QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores

Ganesh Venkatesh+, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata+, Michael Bedford Taylor and Steven Swanson

http://greendroid.org

Department of Computer Science and Engineering
University of California, San Diego

ABSTRACT

Transistor density continues to increase exponentially, but power dissipation per transistor is improving only slightly with each generation of Moore's law. Given the constant chip-level power budgets, this exponentially decreases the percentage of transistors that can switch at full frequency with each technology generation. Hence, while the transistor budget continues to increase exponentially, the power budget has become the dominant limiting factor in processor design. In this regime, utilizing transistors to design specialized cores that optimize energy-per-computation becomes an effective approach to improve system performance.

To trade transistors for energy efficiency in a scalable manner, we propose *Quasi-specific Cores*, or QsCores, specialized processors capable of executing multiple *general-purpose* computations while providing an order of magnitude more energy efficiency than a general-purpose processor. The QsCore design flow is based on the insight that similar code patterns exist within and across applications. Our approach exploits these similar code patterns to ensure that a small set of specialized cores support a large number of commonly used computations.

We evaluate QsCore's ability to target both a single application library (e.g., data structures) as well as a diverse workload consisting of applications selected from different domains (e.g., SPECINT, EEMBC, and Vision). Our results show that QsCores can provide 18.4× better energy efficiency than general-purpose processors while reducing the amount of specialized logic required to support the workload by up to 66%.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Heterogeneous (hybrid) systems; C.3 [**Special-Purpose and Application-based systems**]

General Terms

Design, Experimentation, Measurement

Keywords

QsCore, Conservation Core, Merging, Dark Silicon, Utilization Wall, Heterogeneous Many-Core, Specialization

1. INTRODUCTION

Although transistor density continues to scale, nearly constant per-transistor power and fixed chip-level power budget place tight constraints on how much of a chip can be active at full frequency at one time. Hence, as transistor density increases with each process generation, so does the fraction of the chip that must be under-clocked or under-utilized because of power concerns [27]. Researchers have termed these passive expanses of silicon area *dark silicon* [14, 11, 15, 17, 21]. Dark silicon is forcing designers to find new solutions to convert transistors into performance.

Recent work [3, 16, 27] has shown that specialization is an effective solution to the problem of using the increasing transistor budget to extend performance scaling. Specialization-based approaches trade a cheaper resource, dark silicon (i.e., area), for a more valuable resource, power efficiency, to scale system performance. Specialized processors improve system performance by optimizing per-computation power requirements and thereby allow more computations to execute within a given power budget.

Conventional accelerators reduce power for regular applications containing ample data-level parallelism, but recent work [27, 21] has shown that specialized hardware can reduce power for irregular codes as well. Moreover, these designs use specialized compute elements even for relatively short running computations. This increases the fraction of program execution on specialized, power-efficient hardware, but 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 require designing a large number of specialized cores, which in turn would make it difficult to tightly integrate all of them with the general-purpose processor. Therefore, for specialization to be effective at targeting a general-purpose system's workload, the specialized processors must ideally be able to support multiple computations, such that a relatively small number of them can together support large fractions of system execution.

To trade silicon for specialization in a scalable manner, we propose a new class of specialized processors, *Quasi-specific Cores* (QsCores). Unlike traditional ASICs, which target one specific task, QsCores can support multiple tasks or computations. Because of this improved computational power, a relatively small number of QsCores can potentially support a significant fraction of a

+ Now at Intel Corporation, Hillsboro, Oregon.

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

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil

system's execution. Moreover, QSCORES are an order of magnitude more energy-efficient than general-purpose processors. The QSCORE design flow is based on the insight that "similar" code segments exist within and across applications, and these code segments can be represented by a single computation pattern. The QSCORE design flow exploits this insight by varying the required number of QSCORES as well as their computational power (i.e., the number of distinct computations they can support) based on the relative importance of the applications and the area budget available to optimize them. This enables QSCORES to significantly reduce the total area and the number of specialized processors compared to a fully-specialized logic approach supporting the same functionality.

In this paper, we address many of the challenges involved in designing a QSCORE-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 be many hundreds of instructions long and contain complex control-flow and irregular memory access patterns, making the task of finding similarity both computationally intensive and algorithmically challenging. The second challenge lies in exploiting these similar code patterns to reduce hardware redundancy across specialized cores by designing generalized code-structures that can execute all these similar code patterns. The third challenge involves making the right area-energy tradeoffs to ensure that the QSCORES fit within the area budget, while minimally impacting their energy efficiency. Addressing this challenge requires 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 onto the QSCORES at runtime.

We evaluate our toolchain by designing QSCORES for the operator functions — `find`, `insert`, `delete` — of commonly used data structures, including a linked-list, binary tree, AA tree, and hash table. Our results show that designing just four QSCORES can support all these data structure operations and provide 13.5× energy savings over a general-purpose processor. On a more diverse general-purpose workload consisting of twelve applications selected from different domains (SPECINT, SAT Solver, Vision, and EEMBC), QSCORES continue to provide ASIC-like energy efficiency (18.4× better than general-purpose processors) while reducing the required number of specialized cores by over 50% and the area required by 25% compared to the fully-specialized logic. Also, at the system level, our approach reduces the 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 overviews the QSCORE-enabled systems. Sections 4 and 5 describe the QSCORE design flow. Section 6 presents our methodology for generating the QSCORE hardware and for evaluating its efficiency. Section 7 analyzes the efficiency of QSCORES. Finally, Section 8 concludes.

2. MOTIVATION

In this section we present some of the recent proposals for designing specialized co-processors and motivate the need for QSCORES. Also, we examine the similarity that is present across applications and describe how our design methodology exploits this similarity.

2.1 Specialized Processor Design

Significant prior work exists on customizing a general-purpose processor or extending it with reconfigurable logic based on a target

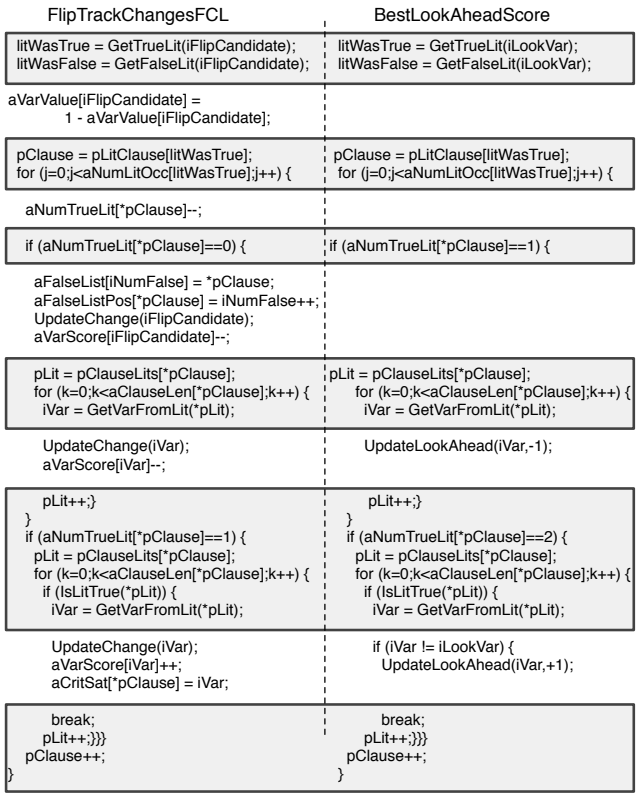

Figure 1: **SAT Solver hotspot code comparison** These two functions in the UBC SAT Solver tool [24] share significant code patterns (highlighted).

application's characteristics [19, 25, 28]. Our work differs from these approaches in that it does not center on the customization of pre-engineered components. Instead, our work builds the co-processors from the ground up based on application characteristics. This approach yields ASIC-like energy-efficiency by eliminating power-hungry abstraction layers, such as a processor's instruction interpreter (fetch, decode, and register file access), or an FPGA's SRAM lookup-tables.

A significant body of work exists on providing application-specific instruction set extensions based on commonly occurring instruction patterns [28, 6, 8, 1, 5]. Our work differs from these prior approaches since we provide specialization at a much broader scope. While these prior approaches focused on short acyclic instruction patterns (10s of instructions), our approach seeks to offload complete functions, typically containing 1000s of instructions, onto a specialized core. This enables us to provide higher energy savings by completely switching off the general-purpose processor pipeline, unlike these prior approaches that require many of the pipeline components to remain active.

Many ASIC-like accelerators [4, 12, 29] have focused on using modulo scheduling to exploit regular loop bodies that have ample loop parallelism and easy-to-analyze memory access patterns. Yehia et al. demonstrate techniques for automatically designing compound circuits that can support multiple regular loops [29]. Their technique focused 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 for which the hotspots tend

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	Tracking	calc_dX, calc_dY, imageBlur
	Disparity	finalSAD, findDisparity, integralImage2D
EEMBC Consumer	RGB/CMYK	CMYfunction
	RGB/YIQ	YIQfunction

Table 1: **Application Set** We use a diverse set of applications to evaluate the effectiveness of QSCORES.

to be much larger, contain complex control constructs, and have irregular memory patterns. The hotspots QSCORES target typically consist of many hundreds of assembly instructions, significantly increasing the exploration space.

In previous work [21, 27], we presented techniques for generating application-specific hardware for the general class of irregular, hard-to-parallelize applications. However, scalability in these systems can be limited by applications with large hotspots that have sufficiently small execution time that they do not justify the area cost of conversion into single-purpose coprocessors. Ideally, to make this area-energy tradeoff more favorable, the co-processors should support multiple applications with similar control/data dependence.

2.2 Application Similarity

We base the QSCORE design methodology on the insight that similar code patterns exist across applications. In this section, we quantify the available similarity and use this to motivate the QSCORE design methodology.

To start, we examine two hotspots in UBC SAT Solver [24] to provide insight about available similarity. Figure 1 shows the source code of the hotspots and highlights the similar code segments present across them. The example shows that while the two hotspots do not have similar control-flow, they do contain similar code segments. Our design methodology exploits these similar code segments to generate computation patterns that can represent both hotspots and supports these generic computation patterns in hardware. This improves the computational power of specialized cores and reduces hardware redundancy across them.

To quantify similarity across applications, we examine a diverse set of applications from SPEC CPU [22], UBC SAT solver [24], EEMBC [10], and SD-VBS [26] benchmark suites (described in Table 1). First, we profile the applications to find the "hotspots" where each application spends most of its time. Then, we build program dependence graphs (PDGs) [13] for the hotspot functions. We find similar code segments across the hotspots by searching for isomorphic subgraphs across their PDGs (see Section 4.2). We quantify the *similarity* between the hotspots' PDGs as the size of matching subgraph.

Our results, shown in Figures 2 and 3, demonstrate that significant similarity exists across applications. Figure 2 bins the hotspot pairs based on the amount of similarity present between them. The data show that most ($> 90\%$) hotspot pairs have some similar code patterns (50% of nodes matched) and more importantly, at least 50% of the hotspot pairs have significant similarity ($> 80\%$ of nodes matched). Figure 3 shows that significant similarity exists both within and across application classes.

To exploit available similarity, our toolchain designs a computation pattern that encompasses the similar code structures and builds a QSCORE for that computation pattern. This design methodology provides the following benefits:

1. **Reduced area requirements** Our approach reduces area requirements by eliminating hardware redundancy across specialized cores that target similar computations. We find these similar computations across hotspots of the same application, across different applications in the same application domain, and even across different application domains (such as Bzip2 and Disparity).
2. **Greater computational power** QSCORES have greater computational power and more flexible control/data flow compared to fully-specialized hardware. In some cases, QSCORE's computational power can extend beyond the code segments for which they were designed. For example, when targeting vision applications, our design flow builds a single QSCORE that supports both `imageBlur` (that uses a five-stage filter [26]) and `edgeFilters` (which uses three stages). The same QSCORE is flexible enough to support other filter depths as well.
3. **Backward compatibility for specialized logic** For many general-purpose applications, older versions may still be in use even though newer versions are available. Our design methodology enables QSCORES to support all previous "in-use" versions. We achieve this by merging the hotspots of previous versions with the latest version and designing one QSCORE for all of them. This allows a system designer to target the latest version without sacrificing legacy support.

3. QSCORE-ENABLED SYSTEMS

Figure 4 (a) shows the high-level design flow for building QSCORE-enabled systems. The process begins with a set of applications that the system designer believes make up a significant fraction of the processor's target workload. We profile these applications to extract the frequently used code segments, called hotspots, which have high application coverage. The QSCORE design flow uses these hotspots to design specialized cores based on the area available for specialization as well as the relative execution coverage of these hotspots. The final step integrates these QSCORES with a general-purpose processor to build a heterogeneous tile. In the general case, a QSCORE-enabled system will consist of multiple heterogeneous tiles, each comprising several QSCORES targeting different workloads.

Hardware-specialization for energy efficiency At the heart of this system design process is the QSCORE design stage (presented in Section 4) that generates specialized cores targeting computations that are commonly run on the system. The QSCORE design flow stage seeks to ensure that the QSCORE set supports all the hotspots of the system's workload while fitting within the area budget available for specialization. This is desirable because QSCORES are an order-of-magnitude more power-efficient than general-purpose processors, and hence, offloading the hotspots onto them can significantly improve the power efficiency of the system.

Application compilation and runtime execution The QSCORE compiler (presented in Section 4.5) automatically retargets application code to benefit from all the specialized cores available without any programmer intervention. At runtime, an application's ex-

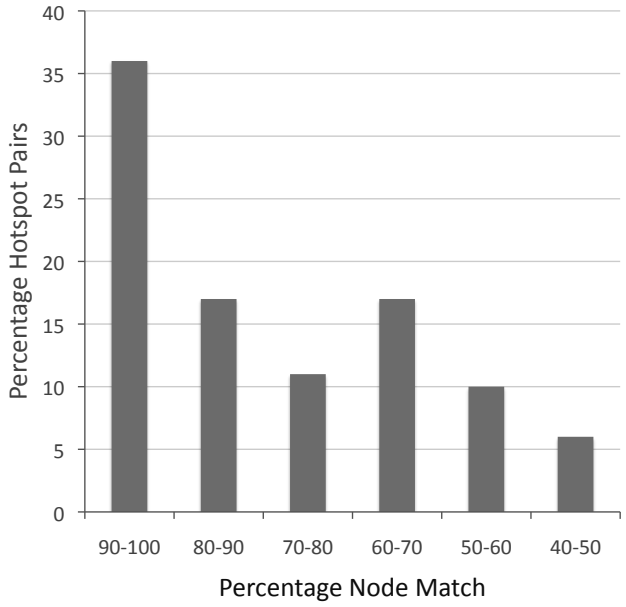

Figure 2: **Similarity available across hotspots** For the applications in Table 1, we bin the hotspot pairs based on the amount of similarity present between them.

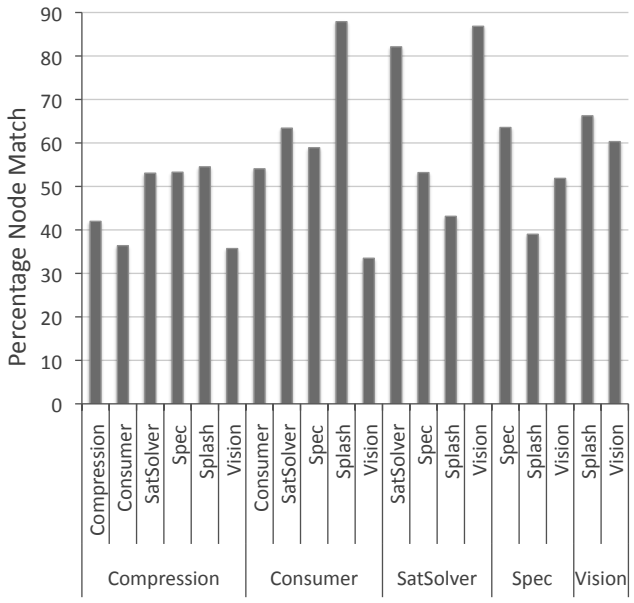

Figure 3: **Quantifying similarity present across application types** The graph quantifies similarity present across different classes of applications.

ecution migrates between the general-purpose processor and QsCores based on whether a matching QsCore exists for the section of code currently executing. This execution migration is possible because of the tight coupling of QsCores and the general-purpose processor via sharing of the L1 data cache and support for arbitrary control transitions between the QsCore and the CPU.

4. QsCore DESIGN FLOW

In this section, we present the details of our QsCore design flow (shown in Figure 4 (b)). The design flow accepts as input the hotspot functions as well as the area budget available for specialization and generates a set of QsCores that implement these hotspots and fit within the area budget.

The design flow starts by constructing a dependence graph corresponding to each of the hot code segments. At each stage, the toolchain selects a dependence graph pair that has similar code patterns present across them, uses these similar code patterns to design a new dependence graph that represents both of the input dependence graphs. It then replaces the input 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 5(a)-(d) depicts the steps involved in the design of a QsCore that can support the sample hot code segments, *computeSum* (computes the sum of the first $\mathtt{n}_1$ numbers) and *computePower* ($\mathtt{c}^{2^{\mathtt{n}_2}}$).

4.1 Dependence Graph Generation

Our toolchain internally represents the application hotspots as PDGs [13], in which nodes represent statements and edges represent control and data dependences. The PDG representation of a hotspot is well-suited for finding matching code regions because it enables us to perform matching based on the program semantics rather than the program code text. Using PDGs, our toolchain eliminates false dependences, preserving only the "true" control and data dependences. Figure 5(a) shows the PDGs for two sample functions, *computeSum* and *computePower*. The solid edges represent control dependences and dashed edges represent data dependences.

The output of this stage is a pool of QsCore PDGs, where each PDG represents an application hotspot. The subsequent stages increase the computational power of the QsCores, enabling them to represent multiple application hotspots.

4.2 Mining for Similar Code Patterns

This stage mines the QsCore pool for code segments that are similar to each other, in that they can be effectively represented by a computation pattern, and hence, do not require multiple specialized cores to support them. The problem of finding similar code segments across hotspots reduces to finding similar subgraphs (subgraph-isomorphism) across their PDGs. Subgraph-isomorphism is a well-studied problem in the field of graph algorithms. We base our algorithm on the FFSM algorithm proposed by Huan et al. [18]. Below, we present a brief description of the FFSM algorithm and how we tailor it to the problem of finding similar code segments.

FFSM takes as input two graphs, G_1 and G_2, where every node in each graph has a *unique ID* as well as a *type* label. The algorithm considers two nodes for matching only if they have the same type. The algorithm begins by selecting a random node in G_1, say n_1, and finds a matching node (of the same type) in G_2, say n_2. Then, the algorithm tries to grow this matched subgraph by comparing the neighbors of n_1 and n_2 and performing other graph transformations on the matched subgraphs (refer to [18] for details). The algorithm returns when it cannot grow the matched subgraph any further.

The QsCore toolchain extends the FFSM matching algorithm to tailor it to the problem of finding similar code patterns in PDGs. First, instead of picking and matching nodes at random, our matching algorithm focuses on finding similar loop bodies. This behavior is desirable since most applications spend most of their execution

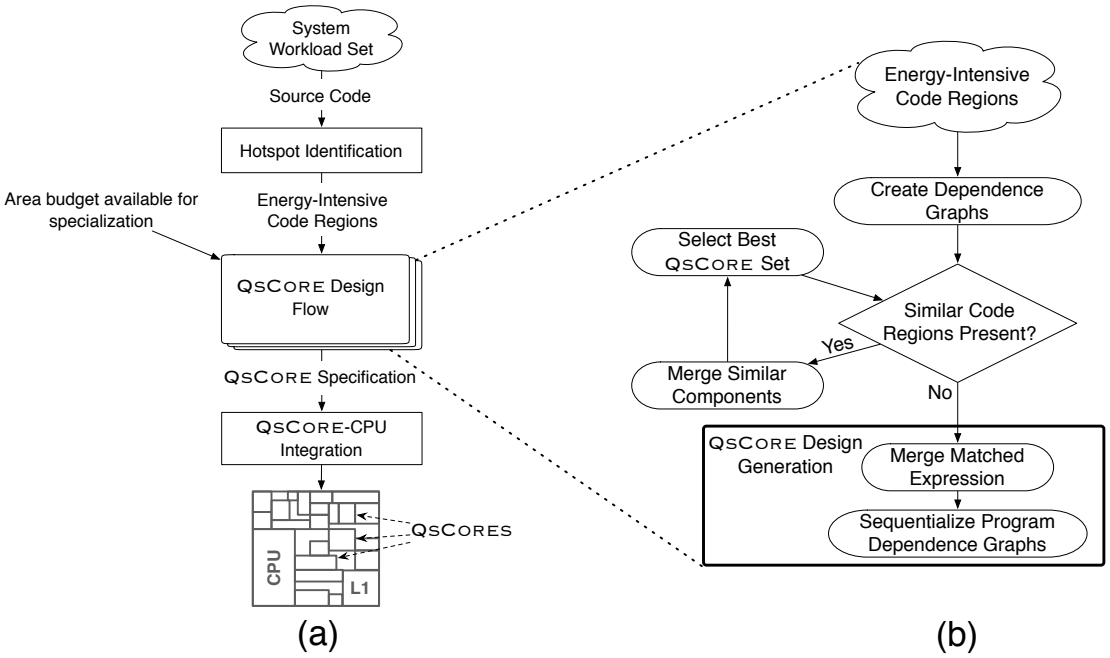

Figure 4: **Overview of QsCore-enabled system** The figure overviews the high-level design flow of a QsCore-enabled system (a) as well as the design flow for generating QsCores (b).

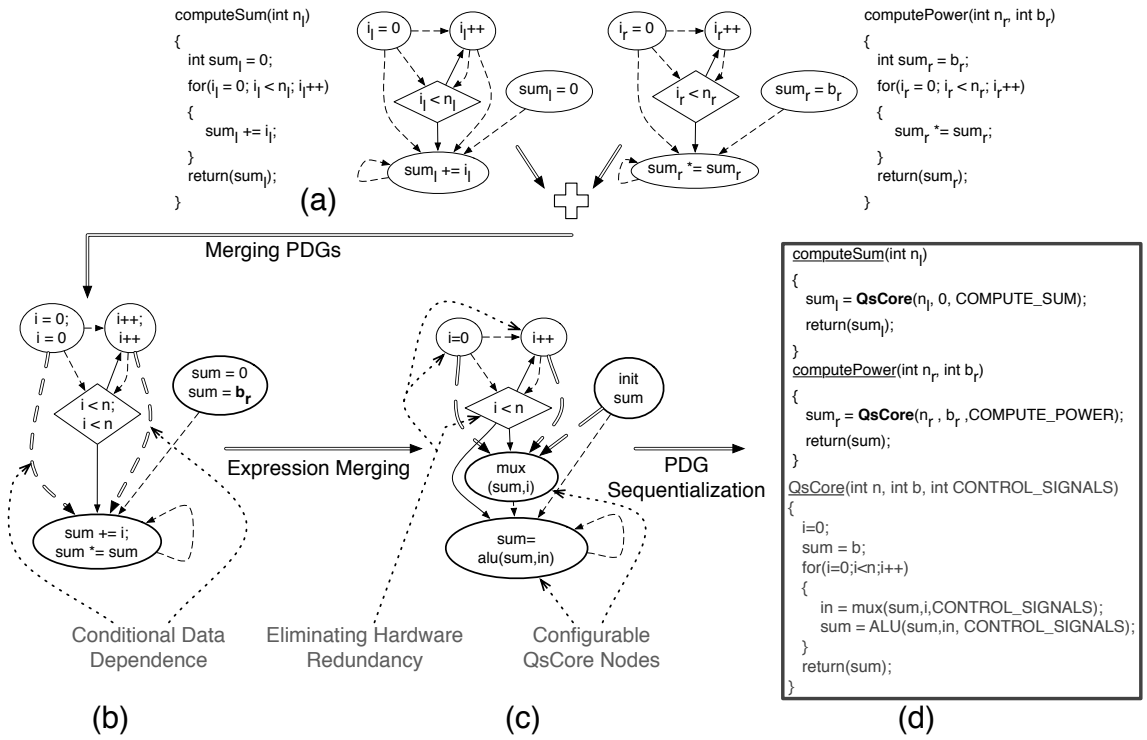

Figure 5: **QsCore design example** This example shows the different stages involved in converting two source code segments, *computeSum* and *computePower* (a), into a QsCore (d). The resultant QsCore can perform the functionality of the two input code segments as well as other functions such as $n!$ and c^{2^n}. In each PDG, solid lines represent control dependences and dashed lines represent data dependences.

time in loops. Second, we use the node type to encode the program structure, reducing the search space by pruning the "illegal" node matches. For example, all the nodes within a loop body should have similar node types and different from the node type of any nested loop nodes within that loop. The node type definition ensures that two nodes 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 have matching control/data edges.

For example, when mining for similar code patterns across the hotspots *computeSum* and *computePower*, shown in Figure 5(a), this stage maps the node $\texttt{sum}_\texttt{l} \texttt{+= i}_\texttt{l}$ in *computeSum* to $\texttt{sum}_\texttt{r} \texttt{*= sum}_\texttt{r}$ in *computePower*, among others (the mapped nodes are drawn and placed similarly in Figure 5(a)).

The output of this stage is a list of dependence graph pairs that have similar code patterns. For each of these similar dependence graph pairs, this stage also produces a mapping of the nodes across them.

4.3 Merging Program Dependence Graphs

This stage designs a new QSCORE PDG for each similar dependence graph pair that the previous stage produces. The new QSCORE dependence graph supports all computations that either of its input dependence graphs can represent. Below, we present the steps involved in designing the new QSCORE's dependence graph.

Creating QSCORE graph nodes The first step produces the QSCORE's dependence graph nodes based on the node mapping from the previous stage. For each pair of mapped nodes, we create a new node consisting of all the expressions contained in the mapped nodes. Figure 5(b) shows the nodes of the newly created dependence graph, where each node consists of an expression from *computeSum* and *computerPower*. For any nodes in the input dependence graph pair that did not get matched, we add them to the QSCORE's dependence graph without any modification.

Adding control and data dependence edges This step produces the data and control dependences of the new QSCORE's dependence graph based on those of the input dependence graph pair. The main challenge in this step is to ensure that the QSCORE's dependence graph is *sequentializable*, that is, there exists a sequential ordering of the nodes that respects all the data and control dependences.

In order to be sequentializable, the QSCORE's dependence graph should not have any circular data/control dependence. Any circular dependence in the new QSCORE's PDG result from node mappings that the previous stage generates. We eliminate circular dependences by removing the least number of node mappings that would break all dependence cycles.

The second invariant we maintain to preserve sequentializability is that each loop body in a QSCORE has a single entry point. To maintain this invariant, we add a dummy control node for any code regions with multiple entries.

Generating the variable set for QSCORES Finally, we create the variable set for the newly created QSCORE (Figure 5(b)). To do this, we build a one-to-one map between variables of the two input dependence graphs based on how they are accessed in the new QSCORE's nodes. For example, in the newly created QSCORE, variable $\texttt{i}_\texttt{l}$ in *computeSum* is mapped to $\texttt{i}_\texttt{r}$ in *computePower* because they are accessed similarly (same read/write access across mapped nodes) in the QSCORE's nodes (shown in Figure 5(b)). For these mapped variables, we create a new variable (i) for the QSCORE's variable set that replaces the usage of the original variables ($\texttt{i}_\texttt{l}$ and $\texttt{i}_\texttt{r}$). We propagate unmapped variables, such as $\texttt{b}_\texttt{r}$ in *computePower*, to the new PDG.

Figure 5(b) shows the new QSCORE's dependence graph that this stage designed by merging the dependence graphs of *computeSum* and *computePower*.

Because a QSCORE's dependence graph represents multiple computations, it contains additional node and edge attributes beyond those found in traditional dependence graphs. We describe these additional attributes below:

Node attributes A traditional dependence graph node contains a list of variables that are defined and used by that node. We extend this node attribute to contain a list of conditionally defined and used variables. For example, in Figure 5(b) the QSCORE node $\langle \texttt{sum += i; sum *= sum} \rangle$ conditionally consumes variable i because only one of its input code segments ($\texttt{sum}_\texttt{l} \texttt{+= i}_\texttt{l}$ from *computeSum*) uses i.

Edge attributes A QSCORE's dependence graph contains conditional data dependences. These conditional edges result from data dependences that are present only in a subset of computations that the QSCORE supports. For example, in Figure 5(b) there is a conditional dependence from source node $\langle \texttt{i++; i++} \rangle$ to destination node $\langle \texttt{sum += i; sum *= sum} \rangle$.

The result of the preceding transformation is a new QSCORE corresponding to each pair of similar PDGs from the previous stage. Each of these newly designed QSCORES 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 QSCORES reduce the area requirements compared to their input dependence graphs because they eliminate hardware redundancy across the input dependence graphs.

The final step of this stage is to, amongst all the QSCORES that this step produces, select the QSCORE that will provide the maximum benefit in terms of area savings and computational flexibility. Section 5 describes our heuristic for selecting the best QSCORE. Finally, this stage replaces the two input dependence graphs with the chosen QSCORE's dependence graph in the QSCORE pool.

At the end of this stage, the toolchain loops back to the second stage (Section 4.2) to find other potential PDG pairs for merging. Eventually, either the pool of QSCORES fits within the target area budget, or the QSCORE set becomes distinct enough that no substantial similarity can be found across them. At that point, the toolchain proceeds to generate the QSCORE specifications in C.

4.4 QSCORE Generation

The fourth stage of the toolchain sequentializes each dependence graph in the QSCORE pool to produce its corresponding QSCORE specifications in C. The two steps involved in this stage are generating valid C-expressions for each QSCORE PDG node and sequentializing the QSCORE's data and control dependences.

Merging expressions This step generates a valid C-expression corresponding to each node in a QSCORE PDG. The QSCORE dependence graph from the previous step can have multiple C-expressions in each node. For example, in Figure 5(b), QSCORE node $\langle \texttt{sum += i; sum *= sum} \rangle$ contains two expressions. To design a valid merged C-expression we build expression trees for each of the expressions in the QSCORE node. We merge the expression trees to form a single expression tree, and this merged tree directly translates to a valid C-expression (see Figure 6). Figure 5(c) shows the result of merging the expressions in Figure 5(b). This step reduces the area requirement and increases the computational power

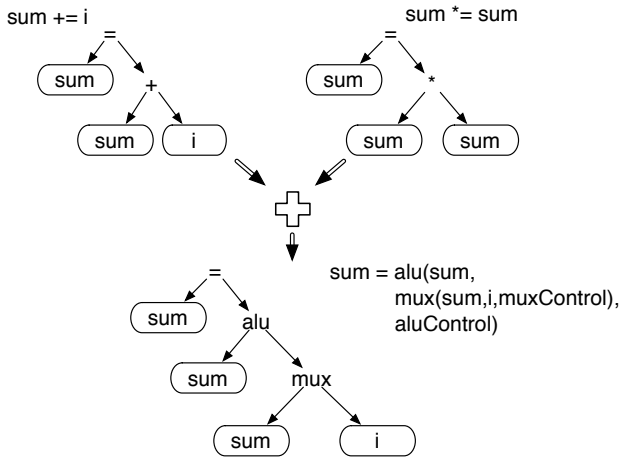

Figure 6: **Expression merging** Merging the two expressions at top produces a single expression that adds a new multiplexor operator and generalizes $+$ and $\times$ operators into an ALU with a separate control input.

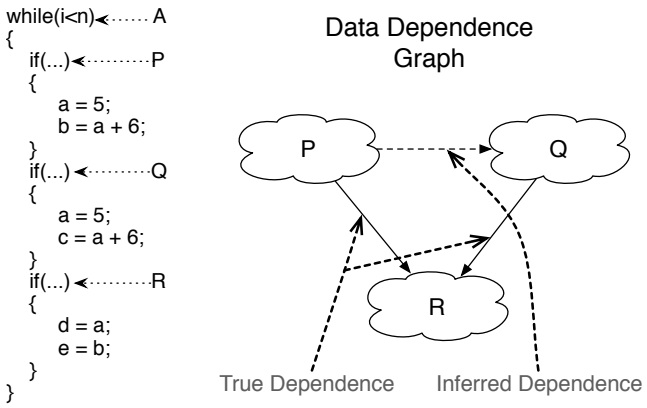

Figure 7: **Inferred dependence example** There is an inferred dependence edge from `P` to `Q` because no valid ordering of `P`, `Q`, and `R` orders `Q` before `P` (see Section 4.4).

of each QSCORE.

Sequentializing QSCORE PDGs PDGs are a inherently parallel representation of a program. However, in order to produce a C representation of the QSCORE, we need an ordering of nodes consistent with the control and data dependence edges. The algorithm for ordering the control edges in a QSCORE's dependence graph is similar to that for traditional dependence graphs, and previously proposed techniques for dependence graphs [13] work in our case as well. Sequentializing a QSCORE's data dependences is more challenging because of conditional data flow edges and conditionally defined-used variables in QSCORE nodes. This stage uses the following technique to sequentialize data dependences.

Our technique orders a QSCORE's nodes in the presence of conditional data dependences without employing backtracking (the computation time for backtracking-based techniques is exponential in the worst case). Our technique avoids backtracking by building *inferred dependences* between sibling nodes of a QSCORE's dependence graph. An inferred dependence is defined as follows: If parent node `A` has child nodes `P` and `Q`, then `P` has an inferred dependence on child `Q` if there exists a third 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 the only valid ordering among them is for child `P` to execute before child `Q`. Figure 7 shows an example. Any valid ordering of the sibling nodes `P`, `Q`, and `R` orders `P` before `Q`, even though there is no dependence between `P` and `Q`.

We use data dependences, inferred dependences, and use/def analysis to sequentialize the data dependences. Our sequentialization algorithm supports conditional data dependences, and the algorithm's computation time scales well with the size and complexity of the QSCORE PDG.

Figure 5(d) shows the C code for the QSCORE that our design flow generates for *computeSum* and *computePower*. Based on the value of the `CONTROL_SIGNALS`, this QSCORE can be configured to support the computations performed by both *computeSum* and *computePower*, as well as other operations such as `n!`, $\texttt{c} * 2^{\texttt{n}}$, and many others, by configuring the control lines to the `mux` and `ALU`.

4.5 Modifying Applications to use QSCORES

Our toolchain also modifies the application code to offload computations to the QSCORES at runtime. The toolchain does this by setting the QSCORE's `CONTROL_SIGNALS` input such that the QSCORE executes the computation needed by the calling application. For example, Figure 5(d) shows how the application code is modified to use the QSCORE. In this example, the *computeSum* function sends the function argument values as well as an additional argument to configure the `ALU` in the QSCORE to perform addition. The toolchain also inserts function stubs in the application code to query the runtime for the availability of a matching QSCORE (not shown in the figure for simplicity). If no matching QSCORE is present and available, then the application defaults to running the computation on the general-purpose processor.

5. QSCORE-SELECTION HEURISTIC

In the previous section we described our design methodology for building QSCORES by merging application hotspots. One challenge in this process is deciding which hotspot pairs to match, since a hotspot can potentially match well with many 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 many different alternatives to consider for merging the hotspots to form a final set of QSCORES. In this section, we present our heuristic to decide which hotspots to merge to make the best area-energy tradeoff at each step.

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

To evaluate the QSCORE b, we define a quality metric $Q_b = \frac{C_b S_b}{A_b P_b}$. To select a good set of QSCORES to build, we will need to compute Q_b for an enormous number of candidate QSCORES. Computing precise values for P_b and S_b in each case is not tractable

since it requires full-fledged synthesis and simulation. To avoid this overhead, we make the following approximations.

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 (see Section 7). Second, we estimate A_b based on the datapath operators and register counts. The resultant area model is able to estimate the area within 10% of the actual area on average with a standard deviation of 14.7%. Finally, we assume that power consumption is proportional to area. This approximation is valid for leakage power. For dynamic power, this is a conservative estimate because the dynamic power of QSCORES is proportional to the amount of logic that the target computation requires, and this amount does not increase with increased QSCORE area.

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

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 selection algorithm** The algorithm for deciding which QSCORE set to build.

We use $X \bowtie Y$ to denote the QSCORE that results from merging the QSCORES for X and Y. While estimating $A_{X \bowtie Y}$ is straightforward, estimating $C_{X \bowtie Y}$ is more challenging because $X \bowtie Y$ can implement code segments beyond just 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 QSCORES, B, we sum the value of Q' for each QSCORE. The goal of the QSCORE design flow is to maximize

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

Algorithm 1 contains the pseudocode for our QSCORE-selection heuristic that starts with a fully-specialized core for each computation and merges them to create the final set of QSCORES. It iteratively selects QSCORE pairs that maximize Q'_B (Line 2) and merges them to form a more general QSCORE that has greater computational power than its input QSCORES.

6. METHODOLOGY

In this section, we describe how our toolchain generates QSCORE hardware and how the QSCORES are integrated with a general-purpose processor. We also explain our simulation infrastructure for measuring QSCORE area, power, and performance.

QSCORE hardware generation The QSCORE hardware is generated using the C-to-hardware compiler developed in [27] that generates synthesizeable Verilog from QSCORE's source code. Our toolchain is built around the OpenIMPACT (1.0rc4) [20], CodeSurfer (2.1p1) [7], and ANTLRWorks (1.3.1) [2] compiler infrastructures and accepts a large subset of the C language, including pointers, `switch` statements, and loops with `break`/`continue`. Below, we briefly overview the C-to-hardware compiler, focussing primarily on how we extend the compiler to support QSCORES. For additional details on the compiler and specialized co-processor's hardware design, please refer to our previous work on conservation cores [27].

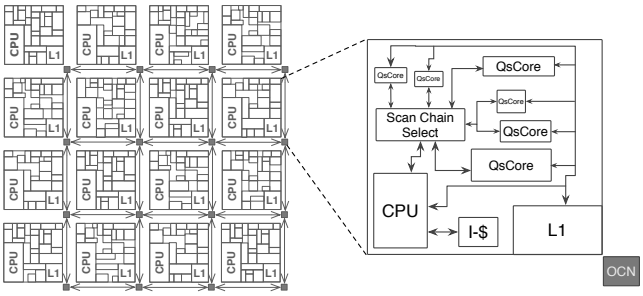

Figure 8: **QSCORE-enabled tiled architecture** A QSCORE-enabled system containing an array of tiles, each of which contains many QSCORES connected via scan chains to the general-purpose processor.

The compiler builds the hardware's datapath and control state machine based on the data and control flow graphs of the QSCORE source code in single static assignment [9] form. In addition to standard C data operators, our hardware compiler also supports QSCORE-specific operations such as `ALU` and *data-selector* (shown as `mux` in Figure 5(d)). The configurable `ALU`s in a QSCORE's dataflow graph are instantiated as functional units in the hardware's datapath. The data-selectors become *mux* operators. To optimize a QSCORE'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.

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

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 QSCORES.

Integrating QSCORES with a general-purpose processor In a QSCORE-enabled system (Figure 8), each tile contains a general-purpose processor, I-cache, D-cache, a set of QSCORES, and interconnect logic. On each tile, QSCORES are connected to a general-purpose processor via scan chains. The general-purpose processor passes input arguments, starts QSCORE execution, and reads QSCORE state via these scan chains. The scan chain interface is slow but scales well, allowing us to connect 10s of QSCORES to a single CPU. The QSCORES share the D-cache with the general-purpose processor enabling data transfer through the L1 cache as well.

Performance and power measurements We use a cycle-accurate simulator to evaluate the QSCORE-enabled system. Our simulator models the complete system including the general-purpose processor, QSCORES, interconnect, runtime, and a coherent memory system.

In order to measure QSCORE power usage, the simulator periodically samples execution and feeds these samples to Synopsys VCS and Synopsys Primetime. We use processor and clock power values for a MIPS 24KE processor in TSMC 45 nm reported in [27]. Finally, we use CACTI 5.3 [23] for I- and D-cache power.

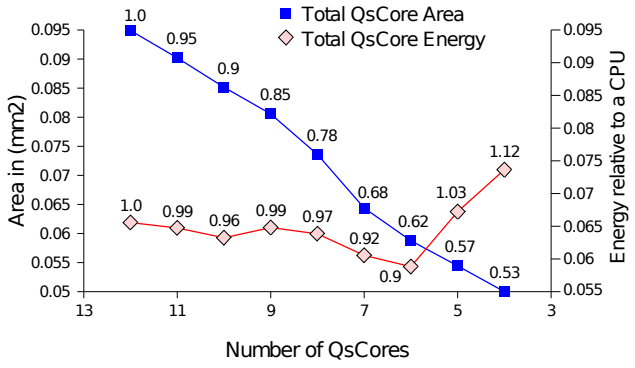

Figure 9: **Impact of varying computational power of QsCores for common data structures** The labels on the area and energy curves show the ratio of area and energy requirements of the QsCore set compared to that of fully-specialized circuits. The axes show the ratio of area and energy versus a general-purpose core.

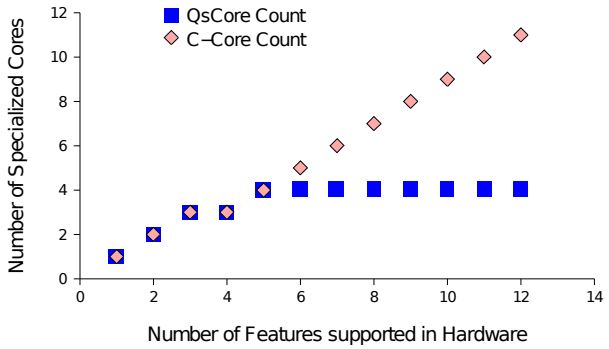

Figure 10: **Scalability of QsCore approach** The graph shows that relatively few QsCores can support multiple commonly used data structures, while c-cores need a new specialized processor for each distinct functionality.

7. RESULTS

In this section we first evaluate our heuristic for QsCore selection by comparing it to the optimal solution found via exhaustive search. Next, we demonstrate that designing relatively few QsCores is enough to support access operations for multiple commonly used data structures. Moreover, these QsCores are an order-of-magnitude more energy-efficient than general-purpose processors. When targeting a diverse workload (Table 1), the data show that QsCores continue to provide ASIC-like energy efficiency while significantly reducing the hardware resource requirements compared to fully-specialized logic. Furthermore, we demonstrate that the QsCore design methodology enables them to support legacy versions of their target applications as well. Finally, we evaluate the energy efficiency of a QsCore-enabled system compared to conventional processors.

7.1 Evaluating the Selection Heuristic

In this section we evaluate our QsCore selection heuristic (Algorithm 1) against exhaustive search. For this study, we use a microbenchmark suite containing code for computing eight mathematical functions: $\sum_{i=0}^{i<n} i$, $n!$, 2^n, a^{2^n}, $\sum_{i=0}^{i<n} a[i]$, $\prod_{i=0}^{i<n} i$, $\sum_{i=0}^{i<n} |a[i]|$, and counting the powers of 2 in an array.

To evaluate our selection heuristic, we build QsCores using Algorithm 1 and also using an exhaustive approach that designs all 255 possible QsCores that can arise from merging subsets of the eight 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 each case, our heuristic built the exact same set of QsCores as exhaustive search.

The low computational complexity and good performance of our selection algorithm allows it to scale to handle large workloads. This ability is critical because we expect a system's target workload to be very large for general-purpose systems.

7.2 QsCore Area and Energy Efficiency

In this section we evaluate the energy and area efficiency of QsCores and their ability to trade-off between area requirement and energy efficiency without compromising on the fraction of the system execution that they support.

Targeting an application domain First, we created QsCores for the `find`, `insert`, and `delete` operations of several data strucures – a linked-list, a binary tree, an AA tree, and a hash table. The results (see Figure 9) demonstrate that our methodology increases the computational power of QsCores, significantly reducing the area requirement while minimally impacting QsCore's energy efficiency. The X-axis plots the number of QsCores that the toolchain designs to support all the data structure access operations. The X-axis values decrease from left to right, so the number of computations that each QsCore supports increases from left to right. The left and right Y-axes show how our toolchain trades off between area and energy efficiency. The results show that just four QsCores are enough support all the data structure operations and that these QsCores are 13.5× more energy-efficient than our baseline general-purpose processor.

The next experiment shows that our toolchain can provide hardware support for an increasing number of functionalities without a corresponding increase in the number of required specialized cores. Figure 10 plots the number of computations supported in hardware against the required number of distinct QsCores. The data show that just four QsCores can support all these operations, while the fully-specialized approach (c-cores) would need to design eleven specialized cores. This 63% decrease in the required number of specialized cores allows us to closely integrate hardware support for a greater number of features in a processor's pipeline. For example, for our scan-chain-based interconnect design, QsCores reduce the interconnect overhead (measured as the number of connections between the CPU and all the specialized cores) by 54% compared to ASICs.

Targeting a general-purpose workload In this section, we design and evaluate QsCores for the hotspots from our diverse workload listed in Table 1. The results are shown in Figure 11. The X-axis plots the number of QsCores required to cover all the application hotspots. The left-most point on the X-axis corresponds to fully-specialized logic, and QsCore's computational power 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%. Also, QsCores re-

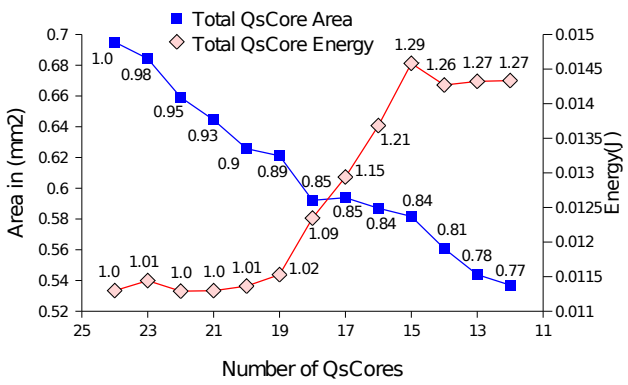

Figure 11: **Impact of varying the computational power of QsCores targeting a diverse workload** Merging QsCores reduces area requirements much more quickly than it impacts efficiency.

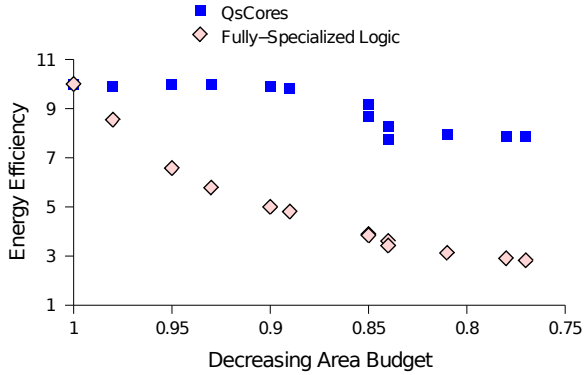

Figure 12: **QsCores vs. ASICs** QsCores can trade-off between energy efficiency and area requirements more effectively than fully-specialized logic because QsCores can save area by increasing computational power rather than removing functionality.

duce 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 subsequent merges, our approach ensures that energy-efficiency degrades gracefully with decrease in the total area budget. These results show that our toolchain can effectively reduce hardware redundancy while providing ASIC-like energy-efficiency.

Figure 12 demonstrates that compared to the recently-proposed conservation core approach [27], QsCores give up very little efficiency in return for substantial area savings. The X-axis plots area compared to implementing our workloads in fully-specialized logic. The Y-axis plots energy efficiency relative to our baseline processor. As the area budget decreases (left to right on the X-axis), our toolchain designs QsCores with greater computational power to ensure that the QsCores continue to cover all the application hotspots. This increase in the generality of QsCores enable them to provide significant energy efficiency even as the area budget decreases, unlike the fully-specialized logic that sees a 4× decrease in energy efficiency.

7.3 Backward Compatibility for QsCores

Finally, we evaluate QsCore's ability to support older application versions. Although hardware designers will build QsCores for the latest version of each target application, they may want the QsCores to support older versions as well. To provide this backward compatibility, our toolchain merges the older application versions with the latest version and builds one QsCore for them.

Previous work [27] presented reconfigurability mechanisms to support newer (and older) application versions. Figure 13 presents QsCore's energy-efficiency across application versions and compares it to that of the technique proposed in [27]. The Y-axis plots the energy efficiency compared to our baseline MIPS processor. The results show that QsCores provide significant energy efficiency 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, QsCores are almost as energy-efficient as the prior approach (less than 10% difference on average).

7.4 Application-Level Energy Savings

This section compares the energy efficiency and energy-delay of a QsCore-enabled system to that of a conventional tiled system. The QsCore-enabled system consists of 12 QsCores and targets the workload shown in Table 1. This experiment models the complete system including the overheads involved in accessing the runtime system and the overheads for offloading computations to the QsCores. Figure 14 summarizes the QsCore characteristics. Figure 15 shows that the QsCore-enabled system can provide significant energy efficiency improvements compared to the baseline system. The data show that, at the application level, QsCores save 45% of energy on average compared to a MIPS processor, and the savings can be as high as 79%. Also, QsCores reduce application energy-delay product by 46% on average. The energy savings for these QsCore-enabled systems are significant, and the energy efficiency will only improve as the transistor budget available for specialization increases.

8. 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 QsCores, specialized co-processors that can support multiple general-purpose computations and can provide significant energy efficiency compared to a general-purpose processor. Our toolchain synthesizes QsCores by leveraging similar code patterns that exist within and across applications. Given a target application set and an area budget, our toolchain can trade off between the computational power of individual QsCores and the area they require, while maximizing energy savings. The results show that our approach can distill the operator functions of multiple commonly used data structures down to just four distinct computation patterns. Moreover, the QsCores designed to support these patterns provide 13.5× more energy efficiency than a general-purpose processor. On a more diverse workload, our approach reduces the required number of specialized cores by over 50% and occupies 25% less area compared to fully-specialized circuits, while providing up to 25× more energy-efficiency than general-purpose processors.

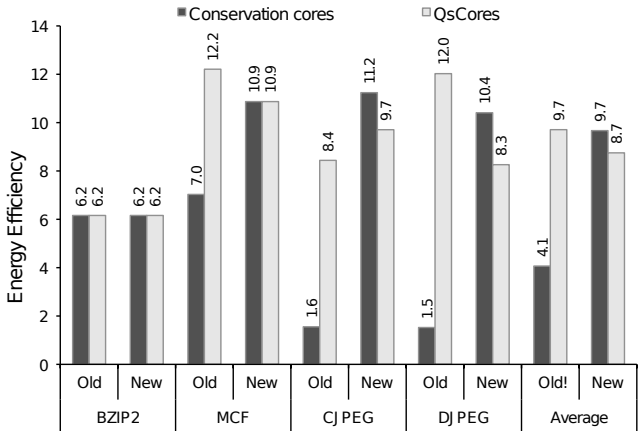

Figure 13: **Building QSCORES for backward compatibility** The energy-efficiency of the older application versions improves significantly (up to 7×) with minimal impact on that of newer ones.

Average QSCORE power	6.7 mW
Average QSCORE energy efficiency (compared to baseline CPU)	23.23×
Average execution time per QSCORE invocation	8951 cycles
Average invocation overhead	316 cycles
Average system execution coverage per QSCORE	4.41%
Average QSCORE area	0.041 mm^2

Figure 14: **QSCORE statistics for workload shown in Table 1**

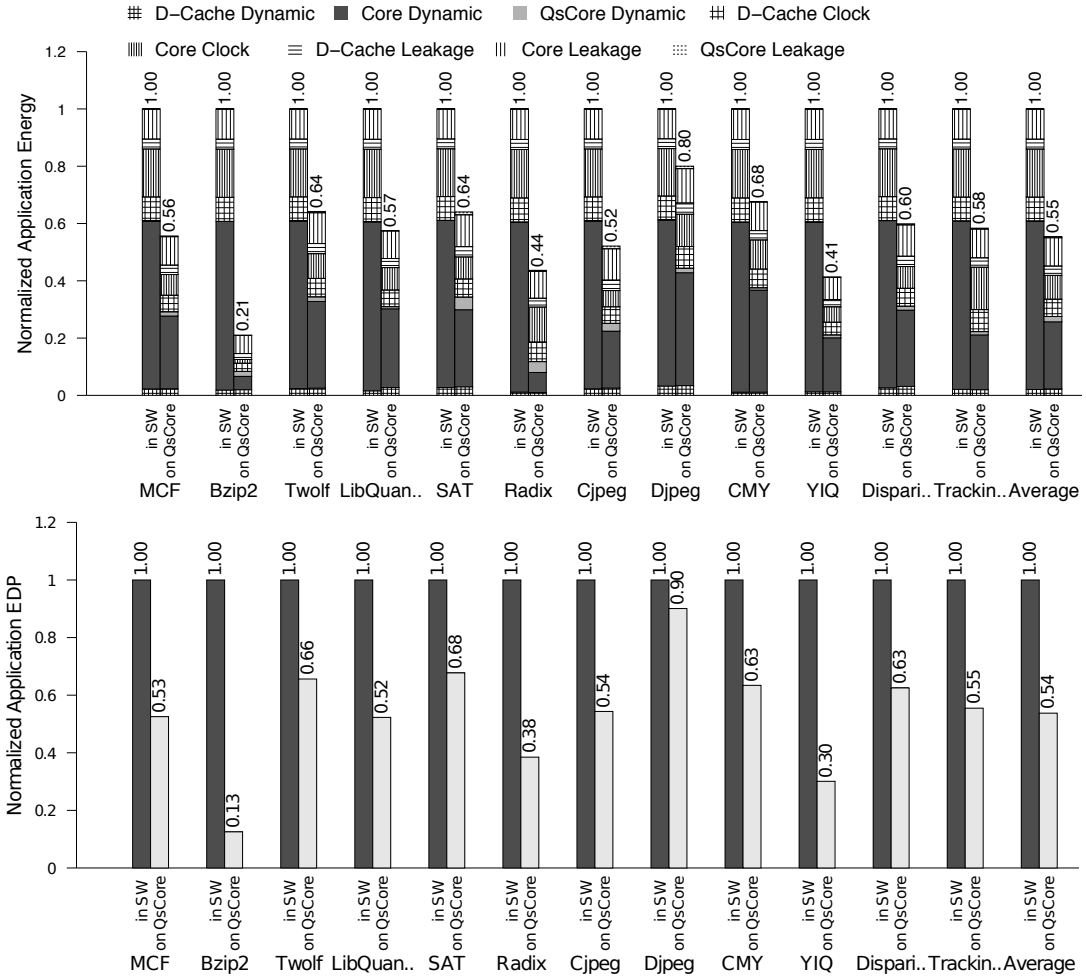

Figure 15: **QSCORE-enabled system energy efficiency** The graphs show the energy and EDP reductions by QSCORES compared to an in-order, power-efficient MIPS core ("SW"). Results are normalized to running on the MIPS core (lower is better). The QSCORES can reduce the application energy consumption and energy-delay product by up to 79% and 83%, respectively.

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. QsCores allow designers 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 a system's execution.

9. REFERENCES

[1] U. D. Bordoloi, H. P. Huynh, S. Chakraborty, and T. Mitra. Evaluating design trade-offs in customizable processors. DAC '09, pages 244–249, New York, NY, USA, 2009. ACM.

[2] J. Bovet and T. Parr. ANTLRWorks: An ANTLR grammar development environment. *Softw. Pract. Exper.*, 38(12):1305–1332, 2008.

[3] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *MICRO*, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

[4] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. *ISCA*, 0:389–400, 2008.

[5] N. Clark, B. Jason, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An architecture framework for transparent instruction set customization in embedded processors. ISCA, pages 272–283, Washington, DC, USA, 2005. IEEE Computer Society.

[6] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. *MICRO-36*, pages 129–140, 3-5 Dec. 2003.

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

[8] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. FPGA '04, pages 183–189, New York, NY, USA, 2004. ACM.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35. ACM Press, 1989.

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

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceeding of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

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

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[14] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. B. Taylor, and S. Swanson. GreenDroid: A mobile application processor for a future of dark silicon. In *Hot Chips*, 2010.

[15] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. B. Taylor. The GreenDroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31(2):86 –95, March/April 2011.

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

[17] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31:6–15, 2011.

[18] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, page 549, Washington, DC, USA, 2003. IEEE Computer Society.

[19] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *Computer*, 35(9):39–47, Sep 2002.

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

[21] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor. Efficient complex operators for irregular codes. In *HPCA '11: Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, pages 491–502, February 2011.

[22] SPEC. SPEC CPU 2000 benchmark specifications.

[23] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, Palo Alto, 2008.

[24] D. A. D. Tompkins and H. H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *SAT*, pages 37–46, 2004.

[25] F. Vahid, G. Stitt, and R. Lysecky. Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer*, 41:40–46, 2008.

[26] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. *IISWC*, 0:55–64, 2009.

[27] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, pages 205–218, New York, NY, USA, 2010. ACM.

[28] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC*, pages 184–188. ACM Press, 2001.

[29] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *HPCA*, pages 277–288, 2009.