Automating Embedded System Design: A Hardware/Software Approach

Background  The field of embedded systems has revolutionized the way we interact, perceive, and communicate information. The diverse characteristics of such devices have facilitated their deployment in many applications: inexpensive cellular phones for communication and accessing information 'on-the-go', reliable pacemakers in the field of medicine, security cameras for surveillance purposes, etc. The key to the attractive characteristics of such systems is the embedded processor (Central Processing Unit (CPU)) which supervises the functions of the complete system. This study explores the design of the central processing unit (CPU) and creates tools that facilitate the seamless execution of different programs on an embedded processor at low costs.

Problem Description  The need to increase computer system performance while staying within power budgets is pervasive. These constraints are particularly important in the embedded domain: embedded systems are increasingly forced to process high-bandwidth raw data while running on battery power. An important strategy for meeting the conflicting goals of high performance and low power is exploiting available parallelism coupled with more efficient architectures and circuits. This insight is exploited by a number of existing architectures such as high-end systems-on-chip (SOCs), multicore general-purpose CPUs, and graphics processing units (GPUs).

Two significant problems remain. First, creating correct and efficient parallel programs is fundamentally difficult: human developers require significant time and expertise to do this, and efforts towards automatic parallelization have achieved only limited success. Second, current architectures cannot meet the performance and power requirements of some kinds of highly demanding applications such as software-defined radio and high-end audio and video processing.

An increasingly important class of compute-intensive applications can be expressed in terms of stream processing where a collection of kernels are applied to input data streams. Streams are temporally ordered data organized into frames. Streaming applications are pervasive and compute-intensive; they include most forms of audio, video, sensor, signal, and network packet processing. For example, speaker independent voice recognition, face recognition and 3G wireless telephony are streaming applications that are difficult or impossible to perform in real-time even on fast general-purpose processors.

Underlying Principles  The major insight that the proposed work exploits is that conventional streaming applications contain two fundamentally different kinds of code: streaming and non-streaming. The non-streaming part of applications is not typically compute-intensive and it executes efficiently on existing general-purpose processors. The streaming part is divided into kernels that communicate across well-structured channels and can be executed in parallel. Furthermore, individual kernels have little internal history and they process stream data in small temporally ordered frames. Kernels have regular and predictable data access patterns, and they typically touch streaming data only once. Taken together, these properties imply that traditional high-frequency processors with LRU-like caches execute streaming codes in an energy-inefficient way, and in many cases, they provide insufficient performance even when energy concerns are disregarded.

0.1 A Partial Solution: Programmable Streaming Coprocessors

Very high performance and very low energy consumption can be achieved by running streaming kernels on streaming coprocessors that are optimized for streaming tasks. The non-streaming code,
such as device drivers, user interfacing, etc. is efficiently executed on low-power CPUs once the compute-intensive streaming tasks are offloaded.

Streaming coprocessors can be implemented in several ways. For example, application specific integrated circuits (ASICs) can achieve maximal performance with very low power consumption, but they have serious drawbacks. The design and fabrication cycles for ASICs are time consuming and costly. ASICs are fixed-function: a feature enhancement or bug-fix cannot be easily deployed to existing devices. My previous work [1] demonstrates that programmable streaming coprocessors offer a superior alternative to ASIC-based coprocessors.

In previous work, the architecture research group has developed a class of domain specific streaming coprocessors that provide very high performance and energy efficiency, while still leaving data processing under software control. The problem with this approach to date is that it significantly complicates the programming model. For example, my face recognition experience required several person-months to manually analyze the code and transform it into three components: code that runs on the host processor, streaming co-processor code, and the interface code that connects the coprocessor and host programs.

1 Proposed Research

In an ideal world, developers would write their code in a streaming language such as Streamit, Brook, SPUR, Cg, etc, but I feel that this ideal world is a long way off. Most embedded applications are written in conventional languages such as C. I, therefore propose to address the following key problem in increasing the performance and reducing the power consumption of compute-intensive streaming applications:

- **CoGenE**: The creation of a Compiler-Generator-Explorer that generates the appropriate streaming core for the streaming code component, generates the object code and a simulator for that core, and evaluates the power and performance characteristics of the result. The design space around the architecture will then be explored by automatically modifying the architecture description based on the simulation results, and the evaluation cycle will continue. The result is a design space power-performance curve over the design space. Users can then select the appropriate solution for their constraints and the chosen device will be synthesized by existing back-end tools that are the result of previous work.

This work requires novel solutions to program analysis problems. CoGenE must analyze control, data access, and communication patterns in the resulting streaming code and map that into efficient execution, memory, and communication resources in order to synthesize the coprocessor architecture, simulator and object code. The proposed research will be successful if we can substantially reduce the amount of time it takes a developer to turn a conventional application into high-performance, low-power system comprising executable code and domain specialized hardware.

2 Preliminary Results

**Architecture Description** Figure 1 shows the internal organization of the ArcFace processor, a streaming co-processor for the face recognition domain. It consists of 8 clock-gated function units.
These include 4 floating point units, 3 integer units, and a register file. The cluster also includes a loop unit, three 8KB dual ported SRAMs, six address generators, local bypass paths between neighboring function units, and a cluster-wide interconnect. The input and output SRAMs are double buffered to allow concurrent host and DSA access. Each execution unit is an arithmetic unit or possibly a register file. Arithmetic units could be internally pipelined.

Address generation, loop control, and multiple execution units all operate concurrently under program control. The compiler generated microcode controls data steering, clock gating (including pipeline registers), and function unit utilization, while permitting single-cycle reconfiguration of the address generators associated with the SRAM ports. The general result of this approach is a cluster that can be tailored to a particular domain to support multiple applications, application phases, or interleaved phases of a single pipelined application. Energy efficiency is primarily due to: minimized communication, activity, and overhead; the creation of function unit pipelines which mimic ASIC flows; and fine-grained clock gating.

The CoGenE compiler The Trimaran compiler (www.trimaran.org) was the starting point for the CoGenE (Compile Generator Explorer) compiler development. Trimaran was chosen since it allows new back-end extensions, and because its native machine model is VLIW. Significant modifications were needed to transform Trimaran from a traditional cache-and-register architecture to meet the needs of our fine-grained cache-less 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. A new back-end code generator was developed 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 rather than for a single register file. Since the compiler controls the programming of the multiplexors 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.

**Benchmarks** We evaluated our design using seven streaming benchmarks. Six codes cover key aspects of face recognition. These include flesh-tone and color coordinate conversion, image segmentation, face detection, and eye-location, and two face identification benchmarks (EBGM and PCA/LDA algorithms). The FIR benchmark was included from the signal processing domain to test the generality of our approach.

**Results Summary** Initial results show that the CoGeNE compiler can deliver 1.65x the throughput of a high performance Pentium 4 processor while providing 10.65x energy savings than a low power XScale processor. Overall, the results demonstrate that face recognition can be performed in real time in an embedded power budget.

**Current/Future Work** During the course of this effort, I have uncovered numerous opportunities for future research. Register and interconnect scheduling can be done in either order. In either case, the second process is limited by decisions made in the first process. I intend to investigate an integrated approach. While the CoGenE compiler does a good job of automating our previous biggest problem, e.g. increased programming complexity, the end goal of this research is to automate as much of the DSA design process as possible. This implies automating the two remaining manual phases of our design process: creation of the architecture description file and splitting the original application suite into host and streaming components amenable to DSA acceleration. I believe that the CoGenE compiler can automatically create the architecture description file and subsequently modify it during design space exploration via the simulation infrastructure. The result will be a set of possible architectures from which the best candidate can be selected. Automatic splitting of the original application codes will be a harder task and I currently do not believe that a fully automated solution is likely. I feel that an interactive tool that significantly aids the process is more likely to succeed.

**3 Potential Impact/Merit**

The success of this research will allow application programmers to design complex embedded systems which exhibit flexibility, high performance, and low power. This will greatly broaden the design community. The tool-set will be disseminated in the public domain. I believe that it will provide a powerful foundation for both education and research. While the specific domain of this research is complex embedded systems, the results will also apply to future domain specialized cores that will necessarily be part of complex system on chip (SOC) and heterogeneous multi-core architectures.

**References**


4