Automating the Design of Embedded Domain Specific Accelerators

Karthik Ramani, Al Davis

UUCS-08-002

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

February 20, 2008

Abstract

Domain specific architecture (DSA) design currently involves a lengthy process that requires significant designer knowledge, experience, and time in arriving at a suitable code generator and architecture for the target application suite. Given the stringent time to market constraints and the dynamic nature of embedded applications, designers face a huge challenge in delivering high performance yet energy efficient devices. In this study, we investigate an automatic design space exploration tool that employs an iterative technique known as “Stall Cycle analysis” (SCA) to arrive at near-optimal energy-performance designs for various constraints ,e.g., minimum area. For each design candidate in the process, the results of code generation and simulation are analyzed to identify bottlenecks to performance (or energy) and provide insight into adding or removing resources for further improvements. Second, we demonstrate the utility of exploration in pruning the design space effectively (from $\geq$1000 points to tens of points) for three application domains: face recognition, speech recognition, and wireless telephony. As compared to manual designs optimized for a particular metric, SCA automates the design of 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). Finally, we discuss the impact of per design code generation in reducing DSA design time from man-months to hours and in identifying superior design points through architectural design space exploration.

1 Introduction

The rapidly growing deployment of embedded systems and the dynamics of the competitive market necessitate the creation of programmable devices optimized for performance and energy (low energy-delay product). Future devices will also be required to deliver the real time demands of increasingly important workloads [18, 8, 16] like face recognition and speech recognition, subsequently referred to as recognition. These diverse application domains demand new functionality and are characterized by intertwined sequential and parallel *kernels*. Given the stringent time to market conditions for embedded machines, the above problems present multi-dimensional challenges to system designers. As applications evolve in complexity and vary in functionality, the kernels present a problem for optimized code generation. Architectural design and further improvements may necessitate changes in compilation and negatively impact code generation.

A popular architectural 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 [13, 20] has been successful in designing energy efficient processors for many application domains. It also entails a significantly modified version of a compiler framework to generate optimized code for such specialized processors. Thus, 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 quality is strongly dependent on user experience, knowledge of the application domain, awareness of architectural design issues and their impact automatic code generation. Most recent DSA studies [20, 13] also generate code manually to exploit certain architectural features thereby, adding months to total design time. Given the complexity of today's applications and designs, this process is error prone, time-consuming, and may be infeasible for large search spaces.

Recent studies have investigated exploration techniques [17, 1, 12, 27, 25] to automate the design of application specific processors or accelerators. Based on the type of architectures explored, these techniques can be classified into three relevant categories. First, studies [17, 25] have investigated analytical techniques for the automation of super-scalar processors for SPEC or media application kernels. While [17] is fast, the algorithm was evaluated for one particular phase within the program. Studies [20, 22] have shown that complex multimedia applications like recognition consists of various kernel phases within the program. In addition, many different algorithms could be employed to perform the functions of the same kernel. While Silvano *et al.* [25] address this issue, their architectural analysis focuses on the memory system and not in great detail on the interconnect and execution units.

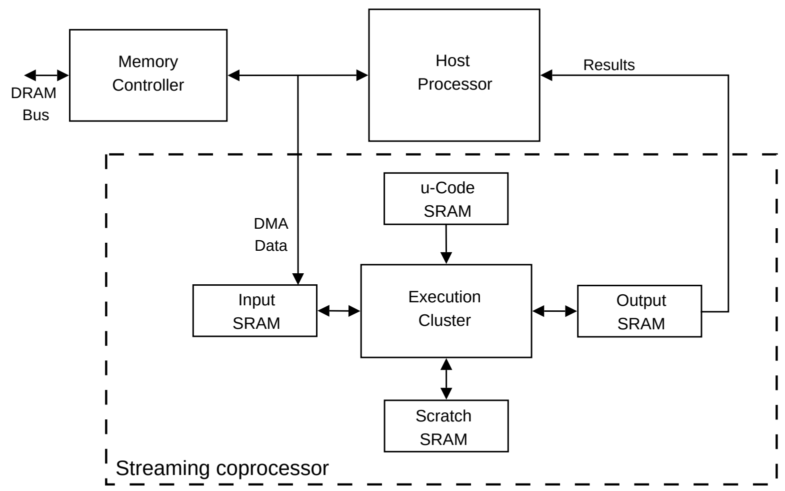

Figure 1: Heterogeneous Multiprocessor Organization

The second class of architectures explored for automation is transparent accelerators [27] for embedded systems. This study balances compilation and accelerator constraints and is similar to our approach in this regard. While their approach is based on instruction set customization, ours is tailored to extract the data flow patterns within the application. The third and final class of architectures, including our study, fall into the category of long word machines [1]. The PICO design system [1] consists of a VLIW GPP and an optional non-programmable accelerator and tries to identify a cost effective combination of the two to provide designs on the cost performance curve. Our approach explores the design of a programmable DSA that satisfies the energy-performance-area constraints for the entire application domain.

This study investigates a simple exploration algorithm to automate the design of long word DSAs for constraints like minimum area, maximum performance, etc. Each design candidate is analyzed for resource addition or removal. A performance bottleneck exists if resource addition or dilation to the candidate delivers a performance improvement. Similarly, if resource removal or thinning reduces energy dissipation significantly, it indicates an energy bottleneck. Our 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 bottlenecks. Bottleneck diagnosis (BD) associates stall cycles in the compilation schedule (or simulator) to bottlenecks (routability issues, insufficient parallelism, etc.) for performance or energy. During diagnosis, we investigate various solutions for each of the bottlenecks. A cost metric is assigned to each solution. The cost metric reflects the improvements in performance, energy, and code generation complexity that the solution potentially presents. In our framework, this information is fed as input to the design selection (DSel) phase. The solution with the lowest cost is selected to generate the next test architecture. This process is repeated in an iterative manner to arrive at near-optimal energy-performance designs for different constraints.

In the second part of the study, we evaluate the effectiveness of exploration for three important application domains: face recognition, speech recognition, and wireless telephony. These domains are fundamentally different in their access, control, and computational characteristics [22, 19, 13] and present a diverse workload [18, 8] to test the generality of the approach. The kernels employed for face and speech recognition are generalized techniques for object recognition and can be adapted to perform other visual recognition applications. In addition to exploring the design space for these domains, we also evaluate the impact of code generation in exploiting the architectural features for each design in the search space. If compilation support is absent for a unit, the search space changes and we show that the final design candidate possesses inferior energy-delay properties with respect to the original design. This demonstrates the benefits of designing programmable hardware for certain application domains. In summary, the main contributions of our study are:

- Exploration Space: This study employs SCA, a tool that explores the space of fine grained programmable VLIW DSAs for diverse application domains to deliver energy efficient soft-real time embedded systems.
- Workload Studies: There seems to be a consensus among industry [8] and academia [16] that recognition is a critical workload for the future. To our knowledge, this is the first study that investigates the automation of DSAs for face and speech recognition. This study also employs different algorithms to perform the functions of the same kernel. For diversity, we also investigate the wireless telephony domain.
- Per Design Code Generation: For each investigated design candidate, we perform optimized compilation and section 5 quantifies the impact of code generation. Results demonstrate the advantages of exposing sophisticated functional units to the compiler and in designing programmable hardware to cater to the needs of the application.

A brief overview of our architectural and compilation methodology is presented in section 2. The exploration process is discussed in section 3. The simulation methodology is presented in section 4. Design case studies, code generation evaluation, and sensitivity analysis follows in section 5. Related work not already discussed can be found in section 6 followed by conclusions in section 7.

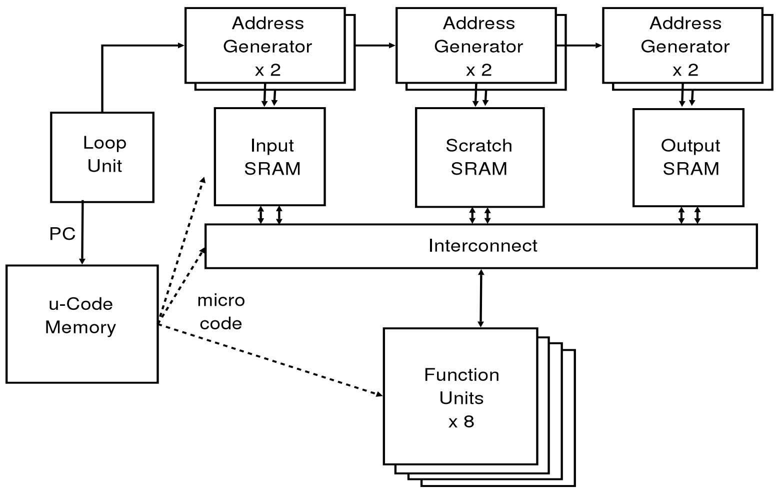

Figure 2: Architectural Design Template for Exploration

2 Framework Description

The automated framework requires two major components upon which the exploration algorithm is implemented: simulation for architectural design and compilation. Recent studies [20, 22] have showcased an architectural design approach that employs a programmable 'ASIC-like' back-end supported by a distributed memory front-end to deliver the performance and energy requirements for embedded face recognition and speech recognition. In this study, design space exploration is performed. Figure 2 shows a template of the architecture. The memory system consists of a loop unit, three distributed SRAMs, and associated address generator units (AGUs). A centralized interconnect is constructed from several layers of multiplexers and connects the memory system and the back-end. The back-end is a cluster consisting of a set of clock-gated function units. A modified trimaran compiler framework [5] is used to generate optimized code and the simulator for performance analysis.

DSA Design Methodology The memory architecture [20, 22] 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) [20] and AGUs provide sophisticated addressing modes which increases the effective instructions committed per cycle (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

Application Domain	Memory system	Back-end	Interconnect	EDP (XScale)
Face Recognition	3 8KB SRAMS, 6 AGUs, 3 HLU contexts	3 INT + 4 FPU + 1 RF	single level	80x
Speech Recognition	3 8KB SRAMS, 6 AGUs, 4 HLU contexts	4 INT + 4 FPU	single level	150x
Wireless Telephony	3 2KB SRAMS, 1 4KB SRAM, 4 AGUs	8 INT	two level	100x

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

the multiple execution units. The HLU also permits modulo scheduling [24] of loops whose loop counts are not known at compile time and this greatly reduces compilation complexity.

The architectural model is effectively a 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 [26]. Figure 3 shows how programmable multiplexers allow function units to be linked into 'ASIC-like' pipelines which persist as long as they are needed. The outputs of each MUX stage and each execution unit are stored in pipelined registers. This allows value lifetime and value motion to be under program control. The compiler generated microcode controls data steering, clock gating (including pipeline registers), function unit utilization, and single-cycle reconfiguration of the address generators associated with the SRAM ports. In the case of frequency limitations, the number of interconnect levels can be increased by employing an additional level of multiplexers and pipelined registers. The overall 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. Studies [20, 13] have demonstrated the effectiveness of this approach for 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.

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 is close to 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 ar-

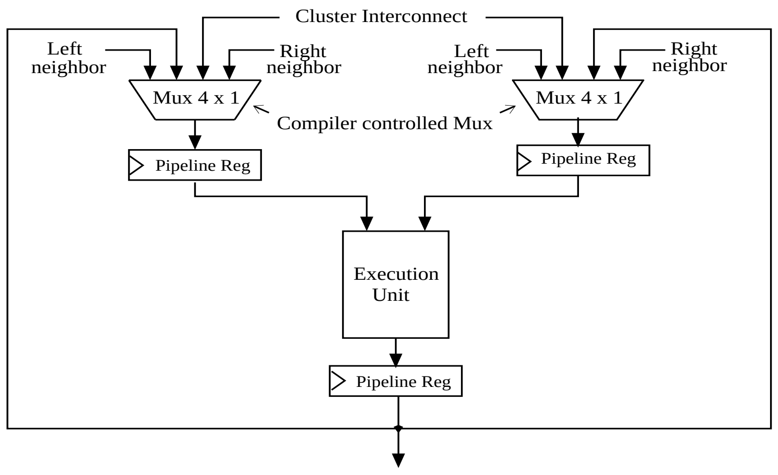

Figure 3: Functional Unit Template

chitecture to meet the needs of our fine-grained cache-less clustered VLIW (Very Long Instruction Word) approach. The result is a compiler that 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.

The program is effectively horizontal microcode which requires that all of the control points be concurrently scheduled in space and time to create efficient, highly parallel and pipelined flow patterns. To solve this problem, the code generator employs integer linear programming (ILP) based interconnect-aware scheduling techniques to map the kernels to the DSA. Compilation techniques for code optimization are based on [3, 21, 22] and these studies have shown that interconnect-aware scheduling delivers the performance and energy benefits of such architectures. 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 HLU. After register assignment, we perform ILP based interconnection scheduling followed by post pass scheduling to resolve conflicts.

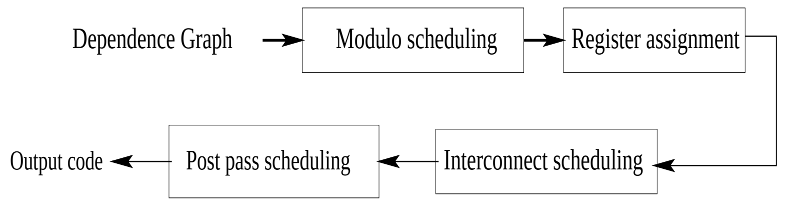

Figure 4: Compilation flow employed in the framework

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 Bottleneck Diagnosis (BD) classifies the overheads in the system into different categories. For each of the overheads, the tool walks through a list of potential solutions. To understand the impact of a potential solution, the notion of cost is employed. In this study, cost is a function of three different metrics: performance, energy, and ease of code generation. The study in [10] employs a similar notion and calculates interaction cost for events and the goal is to optimize latency in super-scalar processors. In contrast, embedded DSA design has stringent constraints on energy dissipation and ease of code generation. Hence, in our study, we assign a cost based on the three metrics. The output of BD is used by the design selection (DSel) phase to select a particular candidate solution. Our framework provides resource dilation or thinning as solutions to bottlenecks. After a low cost solution is selected, we compile and evaluate the new test architecture. To reduce the complexity of exploring the complete design space, DSel breaks down the search space into three subsystems in each iteration: memory system, back-end system, and interconnection network.

3.1 Bottleneck Diagnosis (BD)

To aid the process of overhead classification and quantification, we collect the following available statistics about the application suite using the compiler and the simulator.

- *Function unit utilization rate*: The fraction of total execution cycles in which a particular function unit is utilized.
- *Register file use rate*: The fraction of total execution cycles in which the centralized register file is utilized. For speech recognition, the presence of a scratch memory replaces the register file.
- *Function unit contention*: The 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 creates bandwidth starvation. 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 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 measured using the methodology in [2, 23].

The above statistics are 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:

- 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.
- SRAM Port Contention: Contention in the ports may lead to back-end starvation. This is common in applications with multiple loop contexts.
- 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 and interconnect contention.
- Unit Starvation: An increase in the number of function units may lead to underutilization and may be an energy bottleneck.

- 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 (figure 3). 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.

BD employs program statistics to measure the different sources of overhead in a test architecture. In the second part of BD, the tool associates many potential solutions to each of the overheads. To facilitate design selection, a total cost is associated to each of the potential solutions.

Associating Cost Since each of the overheads can be solved by applying different techniques, we need to associate a cost to each of the solutions and then select the lowest cost solution. The costs are based on performance, energy, and code generation complexity. The framework is built such that the user is prompted for the three costs for each function unit in the library. The total cost of employing a unit is the weighted sum of the three metrics. Assigning weights to each of the metrics can lead to many interesting search spaces and is the subject of future study. We associate a boolean cost for each of the metrics and equal importance is given to each of the metrics. If the increase in size or number of a particular unit delivers an significant improvement in performance then the unit is designated with a low cost for performance, and vice versa. For energy or code generation complexity, we assign a high cost if the unit significantly increases energy or compilation complexity. In cases where the impact is not clear, we assign a high cost. Examples of such cases are dilation in interconnect levels and data width. We illustrate how costs are assigned using the examples of a HLU and a register file. For applications with multiple loop contexts, the HLU [20] automatically updates the loop indices for all the loop contexts and generates indices for the AGU to perform address calculations. The HLU contains its own stack, registers, and adder units. The addition of a HLU has the potential to deliver very high performance while incurring a commensurate increase in energy dissipation. It also provides hardware support for modulo scheduling of loops whose indices are not known at compile time. This reduces code generation complexity. Hence, we assign the unit a low performance cost (0), high energy cost (1), and a low code generation cost(0). The register file is used more as a temporary placeholder for data in the event of routing difficulties. Studies [22, 20] 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. Thus, it is assigned a high performance cost, high energy cost, and a low code generation cost. Table 2 shows the exploration space and the various costs for each functional unit in the library. When there is a tie between two functions with the same cost, dependency takes precedence. For example, AGUs are required for efficient operation of HLUs and so, SCA explores AGUs before HLUs.

3.2 Design Selection (DSel)

The process of design selection could 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 architectural feature at a time. Making too many changes can lead to a feedback loop where the exploration algorithm is stuck in a local minima. To prune the design space efficiently, the search is done along three different subsystems. We first employ solutions that can optimize the memory subsystem followed by the execution back-end. The interconnect subsystem is then optimized. These three steps constitute one iteration and this process is repeated continuously until we arrive at a set of energy-performance near-optimal designs. Such an iteration process also accounts for complex interactions across the different subsystems. 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 process of design selection can be easily understood using the Viola face detection kernel.

High SRAM miss rates cause back-end starvation initially. Although different solutions (adding HLUs, AGUs, increasing ports) can be employed, we employ the low cost solution of an increase in the size of the input and output SRAMs. For a further marginal performance improvement and increased energy dissipation, the choice is to reduce their sizes for greater energy savings at minimal/no performance loss. The marginal performance increase occurs due to the fact that computations in multi-level loops entail complex indirect accesses (Example: $Z[i] = Z[i-1] + \sum_{j=0}^{m} X[j] * Y[W[j]]$). In such cases, loop variable accesses compete with the array data accesses and degrades performance. SCA has the choice of employing HLUs or AGUs as they have similar overall cost. Due to precedence, SCA proceeds with the addition of AGUs before a HLU is employed. Increasing the number of contexts increases the area, complexity, and power dissipation while providing minimal performance improvements. SCA, therefore, attempts to keep this number to a minimum.

During design space exploration, we observe a high contention on the function units. Resource contention occurs initially because the initial test architecture consists of one func-

Component	Range	Performance cost	Energy cost	Code generation cost	Total cost
Data width	16,32, 64 (bit)	1	1	1	3
SRAMs	1, 2, 4,.. 64 (KB) each	0	0	0	0
Ports (SRAMs and RF)	1, 2, 3	1	1	1	3
Hardware loop unit contexts	1, 2, 3, 4, 5	0	1	0	1
AGUs	1-8 per SRAM	0	0	1	1
Register fi le size	8, 16, 32 entries	1	0	0	1
Register fi le number	1,2,3,4,5,6	1	1	0	2
Functional unit type	integer, fbating point	-	-	-	-
Functional unit mix	multiplier, adder, etc.	-	-	-	-
Functional unit number	1-8	0	1	0	1
Interconnect Width	2-5	0	1	1	2
Interconnect levels	1-3	1	1	1	3

Table 2: **Design space and cost for each functional unit variable**

tional unit of each type. Those function units with high utilization and contention are dilated by 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. The register file use rate and interconnect utilization metrics are employed to increase/decrease the size and number of register files. 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. In the advent of a frequency conflict, we add an additional cycle by introducing a pipelined register. This increases compilation complexity 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. Initially, the solution sets for each of the subsystems consist of instances of all units not used for exploration. The steps involved in the process are:

1. *Program Statistics*: Initialize solution set. Collect program statistics to measure the various categories of bottlenecks during compilation and simulation.
2. *Memory Optimization*: A high back-end starvation indicates the need for memory optimizations. Begin by adding unit with lowest cost from the solution set. Remove

item from solution set. Observe use rate metric for unit. For significant performance improvements, dilate resource further. For significant energy increase, employ resource thinning. For marginal performance improvements, move to the next unit in the solution set. Compile and simulate next test design.

3. *Execution Back-end Optimization*: High functional unit contention indicates the need for back-end optimizations. Dilate or thin as needed. Compile and simulate next test design.

4. *Interconnect Optimization*: High interconnect utilization and contention for units or SRAMs indicate the need for dilation or thinning as required. Compile and simulate next test design. When the solution set is empty, go back to step 1.

The iterations are repeated until we arrive at a set of energy-performance near-optimal designs. In cases where the algorithm cannot provide a feasible design, the BD tool returns to the last iteration.

4 Methodology

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 [2] for all predictable structures and empirical models similar to [23] 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 and Evaluation Our benchmark suite, shown in table 3 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 [7]. The speech recognition application consists of three phases that contribute to 99% of total execution time: preprocessing, HMM, and the Gaussian phase [19]. 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.

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

Table 3: **Benchmarks and Description**

A robust exploration algorithm prunes the search space efficiently and evaluates each design candidate. We employ pruning and total exploration time to evaluate the effectiveness of exploration. To effectively compare the performance of different architectures, we employ throughput(number of input frames processed per second) and energy (mJ/input) as the performance and energy metrics respectively. As recommended by Gonzalez and Horowitz [11], the energy-delay product metric is used to compare the efficiency of different processors. For each domain, we identify a set of feasible candidates that satisfy the minimum required performance and the maximum allowable energy budgets. These designs are then investigated for three different constraints: minimum area, best energy-delay product, and highest performance. The resulting design choices are compared against the best manual design from previous studies ([22, 19, 14] and also to the Intel XScale processor [4].

For each design, the compiler generates optimized code by exploiting the capabilities of the new added functional unit. Ideally, the impact of code generation can be estimated by measuring the change in the search space and the new final design candidate in the absence of compilation capability for that new functional unit. A comparison of energy-performance of the new and original design candidate will also help in estimating the relative merits or demerits of casting software functions into specialized hardware.

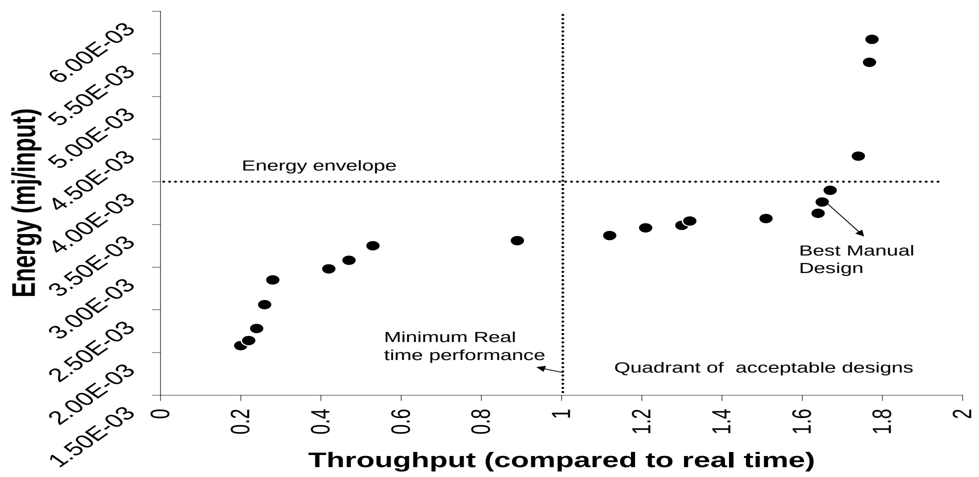

Figure 5: SCA applied to face recognition

5 Results

5.1 CASE STUDY I: Architecture for Embedded Face Recognition

Figures 5 and 6 shows the design points explored by SCA for the seven benchmarks in the face recognition suite. It shows the throughput and energy dissipation 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 fives 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 input and output SRAMS 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 high 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 that is one order of magnitude energy improvement over the energy efficient XScale processor. All designs below this envelope that meet the required performance belong to the ’quadrant of feasible or acceptable designs’. SCA

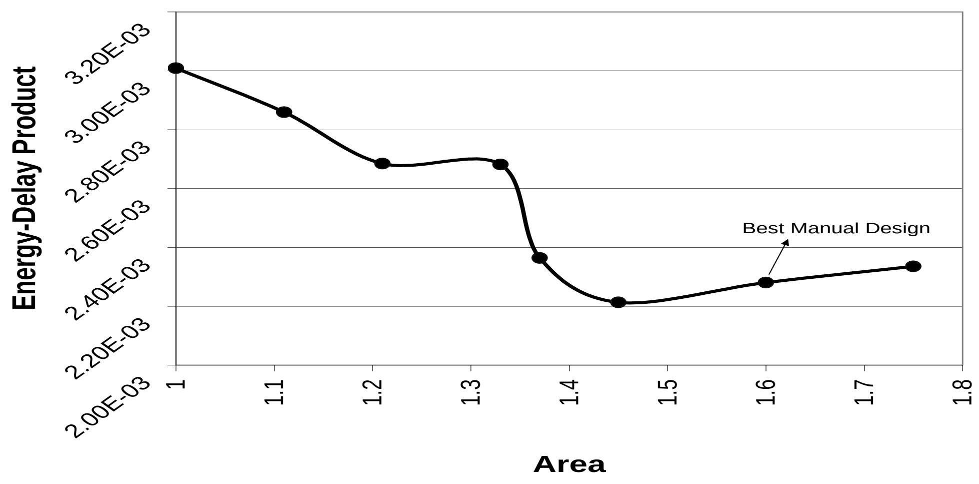

Figure 6: Energy-Delay product comparisons for performance-energy designs

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 [19, 22] 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. The plots show 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 manual design. 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 near optimal designs.

Due to the rapid convergence of exploration, SCA investigates fewer than forty design points from a design space of about a thousand points and the total time for arriving at these feasible designs is 215 minutes on a 1.6 GHz AMD Athlon PC. Combined with a compiler that generates optimized code for the face recognition DSA, our framework significantly reduces designer time.

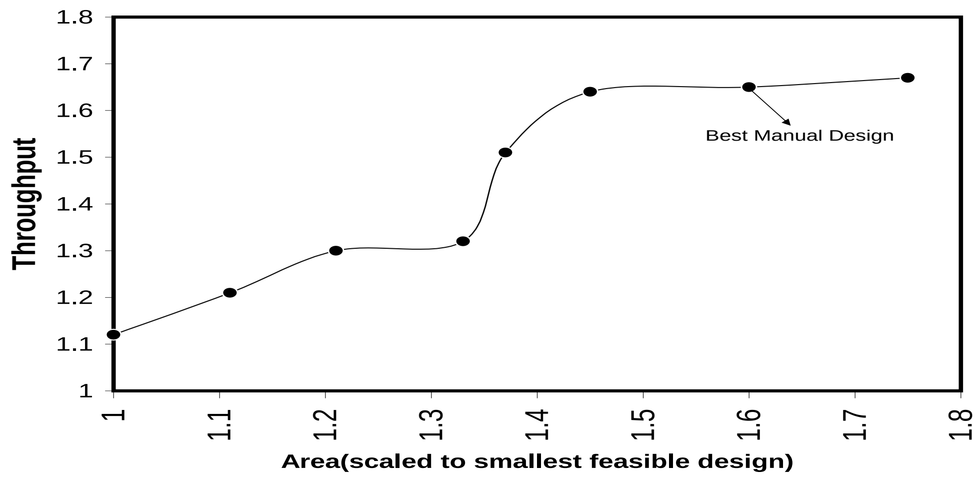

Figure 7: Throughput comparison for performance-energy designs

Quadrant of Feasible Designs SCA selects acceptable designs and we investigate these designs for minimum energy-delay product, minimum area or maximum performance. Figures 6,8, and 7 show the energy-delay product, energy, and throughput plotted as a function of area. Depending on whether the application is for low power or high performance embedded systems, designers can choose:

- *Minimum area design*: Figures 7 and 8 show that the configuration with a 6 way VLIW back-end barely meets the performance requirement and occupies minimum area. It is approximately 75% smaller than the design with highest performance.
- *Minimum energy-delay product design*: 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 an 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

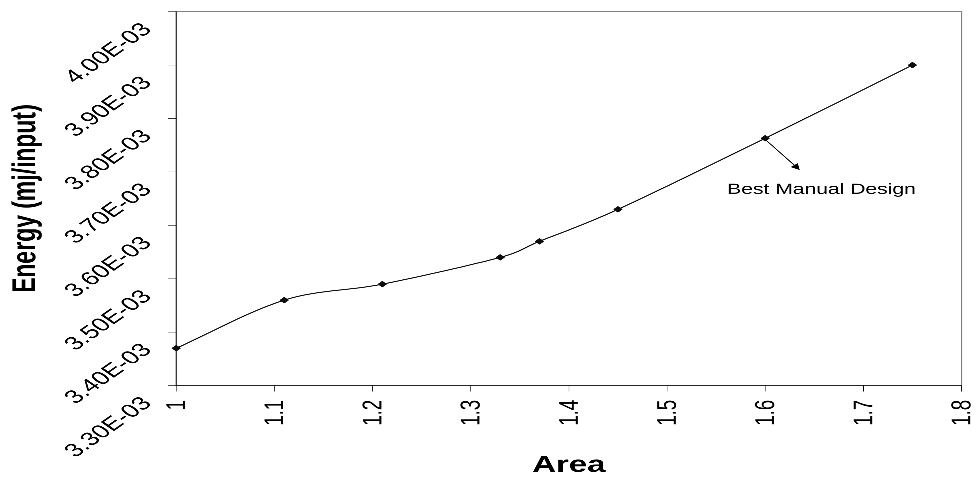

Figure 8: Energy comparisons for performance-energy designs

44% compared to the best manual design [19]. Similarly, the configuration with the highest performance delivers a performance improvement of 38% when compared to the manual design point.

The manually designed architecture was optimized for energy-delay product and a comparison to the best energy-delay design from SCA is only 5% worse. This can be attributed to: i) The original design was hand-scheduled and the width of the multiplexers were larger than that allowed in the design space ii) The framework strictly allows a width that obeys frequency targets and hence, this particular design is not explored by our framework. Nevertheless, SCA identifies a comparable design and this contributes to a significant reduction in designer time.

5.3 CASE STUDY III: A Clustered Architecture for Wireless Telephony

Wireless telephony benchmarks typically use a 16-bit integer data-path. While SCA can be employed to search for different width data paths, the case studies of face and speech recognition required 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 designs from [13, 14], we compare only the energy-delay product improvement over the XScale processor. All designs are from the feasible space and meet the minimum performance requirement. The architecture for wireless telephony is significantly different from those for the recognition algorithms. Support for hardware loops are not necessary in this case and while SCA investigates a configura-

Design Point	Memory SRAM (KB)	Scratch SRAM (KB)	AGUs	HLU contexts	INT units	FP units	RF	Mux levels	Throughput frames/sec	Energy (mj/input)
Face Recognition										
Manual	8	8	2x3	3	3	4	yes	one	1.65	3.76e-03
Minimum Area	8	1	2x3	3	3	3	no	one	1.12	3.46e-03
Minimum ED product	8	4	2x3	3	4	4	yes	one	1.64	3.63e-03
Maximum Performance	8	8	2x3	3	5	5	yes	one	1.68	3.80e-03
No HLU	16	16	2x3	-	8	5	yes(2)	one	1.14	5.4e-03
Speech Recognition										
Manual	8	8	2x3	2	4	4	no	one	1.98	1.1e-03
Minimum Area	8	2	2x3	2	3	2	yes	one	1.03	0.76e-03
Minimum ED product	8	8	2x3	2	4	3	yes	one	1.92	1.13e-03
Maximum Performance	16	16	2x3	2	5	5	yes	one	2.74	3.7e-03
Wireless Telephony										
Manual	4,2	2	4	-	12	-	yes (4)	two	-	100x (EDP)
Minimum Area	2	1	3	-	8	-	yes (1)	one	-	50x (EDP)
Minimum ED product	4	8	4	-	10	-	yes (3)	one	-	120x (EDP)
Maximum Performance	16	8	6	-	16	-	yes (4)	three	-	30x (EDP)
No clustering	16	16	8	-	15	-	yes (5)	one	-	17x (EDP)
All three domains										
Minimum ED product	8	8	6	3	8	4	yes (1)	two	-	-

Table 4: **Best configurations for different constraints, throughput, and energy comparisons for different targets**

tion with a HLU, no performance improvements are observed. 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.

Investigating Clustered Designs The design configuration with minimum area is a single cluster machine that barely meets the performance requirements. While the current configuration is balanced in terms of data throughput across memory and the back-end, there is more available parallelism in the application space. Dilation in SRAM size and the back-end could extract the parallelism. SCA employs dilation and selects a configuration that matches the performance of the manual design [14] at lower energy dissipation. This candidate possesses the best energy-delay characteristics and provides a 17% improvement over the manual design. Further dilation increases pressure on the interconnect and SCA observes diminishing returns in performance. SCA dilates interconnect width until the frequency limits are met. Beyond that, the introduction of a multi-level interconnect facilitates clustering and allows for dilation in back-end functional units. The design with maximum performance consists of a three level interconnect and supports as many as sixteen integer units and four register files. Clustering increases the search space and makes it difficult to manually identify the most optimal designs. This makes a case for tools that explore the design space automatically.

5.4 Impact of Per Design Code Generation

As opposed to previous studies [17] that do not consider the impact of micro-architectural changes on code generation, our study generates optimized code for each of the kernels in the suite and hence guarantees compilation for every design candidate. The final set of design candidates chosen represent a synergy between compilation and architectural design. It also guarantees optimized compilation for the domain. To evaluate the benefit/demerit of per design code generation, we evaluate two interesting scenarios for which the presence of a particular functional unit delivers significant performance and energy improvements.

In the first scenario, we evaluate a face recognition case study where compiler support is disabled for a HLU. In the absence of a HLU, SCA moves to optimize the back-end and successively increases the number of integer units and the size of the register file in the back-end. After performance saturation, it optimizes the front end and increases the size of the SRAMs and observes a performance improvement. Due to the absence of the

HLU, the optimization algorithm successively increases memory size and integer units to account for loop computations, storage, and additional interconnect bandwidth. The extra computations also introduce the problem of port saturation. We have a different set of feasible candidates in this new search. The candidate with the best energy-delay product (shown as No HLU in table 4) meets the performance budget, but is 30% slower while dissipating 35% more energy.

In the second scenario, we limit the number of interconnect levels in the framework for the wireless telephony domain. SCA identifies a different configuration that delivers the performance and the energy requirements for the domain. Nevertheless, this configuration degrades energy-delay product by 44.3% with respect to the original configuration with the best energy-delay product. Multiple interconnect levels reduce congestion for data at the interconnect and hence, delivers performance and energy advantages for this particular domain. It should be noted that imposing this limitation on the other two domains doesn't affect the search results. The above scenarios show the importance of per design code generation and demonstrate the benefits of casting domain specific functions in programmable hardware.

5.5 Sensitivity Analysis: SCA Robustness

An interesting option is to evaluate the robustness of SCA in designing a single DSA for all 16 benchmarks. While convergence was slow, SCA arrived at a energy-delay optimal design for 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 system (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 on the 1.6 GHz Athlon system. When compared to an architecture for one domain, this design consumes more energy but is capable of delivering the real time performance for all three applications.

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. 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 points. Total exploration time was 183 minutes for speech recognition, 215 minutes for face recognition, and 85 minutes for wireless telephony. When compared to months to manually investigate a single design, this is a significant reduction in exploration time given the complexity and number of kernels.

A good exploration algorithm works independent of the choice of initial test architecture. To test the independence of SCA, we chose 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 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 and delivered the same designs for the face recognition and the wireless telephony domains. In the case of speech recognition, the tests lead to similar but not exactly the same configurations. The difference was in the size of the register file and this delivered a marginal energy-delay product difference of less than 2%. This was due to the compiler's ability to generate slightly different schedules for the two candidates. SCA precludes occurrences of a local minimum due to the investigation of various architectural solutions for a bottleneck.

6 Related Work

Grun *et al.* [12] explore memory and interconnect designs across memory and the CPU. In addition to the memory and interconnect subsystems, our approach also explores various back-end configurations. Crovella *et al.* [6] 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 explores many design points in the architecture space for diverse application domains. Other studies [15] have explored machine learning based modeling for design spaces and this could potentially replace the simulator employed in our study. In contrast, our tool automatically searches the design space to arrive at optimal design points for varying constraints. Statistical simulation techniques [9] 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 devices will execute niche application domains like speech recognition, face recognition, etc. As applications evolve, the architecture needs to adapt in a seamless manner to changes in the application. This study investigates 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 near optimal designs for different constraints. We have demonstrated the robustness of the SCA algorithm in exploiting optimized code generation and in reducing user design time, error probability, etc., and by designing systems for three increasingly important application domains. As part of future work, we intend to extend the framework to design DSAs for two application domains that are becoming important: ray tracing and finance modeling. Currently, the grain-size of the back-end execution units is at too large, e.g., Integer and floating point units. We plan to investigate a finer grain size in execution units.

References

[1] S. G. Abraham and B. R. Rau. Efficient design space exploration in pico. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 71–79, New York, NY, USA, 2000. ACM.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture ISCA-27*, June 2000.

[3] G. Burns, M. Jacobs, W. Lindwer, and B. Vandewiele. Exploiting parallelism, while managing complexity using silicon hive programming tools. In *GSPx*, Oct. 2005.

[4] L. Clark. Circuit Design of XScale Microprocessors. In *Symposium on VLSI Circuits (Short Course on Physical Design for Low-Power and High-Performance Microprocessor Circuits)*, June 2001.

[5] T. T. Consortium. Trimaran: An infrastructure for research in instruction-level parallelism. http://www.trimaran.org, 2004.

[6] M. E. Crovella and T. J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. In *Supercomputing '94*. IEEE Computer Society, 1994.

[7] M. T. D. Bolme, R. Beveridge and B. Draper. The CSU face identification evaluation system: Its purpose, features and structure. In *International Conference on Vision Systems*, pages 304–311, April 2003.

[8] P. Dubey. A platform 2015 workload model: Recognition, mining, and synthesis moves computers to the era of tera. *Intel Corporation White Paper*, Feb. 2005.

[9] L. Eckhout. *Accurate Statistical Workload Modeling*. PhD thesis, University of Ghent, 2002.

[10] B. A. Fields, R. Bodík, M. D. Hill, and C. J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 228, Washington, DC, USA, 2003. IEEE Computer Society.

[11] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, Sept. 1996.

[12] P. Grun, N. Dutt, and A. Nicolau. Access Pattern-Based Memory and Connectivity Architecture Exploration . *ACM Transactions on Embedded Computing Systems*, 2(1), February 2003.

[13] A. Ibrahim. *ACT: Adaptive Cellular Telephony Co-Processor*. PhD thesis, University of Utah, December 2005.

[14] A. Ibrahim, M. Parker, and A. Davis. Energy efficient cluster co-processors. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2004.

[15] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 195–206, 2006.

[16] J. Junqua. *Robust Speech Recognition in Embedded Systems and PC Applications*. Springer, 1st edition, May 2000.

[17] T. S. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: an analytical approach. *SIGARCH Comput. Archit. News*, 35(2):402–411, 2007.

[18] B. Liang and P. Dubey. Recognition, mining, and synthesis. *Intel Technology Journal*, 9(2), May.

[19] B. Mathew, A. Davis, and M. Parker. A Low Power Architecture for Embedded Perception. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '04)*, September 2004.

[20] B. K. Mathew. *The Perception Processor*. PhD thesis, University of Utah, August 2004.

[21] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *FPL'03*, 2003.

[22] K. Ramani and A. Davis. Application driven embedded system design: A face recognition case study. In *CASES '07: Proceedings of the 2007 International conference on compilers, architectures, and synthesis for embedded Systems*, pages 103–114, 2007.

[23] K. Ramani, A. Ibrahim, and D. Shimizu. PowerRed: A Flexible Modeling Framework for Power Efficiency Exploration in GPUs. In *Proceedings of the Workshop on General Purpose Processing on GPUs, GPGPU'07*.

[24] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.

[25] C. Silvano, G. Agosta, and G. Palermo. Efficient architecture/compiler co-exploration using analytical models. *Design Automation for Embedded Systems*, 11(1):1–25, 2007.

[26] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

[27] S. Yehia, N. Clark, S. Mahlke, and K. Flautner. Exploring the design space of lut-based transparent accelerators. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 11–21, 2005.