Automating the Design of Embedded Domain Specific Architectures using Stall Cycle Analysis (SCA)

Abstract

The design of domain specific architectures (DSAs) deployed in today’s embedded systems involves a sequential process that requires significant designer knowledge, experience, and time to arrive at a suitable architecture for the target application suite. The subsequent costs incurred conflict with the ever increasing functionality and complexity of embedded applications, and the short time to market requirements for embedded devices. In this study, we propose to create an automatic design space exploration tool that takes an initial architecture and then employs an iterative technique known as 'Stall Cycle analysis' (SCA) to arrive at a performance-energy pareto-optimal design for a given constraint (for example, minimum area). SCA makes the observation that execution cycles of a program consists of cycles of pure computation and stall cycles that occur due to the presence of bottlenecks. Compilation and architectural simulation can be employed to analyze these bottlenecks and provide insight into architectural avenues (adding or removing resources) for further performance improvements (or energy reduction). This iterative technique reduces the search space (from ≥1000 points to tens of points) and its utility is demonstrated by design case studies for three application domains: face recognition, speech recognition, and wireless telephony algorithms. As compared to manual designs optimized for a particular metric, SCA automates the design of performance-energy optimal DSAs for minimum energy-delay product (17% improvement for wireless telephony), minimum area (75% smaller design for face recognition), or maximum performance (38% improvement for speech recognition). To our knowledge, this is the first study that combines compilation and architectural simulation to design compiler friendly DSAs, thereby significantly reducing DSA design time (months to hours).

Keywords: Embedded systems, application and domain specific architectures, design space exploration.
1 Introduction

The rapidly growing deployment of embedded systems and the dynamics of the competitive market necessitate the creation of energy efficient microprocessors in small form factors (minimal silicon area). Every new generation requires new functionality and higher performance. Given the stringent time-to-market requirements, the arrival of complex multimedia applications like face recognition, speech recognition, and wireless telephony protocols have exacerbated the problem. These applications are characterized by intertwined sequential and parallel code (called kernels) phases. All these problems make it a challenge for system designers to create performance-energy-area optimal designs.

A popular solution is to employ a heterogeneous multiprocessor (figure 1), where a general purpose processor (GPP) executes the sequential code and a DSA executes the parallel kernels within the application domain. This approach [8, 12, 14] has been successful in designing energy efficient processors for many application domains. Domain specific architecture (DSA) design is currently a complex process that involves significant user time and experience in understanding the application and arriving at an optimal design. The problem with this methodology is that design optimality is strongly dependent on user experience, knowledge of the application domain, and awareness of the architectural design space. Given the complexity of designs today, this process is error prone, time-consuming, and may be infeasible for large search spaces.

This study explores a novel exploration algorithm to automate the design of performance-energy optimal DSAs for constraints like minimum area, maximum performance, etc. This iterative algorithm is termed
‘Stall Cycle Analysis’ (SCA) and is based on the observation that total execution cycles of a program consists of cycles used for pure computation and stall cycles that occur due to the presence of performance/energy bottlenecks. Understanding these bottlenecks can provide us with insight for further performance improvements (energy reduction). This observation is applied during compilation and simulation and is used to guide the exploration process for designing optimal design points in the architectural phase. As opposed to other studies [10], the design automation process also guarantees optimized compilation and to our knowledge, this is the first study that employs compilation and architectural simulation in a symbiotic manner for design space exploration.

SCA is employed in two phases on a given test architecture. Bottleneck diagnosis (BD) associates stall cycles in the compilation schedule (or simulator) to bottlenecks for performance (energy). Examples of such bottlenecks are insufficient parallelism in hardware, resource contention, resource starvation, routability issues, etc. During diagnosis, we classify and analyze bottlenecks to investigate various architectural solutions. This output of this phase can be used by the system designer in making further architectural choices. In our framework, this information is fed as input to the design selection (DSel) phase. Those solutions that can potentially lead to improved performance (reduced energy) are identified and we add (Dilation) or remove (Thinning) resources to generate the next test architecture. This process is repeated in an iterative manner to arrive at performance-energy pareto-optimal designs for different constraints.

The different phases of the exploration tool are discussed in section 3. Since our design case studies involve the design of ‘ASIC-like’ DSAs, we provide a brief overview of our architectural and compilation methodology in section 2. The simulation methodology is presented in section 4 followed by design case studies in 5. In summary, the contributions of this study are:

- **Bottleneck Diagnosis (BD)**, a tool that identifies architectural avenues for performance improvements (or energy reduction) and helps systems designers in making intelligent design choices.

- **Stall Cycle Analysis (SCA)**, a robust exploration algorithm that efficiently combines optimized compilation and architectural simulation for the design of energy efficient soft real time embedded systems.

- **Design case studies** that employ SCA to automate the design of performance-energy optimal DSAs for minimum energy-delay product, minimum area, and maximum performance. As compared to
time-consuming manual design, SCA provides a richer and better set of design choices to designer while reducing design time significantly (hours on a 1.6 GHz AMD Athlon PC).

2 Framework Description

The automated design framework consists of two major components upon which the exploration algorithm is implemented: architectural design and optimized compilation. Recent studies [12, 15] have shown that an architectural design approach that employs a programmable ‘ASIC-like’ back-end supported by a distributed memory front-end can deliver the performance and energy requirements for embedded face recognition and speech recognition. In this study, design space exploration is performed over such an architectural design space. The performance of these architectures are dependent on the efficiency of the centralized program controlled interconnect (figure 2) that co-ordinates data movement across the memory system and the execution back-end. Since data movement across the interconnect is scheduled at compile time, a brief discussion of the compilation flow is necessary to understanding the complete framework.

**DSA Design Methodology** The memory architecture of the DSA is designed to support the data access, communication, execution unit and control characteristics in the application suite. Specifically, our memory system consists of hardware support for multiple loop contexts that are common in the face recognition suite. In addition, the hardware loop unit (HLU) and address generation units (AGUs) provide sophisticated addressing modes which increase IPC since they perform address calculations in parallel with execution unit
operations. In combination with multiple dual-buffered SRAM memories, this results in very high memory bandwidth sufficient to feed the multiple execution units.

The architectural model is effectively a long word (VLIW) approach but each bit in our program word directly corresponds to a binary value on a physical control wire. This very fine grained approach was inspired by the RAW project [18]. This allows multiple execution units to be chained together to provide "ASIC-like" computation flows by controlling data movement through the communication fabric between execution units, pipeline registers, and the global interconnect (figure 3). The result is a programmable DSA whose energy-delay characteristics approach that of an ASIC while retaining most of the flexibility of more traditional programmable processors. Figure 1 illustrates the complete system architecture. The heterogeneous system consists of a GPP that executes the sequential setup code while the DSA performs kernel acceleration. Other work [13, 7] has demonstrated the effectiveness of a similar approach for the speech recognition and the wireless telephony domains. However, these studies required that the architectures be manually scheduled at the machine language level. As a reference comparison to our exploration technique, table 1 shows the configurations and the energy-delay product (EDP) efficiency (as compared to an Intel XScale processor) of the designs proposed in the above studies for each of the three application domains.

**Compilation Methodology**  Program scheduling for this architecture is a complex task for several reasons. The program is effectively horizontal microcode which requires that all of the control points (register output or load enables, execution opcodes, multiplexer select lines, address context updates, etc.) be concurrently
<table>
<thead>
<tr>
<th>Application Domain</th>
<th>Memory system</th>
<th>Back-end</th>
<th>Interconnect (XScale)</th>
<th>EDP improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face Recognition</td>
<td>3 8KB SRAMS, 6 AGUs, 3 HLU contexts</td>
<td>3 INT + 4 FPU + 1 RF</td>
<td>single level</td>
<td>80x</td>
</tr>
<tr>
<td>Speech Recognition</td>
<td>3 8KB SRAMS, 6 AGUs, 4 HLU contexts</td>
<td>4 INT + 4 FPU</td>
<td>single level</td>
<td>150x</td>
</tr>
<tr>
<td>Wireless Telephony</td>
<td>3 2KB SRAMS, 1 4KB SRAM, 4 AGUs</td>
<td>8 INT</td>
<td>two level</td>
<td>100x</td>
</tr>
</tbody>
</table>

Table 1. Best manual design configurations for the three application domains (INT is integer functional unit and FPU is floating point functional unit).

![Compilation flow](image)

**Figure 4. Compilation flow employed in the framework**

scheduled in space and time to create efficient, highly parallel and pipelined flow patterns. To solve this problem, we have created the CoGenE compiler that employs Integer Linear Programming (ILP) based interconnect-aware scheduling techniques to map the kernels to the DSA. The overall back-end compilation flow is illustrated in 4. After preliminary control and data flow analysis is performed, we identify the inner most loop and load the initiation interval into the hardware loop unit (HLU) and it provides support for modulo scheduling. After register assignment, we perform Integer Linear Programming (ILP) based interconnection scheduling followed by post pass scheduling to resolve conflicts that may arise due to ILP scheduling.

3 Stall Cycle Analysis (SCA)

The idea of stall cycle analysis is based on the observation that total execution cycles of a program consists of two parts: i) pure computation cycles, where useful work is performed by the processor in executing the program, and ii) stall cycles, representing unused work and manifests itself as overhead that is detrimental to performance/energy dissipation of the system. Stall cycles occur due to many different kinds of bottlenecks within the system. Quantifying and analyzing these bottlenecks can help the system designer in understanding the overheads and in investigating architecture design choices that improve the performance or energy dissipation of the system. This process, called as Bottleneck Diagnosis (BD) classifies the overheads in the system into different categories and quantifies their contribution to performance (energy) issues.
This provides us with an understanding of potential avenues for improvement. In our study, the output of this phase is used by the design selection (DSel) process to identify the major culprits to performance degradation or energy dissipation. Once the culprits are identified, we employ an architectural solution to reduce their impact on performance or energy. Our framework provides options to dilate resources, where we add architectural units or increase the size and functionality of an existing unit and then observe the impact of dilation. On the other hand, resource thinning can be employed to remove some units to improve the energy characteristics of the processor. These phases are run iteratively to arrive at performance-energy optimal design points in the architecture space.

3.1 Bottleneck Diagnosis (BD)

To aid the process of overhead classification and quantification, we collect the following statistics about the application suite by continuously monitoring the compiled code. Parameters outside the scope of the compiler (SRAM hit rate, Scratch RAM utilization rate, etc.) are collected through cycle accurate simulation of the test architecture.

- **Function unit utilization rate**: The fraction of total execution cycles in which a particular function unit is utilized. Both per unit and total utilization rate metrics are collected.

- **Register file use rate**: The fraction of total execution cycles in which the centralized register file is utilized. In some cases (speech recognition), the presence of a scratch memory precludes the register file.

- **Function unit contention**: Fraction of cycles in which instruction execution is delayed due to lack of functional unit resources.

- **Interconnect contention**: The fraction of cycles in which contention for interconnect negatively impacts data bypass. This can be quantified by observing the multiplexer utilization rates during the post pass scheduling phase of compilation.

- **SRAM miss rate**: This simulation monitored metric collects the hit rate statistics for the input and output SRAMs.

- **AGU utilization rate**: Fraction of cycles for which the address generator units are employed.
• *Total execution time*: Total execution time in cycles for completion of the program

• *Total energy dissipation*: Total energy dissipated in executing the program in Joules.

The above statistics can be used to classify the different categories of overheads that lead to stall cycles in a given architecture. The major bottlenecks that are detrimental to performance or energy are discussed below.

**Back-end Starvation**  This bottleneck arises due to the inability of the memory system to efficiently deliver data to the back-end. A high SRAM miss rate can be a major performance bottleneck and the solution is to increase the size of SRAMs. In some cases, contention in the ports or lack of hardware support for multiple loop contexts may lead to back-end starvation.

**Insufficient Parallelism in Hardware**  This bottleneck arises because there are more independent instructions that can be issued within a cycle. This will be observed as very high function unit contention or a combination of high function unit utilization and high interconnect contention. An easy solution is to increase the number of functional units in the back-end which also incurs an additional increase in interconnects (multiplexers and pipelined registers).

**Unit Starvation**  An increase in the number of function units may lead to under-utilization and may be an energy bottleneck. Thinning of resources will alleviate this problem.

**Routability**  Cycles may be wasted due to the lack of a route between producer and consumer. This will necessitate storing the value in either a pipelined register or the centralized register file and is both a performance and energy bottleneck. This can be observed by very high contention on the interconnect. The solution is to either increase the width of the multiplexer which may lead to operating frequency issues or to increase the number of interconnect levels between two function units. Due to frequency limitations, this may incur an additional cycle for data transfer and the value may be stored in a pipelined register. A side effect is that it provides a hierarchical interconnect and clustering of execution units.
**Resource Contention**  High contention on a functional unit implies cycles wasted due to waiting on the functional unit. While it is easy to quantify, any of the above bottlenecks can manifest itself as resource contention and careful observation of the other metrics is required to understand this bottleneck.

Thus, bottleneck diagnosis employs the above program statistics to measure the different sources of overhead in a test architecture and suggests different solutions to alleviate their impact in the next test design. This tool can also be employed by a system designer to make further design choices in the architecture space. In this study, we use the output of BD as input to the design selection process. During design selection, we classify the different architectural solutions, giving importance to those solutions that potentially have a greater impact on system performance or energy dissipation. For applications like face recognition, which have an inherent real time performance demand, solutions that can improve performance are classified higher than solutions that impact energy dissipation. Once a particular performance target is reached, we give more importance to energy efficient designs.

### 3.2 Design Selection (DSel)

The process of design selection can choose the next test architecture by either alleviating all bottlenecks in one step or by alleviating the biggest bottleneck in each iteration. We choose to change one or two architectural feature(s) during each iteration. Our process alleviates bottlenecks to memory before removing the bottlenecks in the back-end. Making too many changes in architectural features at every step can lead to a feedback loop where the exploration algorithm is stuck in a minimum and this causes instability and incompleteness in the algorithm. Hence, our algorithm investigates one solution at a time. Below, we investigate different solutions to each of the bottleneck categories in terms of changes in the memory subsystem, the execution back-end, and the interconnection network. It is noteworthy that changes in one subsystem may necessitate changes in the other to facilitate ‘ASIC-like’ execution flows.

**Distributed Memory Selection**  The architectural template consists of a distributed memory system that delivers data to the execution back-end with hardware support for loop contexts and associated AGUs that generate the different address patterns in the application domain. Problems in the memory system can be observed with a high back-end starvation rate (high SRAM miss rates). In the first step, the problem is alleviated by employing an increase in the size of the input and output SRAMs until they start returning
<table>
<thead>
<tr>
<th>Component</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data width</td>
<td>15, 32, 64 (bit)</td>
</tr>
<tr>
<td>SRAMs (input, output, and scratch)</td>
<td>1, 2, 4, 8, 16, 32, 64 (KB) each</td>
</tr>
<tr>
<td>Ports (SRAMs and RF)</td>
<td>1, 2, 3 each</td>
</tr>
<tr>
<td>Hardware loop unit contexts</td>
<td>1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>AGUs</td>
<td>1-8 (increments of one) per SRAM</td>
</tr>
<tr>
<td>Register file</td>
<td>8, 16, 32 entries</td>
</tr>
<tr>
<td>Functional unit mix</td>
<td>multiplier, adder, compare, etc.</td>
</tr>
<tr>
<td>Functional unit number</td>
<td>1-8</td>
</tr>
<tr>
<td>Instruction word width</td>
<td>2-15</td>
</tr>
<tr>
<td>Interconnect Width</td>
<td>2-5</td>
</tr>
<tr>
<td>Interconnect levels</td>
<td>1-3</td>
</tr>
</tbody>
</table>

**Table 2. Design space for exploration**

diminishing gains in performance. In some cases, a choice may be made to reduce their sizes for greater energy savings at a minimal performance loss. The second biggest bottleneck to memory performance in embedded applications arises due to the fact that computations in multi-level loops entail complex indirect accesses (Example: \(Z[i] = Z[i-1] + \sum_{j=0}^{m} X[j] \times Y[W[j]]\)). In such cases, loop variable accesses compete with the actual array data accesses and this degrades performance. SCA chooses to increase the number of AGUs associated with each SRAM to alleviate the problem. Additionally, in cases where we observe very high AGU utilization rates but very little improvements in performance, we choose to increase the number of supported loop contexts. The HLU automatically updates the loop indices in the proper order and reduces loop variable accesses to the SRAMs. Increasing the number of contexts increases the area, complexity, and power dissipation while providing minimal performance improvements beyond a certain number and hence, SCA attempts to keep this number to a minimum. Table 2 illustrates the design space available for the complete framework.

**Execution Back-end Selection** During early design space exploration, we observe a high utilization rate for some of the function units along with high contention on the function units. Insufficient parallelism or resource contention occurs primarily because the initial test architectures consist of one functional unit of each type. Those function units that are not employed are either removed or reduced in number while function units with high utilization and contention are dilated (increments of one for each type). An increase in the number of functional units entails an increase in the length of the VLIW word due to the nature of the architecture and this further increases power dissipation in the instruction cache and interconnect. Hence, thinning may be necessary if energy constraints are not met.
The initial test architecture may contain a centralized register file in one of the functional unit slots. In our architectural methodology, the register file is used more as a temporary placeholder for data in the event of routing difficulties. Studies [15, 12] have shown that the availability of a register file reduces the problem of scheduling over the centralized interconnect and reduces pressure on the scratch SRAM. The register file use rate metric is employed to either dilate or thin the size of the register file.

**Interconnection Network Selection** The centralized interconnect orchestrates data movement across the memory and the execution back-end and is critical to the performance of the complete system. Data movement and life time can be controlled in a fine grained manner by programming the multiplexers and the load enable on the pipelined registers. A high utilization rate on a multiplexer and resource contention on the associated function units is a very good indicator of routability issues. In the first step, we increase the width of the multiplexer while making sure the frequency target is met. An alternative choice would be to add one more level of multiplexing and offers the same functionality. In the advent of a frequency conflict, we add an additional cycle by introducing a pipelined register. This increases the complexity of compilation and may lead to infeasible schedules in some cases. In such events, the algorithm returns to the previous design point.

### 3.3 SCA Exploration Algorithm

The SCA algorithm employs BD and DSel to explore the design space for different application domains. The initial starting point employs a memory subsystem with a one KB input, scratch, and output SRAM with one AGU for each SRAM. We begin with a two way execution back-end consisting of one integer and one floating point functional unit. The search occurs in a number of sequential steps. The memory subsystem is optimized before the execution back-end. Finally, we optimize the interconnect network and this serves to balance the memory and the back-end.

- **Program Statistics**: In the first step, we collect program statistics to measure the various categories of bottlenecks. This information is collected during compilation and from the simulator.

- **Memory Optimization**: Observe SRAM miss rates, AGU utilization and improvement/degradation in memory IPC and energy dissipation compared to previous iteration. Dilate or thin SRAM size. For
high AGU utilization yet marginal performance improvements, increment number of AGUs for that SRAM. Further, preserve or pair down AGUs while increasing the number of loop contexts supported by HLU. Compile and simulate the next test architecture.

- **Execution Back-end Optimization**: Observe functional unit utilization. Remove unused functional units (integer programs like wireless telephony). For very high resource utilization and interconnect contention on a unit type, increase the number and vice versa. Compile and simulate the next test architecture.

- **Interconnect Optimization**: Observe interconnect utilization and contention for each of the units and SRAMs. Increase width in case of routability issues. In case of a frequency issue, increase multiplexing levels with pipelined registers. This may result in infeasible schedules from the compiler. Revert back to previous test architecture in such cases. Compile and simulate the next test architecture.

This process is repeated in an iterative manner until we reach all performance, energy, and area constrained designs for the application domain. In cases where the algorithm cannot provide a feasible design, the BD tool returns to the last feasible iteration, reports infeasible design choices and asks for user input to test further choices.

4 Methodology

**From Trimaran to CoGenE** The application suite is factored into sequential code that runs on the GPP and streaming code in C, which serves as input to our compiler framework. The Trimaran compiler (www.trimaran.org) was the starting point for the compiler development. It was chosen since it allows new back-end extensions, and because its native machine model matches our architectural approach. The infrastructure is flexible enough that each stage of the back end may be replaced or modified to suit one’s needs. Significant modifications were needed to transform Trimaran from a traditional cache-and-register architecture to meet the needs of our fine-grained cache-less clustered VLIW (Very Long Instruction Word) approach. The result is a compiler that takes a streaming code written in C and code generation is parameterized by a machine description file which specifies: the number of clusters, the number and type of functional units in each cluster, the number of levels of inter- and intra-cluster interconnect, and the individual multiplexer configurations. We developed a new back-end code generator that is capable of generating object
code for the coprocessor architecture described by the architecture description file. The code generator includes a modified register allocator that performs allocation for multiple distributed register files. Since the compiler controls the programming of the multiplexers and the liveness of the pipeline registers, register allocation is inherently tightly coupled with interconnect scheduling. Hence, we introduce a separate interconnect scheduling process after register allocation and our scheduling scheme is based on integer linear programming techniques. The effect of compilation can be analyzed with the cycle-accurate simulator and it estimates power using high level parameterizable power models. Our power models (90 nm node) employ analytical models from Wattch [1] for all predictable structures and empirical models similar to [16, 17] for complex structures like the hardware loop unit (HLU). Our area estimates are obtained from Synopsys MCL and design compiler scaled to 90 nm.

Benchmarks Our benchmarks consists of seven kernels from face recognition, three kernels from speech recognition, and six kernels from wireless telephony domains. The face recognition kernels constitute the different components in a complete face recognition application. To increase the robustness of the study, we employ two fundamentally different face recognition algorithms. The EBGM algorithm is more computationally intensive as compared to the PCA/LDA recognition scheme. All the face recognition kernels were obtained from the CSU face recognition suite [3]. The speech recognition application consists of three phases that contribute to 99% of total execution time: preprocessing, HMM, and the Gaussian phase [12]. The kernels from the wireless domain include predominant operations like matrix multiplication, dot product evaluation, determining maximum element in a vector, decoding operations like rake and turbo, and the FIR application. A description of the benchmarks are provided in table 3.

Evaluation Metrics we employ degree of pruning and exploration time as metrics to evaluate the efficiency of design space exploration. Given the complete design space, degree of pruning gives us a measure of the reduction in the size of the exploration space. The total time for exploration evaluates the time taken to arrive at optimal design points for various constraints. To effectively compare the performance of different architectures, we employ throughput measured in terms of the number of input frames processed per second. We employ the energy-delay product as advocated by Horowitz [5] product to compare the efficiency of different processors since both energy and delay for a given unit of work are conflicting constraints for
Table 3. Benchmarks and Description

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face Recognition</td>
<td></td>
</tr>
<tr>
<td>Flesh Toning</td>
<td>pre-processing for identifying skin toned pixels</td>
</tr>
<tr>
<td>Erode</td>
<td>First phase in image segmentation</td>
</tr>
<tr>
<td>Dilate</td>
<td>Second phase in image segmentation</td>
</tr>
<tr>
<td>Viola Detection</td>
<td>Identifies image location likely to contain a face</td>
</tr>
<tr>
<td>Eye Location</td>
<td>Process of locating eye pixels in a face</td>
</tr>
<tr>
<td>EBGM recognition</td>
<td>Graph based computationally intensive matching</td>
</tr>
<tr>
<td>PCA/LDA recognition</td>
<td>Holistic face matching</td>
</tr>
<tr>
<td>Speech Recognition</td>
<td></td>
</tr>
<tr>
<td>Preprocessing</td>
<td>Normalization for further processing</td>
</tr>
<tr>
<td>HMM</td>
<td>Hidden Markov Model for searching the language space</td>
</tr>
<tr>
<td>GAU</td>
<td>Gaussian probability estimation for acoustic model evaluation</td>
</tr>
<tr>
<td>Wireless Telephony</td>
<td></td>
</tr>
<tr>
<td>Vecmax</td>
<td>Maximum of a 128 element vector</td>
</tr>
<tr>
<td>matmult</td>
<td>Matrix multiplication operation (integer)</td>
</tr>
<tr>
<td>dotp, square</td>
<td>Square of the dot product of two vectors</td>
</tr>
<tr>
<td>Rake</td>
<td>Receiving process in a wireless communication system</td>
</tr>
<tr>
<td>Turbo</td>
<td>decoding received encoded vectors</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse response filter</td>
</tr>
</tbody>
</table>

the architect and circuit designer.

5 Results

Embedded designers typically attempt to design a DSA to meet a given performance and energy budget and then optimize the area for the design. SCA is employed in a similar manner and attempts to search through the design space for a set of designs that meet the minimum performance and energy budgets. These designs are then investigated for minimum area, best energy-delay product or highest performance. In the first case study, we evaluate the impact of SCA in designing an optimal domain specific architecture for the face recognition domain. The seven benchmarks required for face recognition are fed as inputs to the framework for iterative exploration. We then discuss the design of optimal DSAs for speech recognition and wireless telephony. Each of the Energy-Delay optimal DSAs are compared to the best manual designs from previous studies and also to industrial design points (wherever applicable) for performance and energy dissipation.

5.1 CASE STUDY I: Architecture for Embedded Face Recognition

Figure 5 shows the design points explored by SCA for the seven benchmarks in the face recognition suite. The plot on the left shows the throughput (frames processed/second) and energy dissipation (mj/input) for each of the design points. We start with the initial design point( 2 way VLIW with 1 KB input and output
SRAMs, 1 AGU/SRAM, No HLU) and observe that its throughput is about five times slower than a real time performance of 5 frames/sec. The minimum real time performance is shown by the vertical line (normalized to 1) and all points to the left of the line do not meet the performance requirements necessary for real time face recognition. As per the optimization algorithm, the memory subsystem is optimized first and SCA successively increases the size of the SRAMS(input and output) to 8 KB with up to 2 AGUs/SRAM. At this point, throughput starts to saturate and this is indicated by a low miss rate and very utilization on AGUs and the interconnect. SCA then investigates the effectiveness of a HLU and successively increases the number of contexts to improve performance. Once memory optimization is complete, SCA then dilates resources in the back-end and we observe significant increases in performance. A configuration with a 6-way VLIW back-end (3 INT + 3 FPU) achieves the minimum required performance for face recognition. All design points to the right of this configuration meet the performance demands, and SCA investigates these designs for energy requirements. The horizontal line shows the maximum allowable energy budget (corresponding to a design that provides approximately one order of magnitude energy improvement over the energy efficient XScale processor) for a DSA in a heterogeneous system. All designs below this envelope that meet the required performance belong to the 'quadrant of feasible or acceptable designs'. SCA further dilates the back-end by employing a centralized register file and by increasing the size of the scratch memory and this enables efficient compilation. Any further increase in resources exceeds the energy envelope.

**Comparison to Best Manual Design** Previous studies [12, 15] have investigated a face recognition DSA that was optimized for energy-delay product. The configuration (three 8 KB SRAMs, with 3 HLU contexts, and a 8 way VLIW (3 INT + 4 FPU + 1 register file)) was shown to be 1.65 times faster than the minimum required real time performance and delivered an energy improvement of 10x as compared to the embedded XScale processor. It can be inferred from the plots that SCA explores this design point in the feasible space. The energy-delay plots demonstrate that the architecture was designed for close to optimal energy-delay characteristics. SCA demonstrates that a similar configuration but with a smaller scratch SRAM (4 KB) and an additional integer unit (Table 4) delivers a 4% energy improvement and a marginal energy-delay product improvement over the manually designed architecture. While manual design is very effective in identifying close to optimal points, this study demonstrates that an intelligent exploration algorithm which searches a wider design space is more robust in identifying optimized designs.
Figure 5. Left: SCA applied to face recognition suite, Right: Energy-Delay product comparisons for feasible performance-energy designs

Exploration Time   Due to the rapid convergence of exploration, SCA investigates fewer than forty design points from a design space of about thousand points and the total time for arriving at these feasible designs is 215 minutes. This demonstrates the efficiency of SCA in intelligently discovering design points during the search phase. Combined with a compiler that generates optimized code for the face recognition DSA, our framework significantly reduces designer time in arriving at performance energy optimal designs.

Figure 6. Area comparison for performance-energy feasible schedules

Investigating Feasible Designs   SCA provides the designer with a set of feasible designs (all points in the quadrant of acceptable designs) and we investigate these designs for minimum energy-delay product, minimum area or maximum performance. Figures 5 and 6 show the energy-delay product (right), area, and
throughput plotted as a function of area for all feasible designs. Depending on whether the application is low power or high performance embedded systems, designers can choose the following designs:

- **Minimum area design**: It can be inferred from figure 6 that the configuration with a 6 way VLIW back-end that barely meets the performance requirement occupies minimum area and is approximately 75% smaller than the design with highest performance.

- **Minimum energy-delay product design**: As discussed above, the configuration with a 9 way VLIW back-end delivers the best energy-delay product and is marginally better than the manually designed system.

- **Maximum performance**: A configuration with a 11 way VLIW back-end delivers the best performance and is approximately 50% faster than the design with minimum area.

### 5.2 CASE STUDY II: Architecture for Embedded Speech Recognition

Table 4 shows the DSA configurations for minimum area, minimum energy-delay product, and maximum performance after selecting the set of feasible designs from the complete space. We observe that the configuration with minimum area reduces energy dissipation by 44% compared to the best manual design (12). Similarly, the configuration with the highest performance delivers a performance improvement of 38% as compared to the manual design point. It can be inferred that SCA provides more flexibility to the designer in choosing feasible design points given the constraints.

The manually designed architecture was optimized for energy-delay product and a comparison to the best energy-delay design from SCA possesses a marginally inferior energy-delay product (5%) as compared to the manual design. This is due to the fact that during exploration, the compiler could not produce a feasible instruction schedule for the manual configuration. This can be attributed to two reasons: i) The original design was hand-scheduled and the width of the multiplexers were larger than that allowed in the design space ii) The manual design had no register file and hence, the compiler’s problem is further exacerbated. Nevertheless, SCA identifies a comparable design with a feasible compiler schedule and this contributes to a significant reduction in designer time thus, precluding hand scheduling of code.
Table 4. Best configurations for different constraints, Throughput, and Energy comparisons for different targets

5.3 CASE STUDY III: A Clustered Architecture for Wireless Telephony

Wireless telephony benchmarks are characterized by integer operations and in most cases employ a 16-bit data-path. While SCA can be employed to search for different width data paths, the case studies of face and speech recognition employed 32 bit data paths and exploration along this dimension was not required. Nevertheless, SCA searches the space for data width and thins resources in such cases. To facilitate comparison against the manual design from [8], we compare only the energy-delay product improvement over the Embedded XScale processor. All designs are from the feasible space and meet the minimum performance requirement. It can be observed that the architecture for wireless telephony is significantly different from the above case studies. Support for hardware loops are not necessary in this case and while SCA investigates a configuration with a HLU, no performance improvements are observed and hence, HLUs appear as an energy bottleneck. Another major difference is the presence of multiple register files and a two level interconnect, leading to a clustered back-end. Each register file supports a few integer units and while SCA does not perform clustering directly, the introduction of a second level interconnect across the functional units provides this functionality.
The design configuration with minimum area is a single cluster machine that barely meets the performance requirements. As expected, its energy-delay product is inferior (2x) to the manual design. A similar observation can be made for the configuration with maximum performance. The introduction of a multi-level interconnect greatly increases the search space and we observe that the SCA based energy-delay optimized design provides a 17% improvement over the manual design (optimized for best EDP). As the design space increases, it becomes increasingly difficult to manually identify the most optimal designs and the wireless telephony case study illustrates this observation.

5.4 Sensitivity Analysis

SCA Robustness: One DSA for all three domains An interesting option to evaluate the robustness of SCA is to design a single DSA for all 16 benchmarks. While convergence was slow, SCA arrived at an energy-delay optimal feasible design for execution all three applications. The configuration is a 12 way VLIW machine (8 INT + 4 FPU) with a centralized register file supported by a well provisioned memory systems (two 8 KB SRAMS for input and output, one 8 KB scratch SRAM, 6 AGUs, HLU with three loop contexts, two level interconnect). SCA limited the design space to around a hundred design points in about 7 hours. As compared to an architecture for one domain, this design consumes more energy but is capable of delivering the real time performance for all the applications.

SCA Convergence Since SCA combines compilation and simulation, investigating many designs for a suite of benchmarks is a time consuming process and is slow compared to purely analytical methods. On the other hand, design space pruning is very efficient and it can be observed from the design space search for face recognition. We observed a similar space search for speech recognition and wireless telephony. Even in the the case of a single DSA for all three domains, SCA investigates around a hundred design points from a design space of at least 3000 design points. Total exploration time was 183 minutes for speech recognition, 215 minutes for face recognition, and 85 minutes for wireless telephony. As compared to months for investigating a single design, this is a significant reduction in exploration time given the complexity and number of kernels.

Choosing Initial Design Points A good exploration algorithm works independent of the choice of initial test architecture. To test the independence of SCA, we choose two starting design points: one that exceeded
the energy envelope required for embedded applications, and another point that is in the middle of the feasible space. For the first test point, SCA performs exploration by successively removing architectural resources and converged to the feasible space in tens ($\leq 20$) of iterations. For the second test point, SCA discovered all the feasible design points by successive addition of resources. Once the design exceeds the energy envelope, SCA provides the designer with those feasible designs. SCA restarts at the initial test point and performs thinning to discover the remaining design points. In both these tests, convergence to the feasible space is sufficiently fast. SCA precludes occurrences of a local minimum due to the investigation of various architectural solutions for alleviating the same bottleneck.

6 Related Work

Brute force design space exploration is a naive technique that generates the set of all designs from the available components and then employs compilation and cycle accurate simulation to arrive at performance and energy values for all these designs. This technique is guaranteed to find optimal designs, but it is time intensive and is often infeasible in many cases. Exploring design spaces in an intelligent manner reduces design time.

Karkhanis et al. [10] explore the design of application specific super-scalar processors for applications like mcf. While their analytical technique is fast, ours is the first study that employs the combination of compilation and cycle accurate simulation. The PICO project [11] explores the design of custom EPIC and VLIW processors for different application domains. It is quantitatively complex to compare the techniques directly due to the differences in architectural design methodology and compilation flow. Grun et al. [6] explore memory and interconnect designs for embedded applications. In contrast to all the above studies, Our application domain is more diverse in its characteristics and SCA reduces the design space effectively, and guarantees feasible and efficient schedules from an optimized compiler. Yehia et al. [19] explore the design of transparent accelerators using simulation and logic synthesis and this balances compilation and accelerator design. Our approach is similar in that our architectural design methodology leverages compilation to achieve energy efficient designs.

Crovella et al. [2] employed the idea of lost cycles analysis for predicting the source of overheads in a parallel program. They employ the tool to analyze performance trade-offs among parallel implementations of 2D FFT. Our algorithm explore many designs points in the architecture space for different application
domains. Other studies [9] have explored analytical predictive modeling for design spaces. As opposed to their work, our tool automatically searches the design space to arrive at optimal design points for varying constraints. Statistical simulation techniques [4] reduce the number of instructions that a simulator executes, thereby reducing design optimization time.

7 Conclusions and Future Work

Future computing devices will likely integrate special purpose processors on a single die to achieve better performance and energy efficiency. These special purpose devices will be catered to execute niche application domains like speech recognition, face recognition, etc. As these applications evolve, the architecture needs to adapt in a seamless manner to changes in the application. In this study, we have proposed the automated design of programmable ’ASIC-like’ architectures for such application domains. An exploration algorithm based on SCA explores the design space of such architectures and arrives at performance-energy optimal designs for different constraints. The bottleneck analysis tool helps the system designer in understanding bottlenecks within a system and provides solutions for further improvement. The design selection stage identifies the most critical architectural changes and guides the design space efficiently to arrive at the next test architecture. We demonstrated the robustness of the SCA algorithm in reducing user design time, error probability, etc., by designing systems for three different application domains. The SCA algorithm was shown to be insensitive to choice of initial design point and converges at optimal designs at all times. In summary, the contributions of the framework are:

- SCA, an exploration tool that provides designers with a set of performance-energy optimal designs for constraints such as minimum area, minimum energy-delay product, or maximum performance.

- SCA discovers intelligent design choices that dramatically reduce the design space from thousands to tens of design points in most cases. Combined with an optimized compiler for the application, this framework reduces designer time significantly.

As part of future work, we intend to extend the framework to design DSAs for two application domains that are becoming increasingly important: ray tracing and finance modeling. We also intend to increase the flexibility of our compiler in investigating different algorithms for interconnect scheduling. This will also improve the robustness of the existing framework.
References


