hotspots" where the application spends most of their time (listed in Table 1). Then, we build Program Dependence Graphs (PDG) [9] for these hotspot functions. We find the similar code segments across these hotspots by searching for isomorphic subgraphs across their PDGs (Section 3.2 discusses the similarity algorithm). We quantify the *similarity* between the hotspots as the fraction of nodes that matched between their PDGs.

Our results, shown in Figures 3 & 4, demonstrate that significant similarity exists across applications. Figure 3 bins the hotspot pairs based on the amount of similarity present between them. The data shows that most of the hotspot pairs ($> 90\%$) had some similar code patterns (50% node matched) and more importantly, at least 50% of the hotspot pairs had significant similarity ($> 80\%$ nodes matched). Also, Figure 4 shows that significant similarity exists within as well as across application classes.

To exploit the available similarity, our tool chain merges application hot spots with similar code structure and builds one QASIC for them. Our design methodology provides the following benefits:

1. **Fewer number of specialized circuits and reduced area requirements:** Our approach realizes these reductions by designing QASICs that can support multiple similar computations. We found these similar computations across hotspots of the same application (such as term_newpos_a and term_newpos_b from Twolf), across different applications in the same application domain (such as different image conversion algorithms), and even across applications from different application domains (such as Bzip2 and Disparity).
2. **Generality:** The QASICs tend to have more flexible control and data flow compared to fully specialized hardware because they are designed to target multiple code segments. As a result, QASIC's computational power can extend beyond the code segments for which they were designed. For example, in our benchmark set, the *imageBlur* kernel uses a 5-stage filter [20] and the *edgeCharacteristics* kernel uses a 3-stage filter. However, the QASIC formed by merging *imageBlur* and *edgeCharacteristics* can execute both the kernels with either filter, providing this QASIC with additional computational power.
3. **Better backward compatibility for application-specific hardware:** In order to remain useful across software versions, application-specific hardware must be able to adapt to changes in the code it supports. We can utilize our design methodology to ensure that QASICs support all the previous "in-use" versions. We achieve this by merging the hotspots of "in-use" application versions with the latest version and design one QASIC for them. This allows the system designer to target the latest version while ably supporting all the previous versions as well. For example, compared to the previous work on supporting multiple application versions [21], our approach can improve the energy savings of *MCF2000* hotspots on the *MCF2006* hardware by 1.8X without affecting the energy-efficiency for *MCF2006*.

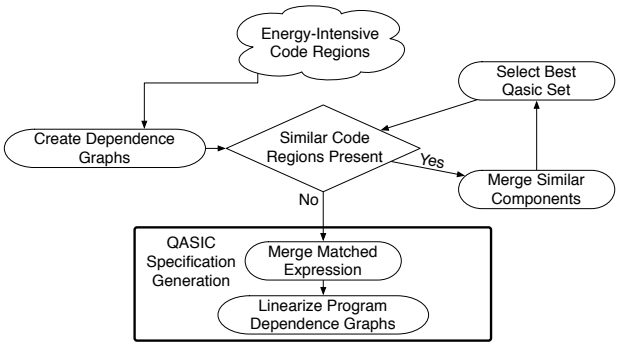

Figure 5. QASIC Design Flow The design flow from application hotspots to QASIC generation is shown.

3. Quasi-ASIC Design Flow

In this section, we present the details of our QASIC design flow (shown in Figure 5). The design flow accepts the target application set and the area budget as input and generates as output a set of QASICs that fit within the available area budget and can support a significant fraction of the execution of target applications.

The design flow starts with a set of dependence graph for each of the application hotspots. At each stage, the toolchain selects the dependence graph pair with similar code patterns, merges the dependence graph pair to build a new dependence graph, and replaces the dependence graph pair with the new merged graph. This process continues until the toolchain is unable to find similar code segments across the dependence graphs or the area budget goals are met. We describe these steps in greater detail below.

Figure 6(a)-(d) shows the process of merging similar hotspots, *computeSum* and *computePower*, to form a QASIC.

3.1 Dependence Graph Generation

Our toolchain internally represents the application hotspots as Program Dependence Graphs(PDG) [13], where nodes represent statements and edges represent control and data dependencies. The PDG representation of the hotspots is better suited for finding matching code regions than control flow graphs or program text because it enables us to perform matching based on the program semantics rather than the program code text. Using PDG, our toolchain gets rid of all the false dependencies, preserving only the "real" control and data dependences. Figure 6(a) shows the PDG for a simple loop that computes sum of first n numbers. The solid edges represent control dependence and dashed ones represent data dependence. Unlike a control flow graph, there is no edge between the nodes $sum = 0$ and $i = 0$ because these statements are independent of each other.

We use the CodeSurfer tool [6] to create PDGs for the given code segments. The output of this stage is a pool of PDGs of the application hotspots. The subsequent steps of the design flow increase

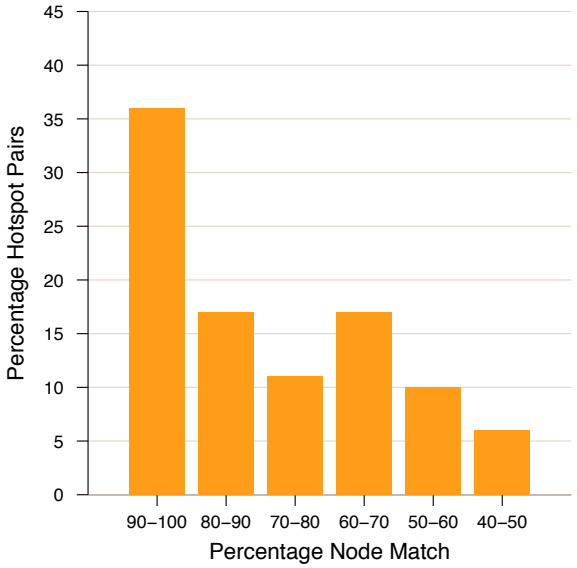

Figure 3. Similarity available across hotspots of diverse application set (Table 1) The X plots shows the amount of similarity present and Y axis shows the percentage of pairs that have certain merge percentage shown on the X axis.

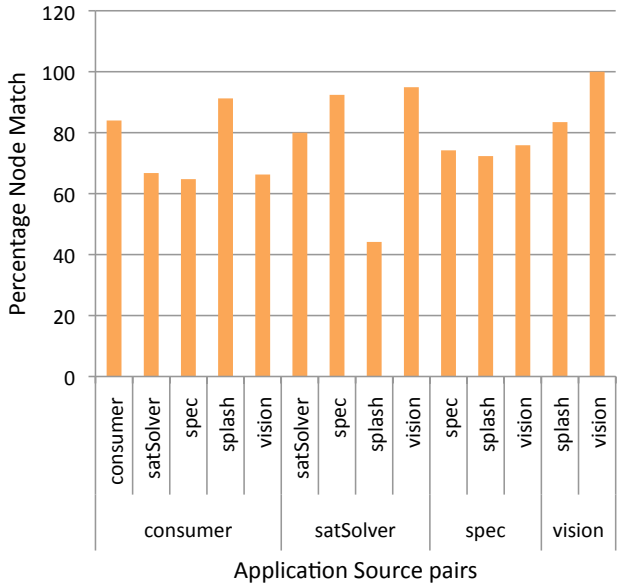

Figure 4. Quantifying Similarity Present Across Application Types. The graph quantifies similarity present across different classes of applications.

the generality of these PDGs and reduce hardware redundancy in this PDG pool until the area budget is met.

3.2 Mining for Similar Code Patterns

This step seeks to find dependence graph pairs from the QASIC PDG pool that are the similar to each other. The problem of finding similar code patterns across application hotspots can be reduced to finding similar subgraphs (subgraph-isomorphism) across their PDGs. The subgraph-isomorphism is a well studied [11] problem in the field of graph algorithms. Our algorithm for mining similar code patterns is based on the FFSM algorithm proposed by Huan et al. [11]. Below, we present a brief description of the FFSM algorithm and the optimizations we made to tailor the algorithm to the problem of finding similar code fragments.

The graph matching algorithm (FFSM) takes as input two graphs, G_1, G_2, where every node in both the graphs have a unique ID as well as type label. The algorithm considers two nodes for matching only if they have the same *type* label. The algorithm begins by selecting a node n_1 randomly in G_1 and finds a matching node n_2 for n_1 in G_2. Then, it tries to grow the matched subgraph by comparing the neighbors of n_1 and n_2 as well as performing other *join* operations on the matched subgraphs (refer to [11] for details). The algorithm returns when it cannot grow the subgraph any further.

The QASIC toolchain extends the *FFSM* matching algorithm in several ways to tailor it to the problem of finding similar code patterns in PDGs. First, instead of picking and matching nodes in a random order, our matching algorithm focuses on finding similar loop bodies. This behavior is desirable since most of the application execution time is spent in loops. Secondly, we use the node type to encode the program structure so as to prune "illegal" matches and reduce the search space. For example, all the nodes within a loop body should have "similar" node type and different from the node type of any nested loop nodes within that loop. This node type definition ensures that two nodes would match only if they perform similar arithmetic operations (for example *addition* and *branch* operations are not similar), similar memory operations (such as array/pointer access), and the control/data edges associated with these two nodes match.

For example, when trying to find similar code patterns across $computeSum$ and $computePower$, shown in Figure 6(a), this stage would map the *sum+=i* node in $computeSum$ to *sum*=sum* of $computePower$, among others.

The output of this stage is a list of dependence graph pairs that have similar code patterns present across them. For each of these similar dependence graph pairs, this stage also produces a mapping of the similar code patterns across them. As we saw earlier in Figure 3, the matching algorithm was able to find substantial similarity across different code segments.

3.3 Merging Program Dependence Graphs with similar code structure

This stage of the QASIC design flow accepts as input the similar dependence graph pairs that the previous stage produces. For each dependence graph pair, this stage merges their mapped nodes to form a new QASIC dependence graph that is capable of supporting all the computations that either of its input dependence graph can support.

The main challenge in this step is to ensure that the QASIC PDGs it produces are linearizable, in that there is a sequential ordering of the PDG nodes that respects all the data and control dependences. To preserve this linearizable property, the QASIC's PDG must be reducible and have no circular data/control dependence. A PDG is reducible if each of its loop body has a single entry point.

As the first step, this stage ensures that there are no circular dependence in the merged PDG using control, data, and *inferred* (ex-

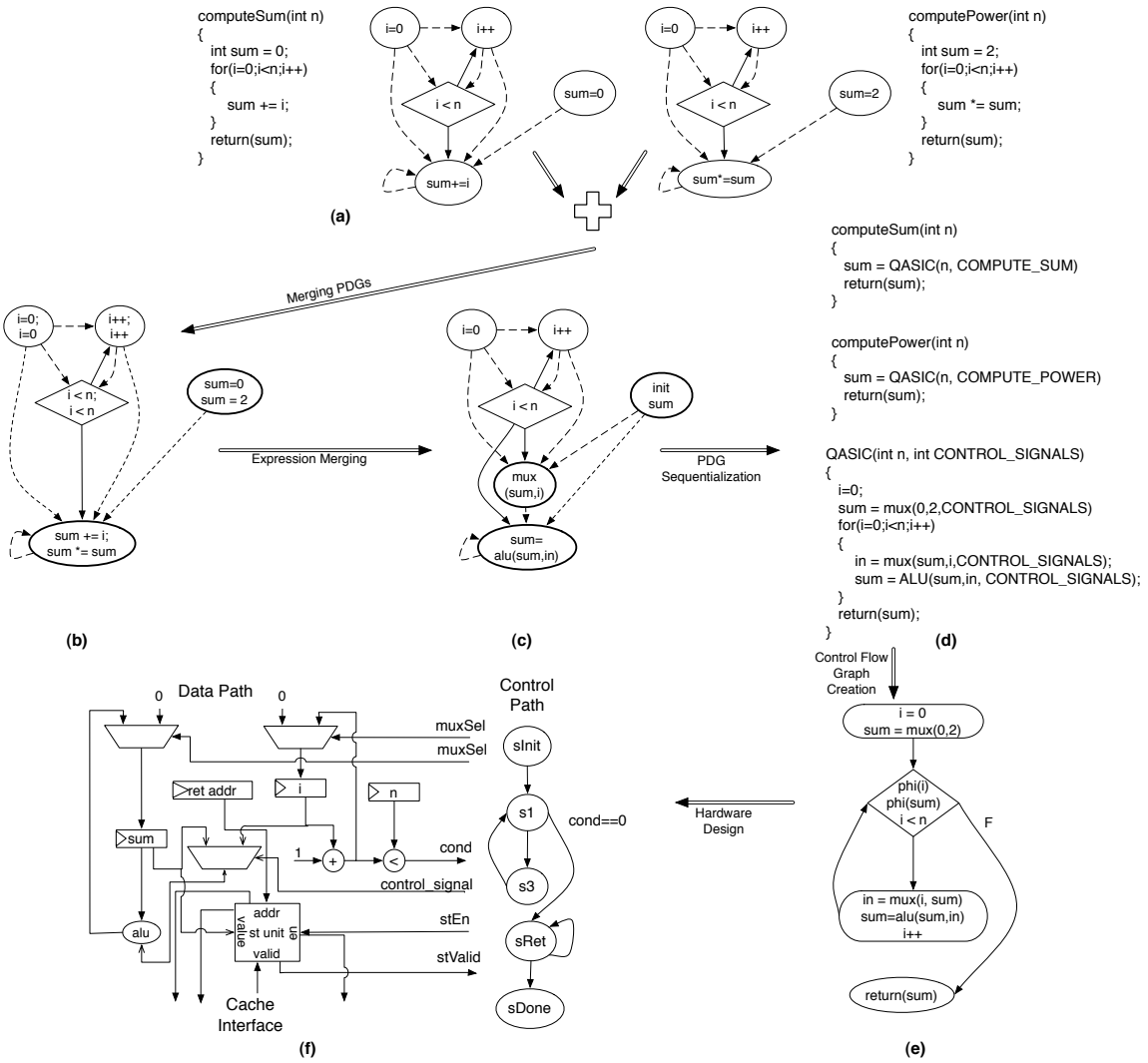

Figure 6. Quasi-Architecture Example An example showing the different stages involved in conversion of source code segments, $computeSum$ and $computePower$ (a) to QASIC hardware (f). The resultant QASIC can perform the functionality of the two input code segments as well as other functions like $n!, c^{2^n}$. The solid lines represent the control dependence and the dashed lines represent the data dependence for the program dependence graphs ((a), (b),(c)).

plained later) dependence edges. We eliminate the circular dependences by removing the least number of node matches that would break all the dependence cycles. The next step is to merge the loop entry points and in the process, ensure that each loop body has only one entry point. Whenever we detect a code region with multiple entry points, we add dummy control nodes as entry points to maintain a reducible control dependence graph. Then, we build a one-to-one map between variables of the two PDGs based on the nodes that got matched. We use this variable map to match the *declaration* and *phi* nodes that were ignored in the Section 3.2 to reduce subgraph matching time. At this point, we merge the two PDGs to form a new PDG of the QASIC that can execute computation corresponding to both the merging pdgs.

Figure 6(b) shows PDG of the QASIC that our toolchain designs by merging the dependence graphs of *computeSum* and *computePower*, shown in Figure 6(a).

The QASIC PDG contains additional node and edge attributes to enable PDG linearization (Section 3.4).

1. **Node Attributes:** Each node in the PDG contains a list of variables defined and used by that node. This node attribute is extended to contain a list of conditionally defined and used variables. For example, in Figure 6(b) $sum\ +\!=\!i$; $sum*\!=\!sum$ node conditionally consumes variable i because only one of its input code segments (*computeSum*) consumes i.
2. **Edge Attributes:** The PDG is augmented with conditional data dependences. The conditional edges result from edges present in only one of the two PDGs being merged. For example, *computeSum* in Figure 6(b) has data dependence between i++ and $sum\ +\!=\!i$ but there is no data dependence between i++ and $sum*\!=\!sum$. This leads to a conditional dependence from i++ to $sum\!+\!=\!i$; $sum*\!=\!sum$ in the QASIC's PDG.

At this point, this stage has designed a QASIC PDG corresponding to each similar dependence graph pair that the previous stage produced. Each of these newly designed QASICs have greater computational power than the two dependence graphs they were formed from because they can support the computations that either of their input dependence graph can support. Moreover, these QASICs re-

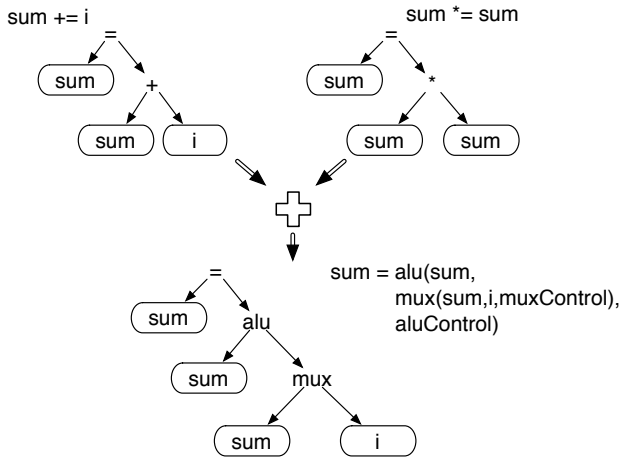

Figure 7. Expression Merging An example showing merging of the expression trees of the two input expressions to build the new merged expression tree and the corresponding expression.

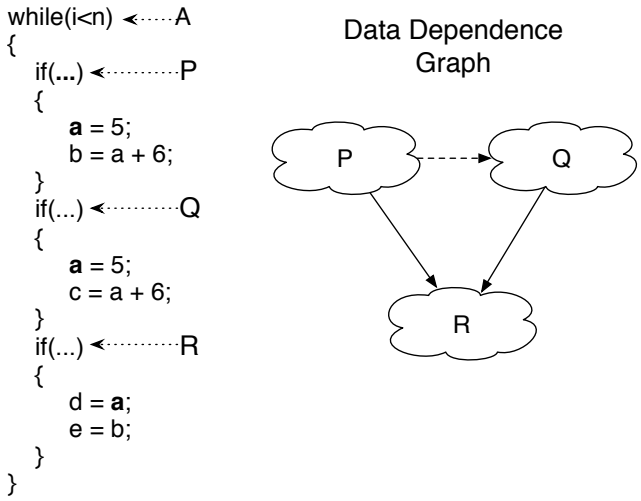

Figure 8. Inferred Dependence Example The solid and dashed lines show true and inferred dependences. There is an inferred dependence edge from P to Q because no valid ordering of P, Q, R orders Q before P (Section 3.4).

quires lesser area compared to its input dependence graphs because they eliminate hardware redundancy across the input dependence graphs by merging similar computations.

The final step of this stage is to, amongst all the QASICs that this step designed, select the QASIC that, when compared to the dependence graph pair they were formed from, will provide the maximum benefits in terms of the area saved and increase in the computational power. Section 4 explains in detail our heuristic for performing this QASIC selection. Once the best QASIC candidate is chosen, this stage replaces the two input PDGs with the chosen QASIC's PDG in the PDG pool.

At the end of this stage, the toolchain loops back to the second stage (Section 3.2) to find other potential PDG pairs for merging. Eventually, the QASIC set becomes distinct enough that no substantial similarity can be found across them. At that point, the toolchain proceeds to generate the QASIC specifications in C.

3.4 QASIC Generation

The fourth stage of the toolchain sequentializes the PDGs of the QASIC set to produce the QASIC specification in C, which is used to generate Verilog code by the backend of our toolchain. The two steps involved in this stage are merging the matched expressions present in each QASIC PDG node and sequentializing the QASIC's data and control dependences.

Merging Expressions This step generates a valid C-expression corresponding to each node of a QASIC. The QASIC PDGs that the previous step designs consist of multiple C-expressions in each of its nodes. For example, in Figure 6(b), a QASIC PDG node contains the expressions *sum += i* and *sum *= sum*. To design this merged C-expression, we build expression trees using ANTLR [2], which are then merged into a single expression tree. This merged tree directly translates to a valid C-expression. This process is shown in Figure 7. This step achieves much of the area reduction seen in our results by reusing datapath operators. For example, in Figure 6(c), the QASIC eliminates hardware redundancy bymerging $i < n$ expression used in both $computeSum$ and $computePower$. Figure 6(c) shows the result of merging expressions for the PDG shown in Figure 6(b).

Linearizing* QASIC *PDGs The PDGs are inherently parallel representation of a program. However, in order to produce QASIC specification in C, we need a valid ordering of nodes consistent with the control and data dependence edges. The ordering of control edges in a QASIC PDG is straightforward and the previously proposed techniques for reducible graphs [13] work in our case as well. Sequentializing the data dependence is more challenging because of the conditional data flow edges as well as conditionally defined-used variables of the nodes. This stage uses the following technique to sequentialize data dependences including the conditional node and edge attributes of QASIC PDG nodes.

The main goal of our technique is to order the PDG nodes in the presence of conditional data dependence without employing backtracking (computation time for backtracking-based techniques can get very expensive as the PDG's size and complexity increases). We use the data dependence edges and use/def analysis to build *inferred dependences* between children of same control node. The inferred dependence is defined as follows: Let us say that parent node A has child nodes P and Q. We say that child P has *inferred dependence* on child Q if there exists a child R of parent node A such that child P produces value b consumed by child R, child Q produces value a consumed by child R, and child P also produces value a. This implies that only valid ordering among them is for child P to execute before child Q. An example of this is shown in Figure 8. The subgraph P produces values a and b, subgraph Q produces values a and c. There is no dependence between P and Q. The subgraph R consumes value a from Q and b from P causing an implicit ordering between P and Q. We use the data dependence, inferred dependence and use/def analysis to linearize the data dependence. To handle additional node and edge attributes of the QASIC PDG, we extend the data structures used in data dependence serialization algorithm to make them aware of these additional attributes. Our linearization algorithm based on inferred dependence lends itself to easily support conditional data dependence and the algorithm's computation time scales well w.r.t. the size and complexity of the QASIC PDG.

The final linearized QASIC formed by merging $computeSum$ and $computePower$ is presented in Figure 6(d). Based on the

value of the CONTROL_SIGNALS, this QASIC can be configured to support the computations performed by $computeSum$ as well as $computePower$. In addition, this QASIC can also be configured to perform other operations such as factorial, c^{2^n}, besides $computeSum$ and $computePower$ by configuring the control lines to the *mux*, *ALU* and *init* value of *sum*.

3.5 Modifying Application Code to utilize QASICs

Our toolchain also modifies the application code to allow it to offload the computations on to the QASIC at the runtime. The toolchain does this by determining a valid setting of the QASIC's CONTROL_SIGNAL input that would allow the QASIC to execute the computation performed by the application. For example, Figure 6(d) shows how the application code is modified to use the QASIC. In this example, the $computeSum$ function sends the function argument values as well as an additional argument that would configure the ALU in the QASIC to perform addition. Moreover, the toolchain also inserts stubs in the application code to query the runtime for the availability of a matching QASIC(not shown in the figure for simplicity). In case no matching QASIC is present or available, then the application defaults to running the software version of the function on the general-purpose processor.

4. QASIC-selection heuristic

In the previous section, we described our design methodology for merging application hotspots and building QASICs for them. Section 3.3 described our methodology for merging two hotspots with similar code structures. However, a hotspot can match well with multiple other hotspots. For example, *integralImage2D* hotspot in Disparity matches well with multiple hot spots (*slave_sort* in Radix, *findDisparity* in Disparity) belonging to different application classes and having different code sizes. In general, there are exponentially different alternatives for merging the application hotspots to form the final QASIC set. In this section, we present our heuristic to decide which hotspots to merge to make the best area-energy tradeoff at each step.

Our tool chain's goal is to find the set of QASICs that will most significantly reduce the power consumption while fitting within the available area budget. The reduction in power consumption that a QASIC can deliver is a combination of its power efficiency and fraction of programs that it executes. Formally, a QASIC b occupies area A_b, consumes power P_b, has speedup S_b, and has coverage C_b (relative application importance).

To evaluate b, we define a quality metric $Q_b = \frac{C_b S_b}{A_b P_b}$. To select a good set of QASICs to build, we will need to compute Q_b for an enormous number of candidate QASICs. Computing precise values for S_b and P_b in each case is not tractable since it requires full-fledged synthesis and simulation. To avoid this overhead, we make two assumptions.

First, we conservatively assume that the speedup, S_b, is always 1. We synthesized and simulated fully specialized hardware for fragments from *integer programs* and found that the they are typically no more than twice as fast as a general purpose processor. We estimate A_b based on the datapath operators and register counts. Next, we assume that power consumption is proportional to area. This approximation is valid if we assume constant activity factors, constant clock frequencies, and circuit capacitance that grows linearly with circuit area. The C_b value is estimated based on system-level profiling.

Although there is invariably some error introduced due to these approximations, we believe that the loss of accuracy is more than compensated by the improvements in computational tractability.

With these assumptions we can approximate Q_b as $Q'_b = \frac{C_b}{A_b^2}$.

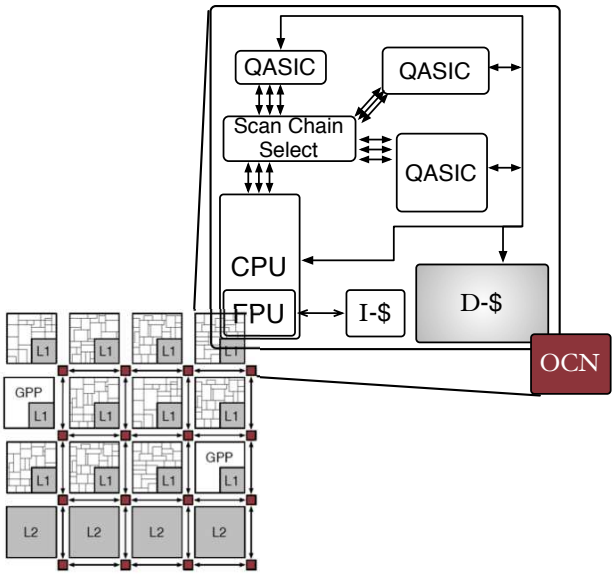

Figure 9. QASIC-enabled Tiled Architecture A QASIC-enabled system containing an array of tiles, each of which contains many QASICs connected via scan chains to the general purpose processor.

We use $X \bowtie Y$ to denote the QASIC that results from merging the QASICs for X and Y (Section 3). While estimating $A_{X \bowtie Y}$ is straightforward, estimating $C_{X \bowtie Y}$ is more challenging because $X \bowtie Y$ can implement other code segments beyond X and Y. Currently, we set $C_{X \bowtie Y} = C_x + C_y$ as a conservative estimate. In the future, we intend to use some form of cross-validation to measure $C_{X \bowtie Y}$ more accurately to promote generality.

To evaluate the quality of a set of QASICs, B, we sum the value of Q' for each QASIC. The goal of the QASIC design flow is to maximize

$$Q'_B = \sum_{b \in B} Q'_b = \sum_{b \in B} \frac{C_b}{A_b^2} \quad \text{subject to} \quad \sum_{b \in B} A_b < A_{\text{budget}}. \tag{1}$$

Algorithm 1 contains the pseudo-code for our QASIC-selection heuristic that starts with a fully specialized ASIC for each fragment and merges them to create QASICs. It iteratively selects QASIC pairs that maximize Q'_B (Line 2) and merges them to form a more general QASIC that has a greater computational power than its input QASIC pair.

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

Algorithm 1. Greedy clustering algorithm The algorithm for deciding which QASICs to build. B is initially the set of fully-specialized ASICs, one for each of the fragments selected by the profiler.

5. QASIC Architecture Design

In this section, we describe our hardware generation compiler and integration of these QASICs with general purpose processor.

5.1 QASIC Hardware Generation

Our hardware generation compiler is built on the C-to-HW compiler proposed in c-cores [21] for generating Verilog from C source code. This compiler also generates a cycle-accurate module for our architectural simulator. Figure 6(d-f) presents this hardware generation step for the example QASIC designed in Section 3.3.

The compiler builds the hardwares datapath and control state machine based on the data and control flow graphs of the QASIC source code in Static Single Assignment [7] form. In addition to standard C data operators, our hardware compiler also supports QASIC-specific operations such as `ALU` and *data-selector* (shown as mux in Figure 6(d)). The generalized arithmetic operations, `ALUs`, in a QASIC's dataflow graph are instantiated as functional units in the hardware's datapath. The *data-selectors* in a QASIC's dataflow graph are instantiated as a *mux* operator. To optimize the QASIC's energy-efficiency, the computation of the data-selector's inputs are predicated on the data-selector's control signal. Hence, based on the control signal, only one of the inputs is computed.

The memory operations in the dataflow graph are instantiated as load/store units in the hardware datapath. The load-store units connect to same coherent data cache as the general purpose processor and perform all the memory operations in program order.

5.2 Integrating QASICs with General-Purpose Processors

The QASICs can be used to extend any general purpose system to provide higher specialization and optimize energy consumption per instruction. In this paper, we describe how tiled-architectures like RAW [16] can be extended with QASICs. In a QASIC-enabled system (Figure 9), each tile contains a general-purpose processor, I-Cache, D-Cache, set of QASICs and interconnect logic. The system includes 100s of QASICs designed for key functions of the target system's workloads. The general purpose processor provides the safety net for code regions not covered by any of the QASICs.

On each tile, QASICs are connected to a general-purpose processor via scan chains. The general-purpose processor passes input arguments, starts QASIC execution as well as receives QASIC task completion notifications via these scan chains. The scan chain interface is slow but scales well, allowing us to connect 10s of QASICs to a CPU. The QASICs shares the D-Cache with the general-purpose processor enabling data transfers through the L1 cache as well.

6. Methodology

In this section, we provide details of our synthesis toolchain and simulation infrastructure for performance and power measurements.

Synthesis 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). The toolchain processes synthesizeable Verilog to generate placed and routed QASICs.

Performance and Power measurements We extend the cycle-accurate simulator presented in [21] to measure QASIC's performance. Our simulator models the complete QASIC-enabled system including the general-purpose processor, QASICs, interconnect, runtime and a coherent memory system.

In order to measure QASIC power usage, the simulator periodically samples execution of QASIC and feeds these samples to Synopsys VCS logic and Synopsys Primetime. We use processor and clock power values for a MIPS 24KE processor in TSMC 45nm reported in [21]. Finally, we use CACTI 5.3 [17] for I- and D-cache power.

7. Results

In this section, first we evaluate our QASIC selection heuristic by comparing it to the optimal solution found via exhaustive search. Next, we demonstrate that relatively few QASICs can support all the commonly used operations on multiple data structures. Moreover, these QASICs provide an order-of-magnitude more energy efficiency than general-purpose processors. In addition, for a more diverse application set (Table 1), the data shows that our methodology can significantly reduce the required number of specialized circuits as well as the area requirements compared to fully-specialized logic while continuing to provide ASIC-like energy efficiency. Finally, our results show that QASICs can effectively support the legacy application versions and can provide significant improvements in their energy efficiency compared to the previous work in this area [21].

7.1 Evaluating the Clustering Heuristic

In this section, we evaluate our QASIC selection heuristic (Algorithm 1) against exhaustive search. We use a micro-benchmark set containing functions for computing eight mathematical functions — $\sum_{i<n}^{i=0} i$, $n!$, 2^n, a^{2^n}, $\sum_{i<n}^{i=0} a[i]$, $\prod_{i<n}^{i=0} i$, $\sum_{i<n}^{i=0} |a[i]|$, and count of powers of 2 in an array.

To evaluate our heuristic, we build QASICs using the heuristic and also using the exhaustive approach that will design all of the 255 QASICs that can arise from merging subsets of the 8 programs.

We compare our algorithm to the exhaustive search approach under different design constraint scenarios. These different scenarios were formed by varying the area budget available for specialization as well as by varying the relative importance assigned to each of the eight computations. In all the cases, our heuristic built the same set of QASICs as exhaustive search.

The low computational complexity and near optimality of our heuristic algorithm allows it to scale to handle large target application sets. This ability is critical because we expect the system's target workload set to be very large in general.

7.2 Evaluating QASIC's Area and Energy Efficiency

In this section, we evaluate the ability of our toolchain to support a significant fraction of the target system execution in hardware using a relatively small number of QASICs that fit within a limited area budget.

7.2.1 Evaluating QASIC's ability to target an application domain

The first experiment designs QASICs for the `find, insert, delete` operations for the commonly used data structures, namely link-list, binary tree, AA tree, and hash table. Figure 10 shows how the toolchain varies the QASIC design based on the area budget available for specialization. The X-axis plots the number of QASICs that the toolchain designs to fit within the available area budget. The left and right Y-axes show how our toolchain trades between area and energy efficiency. The results show that relatively few QASICs can support all these 12 data structure operations while improving the energy efficiency by more than 13.5× compared to our baseline general-purpose processor.

The next experiment shows that our toolchain can provide hardware support for increasing number of functionalities without a corresponding increase in the required number of specialized cores. Figure 11 plots the increase in the functionality supported in hardware against the required number of distinct QASICs. The data shows that just four QASICs can support all these operations, while the previous work, c-cores, would need to design eleven specialized cores. This 63% decrease in the required number of specialized cores allow us to closely integrate hardware support for greater number of features with a processor pipeline. For example, for our

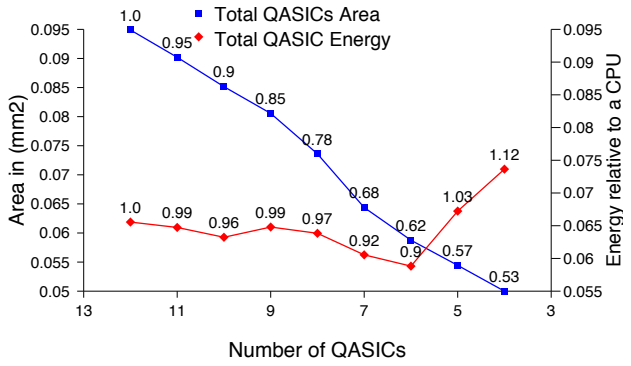

Figure 10. Impact of generalization on the area and energy Efficiency of QASICs targeting commonly used data structures The labels on the area and power curves show the ratio of area and energy requirements of the QASIC set compared to that of fully-specialized circuits.

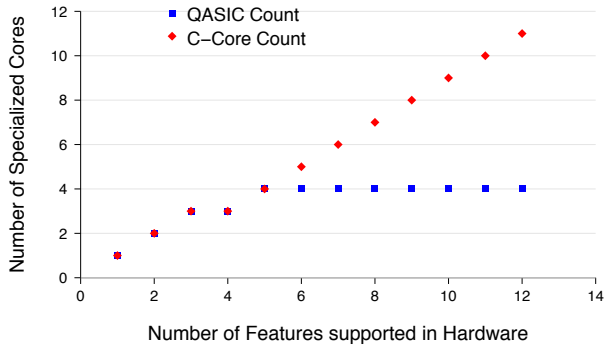

Figure 11. Scalability of QASIC's approach The graph shows that relatively few QASICs can support multiple commonly used data structures, while c-cores need a new specialized processor for every distinct functionality.

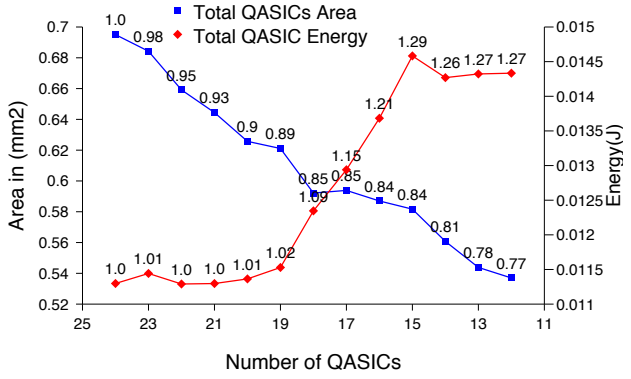

Figure 12. Impact of generalization on QASIC Area and Energy Efficiency for the Benchmark Set The labels on the area and energy curves show the ratio of area and energy requirements of the QASIC set compared to that of fully-specialized circuits.

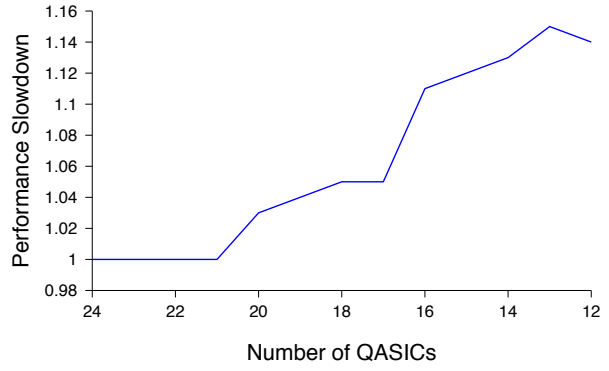

Figure 13. Impact of generalization on QASIC Performance for our Benchmark Set Y axis shows the ratio of execution time of QASICs compared to that of fully-specialized circuits while X axis plots the QASIC count.

scan chain based interconnect design explained in Section 5.2, QASICs reduce the interconnect overhead (measured as the number of connections between the CPU and all the specialized cores) by 54% compared to c-cores.

7.2.2 Evaluating QASIC's ability to target a diverse workload

In this section, we evaluate our QASIC design flow by using it to design QASICs for the hotspots from our diverse workload listed in Table 1. The results are shown in Figure 12. The X-axis plots the number of QASICs required to cover all the application hotspots. The left-most point on the X-axis corresponds to fully-specialized logic, and hardware generality (i.e., the average number of computations that a QASIC supports) increases from left to right. The results show that our toolchain can reduce the number of specialized co-processors required to cover all application hotspots by over 50% (which in turn reduces the interconnect overhead by 1.58×). Also, QASICs reduce the total area requirements by 22% compared to that of fully-specialized hardware, while incurring a 27% increase in energy consumption. The first few merges result in area reduction without any impact on power consumption. This is because leakage energy goes down as the area is reduced and that offsets any increase seen in dynamic energy. For the subsequent merges, our approach ensures that energy-efficiency degrades gracefully with decreases in the total area budget. These results show that our toolchain can effectively reduce hardware redundancy while providing ASIC-like energy-efficiency.

Application-level energy savings Figure 14 shows that, compared to the baseline tiled system, a QASIC-enabled system consisting of just thirteen QASICs can provide significant energy efficiency improvements for our diverse workload. This experiment models the complete system including the overheads involved in accessing the runtime system as well the overheads for offloading computations on to the QASICs. The data show that, at the application level, QASICs save 45% of energy on average compared to a

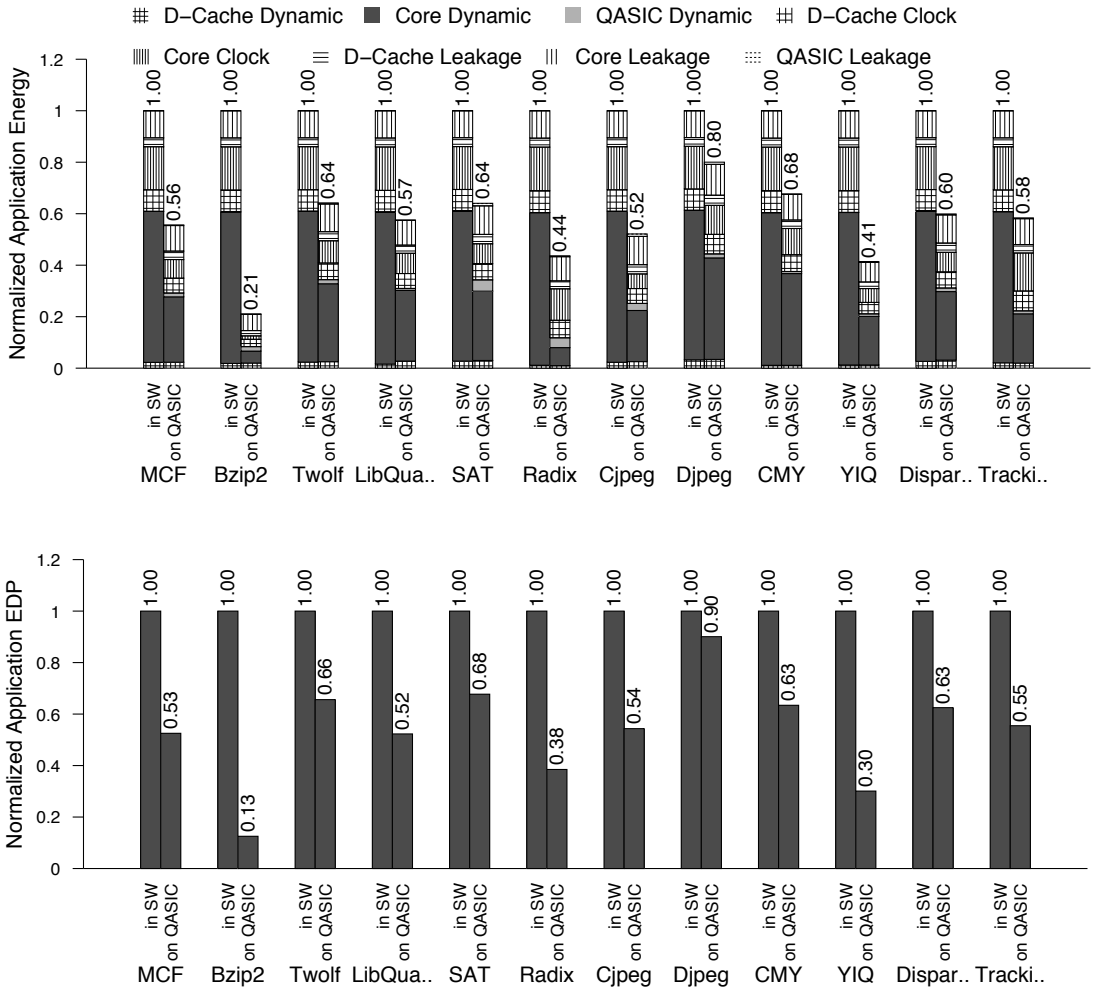

Figure 14. Quasi-ASIC enabled System Energy Efficiency The graphs show the energy and EDP reductions by QASICs compared to an in-order, power efficient MIPS core ("SW"). Results are normalized to running on the MIPS core (lower is better). The QASICs can reduce the application energy consumption and energy-delay product by up to 79% and 83% respectively.

MIPS processor, and the savings can be as high as 79%. Also, QASICs reduce application energy-delay product by 46% on average. The energy savings for these QASIC-enabled systems are significant, and as we saw in the previous section the energy-efficiency will only improve as the transistor budget increases.

7.3 Backward Compatibility for QASICs

In this section, we evaluate QASIC's ability to support older application versions. The specialized hardware should be able to support multiple application versions in order to remain useful for a long time. We assume that while hardware designers will design specialized hardware for the latest version of each application, they will also want to support older "in-use" versions of the application as well. To provide this backward compatibility, our toolflow merges the older application versions with the latest version and builds one QASIC for all of them.

Previous work, conservation cores [21], presented reconfigurability mechanisms to support newer (and older) application versions. Figure 15 presents QASIC's energy-efficiency across application versions and compares it to that of the technique proposed in [21]. The Y-axis plots the energy-efficiency compared to an in-order MIPS processor. The results show that QASICs provide significant energy-improvements compared to the baseline processor for all the application versions and can be up to 7× more energy-efficient than conservation cores for older application versions. Also, for the latest application version, QASICs are almost as energy-efficient as conservation cores (less than 10% difference on average).

8. Related Work

The topic of code similarity has been studied in the past in other contexts like for detecting plagiarism [14] and in software engineering tools [13]. The main difference between our goals and that of these previous research works is that we want to find similarity among diverse pieces of code to improve hardware reusability while the previous research focuses on finding exact or almost same code segments in a body of code that is expected to have code reuse. While there is much we can learn from the techniques proposed in these papers, we would have to significantly augment and enhance them to make it suitable for the purpose of this project.

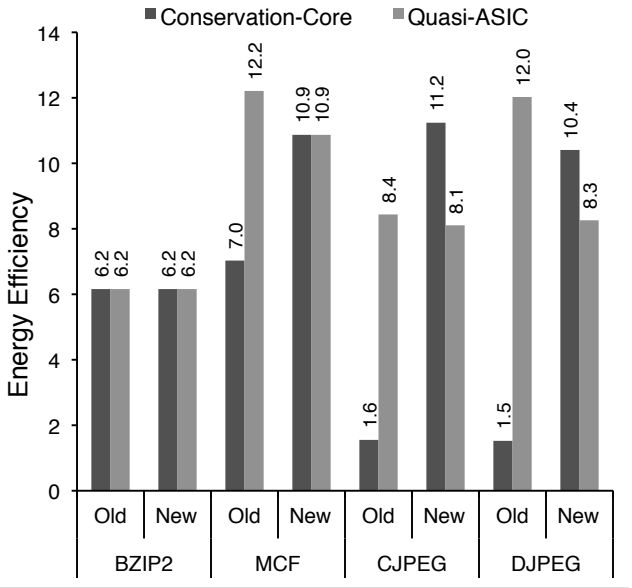

Figure 15. Building QASICs for backwards compatibility The energy-efficiency of the older versions improves significantly (up to 7×) with minimal impact on that of newer ones.

Recent work has focused on providing hardware support for regular embedded loops as well as irregular integer codes [8, 21, 24]. The QASIC's design goals as well as target application set differs from these previous efforts on automatically designing specialized circuits. We differentiate our work from these previous approaches in Section 2. The ideas presented in these papers can be used to augment our work to enable QASICs to better exploit the available parallelism in the target functions as well as to ensure that QASIC's longevity is comparable to general-purpose processors.

In the past, there has been some work on building application specific processors or reconfigurable logic like PICO [12], CriticalBlue [1], WARP [19] and application specific instruction set extensions [5, 22]. Our work differs significantly from these approaches in that our approach does not use any of pre-engineered components but builds the co-processors ground up based solely on the application characteristics. Our application-specific design process yields an energy efficiency closer to an ASIC rather than a reconfigurable fabric or co-processor.

9. Conclusion

Technology scaling trends will continue to increase the number of available transistors while reducing the fraction that can be used simultaneously. To effectively utilize the increasing transistor budgets, we present QASICs, specialized co-processors that can support multiple general-purpose computations and can provide significant energy efficiency compared to a general-purpose processor. Our toolchain designs these QASICs by leveraging the insight that similar code patterns exist within and across applications. Given a target application set and an area budget, our toolchain varies the computational power of these QASICs such that a significant fraction of the execution is supported in hardware without exceeding the area budget. Our results show that designing just four QASICs can support operator functions of multiple commonly used data structures and moreover, these QASICs provide 13.5× more energy efficiency than our general-purpose processor. On a more diverse workload, our approach reduces the required number of specialized cores by over 50% and occupies 23% less area compared to fully-specialized circuits while providing ASIC-like energy efficiency (within 1.27X on average).

Specialization has emerged as an effective approach to combat the dark silicon phenomenon and enable Moore's Law-style system performance scaling without exceeding the power budget. QASICs enable a system designer to provide this specialization in a scalable manner by improving the computational power of specialized processors such that, a relatively few of them, combined, can support a significant fraction of the target workload execution in hardware.

References

[1] http://www.criticalblue.com/products/index.html.

[2] J. Bovet et al. "Antlrworks: an antlr grammar development environment." *Softw. Pract. Exper.*, 2008.

[3] K. Chakraborty, et al. "Computation spreading: employing hardware migration to specialize cmp cores on-the-fly." In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[4] E. S. Chung, et al. "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.

[5] N. Clark, et al. "Processor acceleration through automated instruction set customization." *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 3-5 Dec. 2003.

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

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

[8] K. Fan, et al. "Bridging the computation gap between programmable processors and hardwired accelerators." In *HPCA*, 2009.

[9] J. Ferrante, et al. "The program dependence graph and its use in optimization." *ACM Trans. Program. Lang. Syst.*, 1987.

[10] R. Hameed, et al. "Understanding sources of inefficiency in general-purpose chips." *SIGARCH Comput. Archit. News*, June 2010.

[11] J. Huan, et al. "Efficient mining of frequent subgraphs in the presence of isomorphism." In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, 2003.

[12] V. Kathail, et al. "Pico: automatically designing custom computers." *Computer*, Sep 2002.

[13] R. Komondoor et al. "Semantics-preserving procedure extraction." In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2000.

[14] C. Liu, et al. "Gplag: detection of software plagiarism by program dependence graph analysis." In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006.

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

[16] M. B. Taylor, et al. "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams." In *ISCA*, 2004.

[17] S. Thoziyoor, et al. "Cacti 5.1." Tech. Rep. HPL-2008-20, HP Labs, Palo Alto, 2008.

[18] D. A. D. Tompkins et al. "UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT." In H. H. Hoos et al., eds., *Theory and Applications of Satisfiability Testing: Revised Selected Papers of the Seventh International Conference (SAT 2004, Vancouver, BC, Canada, May 10–13, 2004)*, vol. 3542 of *Lecture Notes in Computer Science*, 2005.

[19] F. Vahid, et al. "Warp processing: Dynamic translation of binaries to fpga circuits." *Computer*, 2008.

[20] S. K. Venkata, et al. "Sd-vbs: The san diego vision benchmark suite." *IISWC*, 2009.

[21] G. Venkatesh, et al. "Conservation cores: reducing the energy of mature computations." In *ASPLOS*, 2010.

[22] A. Wang, et al. "Hardware/software instruction set configurability for system-on-chip processors." In *DAC*, 2001.

[23] S. C. Woo, et al. "The splash-2 programs: characterization and methodological considerations." In *ISCA*, 1995.

[24] S. Yehia, et al. "Reconciling specialization and flexibility through compound circuits." In *HPCA*, 2009.