Exploring Register File Size and Memory Configurations in ASIP Synthesis

Research Proposal

submitted by

Manoj Kumar Jain
(99RCS002)

under the supervision of

Professor M. Balakrishnan & Professor Anshul Kumar

May 2001

Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
New Delhi 110 016 (India)

Contents

List of Figures

1 Introduction

An Application Specific Instruction Set Processor (ASIP) is a processor designed for one particular application or for a set of specific applications. An ASIP exploits special characteristics of the given application(s) to meet the desired performance, cost and power requirements. ASIPs are a balance between two extremes: Application Specific Integrated Circuits (ASICs) and general programmable processors. ASIPs offer the required flexibility (which is not provided by ASICs) at a lower cost than general programmable processors. Thus ASIPs can be efficiently used in many embedded systems such as digital signal processing, servo-motor control, automatic control systems, avionics, cellular phones etc.

Interest in synthesis of ASIPs has increased considerably and a number of methodologies have been proposed in the last decade. We have studied these approaches [1] and identified following as the typical design flow for ASIP synthesis.

- An application written in a high-level language is analyzed statically and dynamically. The analyzed information is stored in a suitable intermediate format, which is used in the subsequent steps.
- Almost all the approaches consider a parameterized model for design space exploration. Inputs from the application analysis step are used along with the range of architecture design space to select a suitable architecture(s) by a design space explorer. The selection process typically can be viewed to consist of a search technique over the design space driven by a performance estimator.
- The instruction set is generated either by synthesis or by a selection process.
- A retargetable compiler is used to generate code. The hardware is synthesized using the ASIP architecture template and instruction set architecture starting from a description in VHDL/ VERILOG using standard tools.

The performance estimation which drives the architecture exploration process is simulator based or analytical (scheduler based). Each approach has its advantages over the other. The analytical approach is relatively faster but the techniques reported so far can handle simple architecture models. On the other hand simulator based approach is more time consuming but it can produce more accurate results for a variety of architectures. With the increase in dimensions of design space, the analytical approaches become more relevant, because of their fast estimation. We have observed that while exploring the architecture space, storage organization is being considered to a certain extent by the simulator based performance estimators but it is ignored by the analytical approaches. These observations are further elaborated in the following section. In our work we emphasize on exploring storage organization using an analytical approach.

2 Approaches in ASIP Synthesis

As indicated above, the two classes of approaches to ASIP synthesis based on the technique used for performance estimator for driving the configuration selection process are :

- Simulator based
- Analytical (Scheduler based)

2.1 Simulator based

Such approaches [2, 3, 4, 5, 6] estimate performance by generating code of the given application for the target architecture. This code is then simulated using an instruction set simulator to get performance estimation. An illustrative approach of this class suggested by Gloria et al [2] (figure 1) is described briefly here.

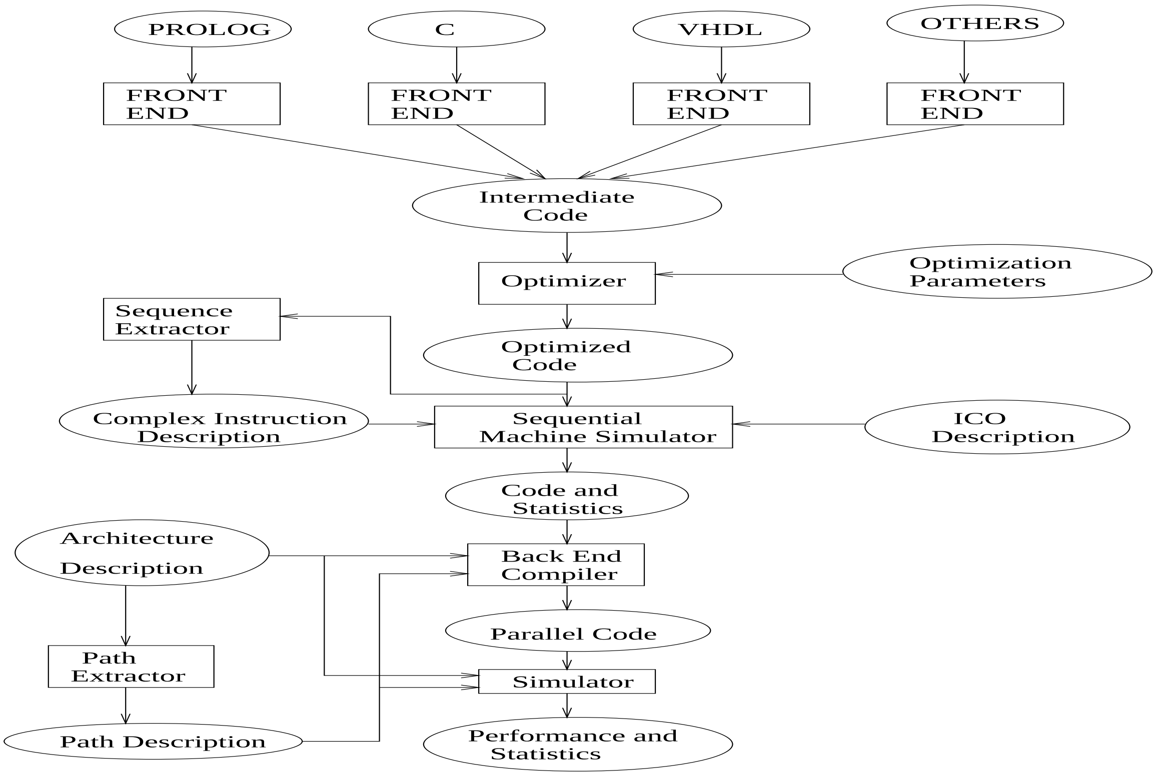

Figure 1: The architecture evaluation system : Glario [2]

The input application defined in high level language is translated into an intermediate code composed of simple instructions directly expressing primitive hardware functionalities. Some optimizations like loop unrolling, data-dependence analysis, variable renaming and loop interchanging are applied to increase the degree of parallelism. A sequential code simulation is performed to validate the code, and to extract some early features which can influence preliminary architectural choices. e.g.

frequently executed sequence of instructions can be substituted by introduction of new instruction. A parametric compiler is used to generate object code of the application for the specified architecture. A simulator is used to estimate the performance.

2.2 Analytical

Such approaches [7, 8] take a very simple architecture model in the input. A suitable internal representation of the input application is generated. Application is run on host machine to gather profile information. A suitable processor configuration is chosen by an explorer using a scheduler for performance estimation. An illustrative approach of this class suggested by Ghazal et al [8] (figure 2) is briefly described here.

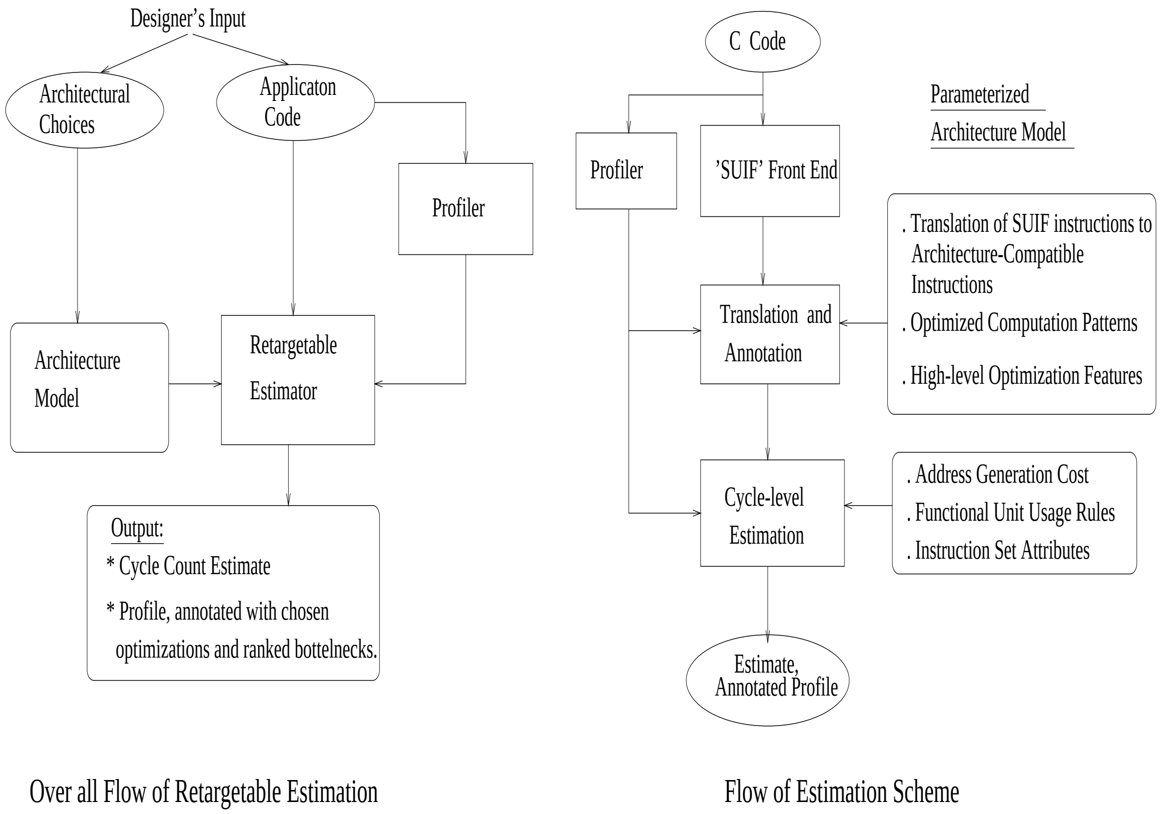

Figure 2: Retargetable Estimation Scheme : Ghazal [8]

Each processor architecture considered for evaluation is described by the architecture model (designer can chose from a database of previously captured architecture or can enter his own). Input application is described in high level language (C). Prediction of optimal run-time behavior is achieved through profiling and a series of passes that search for generic optimizations, and more importantly, for the application-specific optimization features. The estimator provides the designer with a total cycle count estimate, and a profile of the code, annotated with the chosen optimizations and a ranking of the dominant code segments.

2.3 Simulator based v/s Analytical Approaches

Each class of approach has benefits and limitations.

- An efficient retargetable compiler is required for simulator based approaches which may provide an optimized code for each of the architecture choice (it is very difficult task, because optimized code generation needs in depth knowledge of architectures and in depth knowledge of all architectures being explored is not possible). Analytical approaches does not require assembly code generation.
- Simulator approaches take more time, so with the increase in the size of design space it becomes infeasible.
- Analytical approaches estimates are not as accurate as of simulator based approaches.
- Very limited architectural features are currently being explored by analytical approaches.

We find that analytical approaches are more desirable with increasing dimensions and size of the design space to be explored. Since the analytical approaches have not explored storage organization while selecting suitable configuration, we propose to focus on exploring register file size and various memory configuration in our analytical approach.

3 Exploring Register File Size and Memory Configuration

We now look at some work done on exploring storage architecture using simulator based approaches for ASIP synthesis. Gloria et al [2] consider register bank organization and various memory hierarchy structures (I-cache, D-cache, bank interleaving etc) while evaluating application specific architecture in their compiler-simulator based framework. A parametric compiler is used which takes architecture description (including register bank and memory hierarchy) as the input and generates code for particular architecture. Some static information including maximum number of simultaneously live variables is extracted from the generated code for particular architecture. Dynamic information extracted includes most frequently executed parallel instructions. An event driven simulator is used to get performance and resource utilization statistics.

The approach suggested by Gong et al [3] considers register file (number of registers and number of ports) and flat memory modules in their architecture module. A parallelizing compiler is used to extract program parallelism on the code generated by GCC. A ultra-fine scheduler is used to exploit hardware parallelism offered. It produces parallel control code taking the code generated by GCC or by parallelizing compiler in the input. A simulator is used to obtain performance metrics.

Kin et al [5] consider size of cache memory units in their parameterized architecture model. A retargetable ILP compiler (IMPACT) and a simulator (L_{SIM}) was used to get performance.

Honma et al [9] proposed an interesting approach to optimize register count in ASIPs. The method dynamically analyzes the application program only once and uses the static analysis result to estimate the execution cycles. Total execution cycles of the application was considered as a sum of execution cycles needed to perform original functionality of the application program and extra execution cycles due to insufficient register count to perform the load/ store instructions, which can be estimated by using the basic blocks frequency count and the number of load/ store instructions in each basic block. Their method needs generation and investigation of object code of the application program on particular architecture to find the number of load/ store instructions in basic block.

Mishra et al [6] proposed a language based approach to explicitly capture the memory sub system configuration and perform exploration of the memory architecture to trade-off cost versus performance. The memory system is described in an architecture description language. The structure used represents the abstract memory modules (e.g. caches, stream buffers, RAM modules), their connectivity and characteristics (e.g. cache properties). This description which contains information of processor core and memory subsystem) is used to generate a customized compiler and a customized simulator.

We plan to develop a fast retargetable performance estimator considering register file size and various memory configurations without generating object code for target architecture as well as without using retargetable compiler and simulator.

4 Proposed Methodology

Work proposed during this study will be integrated to the ASIP design methodology (ASSIST), being developed in a DST-DAAD project. The overall flow diagram of ASSIST methodology is shown in figure 3.

The application is analyzed with the help of a profiler to extract application parameters. Design space exploration is an iterative process and it starts with a basic configuration (or minimal) that would be synthesized. The performance estimator (where this study is going to contribute) estimates performance based on present processor and memory configuration, application parameters and input models. The configuration selector compares estimates to the user specified constraints to generate next potential configuration. This process is iterated until a satisfactory configuration is generated which is used by a retargetable compiler generator in generating a customized compiler and by a VHDL synthesizer to generate a synthesizable VHDL code for processor.

Our work will be focused to develop an efficient retargetable performance estimator for RISC class of architectures, considering the flexibility is storage organization, namely register file size (number of registers) and various memory configurations.

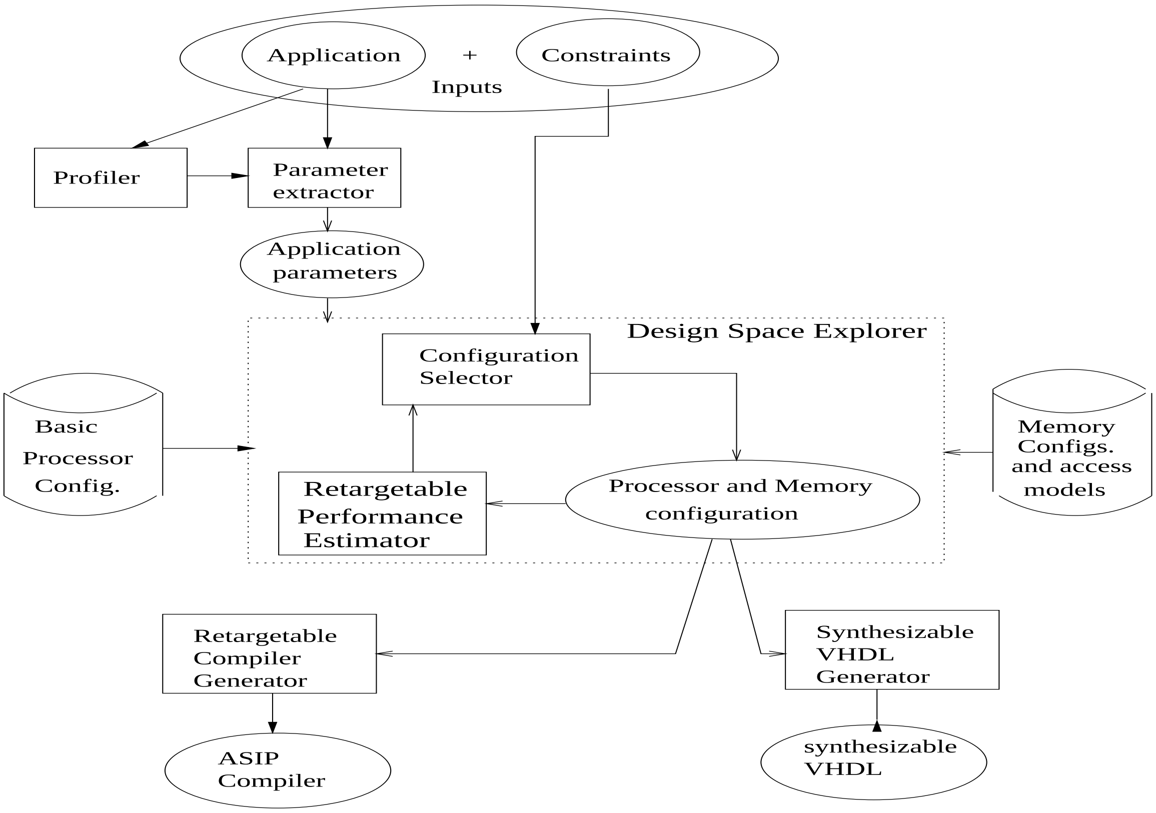

Figure 3: ASSIST: Flow diagram of ASIP design methodology

5 Approach to be followed and Status of Work

We plan to develop a fast retargetable performance estimator considering register file size and various memory configurations. A brief description of our overall framework is described in last section.

5.1 Optimizing Register File Size

Optimizing register file size is very important in processor design. Very small register file size degrades performance significantly, whereas larger register files incurs overheads in terms of cost, area, power, instruction width etc. My Dortmund visit was used to study the influence of register file size variation (3 to 8) on power, performance and code size [10]. Study is made for ARM7TDMI processor in a simulator based framework. Results emphasizes the need for optimizing register file size. We are evolving a suitable analytical approach to study the impact of register file size variation on performance in a very short time. Our approach does not need assembly code generation for target architecture or simulator.

5.1.1 Study the Impact of Register File Size on Power and Performance

As indicated above, we studied the impact of register file size variation on power, performance and code size. Static information about spilling is also gathered. Experimental framework include a parameterizable code generator *encc* being developed at University of Dortmund, Germany [11]. Front end assembler and linker are used to produce an executable code. Trace is generated using a simulator front end. The generated trace is analyzed by an analyzer to get performance, power and other statistical information. Two different memory configurations were studied, one considering both instruction and data are stored in off-chip memory whereas other considering on-chip memory for instructions. Selected benchmarks applications were analyzed to find a suitable reasoning for the observed behavior of the results. Detail results are available in [10]. Results indicated that an increase in the number of registers by one can result up to 57.5% performance improvement and up to 62.9% reduction in energy saving. Results clearly shows the importance of choosing an optimal register file size.

5.1.2 An Analytical Approach to Optimize Register File Size

At present we are evolving an analytical approach for quick performance estimation. Our proposed approach is described here.

Overview First we will profile the application using gprof to find execution count of each basic block. We will find base performance (without spilling, assuming unlimited registers are available) of the application by finding schedule for each basic block using a resource constrained scheduler (as implemented by Gupta et al [7]). Apart from infinite size register file this scheduler assumes that various blocks will be executed sequentially and operations with in a basic block will be executed in parallel limited by data dependencies. This scheduler takes a very simple architecture description (mainly operation and FU latencies). Once the execution time of a block is found (in terms of cycles), it is multiplied with its frequency count. This way execution time of all blocks are estimated and added to get total execution time in terms of number of cycles. Then we will find overhead due to insufficient registers and it will be added to the base execution time to get total execution time.

Overhead Estimation We would like to use concept of register reuse chain (RRC) as given by Zhang et al [12]. A RRC contains the set of nodes which can be assigned to same register. In this paper the RRCs are constructed for the purpose of register allocation. We would use these chains to estimate the increase in the schedule length and spilling overheads for a given limit on number of registers. Initially each RRC will have all the variables whose life can not be overlapped by the any possible schedule. If such RRCs are less than number of registers available then there will not be any overhead. Variables in different RRCs doesn't mean that their life time will always overlap, but it means that there exist some schedule in which life time of nodes of two RRCs will overlap.

Use of RRCs for register allocation is as follows. A directed conflict graph is generated by detecting dependence violation. A node in conflict graph represent a RRC whereas an edge represent that corresponding RRCs cannot be merged. An edge has a direction that represents the way two chains can not be merged. An edge from chain[i] to chain[j] means that chain[i] cannot reuse the register assigned to chain[j]. Thus these two chains cannot be merged in such a way that the tail of chain[j] is connected to the head of chain[i]. A bidirectional edge between nodes implies that these nodes cannot be merged into one RRC either way. This directed conflict graph is then converted into an undirected graph. Graph-coloring algorithm is used to assign colors to the chains. The chains assigned to same color are merged into a single chain.

The merge process generates additional dependencies. Performance overhead due to these additional dependencies is to be estimated. If the merge process as described above cannot reduce the number of RRCs to the level of number of registers, then option for spilling a variable to be considered and its overhead to be estimated. The estimated overhead is multiplied by the frequency count of the basic block to get total overhead for that particular basic block. It is to be repeated for all basic blocks to get overall performance overhead. We will implement and validate our strategy.

5.2 Optimizing Memory Configuration

We are in the process to find a suitable strategy to explore various memory configurations (e.g. cache should present or not, cache (instruction/ data) sizes, on-chip off-chip configurations etc). We plan to consider this feature also in our fast retargetable performance estimator. Once we design a strategy for this, we will implement and validate the same.

6 Acknowledgments

This research work is partially supported by DST-DAAD cooperation project (Project code MCS-216) **ASSIST** (**AS**ip **S**ynthes**IS** Me**T**hodoloy) and NRB sponsored embedded systems project **ASSET** (**A**utomated **S**ynthesi**S** of **E**mbedded Sys**T**ems).

References

[1] Jain, M.K.; Balakrishnan, M.; Anshul Kumar : “ASIP Design Methodologies : Survey and Issues ”, Proceedings of the Fourteenth International Conference on VLSI Design, 2001, 3-7 Jan. 2001, Pages: 76-81.

[2] Gloria A. D.; Faraboschi, P. : “An evaluation system for application specific architectures ”, Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture. (Micro 23), 27-29 Nov. 1990, Pages: 80-89.

[3] Gong, J.; Gajski, D.D.; Nicolau, A. : “Performance evaluation for application-specific architectures ”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Dec. 1995 Vol. 3 Issue: 4, Pages: 483-490.

[4] Kienhuis, B.; Deprettere, E.; Vissers, K.; van der Wolf, P. : “The construction of a retargetable simulator for an architecture template ”, Proceedings of the Sixth International Workshop on Hardware/Software Codesign 1998 (CODES/CASHE ’98), 15-18 March 1998, Pages: 125-129.

[5] Kin, J.; Chunho Lee; Mangione-Smith, W.H.; Potkonjak, M. : “Power efficient mediaprocessors: design space exploration ”, Proceedings of the 36th Design Automation Conference 1999, 21-25 June 1999, Pages: 321-326.

[6] Mishra, P.; Grun, P.; Dutt, N.; Nicolau, A. : “Processor-Memory Co-Exploration driven by a Memory-Aware Architecture Description Language ”, Proceedings of the Fourteenth International Conference on VLSI Design, 2001, 3-7 Jan. 2001, Pages: 70-75.

[7] Gupta, T.V.K.; Sharma, P.; Balakrishnan, M.; Malik, S. : “Processor evaluation in an embedded systems design environment ”, Proceedings of Thirteenth International Conference on VLSI Design 2000, 3-7 Jan. 2000, Pages: 98-103.

[8] Ghazal, N.; Newton, R.; Jan Rabaey. : “Retargetable estimation scheme for DSP architecture selection ”, Proceedings of the Asia and South Pacific Design Automation Conference 2000 (ASP-DAC 2000), 25-28 Jan. 2000, Pages: 485-489.

[9] Honma, Y.; Imai, M.; Shiomi, A.; Hikichi, N.: “Register Count Optimization in Application Specific Integrated Processors ”, Proc. of Second Asia Pacific Conf. on Hardware Description Languages (APCHDL ’94), Oct. 1994, Pages: 263-266.

[10] Jain M.K.; Wehmeyer L.; Steinke, S.; Marwedel P.; Balakrishnan, M. : “Evaluating Register File Size in ASIP Design ”, accepted for presentation in Ninth International Symposium on Hardware/ Software Codesign, CODES 2001, 25-27 April, 2001, Copenhagen, Denmark.

[11] Wehmeyer L.; Jain M.K.; Steinke, S.; Marwedel P.; Balakrishnan, M. : “Using an Energy Aware Compiler Framework to Evaluate Changes in Register File Size towards ASIP-Design ”, accepted for presentation in Fifth International Workshop on Software and Compilers for Embedded Systems, SCOPES 2001, 20-22 March, 2001, St. Goar, Germany.

[12] Zhang Y.; Lee H.J. : “Register Allocation Over a Dependence Graph ”, proceedings of the Second International Workshop on Compiler and Architecture Support of Embedded Systems, CASES 1999, 1-3 Oct, 1999.