Customizing Embedded Processors for Specific Applications

Anshul Kumar, M. Balakrishnan, Manoj Kumar Jain and Anup Gangwar

Department of Computer Science and Engineering,
Indian Institute of Technology Delhi, Hauz Khas, New Delhi-110016, India
{anshul,mbala,manoj,anup}@cse.iitd.ernet.in

Abstract Complexity of embedded applications is growing rapidly. This growth is accompanied by severe implementation contraints on cost, size, performance as well as power. This has resulted in search for expanding the architectural design space for implementing such complex applications. The last decade has seen the growth of Application specific instruction processors(ASIP) as an alternative to general purpose processors on one hand and Application specific integrated circuits (ASIC) on the other. ASIPs are considerably more flexible than ASICs while being more performance and power efficient than general purpose processors. The key issue in designing the ASIPs relate to customization i.e. identify the critical application characteristics which need to be supported by special hardware to create an application specific processor. In this paper we address two specific research problems; storage customization of a RISC processor and design of clustered VLIW processor.

A simple RISC processor is characterized by a single functional unit, pipelined instruction execution, a single fixed register file and unified cache and memory for data and instruction. We propose techniques for customizing the register file size, number of rgister windows used to store contexts, cache size and trading off cache with scratch pad memory. This is implemented in a single integrated framework and results in an application specific storage optimization. This approach has been validated by actual experiments on both ARM as well as LEON (a varient of SUN-SPARC) processors.

VLIW processors are extremely effective in applications with a large fine grain parallelism. These are preferred over superscalar in embedded systems because the task of identifying instruction level parallelism is handled at compile time rather than at runtime to save on power and chip area. The number of functional units in a VLIW is constrained by the number of ports in the register file as in each clock, operands need to be supplied to all the FUs simultaneously. The register file delay as well as power consumption increases as a square of the number of ports and due to this, researchers are considering clustered VLIW processors as an alternative. These have clusters of FUs (homogeneous or heterogeneous) connected to each register file. Now the data transfer time between these clusters depends on the interconnetivity and thus the design space is considerably enlarged. Further, operation-operator binding is now much more closely coupled to the value-register binding as the operand not being in the the associated register file can have considerable impact on performance. In this paper we define a range of clustered VLIW architectures followed by techniques for performance evaluation of these processors. Results show that the normally used RF-RF transfers perform very poorly.

1 Introduction and Role of Architecture Customization

Today Embedded Systems on a Chip (SoCs) can be found in a large variety of applications like image processing, computer vision, networking, wireless communication etc. Typically these applications demand real time processing and high throughput. Many of these applications, specially in the domain of image processing and computer vision, also offer good amount of functional and data parallelism.

Architecture customization leads to design solutions which are cheaper cost-wise as well as satisfy the constraints on performance and power consumption tightly. The *General Purpose Computing* domain doesn't offer much opportunity for architecture customization as it is not known in advance which application will run on the target architecture. However, in embedded systems, the application is known in advance and as a consequence it is possible to analyze the application rigorously and fine tune the architecture. Thus Application Specific Instruction Processors (ASIPs) are important components of SoCs. An ASIP exploits special characteristics of the given application(s) to meet the desired performance, cost and power requirements. ASIPs offer the required flexibility (which is not provided by ASICs) at a lower cost than general programmable processors.

Architecture customization or specialization, in embedded system can be *System Specialization* or *Component Specialization*. In System Specialization, the system is considered as a whole with multiple threads of control. The application is partitioned and is mapped to the control components and the data components. In Component Specialization the individual component e.g.. processor, memory, interconnection are considered with a single thread of control. Component specialization may involve one or more of the following:

1. *Instruction Set Specialization:* concentrate on instruction level parallelism , operations which can be chained (multiply-add), instruction size reduction, time and power reduction.
2. *Data Path Specialization:* concentrates on word length adaptation, number of FUs, application specific FUs, register file size and register structure.
3. *Memory Specialization:* concentrates on number and size of memory banks, number and size of access ports, access patterns (paged mode , inter-leaving, cache organization and special memory introduction e.g. scratch pads, FIFO.
4. *Interconnect Specialization:* includes interconnection paths and protocols. The functionality, in terms of power and cost is reduced as compared to some standard bus.

Extensive research has been done on designing application specific processors over the last decade. Section 2 summarizes various approaches followed by different researchers. The next two sections deal with two areas which were relatively less explored and have been the focus of research at IIT Delhi. These are: customizing on-chip storage (Section 3) and customizing high performance ASIPs (Section 4).

2 Designing Customized Processors: Alternative Approaches

2.1 ASIP Design Methodology

Gloria et al. [12] defined some main requirements of the design of application-specific architectures. Important among these are as follows.

- Design starts with the application behavior.
- Evaluate several architectural options.
- Identify hardware functionality to speed up the application.
- Introduce hardware resources for frequently used operations only if it can be supported during compilation.

Various methodologies have been reported in literature to meet these requirements. It is found that typically there are five main steps followed in the synthesis of ASIPs. The steps are shown in Figure 1.

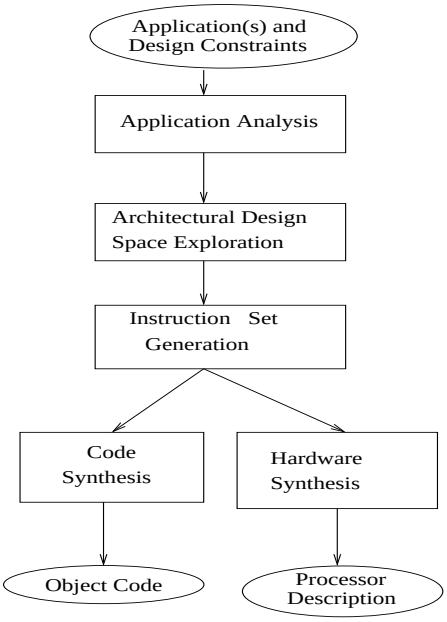


Figure 1. Flow diagram of ASIP design methodology

Application Analysis: Input in the ASIP design process is an application or a set of applications, along with their test data and design constraints. It is essential to analyze the application to get the desired characteristics/ requirements which can guide the hardware synthesis as well as instruction set generation. An application written in a high level language is analyzed statically and dynamically and some parameters useful for the subsequent steps are extracted. These parameters include data types and their access methods, execution counts of the operators and functions, frequency of the instruction sequences, life time of the variables, data access requirements, temporal and spatial locality in the program, the frequency of individual instructions and the sequences of contiguous instructions, average basic block size, number of operation patterns such as Multiply-Accumulate (MAC) operations, ratio of address computation instructions to data computation instructions, ratio of input/output instructions to the total instructions, average number of cycles between generation of a scalar and its consumption in the data flow graph etc.

Architectural Design Space Exploration: First a set of possible architectures is identified for a specific application(s) using output of step 1 as well as the given design constraints. Performance of possible architectures is estimated and suitable architecture satisfying performance and power constraints and having minimum hardware cost is selected.

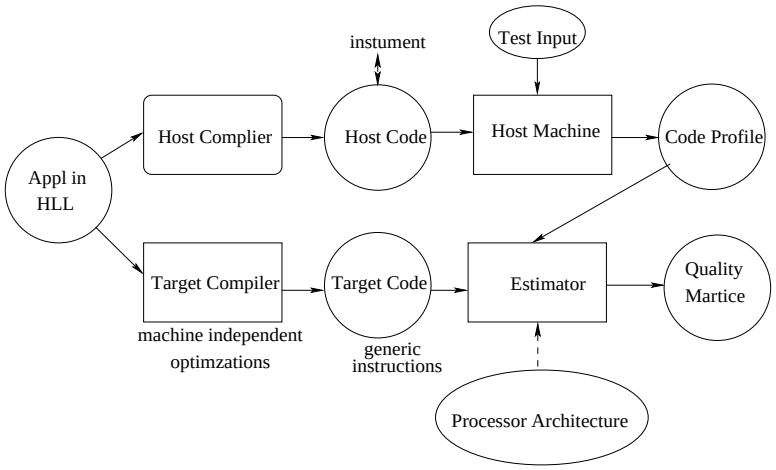


Figure 2. Scheduler Based Approach

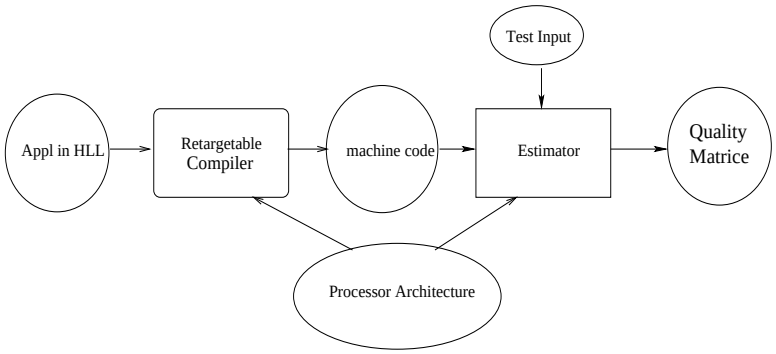


Figure 3. Simulator Based Approach

Broadly there are two type of approaches for performance estimation. These are *scheduler based* and *simulator based*. In *scheduler based* approach the problem is formulated as a resource constrained scheduling problem with the selected architectural components as the resources and the application is scheduled to generate an estimate of the cycle count. Profile data is used to obtain the frequency of each operation, as shown in Figure 2. In *simulator based* approach, a simulation model of the architecture based on the selected features is generated and the application is simulated on this model to compute the performance as shown in Figure 3. Using a retargetable, a spe-

cific simulator instance can be derived for each architecture instance of the architecture template.

Instruction Set Generation: Instruction set is generated for that particular application and for the architecture selected. This instruction set is used during the code synthesis and the hardware synthesis steps.

One approach for instruction set generation is to synthesize an instruction set for a particular application based on the application requirement, quantified in terms of the required micro-operations and their frequencies. Micro-operations can be suitably grouped to form instructions of different complexity depending upon the performance requirement. Alternatively, it may be assumed that a super-set of instructions is available and a subset of these is selected to satisfy the performance requirements within the architectural constraints. More complex instructions can also be created by pattern matching. A set of template patterns is extracted from the instruction set and the graph representing the intermediate code of the application is covered by these patterns. Each pattern defines a complex instruction.

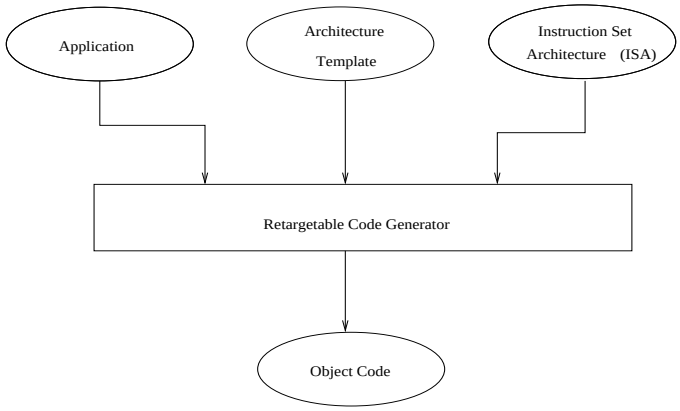


Figure 4. Retargetable Code Generator

Code Synthesis: Synthesizing a customized processor gives rise to the need for generating the object code which will be specific to the new Processor Architecture. Code Synthesis could be done by using a Retargetable Code Generator which will take application, the architecture template and the instruction set as inputs and generate code for the specific processor. Figure 4 illustrates Retargetable Code Generator.

Code Synthesis could also be done using the Customized Compiler generated by a compiler generator for a range of architectures. The customized compiler will take the Application and the code generated by a Retargetable Compiler. Figure5 shows this approach. Retargetable Compiler requires architecture template and instruction set architecture as input and produces a customized compiler.

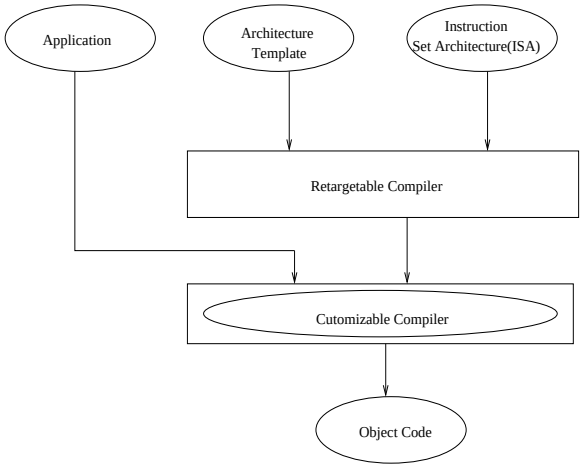


Figure 5. Compiler Generator

Hardware Synthesis: In this step the hardware is synthesized using the ASIP architectural template and instruction set architecture starting from a description in VHDL/ VERILOG using standard tools.

Every methodology does not emphasize all these steps. Some of them consider the processor micro-architecture to be fixed while only generating the instruction set within the flexibility provided by the micro-architecture, e.g. [15], while others consider the process of instruction set generation only after the parallelism and functionality of the processor micro-architecture is finalized based on the application, e.g. [17] etc.

2.2 Architectural Design Space

A good parameterized model for the architecture is very important for design space exploration. Different values can be assigned to the parameters (keeping design constraints into consideration), to derive various architecture instances. The design space will depend on the number of parameters and the range of values which can be assigned to these parameters.

The parameterized architecture model invariable includes the number of functional units of different types, storage units and interconnect resources, structure of pipelining in the pipelined functional units, component like buffers, controllers, routers and their composition rules.

Kin et al [24] consider issue width, number of branch units, number of memory units, the size of instruction cache and size of data cache etc. in the model. Gupta et al [16] use a model which include parameters like number of registers, number of operation slots in each instruction, concurrent load/store operations and latency of functional units and operations. Ghazal [11] include optimizing features like addressing support, instruction packing, memory pack/ unpack support, loop vectorization, complex arithmetic patterns such as dual multiply-accumulate, complex multiplication etc in their architectural model. This results in a large design space. They have developed a retargetable estimator which takes advantage of such an architecture model.

Architectures considered by different researchers also differ in terms of the instruction level parallelism they support. For example [7] and [23] do not support instruction level parallelism, whereas [13] and [16] support VLIW architecture and [11] and [24] support VLIW as well as super scalar architecture.

Most of these approaches consider only a flat memory. Only [24] addresses consideration of instruction and data cache sizes during design space exploration, but the range of architectures explored is rather limited. Similarly no approach considers flexibility in terms of number of stages in a pipeline, though a pipelined architecture is considered in [7] and [12].

3 Customizing On-chip Storage

As discussed in the previous section, design space exploration is driven by performance estimations. These estimates are generated using a simulator based or scheduler based framework. Simulator based technique needs a retargetable compiler to generate code for different processor configurations to be explored. Further, simulating the generated code is slow.

On the other hand, the scheduler based approach is much faster and quite suitable for an early design space exploration. However, on-chip storage which includes register files and cache is not explored by the scheduler based approaches reported so far. Thus, the main focus of our work has been to include on-chip storage exploration as part of the design space exploration.

Our previous study [20] using a retargetable code generator and standard simulator has indicated that choice of an appropriate number of registers has a significant impact on performance and energy in certain cases. In ASIP synthesis, it is important to select appropriate on-chip storage after examining trade offs between various choices. For example trade-off between register file size and cache size can be evaluated. Similarly, for a given register file size, trade-off between number of register windows and window size can be evaluated. Our approach is to first find the execution time ignoring the influence of storage constraints on it and then add overheads due to register spills because of limited register file, window spills and restores due to limited register windows and cache misses.

In scheduler based performance evaluation, register allocation is the key step which helps in determining the influence of limited register file size on the performance. A usual approach is to perform register allocation either after scheduling like a typical compiler[14], or solve register allocation and scheduling in an integrated manner[5]. However, register allocation before scheduling is desirable when minimizing of register file size is more important than the length of the code sequence. Our technique does register allocation before scheduling using the concept of register reuse chains [36] with significant extensions.

3.1 Integrated On-chip Storage Evaluation Methodology

Our approach to storage exploration as a part of the overall design space exploration is shown in figure 6. Cycle count for application execution on the chosen processor and

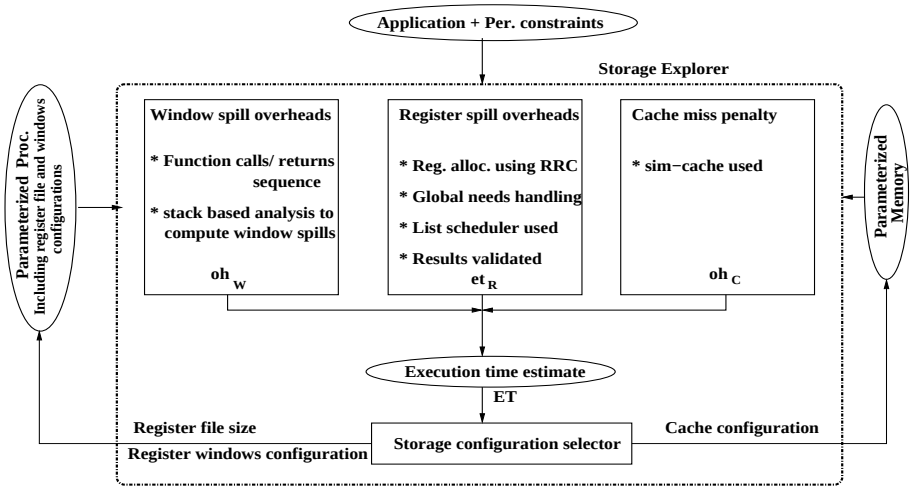


Figure 6. Storage exploration technique

memory configuration is estimated using a parameterized model for processor as well as memory. Parameters of data cache include size, line size, associativity, replacement policy and access time. Processor configuration specification includes register file and windows organization along with pipeline information and functional unit (FU) operation capability and latency.

Register allocation is done on unscheduled code using the concept of reuse chains [36] with significant extensions [21].

Overall execution time estimate (ET) for an application for the specified memory and processor configuration can be expressed as follows.

$$ET = et_R + oh_W + oh_C \tag{1}$$

where
et_R : Execution time including overhead due to limited size of register file R,
oh_W : Additional schedule overhead due to limited windows, and
oh_C : Additional schedule overhead due to cache misses.

et_R can be further expressed by the following equation:

$$et_R = bet + oh_{dep} + spill_R * t_R \tag{2}$$

Computation of et_R is described in the next subsection. oh_W can be further expressed by the following equation:

$$oh_W = spill_w * t_W \tag{3}$$

where
$spill_w$: Number of window spills, and

t_W : Delay associated with each register window spill.
oh_C can be further expressed by the following equation:

$$oh_C = miss_C * t_C \tag{4}$$

where
$miss_C$: Number of cache misses, and
t_C : The cache miss penalty.
t_W is computed by knowing register window size and the latency of 'store' instruction. t_C is computed using block size and the delays associated in each data transfer. Storage configuration selector selects suitable processor and memory configuration to meet the desired performance by knowing all the execution time estimates.

3.2 Execution Time Estimation with Limited Registers

Input application (written in C) is profiled using *gprof* to find execution count of all basic blocks for each function. These execution counts are used to multiply with the estimated execution times.

For each basic block B, local register allocation is performed taking the data flow graph and number of registers to be used for local register allocation (say k) as input using a modified register reuse chains approach [21]. Data flow graph may be modified because of additional dependencies as well as spills inserted during register allocation. This modified data flow graph is taken as input by a priority based resource constrained list scheduler, which produces schedule estimates. This estimate is multiplied by the execution frequency of block B to compute local estimate ($LE_{B,k}$) for this block.

Local estimates are produced for all the basic blocks contained in a function, for the complete range of register file sizes to be explored. Schedule overheads needed to handle global needs with limited number of registers are computed using life time analysis of variables. For each block, we need information on variables used, defined, consumed, live at entry and exit points of this block. This additional global needs overhead is also generated for the complete range of number of registers for each basic block. Then, we decide on the optimal distribution of the registers available (say n) into registers to handle local register allocation (k) and registers to handle global needs ($n-k$), such that overall schedule estimate for that block is minimized.

Overall estimate for a block B can be expressed as

$$OE_B = min_k(LE_{B,k} + GE_{B,n-k}) \tag{5}$$

where OE_B is the total schedule estimate for basic block B, and $GE_{B,n-k}$ is the overhead to handle global needs with $n-k$ registers. OE_B values for all blocks are summed up to produce estimates at the function level. Estimates for all functions are added together to produce overall estimate for the application i.e. et_R. So et_R can be expressed as

$$et_R = \sum_{for\ each\ function} \ \sum_{for\ each\ basic\ block\ B} (OE_B) \tag{6}$$

3.3 Overhead of Limited Reg. Windows

Processors with register window scheme typically assume a set of registers organized as a circular buffer. When a procedure is called (means a context switch), the tail of the buffer is advanced in order to allocate a new window of registers that the procedure can use for its locals. On overflow, registers are spilled from the head of the buffer to make room for the new window at the tail. These registers are not reloaded until a chain of returns makes it necessary. We consider the context switches that are due to function calls and returns.

The number of register window spills and restores can be estimated by examining a trace of calls and returns from the application execution[6]. Trace is given as input to a simple stack based analyser to compute window spills and restores. Trace is used only once and simultaneously spills and restores for different number of register windows are computed[27].

Window spills ($spill_W$) due to limited number of register windows computed in this manner are used in (equation 3), the execution penalty due to window spills and restores (oh_W) is computed using window size and latency associated with load and store operation for the processor. This additional penalty is added into the estimates produced by the technique to produce overall execution time estimates.

3.4 Overhead of Cache Misses

We observe that usually the number of memory locations required to store spilled scalar variables and register windows is small compared to the total number of cache locations. Therefore, we assume that the spilling overhead is insensitive to the cache organization. This observation allows us to estimate the two independently.

To know the memory access profiles (total number of accesses, hits, misses etc.) we need to generate addresses to which memory accesses are made and then a simulator is required to simulate those accesses. Since memory access patterns are typically application dependent, we can use any standard tool set to find memory access profiles. Once we know the number of memory misses for a particular cache, knowing the block size and delay information we compute the additional schedule overhead due to cache misses in a straight forward manner.

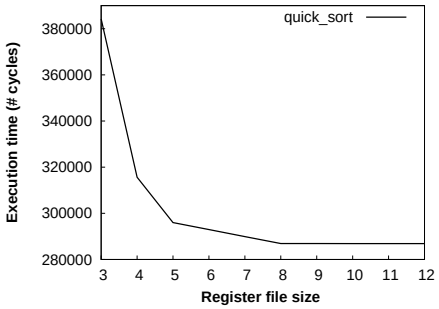
We used *simplescalar* tool set [2] to compute the number of cache misses for different cache sizes. It assumes a MIPS like processor with some minor changes. Since we do not require timing information from the simplescalar tool, we use *sim-cache* simulator which is fast compared to *sim-outorder*. *sim-cache* does not use processor timing information and it gives only cache miss statistics. It takes only address trace along with cache configurations as input.

3.5 Results

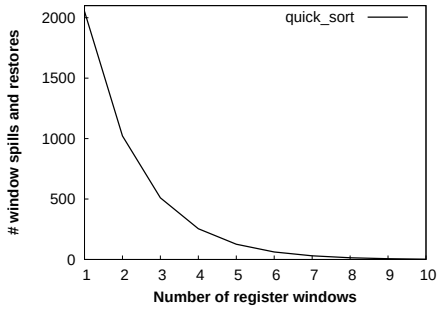
Figure 7(a) shows the impact of variation of register file size on execution time for *quick_sort*. Results indicate that execution time decreases with an increase in the register file size, but saturating after a certain register file size is reached. Variation of number of window spills and restores required with different number of register windows is

shown in figure 7(b). Curve saturates at 9 windows. Trade-off between the number of registers and window sizes is shown in Figure 7. Here we assumed that register file will be distributed in windows of equal sizes.

On one end, when the number of windows is small, the time overhead due to context switches dominates the cycle count. At the other extreme, when the number of windows is large for the same total number of registers, the individual window size becomes small and the overhead due to load and stores (within a context) dominates the cycle count.



(a) Impact of Register file size on execution time



(b) Impact of number of registers windows

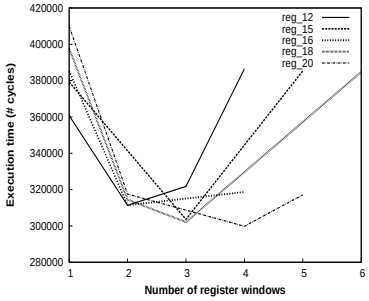


Figure 7. Trade-off between number of windows and their sizes

Execution time estimates for various benchmark applications for different register file sizes and different data cache sizes are shown in Figure 8. Based on the generated execution time estimates and the input performance constraint, suitable configurations can be suggested. For example, if the application, *matrix_mult*, should not take more than $1.0E+05$ cycles, then one of the following configurations can be suggested.

1. 12 registers with 4K data cache
2. 15 registers with 2K data cache
3. 20 registers with 1K data cache

To validate the accuracy of our estimator, performance estimations with varying on-chip storage configurations for selected benchmarks applications were done with

three real processors. These are: ARM (*ARM7TDMI* a RISC) [1], Trimedia (*TM-1000* a VLIW) [3] and *LEON* (a processor with register windows) [10]. *TM-1000*'s five-issue-slot instruction length enables up to five operations to be scheduled in parallel into a single VLIW instruction. To know correctness of our techniques, we chose to validate our result against the numbers produced by standard tool sets.

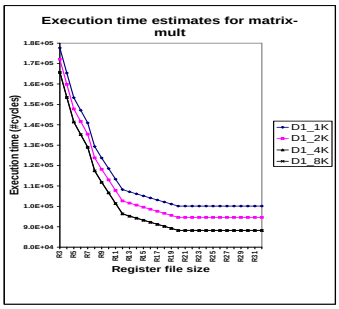


Figure 8. Results for *matrix-mult*

Validation shows that our estimates are within 10% compared to the actual performance numbers produced by standard tool sets. The actual figures were 9.6%, 3.3% and 9.7% for *ARM7TDMI*, *TM-1000* and *LEON* respectively. Further, this technique was nearly 77 times faster compared to the simulator based technique. Validation results for *LEON* is shown in figure 9.

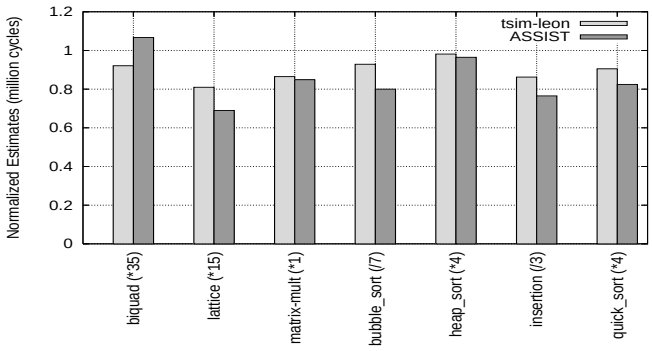


Figure 9. Validation on *LEON*

Results generated were also validated against VHDL level simulation for collision detection application which is developed at IIT Delhi for detecting collision of an object with Camera [26]. The execution time estimates produced by our estimator (443278 cycles) are within 10.33% compared to the estimates produced by tsim (494375 cycles) and within 5.26% compared to the estimates produced by VHDL simulation.

3.6 Benefits of RF Customization

Optimizing register file size is useful in many ways. Some of them are listed here.

- If smaller register file can be used then register address needs fewer bits. Thus, we can think of either reducing the instruction width or providing more room for op-code so new application specific instructions could be easily accommodated.
- Reduction in the switching activity and thus saving in terms of power consumption.
- In case, instruction width cannot be reduced by sparing some registers, these registers or their addresses could be used efficiently for specific purposes. For example, hard-wiring some registers with fixed values will help removing some moves. There are other possibilities as well.

Interestingly, spare register addresses may be used to address co-processor registers. Interface between processor and coprocessor is responsible for transmitting operands from processor to co-processor and results from co-processor to processor. Design of these interfaces differ from processor to processor interface.

Some processors, like MIPS, allow direct transfer of values from main register file to co-processor register file and vice-verse. By using addresses of ‘spare’ registers predicted by our technique, to address co-processor registers, such data transfers can be saved.

Processor like *LEON* do not allow transfer of values directly between processor register file to co-processor register file, so this communication is through memory. This means, the operand values are first stored into memory and then, these values are loaded into co-processor register file. After co-processor had produced results, these results will be first stored into memory and then, they are loaded into main register file. Again by using addresses of ‘spare’ registers, significant benefits can be achieved in this case, because all such loads and stores can be avoided.

4 Customizing High Performance ASIPs

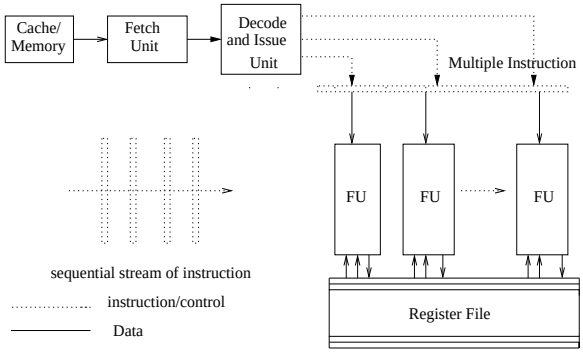


Figure 10. ILP in Superscalar Processors

The Instruction Level Parallelism allows two architectural options: superscalar (Figure 10) and VLIW (Figure 11). The Instruction Level Parallelism in Superscalar Processor is Hardware Driven It requires a complex decode and issue unit to handle the parallel execution of multiple instruction . There is a clear trade-off between the speed and complexity of this approach. The advantage of superscalar processor is code compatibility as the major decision of issuing multiple instructions to the execution unit is handled by the hardware unit . This also avoids the need for retargetable compiler. So any off the shelf compiler is sufficient for generating the code for superscalar processor. But the advantages of Superscalar processor doesn't fall in the ASIP domain as ASIPs requires retargetable compilers or compiler generator. The space available for customization in Superscalar Processors is very less. VLIW Processor is software driven where the partitioning of the application is done by the compiler which generates the object code for the specific VLIW architecture. VLIW Processors, with simplified hardware allows for several customization options and are suitable as ASIPs.

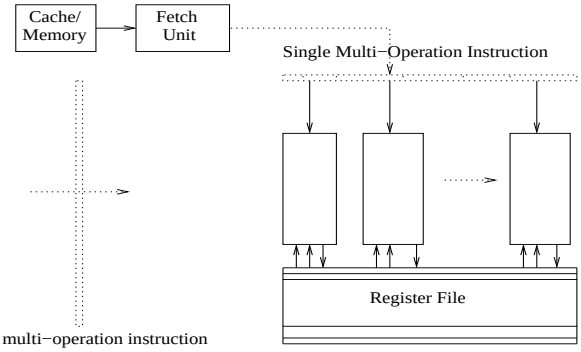


Figure 11. ILP in VLIW Processors

4.1 Architecture Design Space for custom VLIWs

VLIW processors lend themselves naturally to customization due to simplified hardware, however they suffer mainly from code expansion which needs to be tackled with a good instruction encoding scheme. The key specialization domains for a VLIW processor are [19, 30]:

1. No. and types of functional units
2. Register file structure
3. Interconnection network both between FUs and between register files
4. Instruction encoding for NOP compression

These have been illustrated in Figure 12. We next discuss each of these in detail.

Functional Units: The functional units lend themselves naturally to specializations [18, 29]. The core set of FUs shown in Figure 12, support the usual fine grain operations like add, multiply, compare etc. augmented by application specific extensions. These extensions are centered around some medium or coarse grain FUs defining new

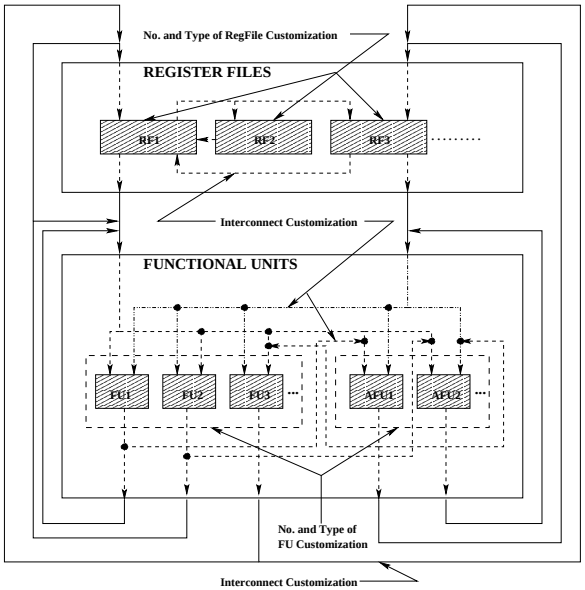


Figure 12. Customization in VLIW ASIPs

instructions for implementing some critical functionality of the specific application. Actually the fine grain core may not be absolutely rigid, it may be generic in some limited sense. The presence of core makes things easier by providing a default implementation for any part of the application.

At the one end of the spectrum there are multiple input single output units(MISO) without any memory accesses and control. This is the simplest generalization of basic fine grain operations which typically take one or two inputs and produce one result. The next conceivable generalization is to allow multiple outputs to be produced by an FU, making it a **MIMO** or a multiple input multiple output unit. The cycles in which various operands of a MIMO are input and results are output, relative to the beginning cycle, define the *I/O time-shape* of such a MIMO [8]. If the cycles in which I/O occurs are fixed, the time-shape is considered to be *rigid*. On the other hand, if the I/O operations are hold-able, that is the cycles in which they occur could be delayed, the time-shape is said to be *flexible* and it eases scheduling. A Basic MIMO takes all its operands from register files, therefore a further extension can be in terms of considering **MIMOs with load/store** which are capable of accessing the memory. Permitting conditionals within a FU further enhances its scope but it causes the latency of the FU to be variable which makes pure VLIW kind of scheduling difficult, one can even think of mapping loops to an FU with even the loop control inside the FU but again this will require run time hand shaking mechanism. Table 1 summarizes these possibilities.

Consider a Data Flow Graph for representing the operation in a program. In a Data Flow Graph each machine instruction is represented by a node. The appropriate subgraph can be executed as a single new machine instruction handled by the AFU. If this ad-hoc function unit (AFU) completes the execution faster then there is Gain. There

Name	Sources	Destinations	I/O Policy
MISO	Multiple (RF)	Single (RF)	Flexible or Rigid
MIMO	Multiple (RF)	Multiple (RF)	Flexible or Rigid
MIMO with LD/ST	Multiple (RF/Mem)	Multiple (RF/Mem)	Flexible or Rigid for RF, Block LD/ST at beginning and end of operation for Mem

Table 1. Architectural spectrum of custom FUs

could be various possible sources of Gain - *Spatial Computation*, *Chaining of operations*, *Exploiting Constants*, *Reducing Precision* (by performing bit-width analysis of variables declared in the program), *Arithmetic Optimization* (by exploiting arithmetic properties for efficient chaining of arithmetic operations). The parallelism at the basic block is relatively small. However, the classical ILP techniques *Predication* and *Loop Unrolling* could be used to expose large amounts of parallelism. In a recent study we have demonstrated that many common media applications can exhibit an ILP of as much as 20 on a VLIW processor with 16 ALUs and 8 LD/ST units [4].

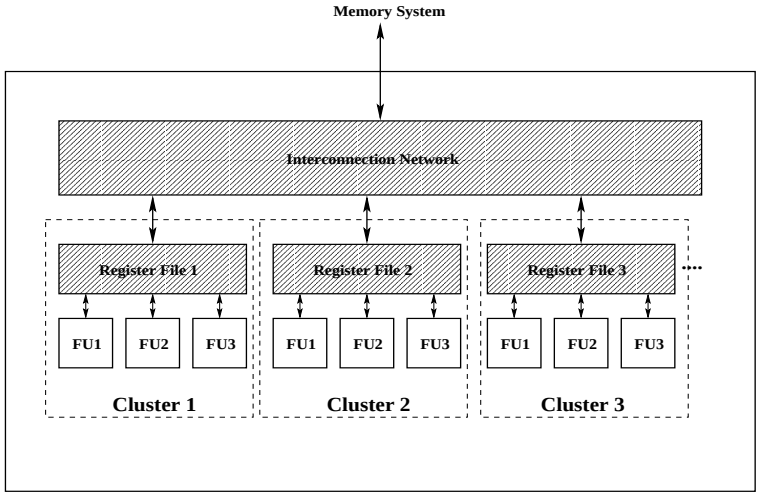


Figure 13. A Clustered VLIW Processor

Register File Structure: Traditionally, VLIW datapaths have been based on a single register file shared by all functional units. Unfortunately this single register file organization doesn't scale well with the large number of functional units of a high performance VLIW processor. For N arithmetic units connected to a register file the area, delay and power grow as N^3, $N^{3/2}$ and N^3 [31]. In short, as the number of functional units increases, the internal communication between functional units becomes the dominant factor for area, delay and power requirements. In order to overcome this limitation

the functional units can be *clustered* together, wherein they can read from or write to only a subset of registers. Figure 13 shows a typical clustered VLIW architecture. However this gain does come at a cost. In particular it may lead to data copying from one set of registers to another and thus increase execution latency. Since in embedded systems the application is known before hand it should be possible to customize the clusters in order to reduce this additional copying and the increase in latency using powerful optimizing compilers and interconnect analysis.

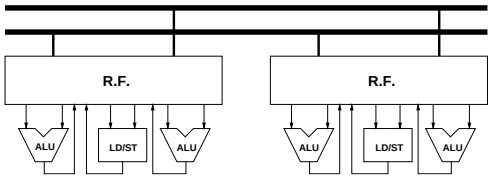


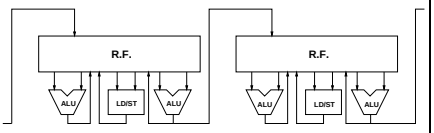
Figure 14. RF-to-RF ($\delta = 1$)

Interconnection Network: Clustered VLIW processors can be classified on the basis of their inter-cluster communication structures. At the top level we can divide them into two sub-categories: a) Those supporting inter-cluster RF-to-RF copy operations and b) Those supporting direct inter-cluster communication between FUs and RFs. There is very little variety in the architectures supporting RF-to-RF copy operations. At most these can be classified on the basis of the interconnect mechanism which they use for supporting these transfers. The examples of such architectures are: Lx [9], NOVA [19], Sanchez et. al. [32], IA-64 [34]. An example RF-to-RF architecture is shown in Figure 14.

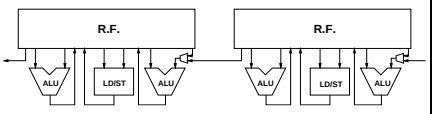
We use the RF→FU (read) and FU→RF (write) communication mechanisms to classify direct inter-cluster communication architectures. The reads and writes can be from either the same cluster or across clusters. The communication can be either using a point-to-point network, a bus or a buffered point-to-point connection. An underlying assumption is that FUs always have one path to their RF (both read and write) which they may or may not use. A few examples of these architectures are shown in Figures 15 and 16. The architecture shown in Figure 16(a) has been used by Siroyan [33], Transmogrifier [25] etc.

Our complete design space of clustered architectures is shown in Table 2. The Columns 1 and 2 marked as *Reads* and *Writes* denote whether the reads and writes are across (*A*) clusters or within the same (*S*) cluster. Columns 3 and 4 marked as *RF→FU* and *FU→RF*, specify the interconnect type from register file to FU and from FU to register file respectively. Here, *PP* denotes *Point-to-Point* and *PPB* denotes *Point-to-Point Buffered*. This table also shows in Column 5, the commercial or research architectures which have been explored in this complete design space. For example the TiC6x is an architecture, which reads across clusters and writes to the same cluster; it uses buses for reading from RFs and point-to-point connections for writing back results to RFs.

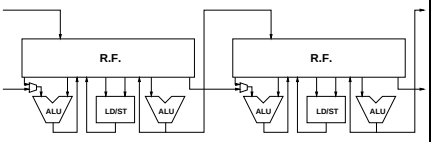
We would like to contrast here our classification with what has been presented in [35]. They have only considered five different communication mechanisms and have not presented a classification. The *bus-based* and *communication FU* based interconnects



(a) Write Across-1

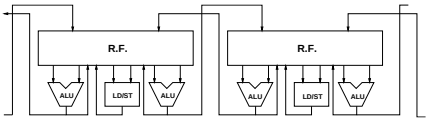


(b) Read Across-1

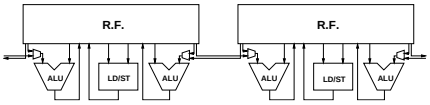


(c) Write/Read Across-1

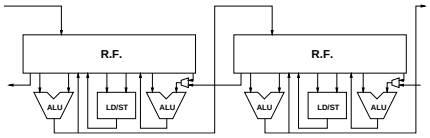
Figure 15. Architectures with ($\delta = N_{Clusters}$)



(a) Write Across-2



(b) Read Across-2



(c) Write/Read Across-2

Figure 16. Architectures with ($\delta = N_{Clusters}/2$)

Reads	Writes	RF→FU	FU→RF	Available Archs.
S	S	PP	PP	TriMedia, FR-V, MAP-CA, MAJC
A	S	PP	PP	
A	S	Bus	PP	Ti C6x
A	S	PPB	PP	
S	A	PP	PP	Transmogrifier, Siroyan, A|RT
S	A	PP	Bus	
S	A	PP	PPB	
A	A	PP	PP	
A	A	PP	Bus	
A	A	PP	PPB	
A	A	Bus	PP	
A	A	Bus	Bus	
A	A	Bus	PPB	
A	A	PPB	PP	
A	A	PPB	Bus	
A	A	PPB	PPB	

Table 2. Overall Design-Space for Direct Communication Architectures

which they have considered are part of the RF-to-RF type communication domain in our classification. The *extended results* type architecture is a *write across* architecture in our classification and *extended operands* is basically a *read across* type of architecture as per our classification. However, here again, they have considered only one type of interconnect, point-to-point, whereas others such as point-to-point buffered, buses are also possible. These have been shown in Table 2.

It can be clearly seen from Table 2 that a large range of architectures has not been explored. For each of these architectures an important metric is the maximum hop distance between any two clusters (δ). For example in case of architecture in Figure 15(a) $\delta = 8$ and for architecture in Figure 16(b), $\delta = 4$ assuming an 8-cluster configuration. δ is not an independent parameter, it can be calculated from the architecture type and number of clusters ($n_{clusters}$).

VLIW Design Space Exploration Methodology Figure 17 shows the overall design space exploration methodology. The Trimaran system is used to obtain an instruction trace of the whole application from which a DFG is extracted for each of the functions. Trimaran also performs a number of ILP enhancing transformation. This trace is fed to the DFG generating phase, which generates a DFG out of this instruction trace. The chain detection phase finds out long sequences of operations in the generated DFG. The clustering phase, which comes next, forms groups of chains iteratively, till the numbers of groups is reduced to the number of clusters in the architecture. The binding phase, binds these groups of chains to the clusters. It needs to be noted that the operation to FU binding is done in two steps. First operation to cluster binding is done and next operation to FU in a cluster binding is done. Since, during clustering, the partial schedules are calculated (explained in detail later), the typical phase coupling problem [22, 28], is contained to a large extent, while still keeping the overall problem size manageable. Lastly, a scheduling phase schedules the operations into appropriate schedule steps.

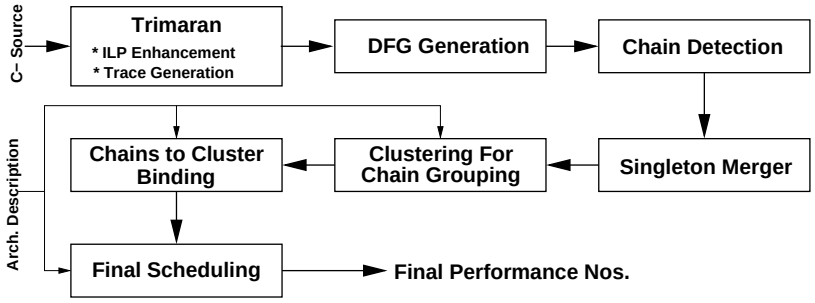


Figure 17. DSE Framework

4.2 Experimental Results and Observations

Our framework is very flexible, as can be seen from the results. It supports architecture exploration for a wide design space. However, results based on several parameters could not be presented in this paper due to paucity of space e.g. those with each cluster connected to two or more clusters for each of the architecture types, those having dedicated

communication FUs for inter-cluster communication etc. The architectures for which we present results are shown in Figures 14, 15 and 16. For the RF-to-RF architectures it is assumed that the number of buses is 2 and the bus latency is 1.

			$\delta = 4$			$\delta = 2$		
Bench.	**PV**	**RF**	**WA.1**	**RA.1**	**WR.1**	**WA.2**	**RA.2**	**WR.2**
matrix_init	9.80	6.28	6.12	6.12	6.12	6.12	6.12	6.12
biquad	10.67	2.61	8.56	8.47	8.75	8.75	8.85	9.06
mm	11.68	3.19	6.81	9.28	8.48	8.48	7.80	8.52
insert_sort	9.45	4.09	7.22	7.19	7.80	7.92	7.87	7.94
h2v2_fancy	9.71	2.04	6.02	5.64	6.85	6.84	6.17	6.67
encode_one	10.28	2.19	6.46	6.50	7.50	6.72	6.65	6.69
h2v2_down	11.06	3.43	5.61	5.49	5.94	5.94	5.50	5.90
form_comp	10.75	5.18	5.21	5.43	5.61	5.61	5.52	5.58
decd_mpeg1	10.90	2.16	7.43	7.60	8.61	8.61	8.14	8.67
dist1	7.44	2.62	6.41	7.00	7.02	7.15	6.41	7.15
pred_zero	9.48	4.88	5.26	5.37	5.57	5.57	5.53	5.57
pred_pole	9.56	5.49	5.16	5.16	5.38	5.27	5.16	5.27
tndm_adj	10.19	5.71	6.85	6.34	6.85	6.85	6.73	6.85
update	9.70	2.20	6.31	6.36	7.36	7.36	7.16	7.43
g721enc	10.50	3.62	6.54	6.34	6.72	6.69	6.60	6.63
ILP as Fraction of PV (%)								
Avg. (%)	-	35.32	65.56	65.45	70.57	69.64	66.56	69.82
Max. (%)	-	64.08	86.16	94.09	94.35	96.1	86.16	96.1
Min. (%)	-	19.82	48.47	49.64	52.19	52.19	49.73	51.91

Table 3. ILP for (8-ALU, 4-MEM) 4-Clust Architectures

The obtained ILP numbers for the various architectures for different cluster configurations are shown in Tables 3 and 4. Here the different benchmarks are the actual functions from the benchmark suites of DSP-Stone and MediaBench. The register file for each of the architectures has sufficient number of ports to allow simultaneous execution on all the FUs e.g. for a monolithic RF architecture the RF has $24 * 3 = 72$ *Read* and $24 * 1 = 24$ $Write$ ports (assuming 1 read port for predicate input). This is done so as to ensure that there is no concurrency loss due to insufficient number of ports and the effect of interconnection architecture stands out. Abbreviation *PV* (column 2) has been used in this table for *Pure VLIW (or single RF)*, *RF* (column 3) for RF-to-RF, *WA* for *Write Across* (column 4 and 7), *RA* for *Read Across* (column 5 and 8) and *WR* for *Write/Read Across* (column 6 and 9). The numbers in these tables, show the ILP for each of the benchmark functions e.g. for insert sort for a 4-cluster configuration, the ILP for a write across architecture (Figure 15(a)) is 7.22, for read across (Figure 15(b)). From the results we conclude the following:

1. Loss of concurrency vis-a-vis pure VLIW is considerable and application dependent.

			$\delta = 8$			$\delta = 4$		
Bench.	**PV**	**RF**	**WA.1**	**RA.1**	**WR.1**	**WA.2**	**RA.2**	**WR.2**
matrix_init	20.42	11.67	11.14	11.14	11.14	11.14	11.14	11.14
biquad	21.64	2.00	12.98	13.91	15.58	15.58	15.90	16.23
mm	19.64	2.12	9.90	13.06	15.60	15.60	14.69	14.69
insert_sort	18.94	2.21	11.03	12.32	13.48	13.48	13.39	13.62
h2v2_fancy	19.38	2.07	10.00	9.63	13.04	13.08	11.71	12.55
encode_one	19.14	2.18	12.02	13.04	14.44	14.39	14.39	14.70
h2v2_down	19.04	2.05	4.90	6.45	6.35	6.35	6.49	6.45
form_comp	21.72	4.63	7.85	9.91	10.86	10.98	10.98	10.98
decd_mpeg1	21.81	2.18	9.45	10.24	12.40	12.52	12.29	12.64
dist1	7.44	3.38	5.03	5.72	7.02	7.02	5.47	6.64
pred_zero	19.20	4.98	5.37	5.41	5.82	5.77	5.77	5.77
pred_pole	19.85	3.35	9.92	9.92	10.32	10.32	10.32	10.32
tndm_adj	17.95	2.58	11.78	11.42	12.16	12.16	12.16	12.16
update	19.39	3.46	11.20	11.36	13.25	13.03	12.42	12.42
g721enc	21.14	2.07	13.64	12.47	14.17	14.17	13.14	13.51
ILP as Fraction of PV (%)								
Avg. (%)	-	18.23	49.70	54.46	62.81	62.66	60.10	61.77
Max. (%)	-	57.15	67.61	76.88	94.35	94.35	75.18	89.25
Min. (%)	-	9.09	25.74	28.18	30.31	30.05	30.05	30.05

Table 4. ILP for (16-ALU, 8-MEM) 8-Clust Architectures

2. In few cases the concurrency achieved is almost independent of the interconnect architecture. This denotes that a few grouped chains in one cluster are limiting the performance along with a few critical transfers.
3. For applications with consistently low ILP for all architectures the results are poor due to large number of transfers amongst clusters.
4. In some cases the performance in case of $n_{clusters} = 4$ architecture is better than performance in case of $n_{clusters} = 8$ architecture (dist1). This is because of the reduced hop distance amongst clusters and this behavior is common across different architectures. In such cases communication requirements of the application are spread across all the clusters, so, as the average number of hops comes down the performance increases. This happens even though the supported concurrency has decreased due to less number of FUs.

Figure 18, shows the variation of average ILP as fraction of Pure VLIW versus number of clusters for each of the architectures. While for $n_{clusters}$ less than 8, the behavior of different interconnection networks is not brought out, once $n_{clusters}$ grows beyond 8, the superiority of *WR.2* and *WA.2* is clear. Both of these are able to deal well with the applications which require heavy communication. The minimum loss of concurrency in these cases is around 37% and 30% for architectures with $n_{clusters} = 8$ and $n_{clusters} = 4$ respectively. It needs to be noted that *WR.2* is the type of interconnect employed by most commercial as well as research architectures as shown in Table 2. The performance of *RA.2* in general, is inferior to *WA.2* because if there is more than one

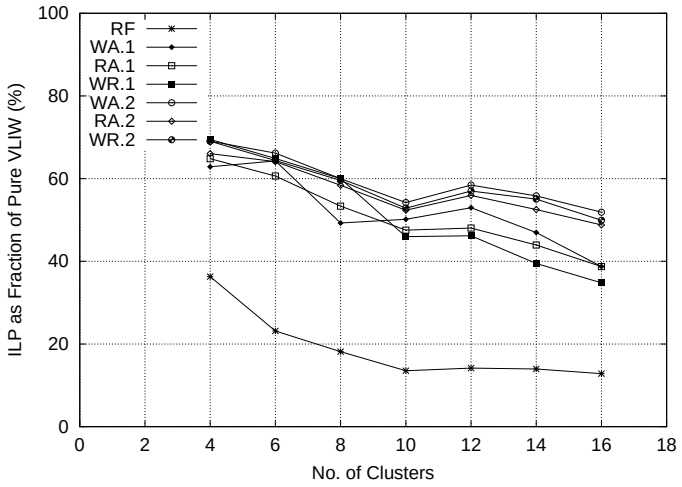


Figure 18. Average Variation in Performance With $n_{clusters}$

consumer of the propagated value in the target cluster, the value first needs to be moved to target cluster using the available read path, which amounts to an additional processor cycle. It is interesting to note that the performance of RF-to-RF type of architecture is quite poor, with an average loss of 81% for $n_{clusters} = 8$ and 64% for $n_{clusters} = 4$ and deteriorates further with increase in number of clusters. This is ignoring the latency of such transfers using global buses (assumed bus latency is 1) vis-a-vis local transfers. It would be quite interesting to identify application characteristics by which the suitability of an architecture can be established.

5 Conclusions

In this paper, we have presented five key steps in the process of designing ASIPs. We have presented a survey on the work done while attempting classification of the approaches for each step. Performance estimation is based either on scheduler based or simulation based technique. Instruction set is generated either through synthesis or a selection process. Code is synthesized either by a retargetable code generator or by a custom generated compiler.

We have developed a complete strategy to explore on-chip stor-age architecture for Application Specific Instruction Set Processors. This work involves deciding a suitable register file size, number-ber of register windows and on-chip memory configurations. Our technique neither requires code generator nor a simulator for a specific target architecture. Further, the processor description required for re-targeting is very simple. Apart from on-chip storage related parameters, we can also vary number and types of functional units.

Furthers, we have proposed and implemented a framework for evaluation of inter-cluster interconnections in clustered VLIW architectures. We have provided a concrete classification of the various inter-cluster interconnection mechanisms in clustered

VLIW processors. We have evaluated a range of clustered VLIW architectures and results conclusively show that application dependent evaluation is critical to make the right choice of an interconnection mechanism.

Acknowledgments The authors would like to acknowledge the partial support of *Naval Research Board*, Govt. of India towards the Multi-processor Embedded System Project at IIT Delhi of which this work is a part.

References

1. ARM Ltd. Homepage. `http://www.arm.com`.
2. Simplescalar Homepage. `http://www.simplescalar.com`.
3. Trimedia Homepage. `http://www.trimedia.com`.
4. M. Balakrishnan, Anup Gangwar, and Anshul Kumar. mpact of Inter-cluster Communication Mechanisms on ILP in Clustered VLIW Architectures. In *WASP*, 2003.
5. D. A. Berson, R. Gupta, and M. L. Soffa. URSA: A Unified Resource Allocator for registers and functional units in VLIW architectures. In *Proceedings of the IFIP WG 10.3 (Concurrent Systems) Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, Fl.*, pages 243–254, January 1993.
6. V. Bhatt, M. Balakrishnan, and A. Kumar. Exploring the Number of Register Windows in ASIP Synthesis. In *Proc. of VLSI/ ASPDAC 2002*, pages 223–229.
7. N. N. Binh, M. Imai, and A. Shiomi. A new HW/SW partitioning algorithm for synthesizing the highest performance pipelined ASIPs with multiple identical FUs. In *DAC-96*, pages 126–131, September 1996.
8. N. G. Busa, A. van der Werf, and M. Bekooij. Scheduling coarse grain operations for VLIW processors. In *Proceedings of Asia and South Pacific Design Automation Conference, Yokohama*, 1998.
9. Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred (Mark Owen) Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *International Symposium on Computer Architecture (ISCA'2000),*, 2000.
10. Jiri Gaisler. LEON: A Sparc V8 Compliant Soft Uniprocessor Core. http://www.gaisler.com/leon.html.
11. N. Ghazal, R. Newton, and J. Rabaey. Retargetable Estimation Scheme for DSP Architecture Selection. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 485–489, January 2000.
12. A. D. Gloria and P. Faraboschi. An evaluation system for application specific architectures. In *Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture. (Micro 23)*, pages 80–89, November 1990.
13. J. Gong, D. D. Gajski, and A. Nicolau. Performance Evaluation for Application-Specific Architectures. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 3 Issue 4*, pages 483–490, December 1995.
14. J. R. Goodman and W. C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the ACM International Conference on Super Computing, (ICS 1988)*, pages 444–452, 1988.
15. M. Gschwind. Instruction set selection for ASIP design. In *CODES-99*, pages 7–11, May 1999.
16. T. V. K. Gupta, P. Sharma, M. Balakrishnan, and S. Malik. Processor Evaluation in an Embedded Systems Design Environment. In *Proceedings of the Thirteenth International Conference on VLSI Design*, pages 98–103, January 2000.

17. S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *DAC-98*, pages 510–515, June 1998.
18. Paolo Ienne, Laura Pozzi, and Miljan Vuletic. On the limits of processor specialisation by mapping dataflow sections on ad-hoc functional unit. Technical Report 01/376, CS Technical Report, December 2001.
19. M. Jacome and G. de Veciana. Design challenges for new application specific processors. In *IEEE Design and Test of Computers*, number 2, pages 40–50, 2000.
20. M. K. Jain, L. Wehmeyer, S. Steinke, P. Marwedel, and M. Balakrishnan. Evaluating Register File Size in ASIP Design. In *Proceedings of the Ninth International Symposium on Hardware/ Software Codesign, (CODES 2001)*, pages 109–114, April 2001.
21. M.K. Jain, M. Balakrishnan, and Anshul Kumar. An Efficient Technique for Exploring Register File Size in ASIP Synthesis. In *Proc. of CASES 2002*, pages 252–261.
22. Krishnan Kailas, Kemal Ebcioglu, and Ashok K. Agrawala. CARS: A new code generation framework for clustered ILP processors. In *HPCA*, pages 133–144, 2001.
23. B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. The construction of a retargetable simulator for an architecture template. In *(CODES/CASHE-98)*, pages 125–129, March 1998.
24. J. Kin, Chunho Lee, W. H. Mangione-Smith, and M. Potkonjak. Power efficient mediaprocessors: design space exploration. In *DAC-99*, pages 321–326, June 1999.
25. D. Lewis, D. Galloway, M. Ierssel, J. Rose, and P. Chow. The transmogrifier-2: A 1-million gate rapid prototyping system, 1997.
26. S. K. Lodha and S. Gupta. A FPGA based Real Time Collision Detection and Avoidance. In *B. Tech. Thesis, Department of Computer Science and Engineering, IIT Delhi*, 1997.
27. M. Balakrishnan, Manoj Kumar Jain, and Anshul Kumar. Integrated On-chip Storage Evaluation in ASIP Synthesis . In *VLSI*, 2005.
28. Emre Ozer, Sanjeev Banerjia, and Thomas M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *International Symposium on Microarchitecture*, pages 308–315, 1998.
29. Laura Pozzi. Methodologies for Design of Application Specific Reconfigurable VLIW Processors, PhD Thesis, Politechnico di Milano, 2000.
30. B. Ramakrishna Rau and Michael S. Schlansker. Embedded computer architecture and automation. *IEEE Computer*, 34(4):75–83, April 2001.
31. Scott Rixner, William J. Dally, Brucek Khailany, Peter R. Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *Proceedings of 6th International Symposium on High Performance Computer Architecture*, pages 375–386, May 2000.
32. Jesus Sanchez and Antonio Gonzalez. Instruction scheduling for clustered VLIW architectures. In *International Symposium on System Synthesis (ISSS'2000),*, 2000.
33. Siroyan. http://www.siroyan.com.
34. Peter Song. Demystifying EPIC and IA-64. *Microprocessor Report*, 1998.
35. Andrei Terechko, Erwan Le Thenaff, Manish Garg, Jos van Eijndhoven, and Henk Corporaal. Inter-Cluster Communication Models for Clustered VLIW Processors. In *9th International Symposium on High Performance Computer Architecture, Anaheim, California*, pages 298–309, February 2003.
36. Y. Zhang and H. J. Lee. Register Allocation Over a Dependence Graph. In *Proceedings of the Second International Workshop on Compiler and Architecture Support of Embedded Systems, (CASES 1999)*, October 1999.