Exploring Storage Organization in ASIP Synthesis

Manoj Kumar Jain, M. Balakrishnan and Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology Delhi, India
{manoj,mbala,anshul}@cse.iitd.ernet.in

Abstract

Performance estimation which drives the design space exploration is usually done by simulation. With increasing dimensions of the design space, simulator based approaches become too time consuming. In the domain of Application Specific Instruction set Processors (ASIP), this problem can be solved by scheduler based approaches, which are much faster. However, existing scheduler based approaches do not help in exploring storage organization. We present a scheduler based technique for exploring register file size, number of register windows and cache configurations in an integrated manner.

Performance for different register file sizes are estimated by predicting the number of memory spills and its delay. The technique employed does not require explicit register assignment. Number of context switches leading to spills are estimated for evaluating the time penalty due to a limited number of register windows and cache simulator is used for estimating cache performance.

The proposed technique has been validated for several benchmarks over a range of processors by comparing our estimates to the results obtained from standard simulation tools. The processors include ARM7TDMI, LEON and Trimedia (TM-1000).

Keywords

ASIP synthesis, storage exploration, scheduler based technique, register file, register windows, cache configurations.

1. Introduction and Objectives

A typical ASIP design flow includes key steps as application analysis, design space exploration, instruction set generation, code generation for software and hardware synthesis [1]. Design space exploration is driven by performance estimations. These estimates are generated using a simulator based [2, 3] or scheduler based framework [4, 5]. Simulator based technique needs a retargetable compiler to generate code for different processor configurations to be explored. Simulating the generated code is slow. Further, there is a well known trade off between retargetability and code quality in terms of performance and code size compared to hand optimized code. Therefore, in our opinion, simulation based approach is not suitable for early design space exploration.

On the other hand, in scheduler based approach, the application code is scheduled on an abstract model of the processor being explored to get an estimate of the execution time. Therefore, it is much faster and quite suitable for an early design space exploration. However, storage organization, which includes register files and cache is not fully explored by the scheduler based approaches reported so far. Our previous study [6, 7] 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 this paper, we propose a scheduler based approach to explore register file size, number of register windows for a SPARC like architecture and cache memory configurations in an integrated manner.

In scheduler based approach, register allocation is the key step which helps in determining the influence of limited register file size on the performance. Most of the approaches suggested for register allocation perform register allocation either after scheduling like a typical compiler, or they try to solve register allocation and scheduling in an integrated manner. Goodman et al. [8] suggested DAG driven technique for register allocation, but their approach needs a pre-pass scheduling. Rajiv Gupta et al. [9] propose to solve register allocation and instruction scheduling in an integrated manner. However, register allocation before scheduling is desirable when minimization 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 [10]. Another feature of our technique is its ability to handle global register needs during the estimation.

A technique was proposed to estimate number of register windows spills and restores [11]. This is based on generating a trace of calls and returns from the application execution. In our framework we use this approach for estimating the trade off between number of register windows and window size.

A number of techniques for memory exploration are reported in literature [12, 13, 14, 15]. To the best of our knowledge, no approach considers register file as a part of storage during the design space exploration.

The paper is organized as follows. Our overall methodology for ASIP synthesis and proposed technique for storage organization exploration is presented in the next section. The scheduler based technique for register file size exploration is discussed in section 3. Integrated exploration of register window size and number of register windows for SPARC architecture and memory configurations is presented in section 4. Section 5 presents results of various trade offs along with a case study. Paper is concluded with conclusions and future work in the last section.

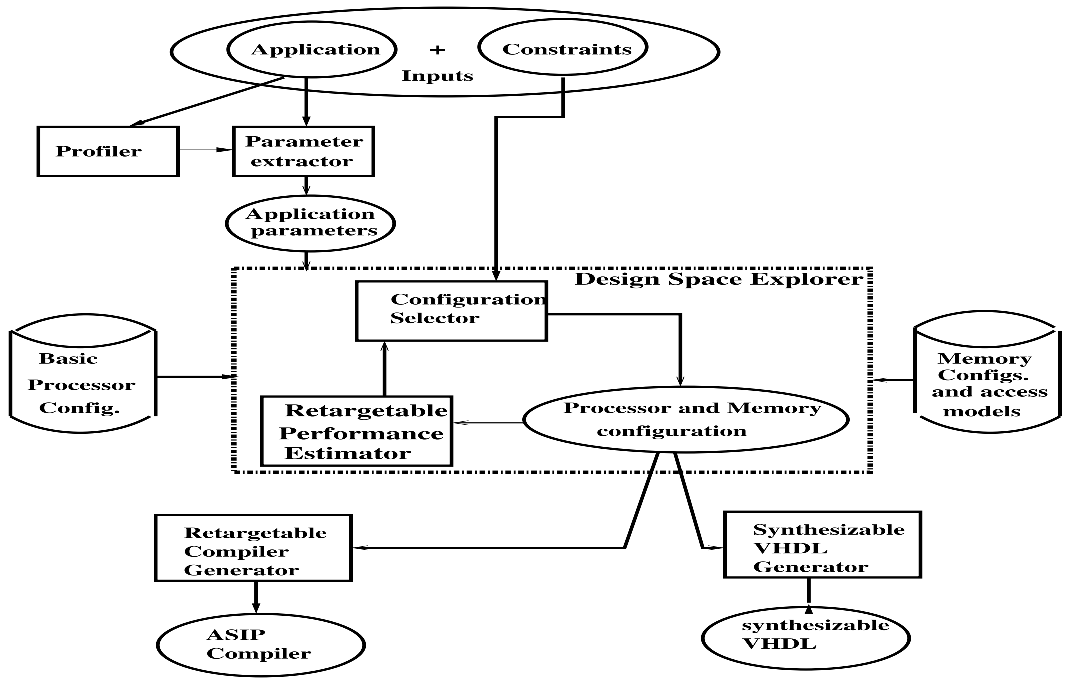

Figure 1. ASIP Synthesis Methodology : ASSIST

2. Methodology

2.1 ASSIST : ASIP Design Methodology

Work reported in this paper is part of the ASIP Synthesis Methodology (ASSIST), being developed in our group. The overall flow diagram of ASSIST methodology is shown in figure 1. The inputs include application behavior in C, performance, power and area constraints, basic processor configuration, pipeline templates and memory access models, power models for various components, area and clock period models.

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 our technique is used) 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 the 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 the customized processor.

2.2 Storage Space Exploration

Storage exploration is part of design space exploration phase of overall methodology. Proposed technique for storage space exploration is shown in figure 2. We estimate the cycle count for application execution on chosen processor and memory configuration. We consider a parameterized model for processor as well as memory. Memory parameters 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 which is described in the next section. A resource constrained list scheduler is used for performance estimation. Global cost is computed based on variable usage analysis. A sequence of function calls and returns during the execution of the application is generated and number of window spills and restores required are computed for a specific number of windows. We use sim-cache simulator of simplescalar tool set [16] to know cache misses statistics.

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

$$ET = et_R + oh_W + oh_C$$

Where
et_R : Execution time when register file contains R registers.
oh_W : Additional schedule overhead due to limited windows.
oh_C : Additional schedule overhead due to cache misses.
et_R can be further expressed as following equation.

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

Where bet is the base execution time considering resources other than storage and oh_{dep} is the overhead due to additional dependencies inserted during register allocation. $spill_R$ is the number of register spills and t_R is the delay associated with each register spill. Computation of et_R is described in the next section.

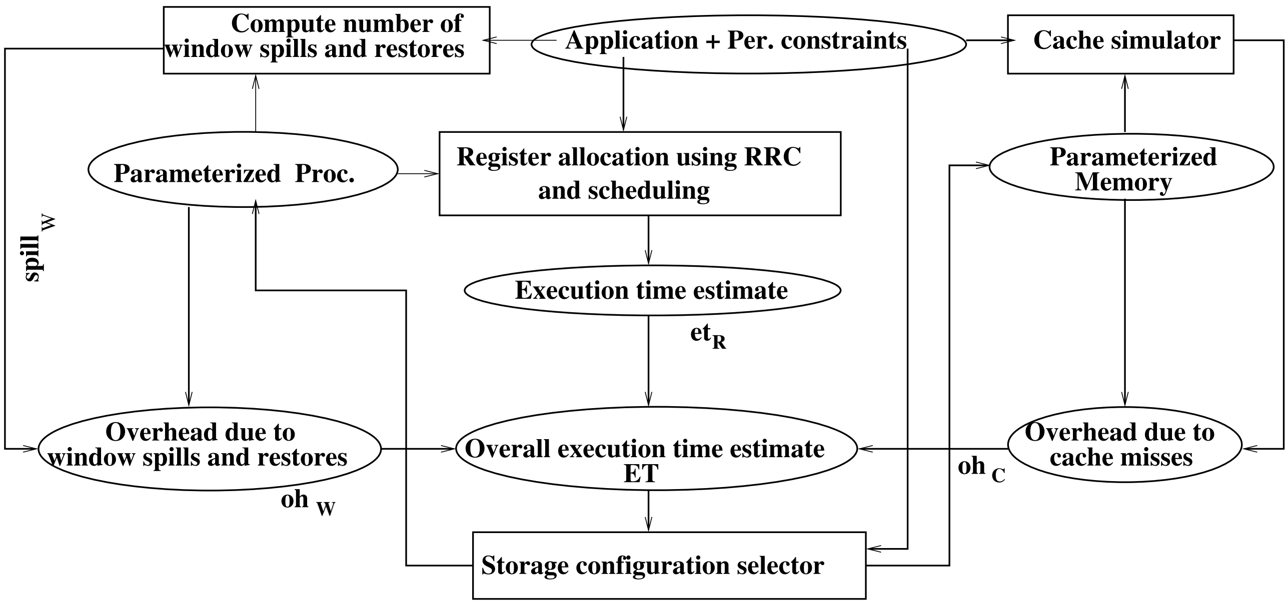

Figure 2. Storage exploration technique

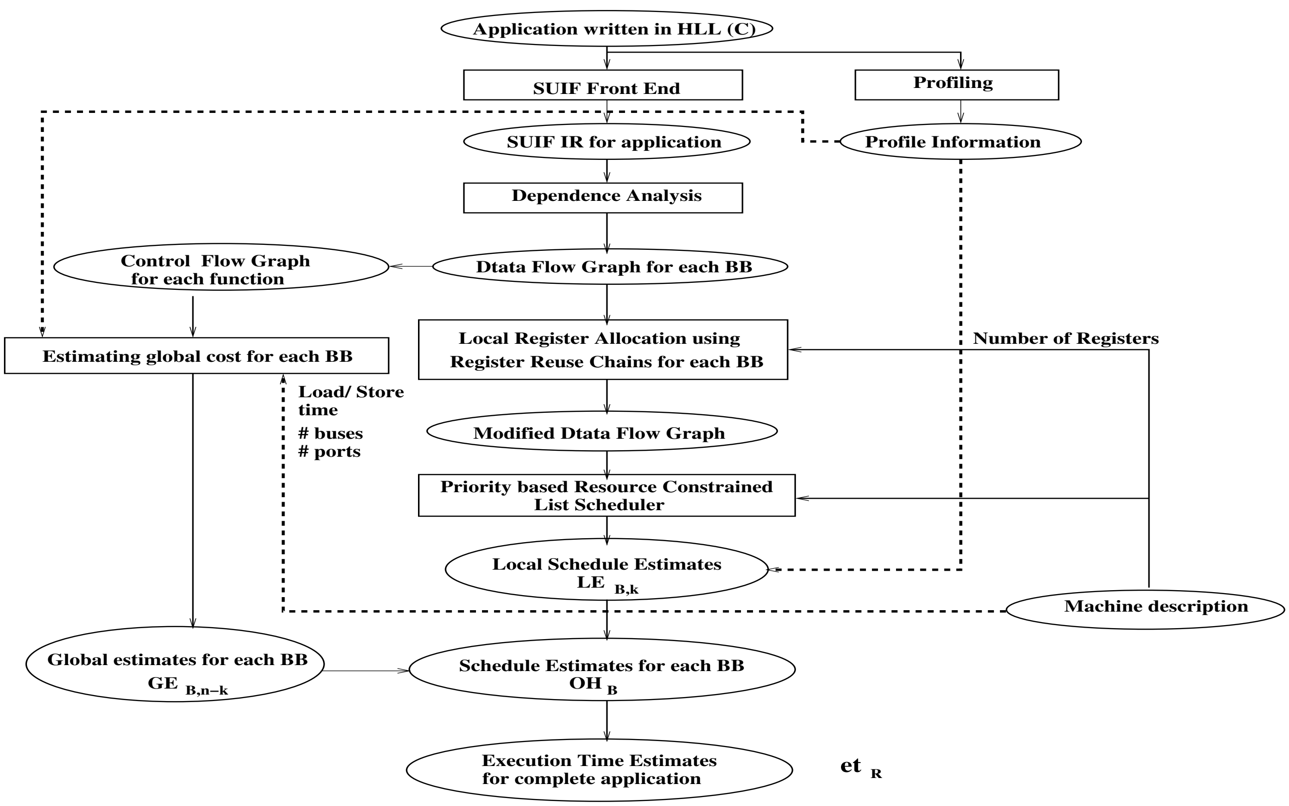

Figure 3. Flow diagram of our performance estimation scheme

oh_W can be further expressed as following equation.

$$oh_W = spill_w * t_W$$

Where $spill_w$ is the number of window spills and t_W is the delay associated with each register window spill.
oh_C can be further expressed as following equation.

$$oh_C = miss_C * t_C$$

Where $miss_C$ is the number of cache misses for the specific cache configuration and t_C is the cache miss penalty. Computation of oh_W and oh_C is explained in the section 4. Storage configuration selector selects suitable processor and memory configuration to meet the desired performance by knowing all the execution time estimates.

3. Estimating execution time with limited regs

3.1 Overview

Figure 3 shows the procedure for estimating the execution time with limited register file size. Input application (written in C) is profiled using *gprof* to find execution count of each basic block as well as functions. Intermediate representation is generated using *SUIF* [17]. Control and dependency analysis is done using this intermediate representation. Control flow graph is generated at the function level whereas the data flow graph is generated at the basic block level.

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 register reuse chains which is described in the next subsection. 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, and 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 for all the blocks within a function for complete range of register file sizes to be explored are produced. Schedule overheads needed to handle global needs with limited number of registers are computed using variable life time analysis. For each block, we need information on variables used, defined, consumed, live at entry point to this block and live at exit point to this block. This additional global needs overhead is also generated for complete range of number of registers. 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 minimum.

Overall estimate for a block B (OE_B) can be expressed as

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

where $GE_{B,n-k}$ is the overhead to handle global needs with $n-k$ registers. A weighted sum of OE_B values for all blocks produce estimates at the function level. Estimates for all functions are added together to produce overall estimate et_R for the application. The following subsections briefly describes register allocation using register reuse chains and addressing global register needs. More details are available in [18].

3.2 Register Allocation using Reg. Reuse Chains

A register reuse chain or RRC is an ordered set of nodes in a data dependence graph in which all the nodes share the same register. We use these chains to estimate the increase in the schedule length and spilling overheads for a given number of registers. Initial register reuse chains are formed in a manner that no additional dependency is generated. If the number of chains is higher than the number of registers available for local allocation say (k), then chains are merged to reduce the number to the desired level. The chain merging is driven by merging cost which is the cost in terms of spills and schedule overheads to be paid if the said chains are merged. Attempt is made to introduce minimum number of spills in the process of merging. The result of such chain merging would be insertion of spill instructions as well as additional dependencies. The data flow graph is modified in the process. Our scheduler (priority based resource constrained list scheduler) takes care of this modified data flow graph.

For initial chain formation and converting them into dependence conservative chains we have adopted the ideas suggested by [10]. Since spilling was not considered in [10] they allow merging of two chains only if it is possible to merge them without spilling a register value into memory. Our technique considers options of spilling a register value into memory and merging cost is computed on the basis of how many additional loads and stores would be required for the said merging. The merging cost of chain c_j to reuse register assigned to chain c_i can be expressed as the following equation. Illustrative examples can be found in [18].

$$\begin{aligned} M_{i,j} &= \infty \qquad if\ \exists n_1 \in c_i, n_2 \in c_j | path(n_2, n_1) \\ &= \Big[||\{< n1, n2 > \in DAG | n_1 \in c_i, n_2 \in c_j\}|| + \\ &\quad ||\{n_3 | < last(c_i), n3 > \in DAG, path(last(c_j), n3)\}|| \Big] * \\ &\quad (cost_{load} + cost_{stores}) \\ &\qquad otherwise \end{aligned}$$

3.3 Addressing Global Register Needs

The additional schedule overhead due to global needs ($GE_{B,n-k}$) for basic block B, when out of a total n registers, k are used for local allocation, is split into the store component ($GS_{B,n-k}$) and load component ($GL_{B,n-k}$) as follows.

$$GE_{B,n-k} = GS_{B,n-k} * t\_store + GL_{B,n-k} * t\_load$$

Where
$GS_{B,n-k}$: number of additional stores.
$GL_{B,n-k}$: number of additional loads.
t_store : time required for one store.
t_load : time required for one load.
t_store and t_load are extracted from input machine description.
$GS_{B,n-k}$ and $GL_{B,n-k}$ are estimated as described in [18].

Using the variable usage information, we identify the variables which are live at entry and exit point of each basic block as well as variables consumed in each block and variables that need to be kept live across but not used in the block. Based on such analysis we compute the total register needs at any point of time for each block. If such needs exceeds the availability of registers, excess are spilled to memory and accessed back when needed. Number of loads and stores required are computed using the above information and execution counts of blocks.

4. Overheads due to Limited Register Windows and Cache Misses

4.1 Register Window Spills

While estimating performance for a processor with multiple register windows, such as SPARC, we need to account for the overheads due to register window spills.

It was shown in [11] that the number of register window spills and restores can be computed for a given number of register windows from the sequence of function calls and returns made during execution of the application. For generating the sequence of function calls and returns, the Input application is instrumented and the trace of function calls and returns is recorded. Once such a call/ return trace is available to us, we perform a stack based analysis for finding the number of window spills and restores for the given number of register windows. Depending on the register file size (total number of registers available in the register file) and distribution of registers into registers for global use as well as registers to store in/ out parameters and for local use, it is decided that how many registers could be used when registers are allocated within a particular function. We use our technique proposed in the last section to estimate execution time when each function gets a specified number of registers for allocation. Window spills ($spill_W$) due to limited number of register windows is computed separately and execution penalty due to window spills and restores (oh_W) is computed. This additional penalty is added into the estimates produced by the technique to produce overall execution time estimates.

4.2 Cache Miss Overheads

In the previous discussion we have considered the effect of limited number of registers and register windows on performance. In this subsection we consider cache to be a part of storage hierarchy along with register file for design space exploration.

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, 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, so 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 using the following equation.

$$d = \alpha_1 + \alpha_2 * (b - 1)$$

where α_1 is the time required (in terms of processor clock cycles) for transferring first data from next level memory to cache and α_2 is the time required for each of the remaining data transfers. b is the block size. This additional delay is added with the base execution time estimated from our methodology described earlier to get the overall execution time estimates.

We have used *simplescalar* tool set [16] to compute the number of cache misses using different cache sizes. For each cache size we considered all the possible combinations of number of blocks and block sizes and chosen the combination which produces minimum additional delay. Since we do not require timing information from the simplescalar tool, we use *sim-cache* simulator which is fast compared to *sim-outorder*.

5. Results

Performance estimations with varying register file sizes for selected benchmarks applications were done. Two processors namely ARM (*ARM7TDMI* a RISC) [19] and Trimedia (*TM-1000* a VLIW) [20] were chosen for experimentation and validation. *TM-1000*'s five-issue-slot instruction length enables up to five operations to be scheduled in parallel into a single VLIW instruction.

Maximum register file size was defined as 16 for *ARM7TDMI*. Results show the performance improvement with larger number of registers. There are two factors responsible for this improvement : reduction in spills and more flexibility in scheduling. After a certain limit, the performance gets saturated and providing additional registers does not improve the performance. Plots for many benchmarks have sharp knee reflecting drastic change in performance at specific number of registers.

To verify correctness of the estimates produced by our technique, we compared it with results from standard tools. Specific register file size is chosen for validation considering the size supported by their standard tool sets.

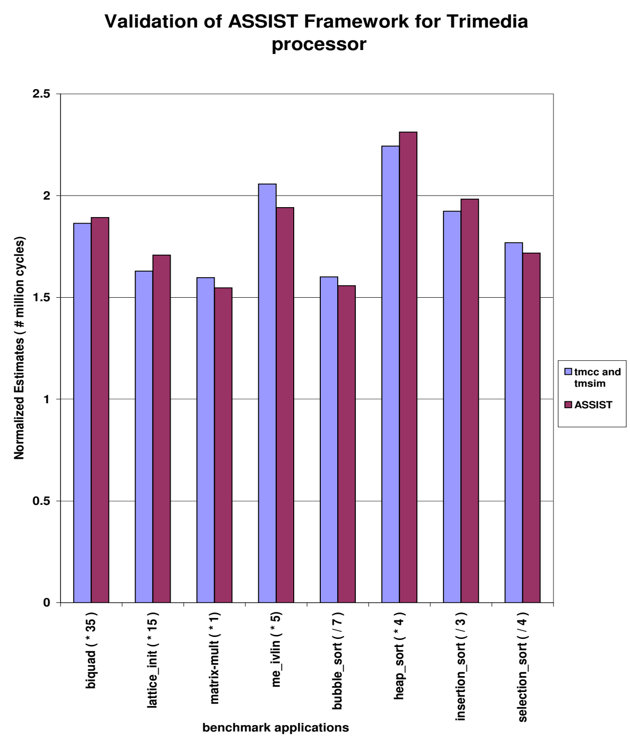

Figure 4. Validation for *TM-1000*

We observe that our technique produced estimates within 9.6% compared to the performance numbers produced by standard compiler and simulator available in the *Armulator* tool set. On the other hand the simulator based technique *encc* produced estimates with an average error of about 29.6%. We have compared the run time elapsed in our technique with *encc* technique. Comparison shows that on an average our technique was 77 times faster compared to *encc*. Of course *encc* also generates code apart from performance estimation whereas ASSIST only generates estimates. But the major time consumed in the *encc* framework is for simulation for which it uses *Armulator*.

To show the retargetability and capability of handling processors which supports multi-issue architecture to exploit the instruction level parallelism, we have chosen Trimedia processor (*TM-1000*). Our technique is capable of exploring number of functional units (FUs) of various types. We observe that our technique produced estimates within 3.3% compared to the performance numbers produced by standard compiler and simulator available in the Trimedia tool set (figure 4). Number in the braces shows scaling factor used to represent results on the same plot.

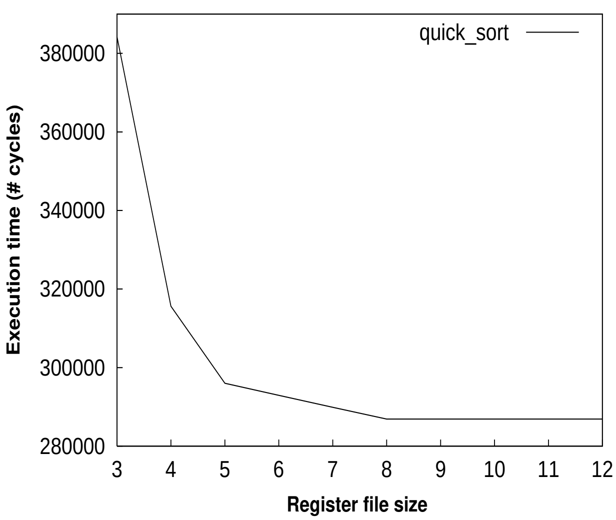

Figure 5. Impact of reg file size on exec. time

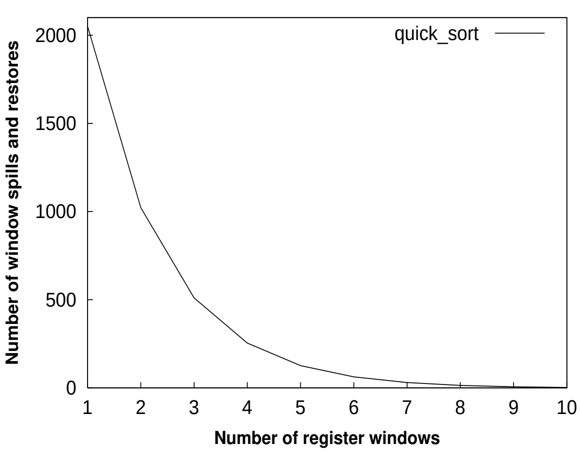

Figure 6. Impact of number of reg. windows

5.1 Trade off Number of Register Windows and Window Size

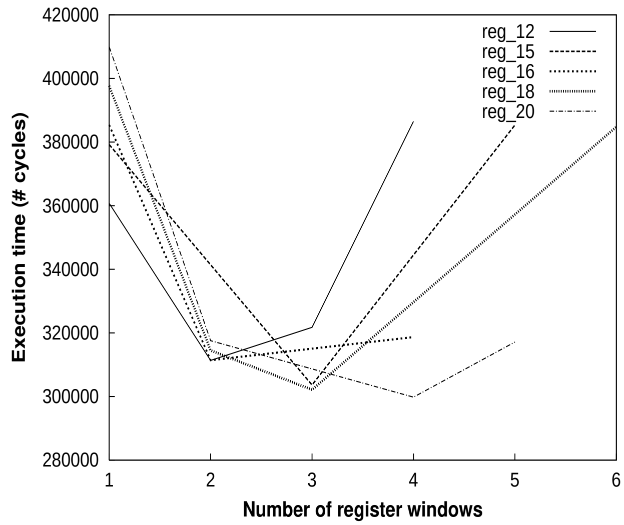

Figure 7. Trade off between number of windows and their sizes

We have generated various trade off curves for our benchmark applications. For *quick_sort* program, figure 5 shows the impact of variation of register file size on execution time. When these data values are generated, only a single register file of specified size is assumed and all the registers were allowed to be used for register allocation for any function. Any schedule overhead due to limited number of register windows was ignored. Variation of number of window spills and restores required with different number of register windows is shown in figure 6. Nine widows are required to remove overheads in terms of window spills and restores due to context switches.

Now when we are interested in trade off between the number of registers and window sizes then we have results as shown in figure 7. Depending on the performance requirement, suitable register file size can be chosen and for the chosen register file size, number of windows and hence the window size (number of registers in a window) can be decided. After certain value of number of register windows the curve starts going up because when larger number of register windows are used, individual window will contain fewer registers to store variables so time spent in storing and retrieving variables dominates over time saved in storing and retrieving contexts.

5.2 Trade off Cache Size and Register File Size

We have generated execution time estimates for various benchmark applications for different register file sizes and different data cache memory sizes. If we observe results produced for *matrix-mult* program for different register file size and memory configurations as shown in figure 8 then some interesting trade offs can be observed. Based on the generated execution time estimates and the input performance constraint, some possible suitable configurations can be suggested. If the application should not take more

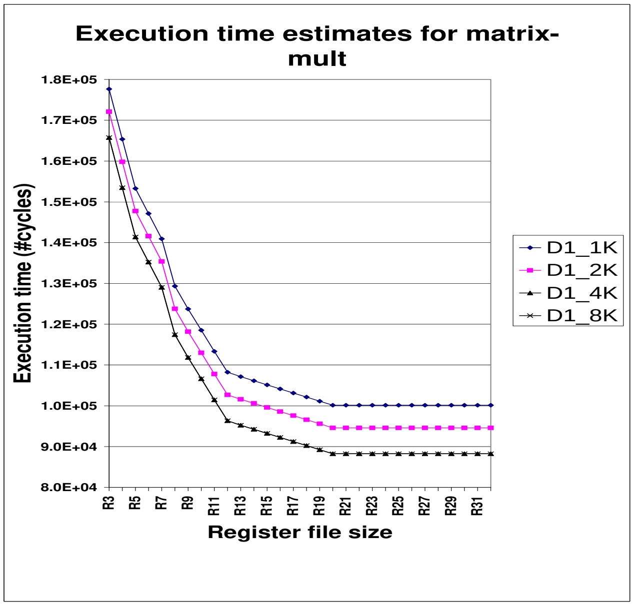

Figure 8. Results for *matrix-mult* for LEON

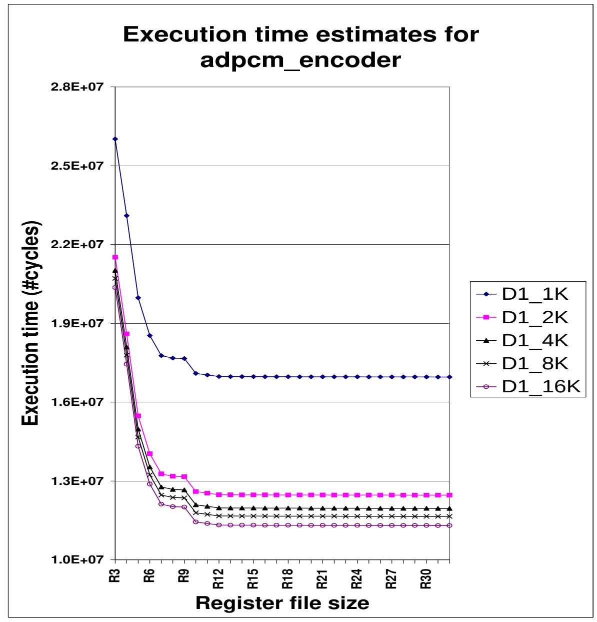

Figure 9. Results for *adpcm rawaudio encoder* for LEON

than $1.0E+05$ cycles then one of the three 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

We have chosen LEON [21] processor for validation, which is a SPARC V8-based architecture. The VHDL model of the LEON is available under the GNU general public license. We have compared the execution time estimates produced by our technique with *tsim* simulator provided with the LEON tool set.

Since number of windows is fixed (8) in *tsim* LEON simulator, we have validated our results for 8 register windows making adjustment for specific compiler strategies. We have considered 4K instruction cache and 4K data cache which is the memory configuration assumed by tsim simulator. On the average our technique produced estimates within 9.7% compared to the performance numbers produced by standard compiler and simulator available in the LEON tool set.

5.3 Case Study

In addition to the small test examples to illustrate and validate our methodology, we have also generated results for a complete media benchmark for LEON processor. The benchmark chosen is *adpcm rawaudio encoder* and *adpcm rawaudio decoder* and the generated results are shown in figures 9 and 10 respectively. Results indicate that increasing data cache size from 1K to 2K significantly improve performance of encoder but not for the decoder. If encoder is to be completed its execution in 16 million cycles then we have to use at least 2K data cache irrespective of register file size. We also observe that there is not significant performance improvement beyond window size 11 for both the applications.

6. Conclusions and Future Work

This work is geared toward deciding a suitable register file size, number of register windows and memory configurations in an ASIP synthesis environment. The assumption is that we have a configurable and synthesizable HDL description of a processor core which can be customized for the application.

Input application is profiled using *gprof* to get execution counts of various blocks. Our technique uses *SUIF* intermediate format to extract dependency and control information. Extending the concept of register reuse chains to predict cost of merging and global needs, we are able to do register allocation before scheduling. Our technique also permits exploration of register file size and windows configuration (number of register windows and window size) in an integrated manner for SPARC like architecture. Further we are able to extend our technique to consider various cache configurations as well. The technique is also capable of generating performance estimates with varying number of FUs of various types.

Performance estimates for a range of register file sizes for three processors namely, ARM (*ARM7TDMI*), LEON and Trimedia (*TM-1000*). Validation shows that our estimates are within 9.6%, 9.7% and 3.3% for these processors respectively, compared to the actual performance produced by standard tool sets. Further, this technique was nearly 77 times faster compared to the retargetable code generator based technique *encc*.

Optimizing register file size is useful in many ways. If smaller register file can be used then register addresses needs fewer bits. This reduction can be used to either reducing the instruction width or providing more room for op codes for new application specific instructions. It may help in reducing the switching activity and thus saving in terms of power consumption. An interesting application of our approach is to map the “saved registers” onto the coprocessor. This would provide a very flexible and “seamless”

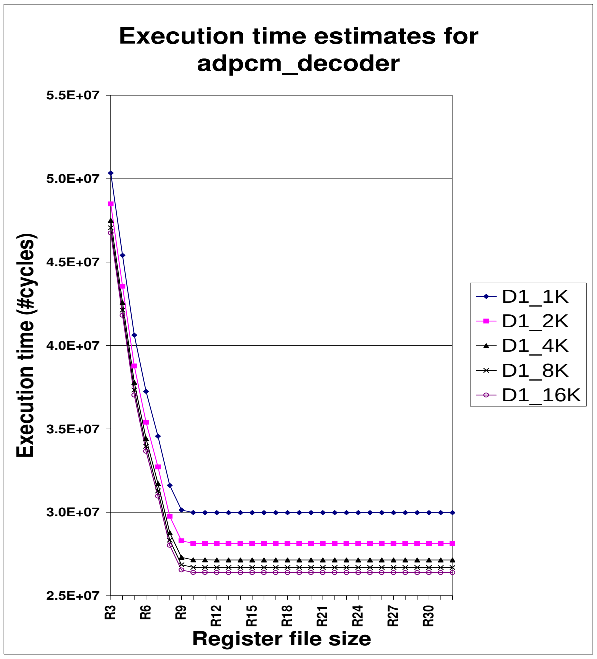

Figure 10. Results for *adpcm rawaudio decoder* **for LEON**

interaction with coprocessor input/ output registers. We are in the process of evaluating this approach by integrating memory exploration with coprocessor synthesis.

Presently our technique has some limitations. Impact of register file size on clock period is ignored. Further, compiler modifications required to support variable register file are not considered. We plan to remove these limitations in future.

7. REFERENCES

[1] M.K. Jain, M. Balakrishnan, and A. Kumar. ASIP Design Methodologies: Survey and Issues. In *Proceedings of the IEEE / ACM International Conference on VLSI Design. (VLSI 2001)*, pages 76–81, January 2001.

[2] 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.

[3] J. Kin, C. Lee, W.H. Mangione-Smith, and M. Potkonjak. Power Efficient Media processors: Design Space Exploration. In *Proceedings of the 36th Design Automation Conference*, pages 321–326, June 1999.

[4] 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.

[5] 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.

[6] 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 Co-design, (CODES 2001)*, pages 109–114, April 2001.

[7] L. Wehmeyer, M.K. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan. Analysis of the Influence of Register File Size on Energy Consumption, Code Size and Execution Time. *IEEE Transactions on Computer Added Design of Integrated Circuits and Systems*, 20(11):1329–1337, November 2001 2001.

[8] 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.

[9] 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.

[10] 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.

[11] V. Bhatt, M. Balakrishnan, and A. Kumar. Exploring the Number of Register Windows in ASIP Synthesis. In *Proceedings of the IEEE / ACM International Conference on VLSI Design and ASP Design Automation Conference. (VLSI/ ASPDAC 2002)*, pages 223–229, January 2002.

[12] B.L. Jacob, P.M. Chen, S.R. Silverman, and T.N. Mudge. An Analytical Model for Designing Memory Hierarchies. *IEEE Transactions on Computers*, 45(10):1180–1194, October 1996 1996.

[13] D. Kirvoski, C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Application-Driven Synthesis of Memory-Intensive Systems-on-Chip. *IEEE Transactions on Computer Added Design of Integrated Circuits and Systems*, 18(9):1316–1326, September 1999 1999.

[14] S.G. Abraham and S.A. Mahlke. Aumomatic and Efficient Evaluation of Memory Hierarchies for Embedded Systems. In *Technical Report HPL-1999-132, HPL Laboratories Palo Alto*, October 1999.

[15] P. Grun, N. Dutt, and A. Nicolau. APEX: Access Pattern based Memory Architecture Exploration. In *International Symposium on Systems Synthesis, (ISSS 2001)*, October 2001.

[16] Simplescalar Homepage. "http://www.simplescalar.com".

[17] SUIF Homepage. "http://suif.stanford.edu/".

[18] M.K. Jain, M. Balakrishnan, and Anshul Kumar. An Efficient Technique for Exploring Register File Size in ASIP Synthesis. In *Proceedings of the Fifthth International Conference on Compilers, Architecture and Synthesis for Embedded Systems, (CASES 2002)*, pages 252–261, October 2002.

[19] ARM Ltd. Homepage. "http://www.arm.com".

[20] Trimedia Homepage. "http://www.trimedia.com".

[21] LEON Homepage. "http://www.gaisler.com/leon.html".