Register File Synthesis in ASIP Design

Manoj Kumar Jain [†] Lars Wehmeyer [*]
Peter Marwedel [*] M. Balakrishnan [†]

Abstract

Interest in synthesis of Application Specific Instruction Set Processors or ASIPs has increased considerably and a number of methodologies have been proposed for ASIP design. A key step in ASIP synthesis involves deciding architectural features based on application requirements and constraints. In this report we observe the effect of changing register file size on the performance as well as power and energy consumption. Detailed data is generated and analyzed for a number of application programs. Results indicate that choice of an appropriate number of registers has a significant impact on performance.

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 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 [5, 9, 2]. 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 [9, 2].

A recent survey of the approaches suggested for ASIP design methodologies during the 90's [7] identified five key steps as follows (fig. 1).

1. Application Analysis
2. Architectural Design Space Exploration
3. Instruction Set Generation
4. Code Synthesis

[*]Department of Computer Science 12, University of Dortmund, Germany. Email : {wehmeyer,marwedel}@ls12.cs.uni-dortmund.de

[†]Department of Computer Science and Engineering, Indian Institute of Technology Delhi, India. Email : {manoj,mbala}@cse.iitd.ernet.in

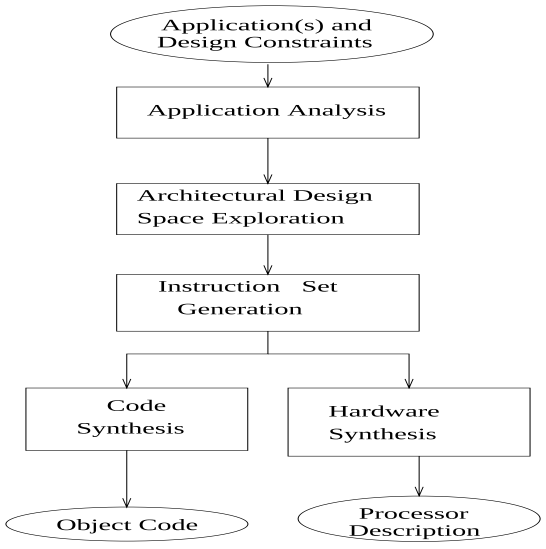
Application(s) and Design Constraints
Application Analysis
Architectural Design Space Exploration
Instruction Set Generation
Code Synthesis
Hardware Synthesis
Object Code
Processor Description

Figure 1: Flow diagram of ASIP design methodology

5. Hardware 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 of ASIP design. 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.

Some approaches attempted to establish a relationship between architectural features and application parameters [7, 4, 3]. Methods are suggested to find the parameters which in turn decide architectural features. Sato et al [9] have developed an Application Program Analyzer (APA) which finds data types and their access methods, execution counts of operators and functions used, the frequency of individual instructions and sequences of contiguous instructions. Gupta et al [4] and Ghazal et al [3] considered application parameters like average basic block size, number of Multiply-Accumulate (MAC) operations, ratio of address computation instructions to data computation instructions, ratio of input/output instructions to total instructions, average number of cycles between generation of a scalar and its consumption in the data flow graph etc. The architectural features considered by these approaches are number of operation slots in each instruction, concurrent load/store operations and latency of functional units and operations, addressing support, instruction packing, memory pack/ unpack support, loop vectorization, complex arithmetic patterns etc. Number of registers were considered in the model, but it was unconstrained.

There is a need to consider more architectural features as well as study the relationship between application parameters and these features in terms of user constraints on cost, performance, power and energy. In this work we consider varying the number of registers for ASIP design space exploration and the attempt is to study its effect at the application behavioural level. A specific architecture (*ARM7TDMI*) along with a compiler (*encc*) and a simulator has been used in this study. The intent is to study the effect of varying register file size on a particular processor and use this to understand the trend for power and performance estimation in a general ASIC synthesis framework.

Section 2 describes the experimental set up used and the procedure adopted to take the observations. Results of the observations are presented in Section 3. The last section concludes the paper with directions for future work.

2 Experimental Setup

Some benchmark programs were chosen and code generation and performance evaluation was performed with varying number of registers for the *ARM7TDMI* processor using the parameterizable compiler *encc* being developed and in use at the University of Dortmund, Germany. The benchmark programs were then analyzed to identify application characteristics responsible for the observed behavior.

2.1 Benchmark Suite

The following applications were selected as benchmark programs. These applications are either from the domain of media applications, DSP or implementations of standard sorting algorithms. An attempt has been made to study applications requiring typical array access patterns. These benchmark programs are available at *http://www.cse.iitd.ernet.in/ manoj/research/benchmarks.html*.

1. *biquad_N_sections* (from DSP domain)
2. *lattice_init* (from DSP domain)
3. *matrix-mult* (multiplication of two m x n matrices)
4. *me_ivlin* (media application)
5. *bubble_sort*
6. *heap_sort*
7. *insertion_sort*
8. *selection_sort*

2.2 The *encc* Compiler

The *encc* compiler was used for code generation and performance evaluation. *encc* was developed for the RISC class of architectures and generates code for reduced energy consumption. It features a built-in power model which is used to take decisions during the compilation process. The compiler was re-targeted from the *ARM7TDMI* to the *LEON* processor. Configuration of the compiler is possible by changing a parameter file which contains several constant declarations and processor specific information. Using this configuration file for the target processor, a customized compiler is generated. In our case, we took the configuration file for the *ARM7TDMI* processor and changed the number of registers in the range from 3 to 8 and generated a compiler for each case so as to be able to compile and evaluate the performance of the benchmark programs.

2.2.1 Functional Description of the *encc* Compiler

Taking an application program written in C an intermediate representation (IR) file is generated using *LANCE* [8]. Some standard optimizations are performed on this IR file using *LANCE* library functions. The optimizations performed by *LANCE* on the IR include constant propagation, copy propagation, dead code elimination, constant folding, jump optimizations and common subexpression elimination.

Taking an IR file as input, the code generator generates a forest of data flow trees for each function. A cover is obtained for each tree based on tree pattern matching. At this stage, the internal power model is used to generate a valid cover with minimal power consumption. A low level intermediate representation is generated. Register allocation, instruction scheduling, spill code generation and peephole optimizations are performed using this representation to generate assembly code. An assembler and a linker are used to create the object code. An instruction set simulator produces outputs required for validation. A trace of instructions is also produced which is analyzed by a trace analyzer. The *encc* provides information on spilled registers as well. The optimization options available are time, energy, size and power. One optimization can be selected at a time.

2.2.2 Power Model used in *encc*

The power model used in the compiler is based on the processor power model developed by Tiwari et al [11], which distinguishes between basic costs and inter-instruction effects. Basic costs consist of the measured current during execution of a single instruction in a loop. An approximate amount is added for stalls and cache misses. The change of circuit state for a different instruction and resource constraints are summed up in the inter-instruction effects. The power model used in *encc* includes the power of the processor P_{proc} and the power dissipation of the memory P_{mem}. For computing the basic power costs and inter-instruction effects, actual measurements have been done for the *THUMB* instruction set. Similarly, power consumption of the memory is based on actual measurements carried out on the off-chip memory of the used evaluation board ATMEL AT91M40400.

Power is calculated using the following equation.

$$P_{total} = \sum_{exec\ instr} P_{proc} + P_{mem}$$

This model is integrated into the compiler to take decisions during instruction selection or optimizations and into the trace analyzer which computes the total amount of energy dissipated during execution of the program under observation.

2.3 The *ARM7TDMI* Processor

The *ARM7TDMI* by ARM Ltd [6] is a 32-bit RISC processor and offers high performance combined with low power consumption. This processor employs a special architectural strategy known as *THUMB*, with the key idea of a 16-bit reduced instruction set. Thus the *ARM7TDMI* has two instruction sets :

1. The standard 32-bit ARM set
2. The 16-bit THUMB set

THUMB code operates on the same 32-bit register set as ARM code so it achieves better performance compared to traditional 16-bit processors using 16-bit registers and consumes less power than traditional 32-bit processors. Various portions of a system can be optimized for speed or for code density by switching between *THUMB* and *ARM* execution as appropriate. The *ARM7TDMI* processor has a total of 37 registers (31 general purpose 32-bit registers and 6 status registers) but these are not visible simultaneously. The processor state and operating mode dictate which registers are available to the programmer. In *THUMB* mode only 8 general purpose registers are available to the user, requiring 3 bits for register coding, thus reducing the instruction size.

2.4 Observations

The number of physical registers was varied in the range from 3 to 8 for the *ARM7TDMI* processor. The number of registers was increased beyond 8 as well, but in that case only assembly code could be generated as no instruction set simulator was available to execute the code. However, we were able to get information about spilling and static code size in such cases. For each different number of physical registers, *encc* was compiled to generate a customized compiler which was then used to generate code and other trace information for our benchmark programs. In a similar way, we have generated spilling information for the *LEON* processor as well. *LEON* is a RISC type of processor having SPARC architecture.

Optimizations for time, energy, size and power were performed for different configurations for all benchmark programs. Since the trends have shown to be similar for various optimizations, for brevity we are presenting results only for time optimization in the next section.

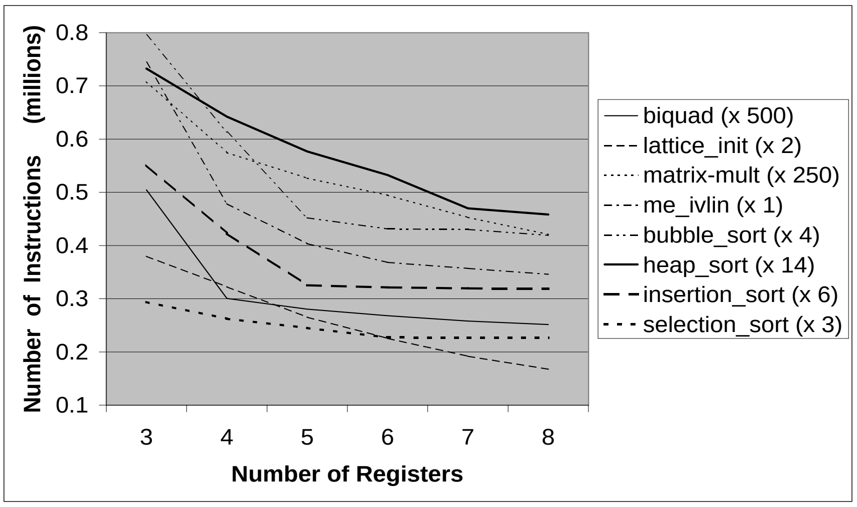

Figure 2: Number of Executed Instructions

3 Results

We present the results obtained for number of executed instructions, number of cycles, ratio of spill instructions to total static code size, power and energy consumption. The results and the following analysis is based on the following two assumptions.

1. Processor cycle time does not change with the change in the number of registers. This implies that change in the number of cycles is directly related to performance.
2. Power consumed by each instruction does not change significantly with the change in the number of registers.

3.1 Number of Executed Instructions

The results obtained for number of executed instructions are shown in figure 2. Values for different programs are scaled to produce the results on a single plot. Scale factors are shown in the figure. This is acceptable since the general trends can still be observed. We can observe one sharp curvature (knee) in some curves. The Curve for the program *biquad_N_sections* has its knee at 4 registers, whereas the programs *bubble_sort* and *insertion_sort* both have their knee at 5 registers. The curves for some of the other programs do not contain such a knee. In the program *biquad_N_sections*, there are two *for* loops with high iteration count. Each contains a statement like

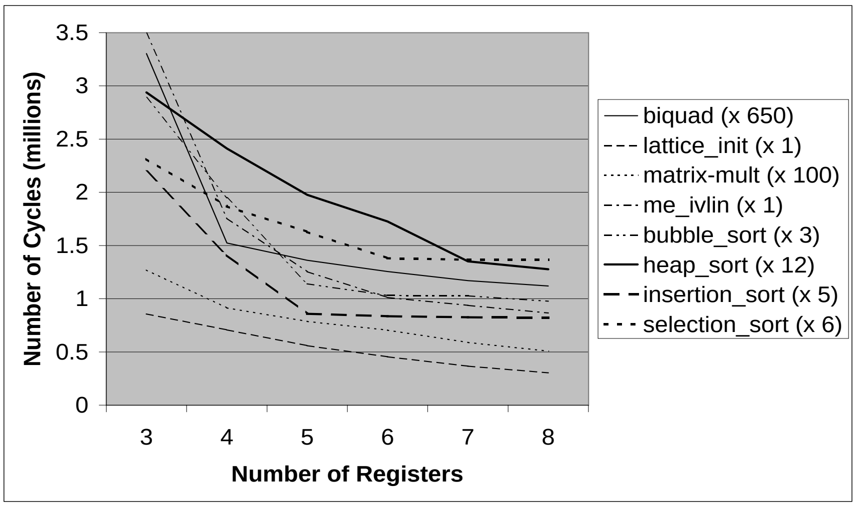

Figure 3: Number of Cycles

some_array[loop_counter] = value; which needs 4 registers for its execution without spilling. One each for storing the value of *loop_counter*, base address of the array *some_array*, offset value and the value to be written into the array. Thus the number of instructions shoots up significantly when we lower the number of physical registers from 4 to 3, since additional spill code has to be inserted within the loop. Looking at the programs *bubble_sort* and *insertion_sort*, we observe that each contains a 2-level nested loop. The statements in the innermost loop in both the cases need 5 registers for execution, that is why we observe a knee at 5 registers in the curves for these programs.

3.2 Number of Cycles

The results obtained for number of cycles are shown in figure 3. Again, the values for different programs are scaled to produce the results on a single plot and scale factors are shown. General behavior of the curves for the number of cycles is similar to that for the number of instructions. Though as we lower the number of registers, more spill instructions are inserted. Since spill instructions consist mainly of multi-cycle load and store instructions, the average number of cycles per instruction increases more than number of instructions. Still, the general shape of the curves is the same. Thus, the same application characteristics are responsible for similar behavior in both number of instructions and number of cycles.

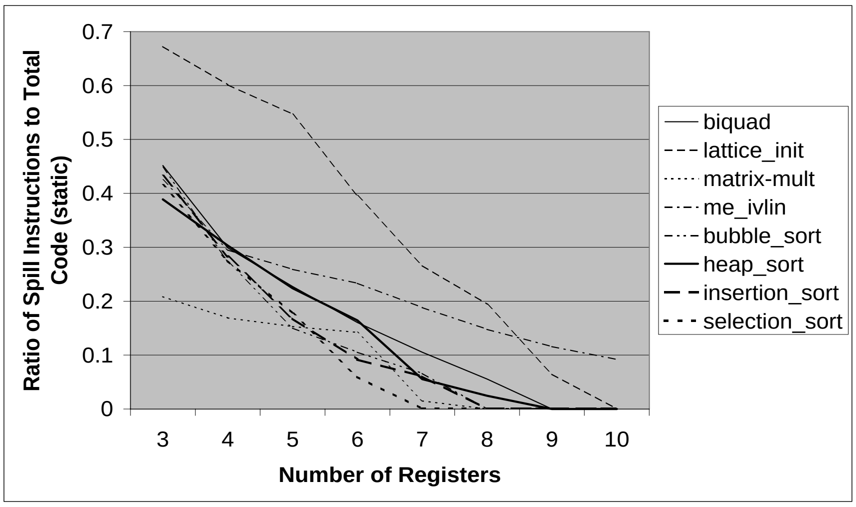

Figure 4: Ratio of Number of Spill Instructions to Total Number of Static Instructions

3.3 Ratio of Spill Instructions to Total Static Code Size

The results obtained for ratio of spill instructions to total static code size is shown in figure 4. The values for the program *lattice_init* are high because of high register pressure. A 2-level nested *for* loop is there. The inner loop contains two statements which needs 6 registers for execution. An interesting feature is observed for this program: the presence of common sub-expressions in two statements of the inner loop. Three additional registers are required to avoid repetition of address calculations and memory accesses. Values for program *me_ivlin* are high due to the large number of variables required to be live for a long time, so spilling is high, but it is continuously decreasing with increasing number of registers. To eliminate all spill code from this program, 19 registers are required. The values are drastically decreasing at 7 registers for the program *matrix-mult*, because 7 registers are sufficient to execute the statement in the innermost *for* loop (3-level nesting).

3.4 Average Power Consumption

We have used two different memory configurations in our study. One considers only off-chip memory, while the other considers on-chip instruction memory and off-chip data memory.

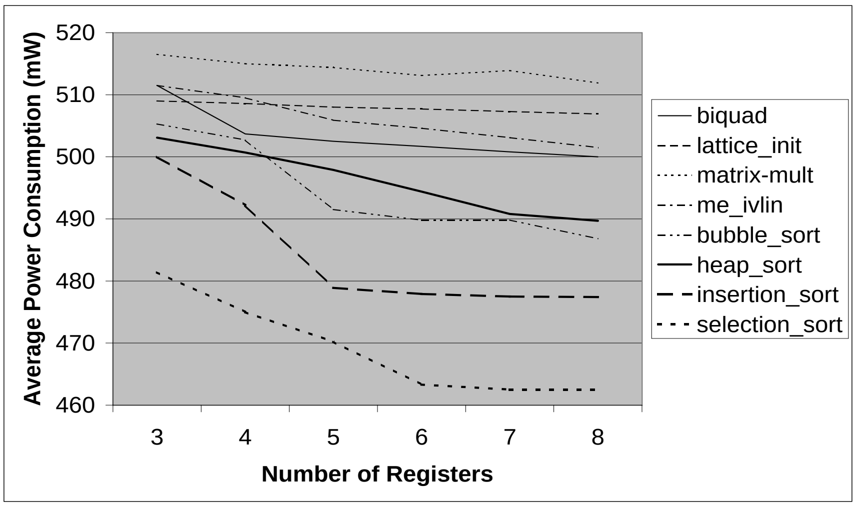

Figure 5: Average Power Consumption based on only Off-chip Memory

3.4.1 Off-chip memory

The results obtained for average power consumption while considering only off-chip memory are shown in figure 5. The power values are highest for the *matrix-mult* program, because the innermost loop (3-level nested looping) contains the statement
*c[i][j] = c[i][j] + a[i][k] * b[k][j];*
which accesses two 2-D array elements for reading and one 2-D array element for reading as well as writing. Since all the arrays are 2-D arrays, the address calculation requires an arithmetic shift left (instead of another expensive multiplication) and an addition. Since one power-hungry multiplication is still required for performing the actual arithmetic operation between the two matrices, the power consumption is high. The values for the program *lattice_init* are also high due to the fact that it is also a memory access intensive application. A 2-level nested *for* loop can be found and the inner loop body contains statements, accessing two 2-D matrices and one 1-D matrix. The values for the program *me_ivlin* are quite high due to high register pressure which leads to more spilling to memory. Since power consumption of the external data memory is significantly higher than the power consumed within the processor, the application's power demands are high. The values for the programs *bubble_sort* and *heap_sort* are similar because memory accesses in both are of similar extent. The values for program *selection_sort* are the lowest, because in selection sort data movement in memory is minimum. For the program *insertion_sort* the amount of data movement in memory is more than that of *selection_sort* but less than that of *bubble_sort*, which justifies its

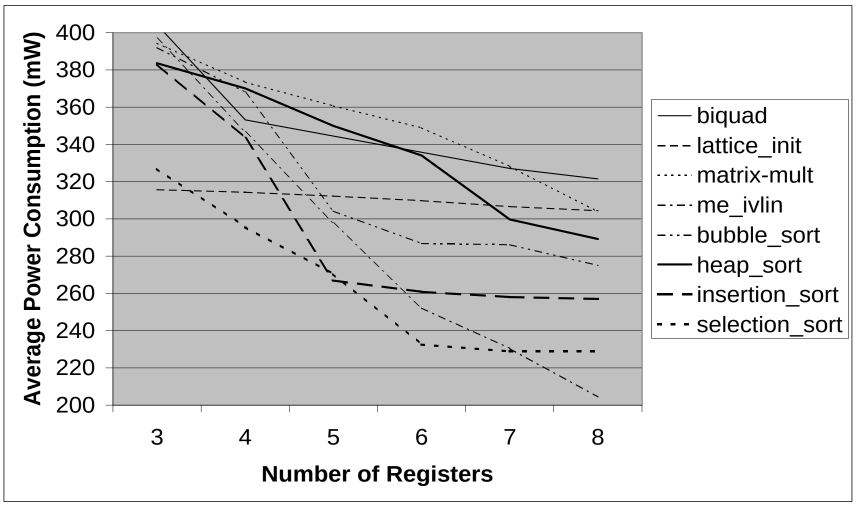

Figure 6: Average Power Consumption based on On-chip Instruction Memory and Off-chip Data Memory

position in the plot.

Our analysis shows that using more registers does not help significantly in saving power consumption, especially for memory intensive applications (e.g. programs *matrix-mult* and *lattice_init*). Though we observe that number of instructions executed and number of cycles taken for execution are being saved considerably with increasing number of registers in our observation range. These applications have higher power consumptions and even providing additional registers could not help in saving it. For other applications, the saving in power consumption is marginal and that gets saturated after a few registers.

3.4.2 On-chip Instruction Memory and Off-chip Data Memory

The results obtained for average power consumption while considering on-chip instruction and off-chip data memory are shown in figure 6. We observe a significant change in power consumption by the applications which are not memory intensive but have high register pressure (e.g. the program *me_ivlin*). In such applications significant spilling is saved by providing additional registers. On chip instruction memory consumes less power compared to off-chip memory used for data accesses. This is due to several reasons: on chip memory is usually smaller, the bus lines that need to be driven are shorter since the boundaries of the chip are not left, and on chip memories for storing instructions are often realised as power saving ROM memories. The aver-

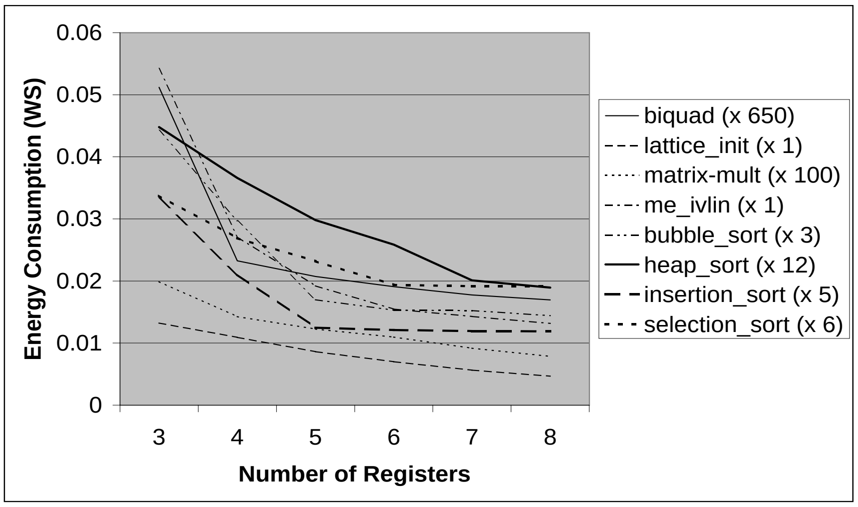

Figure 7: Energy Consumption based on only Off-chip Memory

age power consumption is less for all the benchmark programs compared to the power consumption for other memory configuration (i.e. considering only off-chip memory).

3.5 Energy Consumption

Again, we present results for both memory configurations.

3.5.1 Off-chip memory

The results obtained for energy consumption while considering only off-chip memory are shown in figure 7. Since for this memory configuration the average power consumption is almost constant, so trends of the curves for energy consumption is similar to that for number of cycles required for execution, since E = P * t and the power values do not change very much.

3.5.2 On-chip Instruction Memory and Off-chip Data Memory

The results obtained for energy consumption while considering on-chip instruction memory and off-chip data memory are shown in figure 8. For this configuration the average power consumption is lower in general, and there is significant saving in power consumption while reducing spilling by providing additional registers. This results in a significant reduction in energy consumption with larger number of registers.This

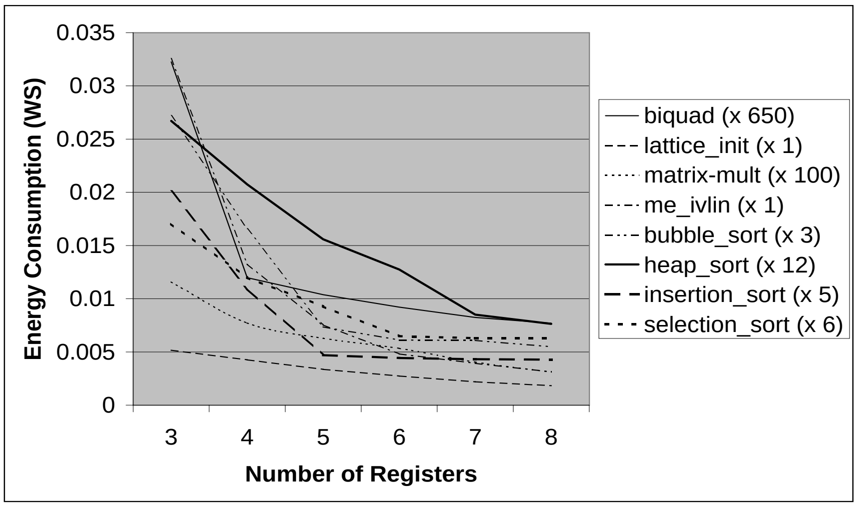

Figure 8: Energy Consumption based on On-chip Instruction Memory and Off-chip Data Memory

difference is visible especially for the applications which are not too memory intensive and having high register pressure such as *me-ivlin*.

3.6 Analysis of Results

We have analyzed the results for number of instructions executed, number of cycles taken for execution, number of spilling instructions inserted in code, power and energy consumption for each program separately. Here we analyze the results for two application programs: *lattice_init* and *me_ivlin*. We used on-chip instruction memory and off-chip data memory while generating results.

Results obtained for program *lattice_init* are shown in figure 9. We find that in this application, the power consumption does not change significantly with change in number of registers, though there is some change in number of spilling instructions. This is due to the fact that this application is memory intensive. The energy consumption shows a steady drop dominated by the reduction in the number of cycles without any pronounced knee.

Results obtained for program *me_ivlin* are shown in figure 10. We can see the change in power consumption for this program as we vary the number of registers. This is because the application is not memory intensive but it has high register pressure, so additional registers helps in saving the spilling and thus reducing the memory accesses. A careful analysis shows two knees in the energy curve, the one at register value 4 is

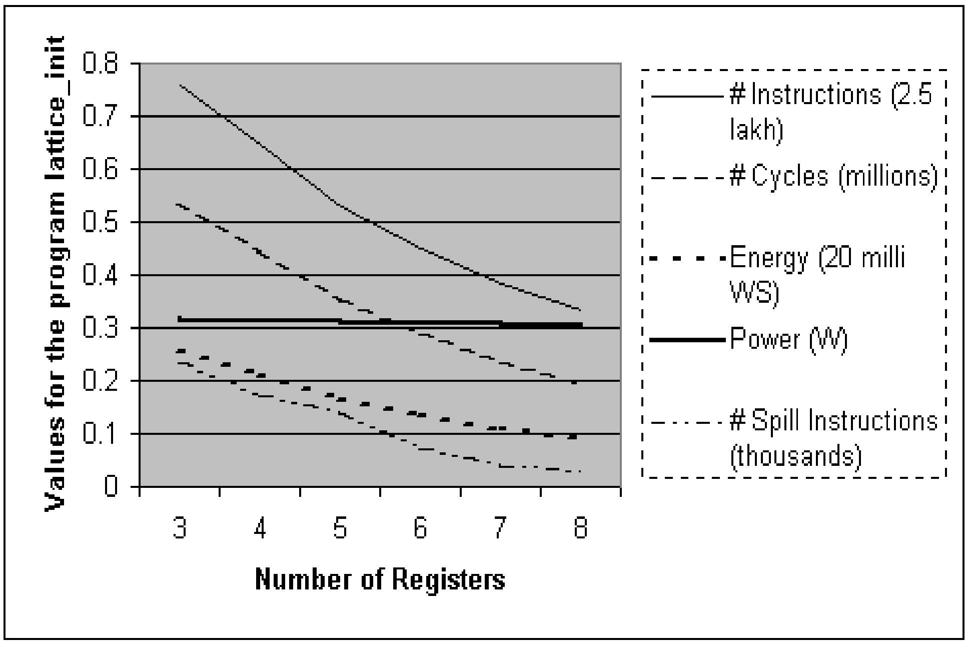

Figure 9: Results for the program lattice_init

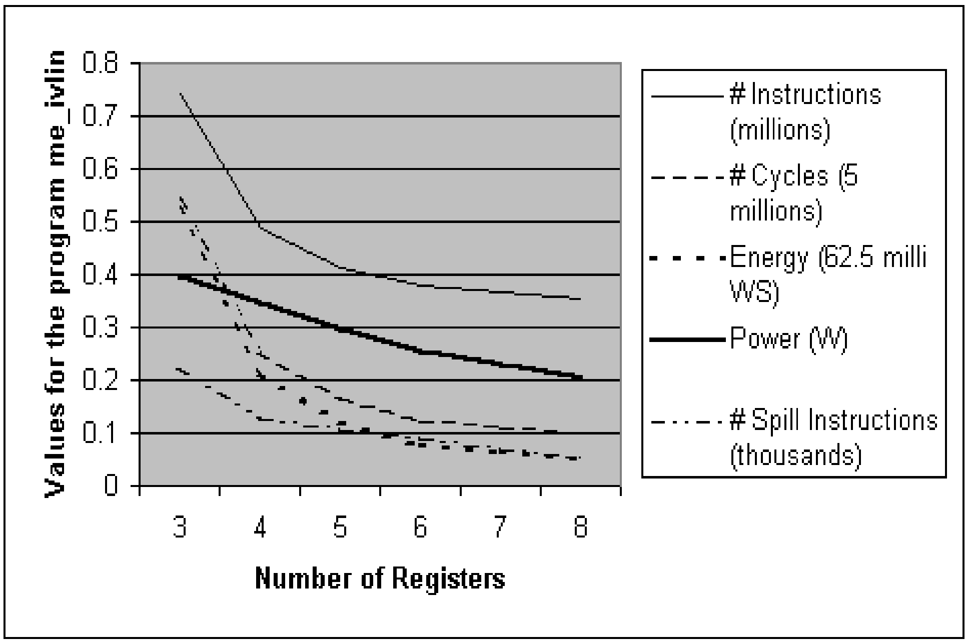

Figure 10: Results for the program me_ivlin

Application	Performance		Power		Energy	
program	Reg. size	% inc.	Reg. size	% red.	Reg. size	% red.
biquad_N_sections	$3 \rightarrow 4$	57.5	$3 \rightarrow 4$	12.6	$3 \rightarrow 4$	62.9
lattice_init	$4 \rightarrow 5$	20.5	$6 \rightarrow 7$	1.0	$4 \rightarrow 5$	21.0
matrix-mult	$3 \rightarrow 4$	29.7	$7 \rightarrow 8$	7.4	$3 \rightarrow 4$	33.4
me_ivlin	$3 \rightarrow 4$	53.4	$5 \rightarrow 6$	15.3	$3 \rightarrow 4$	59.3
buuble_sort	$4 \rightarrow 5$	46.3	$4 \rightarrow 5$	17.3	$4 \rightarrow 5$	55.6
heap_sort	$6 \rightarrow 7$	25.6	$6 \rightarrow 7$	10.3	$6 \rightarrow 7$	33.2
insertion_sort	$4 \rightarrow 5$	44.8	$4 \rightarrow 5$	22.3	$4 \rightarrow 5$	57.1
selection_sort	$3 \rightarrow 4$	22.2	$5 \rightarrow 6$	14.0	$5 \rightarrow 6$	30.1
Average		37.5		12.5		44.1

Table 1: Maximum variation in results for various benchmark programs

due to the knee in the cycle count and the knee at register value 6 is due to the knee in the power curve.

Table 1 shows the maximum percentage increase in performance and reduction in power and energy due to an increase of one register in each of the application programs. We also indicate where this takes place. This table establishes the importance of register file size as an architectural feature as a single register increase results in a performance improvement of up to 57.5% and energy reduction of 62.9%. The power is relatively insensitive to the changes in the number of registers. Furthermore, there is a high degree of correlation between the register file size which gives optimum performance and optimum energy consumption.

4 Conclusion and future work

We changed the number of physical registers for the *ARM7TDMI* processor. A new instance of the *encc* compiler was compiled with the specific number of registers. This generated compiler was used for compiling the benchmark programs. We studied the results obtained for number of instructions executed, cycle time taken for execution and spilling information, power and energy consumption. An increase in the number of registers by one can result in upto 57.5 % of performance improvement and upto 62.9 % reduction in energy consumption. Further there is a high degree of correlation between performance improvement and energy reduction. In the process we found that power does not strongly depend on the number of registers. We have generated spilling information for these application programs in the same range of number of registers on *LEON* processor as well. There is a reasonable correlation in the data generated.

The cost of varying register file size in an ASIP is not linear due to its effect on instruction encoding, instruction bit-width and required chip area. For an effective area-time-power tradeoff, we propose to develop an area model as well. Future work will be to identify and extract application characteristics so that an early estimation of number of ‘optimal’ registers may be done.

Acknowledgements

We acknowledge the guidelines and help provided by Professor Anshul Kumar, Dr. Rainer Leupers and Stefan Steinke. This research work is supported by DST-DAAD cooperation project (Project code MCS-216).

References

[1] Binh, N.N.; Imai, M.; Shiomi, A.; Hikichi, N. : "A hardware/software partitioning algorithm for pipelined instruction set processor ", Proceedings of the Design Automation Conference, 1995, with EURO-VHDL, EURO-DAC '95, 18-22 Sept. 1995, pp. 176-181.

[2] Childers, B.R.; Davidson J.W. : "Application Specific Pipelines for Exploiting Instruction-Level Parallelism ", University of Virginia Technical Report No. CS-98-14, May 1, 1998.

[3] 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, pp. 485-489.

[4] 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, pp. 98-103.

[5] Hoon Choi; In-Cheol Park; Seung Ho Hwang; Chong-Min Kyung : "Synthesis of application specific instructions for embedded DSP software ", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers 1998, 8-12 Nov. 1998, pp. 665 - 671.

[6] http://www.arm.com/

[7] Jain M.K.; Balakrishnan, M.; Anshul Kumar : "ASIP Design Methodologies : Survey and Issues ", to appear in the Proceedings of the IEEE/ACM International Conference on VLSI Design, 2001.

[8] LANCE System.
http://ls12-www.cs.uni-dortmund.de/∼leupers/lanceV2/lanceV2.html

[9] Sato, J.; Imai, M.; Hakata, T.; Alomary, A.Y.; Hikichi, N. : "An integrated design environment for application specific integrated processor ", Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors 1991, ICCD '91, 14-16 Oct. 1991, pp. 414-417.

[10] Stanford compiler group. The SUIF library, version 1.0, 1994.
http://suif.stanford.edu

[11] Tiwari, V.; Malik, S.; Wolfe A. : "Power Analysis Of Embedded Software: A First Step Towards Software Power Minimization ", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1994, ICCAD '94, 6-10 Nov. 1994, pp. 384-390.