# An Efficient Technique for Exploring Register File Size 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 is a crucial operation which drives the design space exploration in Application Specific Instruction Set Processors (ASIP) synthesis. The usual approach to estimate performance is to do simulation. With increasing dimensions of the design space, simulator based approaches become too time consuming. 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. This paper presents a scheduler based technique for exploring register file size in ASIP synthesis.

The performance is estimated by estimating the number of spills for a particular register file size. The concept of register reuse chains is used for local register allocation. Live variable analysis is done across all the blocks to estimate global register needs. The technique is fast, accurate and retargetable and does not require code generation.

Performance estimates for register file sizes between 1 to 8 for ARM (*ARM7TDMI*) and 1 to 128 for Trimedia (TM-1000) were generated for selected benchmarks to validate the proposed technique. Results show that our estimates are within 9.6% for *ARM7TDMI* and 3.3% for *TM-1000* compared to the actual performance produced by standard tool sets. Further, this technique is nearly 77 times faster compared to the simulator based technique *encc*.

## General Terms

Performance, Measurement, Design, Experimentation.

## Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance and analysis, Performance analysis and design aids, Domain-specific architectures.

## Keywords

Register file, ASIP Synthesis, register spill, retargetable estimation, instruction scheduling, design space exploration, liveness analysis, global analysis, storage exploration.


*CASES 2002,* October 8–11, 2002, Grenoble, France.


## 1. INTRODUCTION

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. Thus ASIPs can be efficiently used in many embedded systems such as servo-motor control, avionics, cellular phones etc.

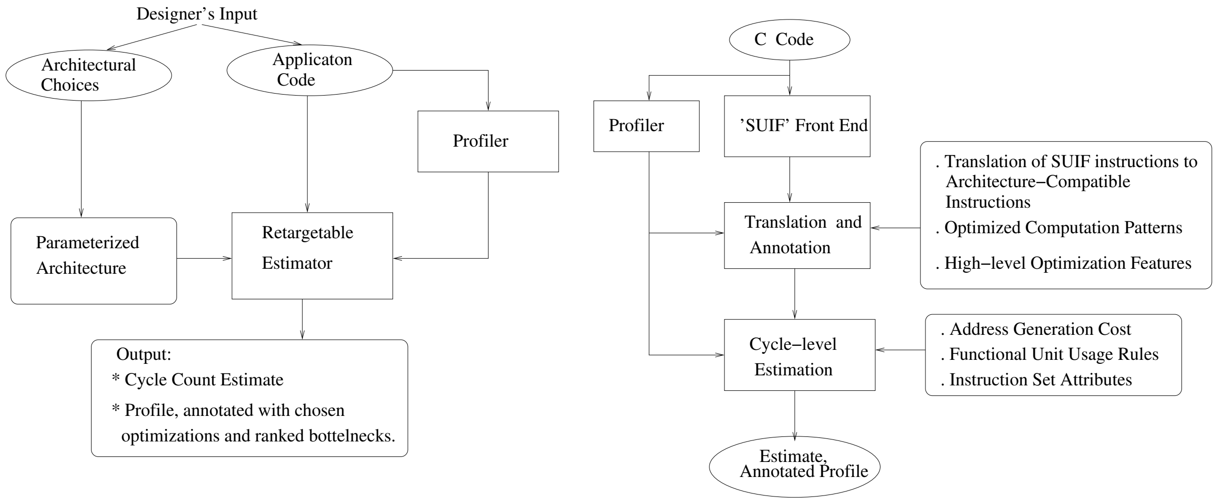
A typical ASIP design flow includes key steps as application analysis, design space exploration, instruction set generation, code generation for software and hardware synthesis [1]. Design space exploration is driven by performance estimations. These estimates are generated using a simulator based or scheduler based framework. Simulating the generated code is slow. On the other hand storage organization is not explored by scheduler based approaches. Our previous study has indicated that choice of an appropriate number of registers has a significant impact on performance and energy [2, 3]. The results show that a single extra register improves the performance and reduces energy significantly in certain cases. The focus of our work is to explore storage architecture in a scheduler based approach. In this paper we focus on register file size exploration.

Proposed technique uses the concept of Register Reuse Chains (RRCs) [4], which makes it possible to do register allocation for unscheduled code blocks. While doing register allocation, attempt is made to minimize the schedule overhead which will occur due to limited registers. A priority based resource constrained list scheduler is used to get local performance estimates of each block. Global analysis is performed at the function level to find additional schedule overhead required to handle global needs which are ignored when the local estimates are generated. This overhead is added with the local estimates to produce the total estimates. Execution counts obtained from *gprof* are used to multiply the estimated execution times.

Related work is presented in the next section. Our overall ASIP design methodology and overview of our technique is presented in section 3 whereas the performance estimation technique is discussed in detail in section 4. Register allocation technique is described with the help of an illustrative example in this section. Results along with their analysis are presented in section 5 followed by conclusions and future work in the last section.

## 2. RELATED WORK

As indicated above, the two classes of approaches to ASIP synthesis based on the technique used for performance estimator for driving the configuration selection process are simulator based and scheduler based. These techniques and their comparison is described here briefly.




Over all Flow of Retargetable Estimation

Flow of Estimation Scheme

**Figure 1: Scheduler based estimation scheme**

## 2.1 Classification of Performance Estimation Approaches

**Simulator based** approaches [5, 6, 7, 8] estimate performance by generating code of the given application for the target architecture. This code is then simulated using a simulator to get performance estimation. These approaches have considered the storage architecture up to a certain extent.

**Scheduler based** estimation approaches like [9, 10] take a very simple architecture model as input. A suitable internal representation of the input application is generated. Application is run on host machine to gather profile information. A suitable processor configuration is chosen by an explorer using a scheduler for performance estimates. An illustrative approach of this class suggested by Gupta et al [9] and Ghazal et al [10] (figure 1) is briefly described here.

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

The main drawback of simulation based approaches is that these take considerable time. Therefore, with the growing size of design space these are less suitable. In contrast to this, the scheduler based approaches are much faster at the cost of some loss of accuracy. Moreover, only a limited set of architectural features are explored by scheduler based approaches. Storage organization is almost ignored by such approaches. We focus on exploring storage organization in scheduler based approach. This paper presents a scheduler based technique for exploring register file size.

## 2.2 Our approach v/s existing approaches for Register Allocation

Register allocation and instruction scheduling are two major compile-time activities [11]. The goal of register allocation is to eliminate unnecessary load/ store instructions and consequently reduce code size as well as power consumption. On the other hand the main goal of instruction scheduling is to increase the computing speed by fully utilizing the resources of a processor.

Register allocation is more important in ASIP synthesis. Less number of bits are required to identify a register when register file size is small. A typical instruction format includes references to two registers. Saving bits in register addressing saves bits in instruction, thus we have more bits available for op codes and more application specific instructions can be incorporated. Alternatively, instruction width and the bus width for accessing instruction cache could be reduced. Even if this bus width is not reduced we will have less bit transitions and thus reducing power consumption. That is why, in our technique we prefer to do register allocation prior to scheduling.

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 integrated manner. Goodman and Hsu [12] suggested DAG driven technique for register allocation, but their approach needs a pre-pass scheduling. Cindy Norris et al [13] and Rajiv Gupta et al [14] proposed to solve register allocation and instruction scheduling in an integrated manner. Global needs are ignored by most of these approaches.

On the other hand, Zhang et al [4] proposed DAG driven technique which performs register allocation before scheduling. They have used the concept of register reuse chains (RRCs) which is also used in our approach. A register reuse chain is a sequence of operation nodes which can be assigned the same register. RRCs are merged to reduce the register requirement. We have used their concept of RRCs with significant extensions. Since their approach does not consider the option for spilling a variable so they don't have a notion of cost of merging. They allow chains to merge only if it is possible to do so without spilling, otherwise the chains are not allowed to merge. We have considered the option of spilling the variables also while deciding on the possibility of merging the chains. We compute merging cost for every pair of reuse chains and systematically merge them. Another major extension is that we also consider the global needs, that is, variables live across basic blocks may be allocated registers. For this live variable analysis is done across the basic blocks and variable usage sets are generated. Global needs is estimated using these variable usage sets.

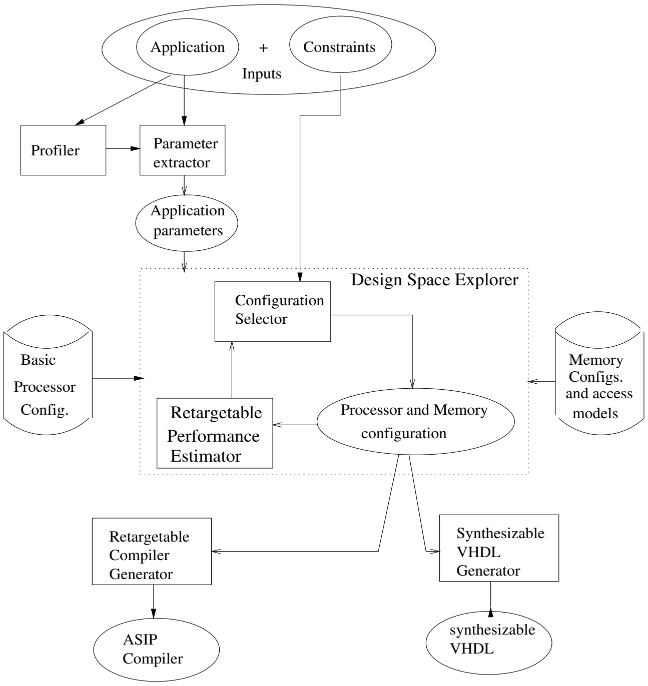



**Figure 2: ASIP Synthesis Methodology : ASSIST**

# 3. OVERALL METHODOLOGY AND OUR TECHNIQUE

Work reported in this paper is a part of the ASIP Synthesis Methodology (ASSIST), being developed.

## 3.1 ASSIST : ASIP Design Methodology

The overall flow diagram of ASSIST methodology is shown in figure 2. 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 processor.

Specifically this research work is focused on developing an efficient retargetable performance estimator for RISC and VLIW class of architectures, considering the flexibility in the register file size.

## 3.2 A Scheduler based Technique to Explore Register File Size

Our performance estimation methodology is shown in fig. 3. 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* [15]. 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 as input. Initial register reuse chains are formed. 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. These costs are computed when compatibility analysis is done. Attempt is made to introduce minimum spill 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. This modified data flow graph is taken as input by a priority based resource constrained scheduler, and performs scheduling based on the list scheduling 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 can be expressed as

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

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.

# 4. PERFORMANCE ESTIMATION

We now describe in detail our technique for estimating performance using an illustrative example.

The input program code written in C along with dependency graph generated using *SUIF* intermediate representation is shown in figure 4. Certain constant values (e.g. x) are needed to perform some operations. These values are first loaded into registers and the operators get these values from the registers, this is the reason we have shown loaded constants as nodes (e.g Lx) in the dependency graph. Ignoring these nodes will under estimate the register needs.

## 4.1 Register allocation using Register Reuse Chains (RRCs)

Register reuse chain or RRC is a sequence of operation nodes which can be assigned the same register. We use these chains to estimate the increase in the schedule length and spilling overheads for a given number of registers.

### 4.1.1 Generating Initial Register Reuse Chains

In the initial register reuse chains, two nodes are assigned to the same chain only if two nodes do not overlap their live ranges in any possible schedule. To find initial RRCs, earliest (in BFS order of the dependency graph) ultimate killing nodes (EUK) for all the nodes are identified. EUK of a node is the earliest node where the

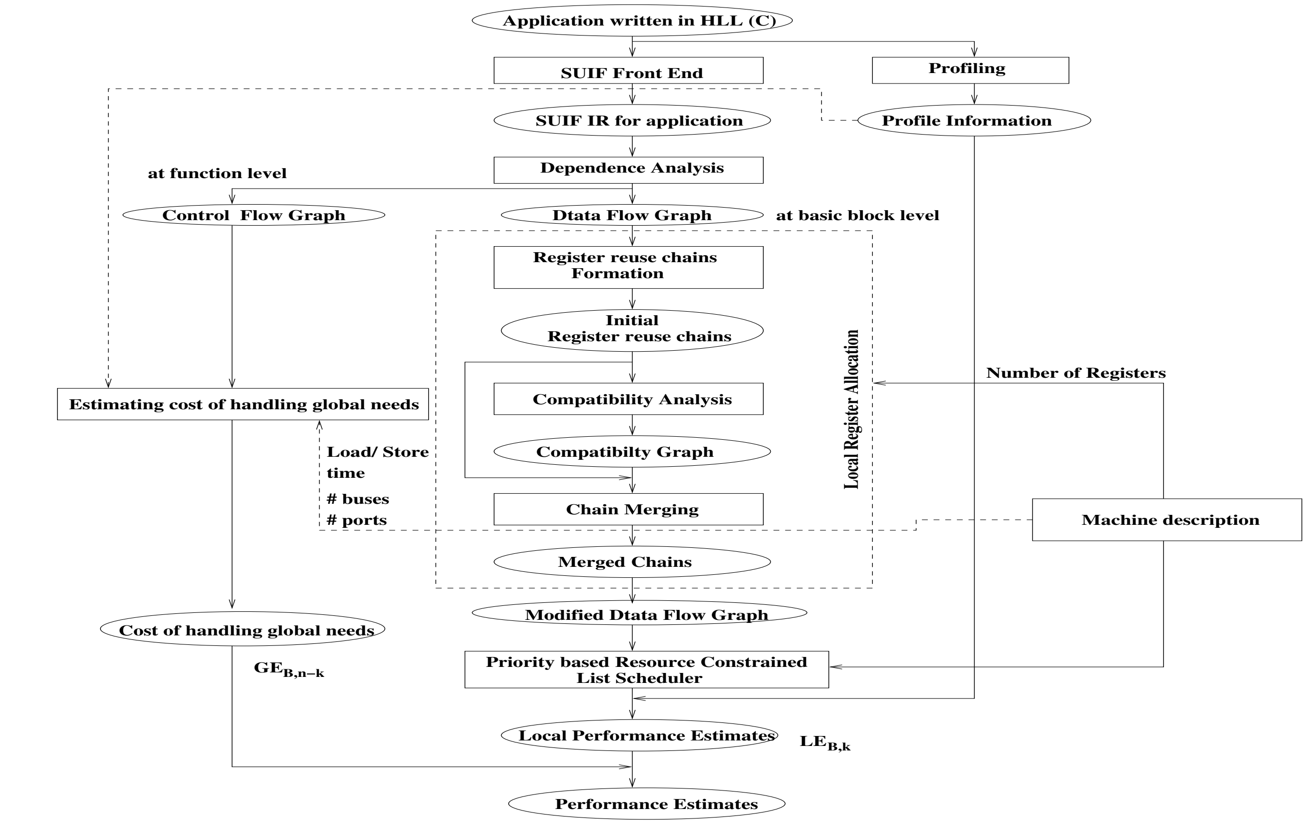



**Figure 3: Flow diagram of our performance estimation scheme**

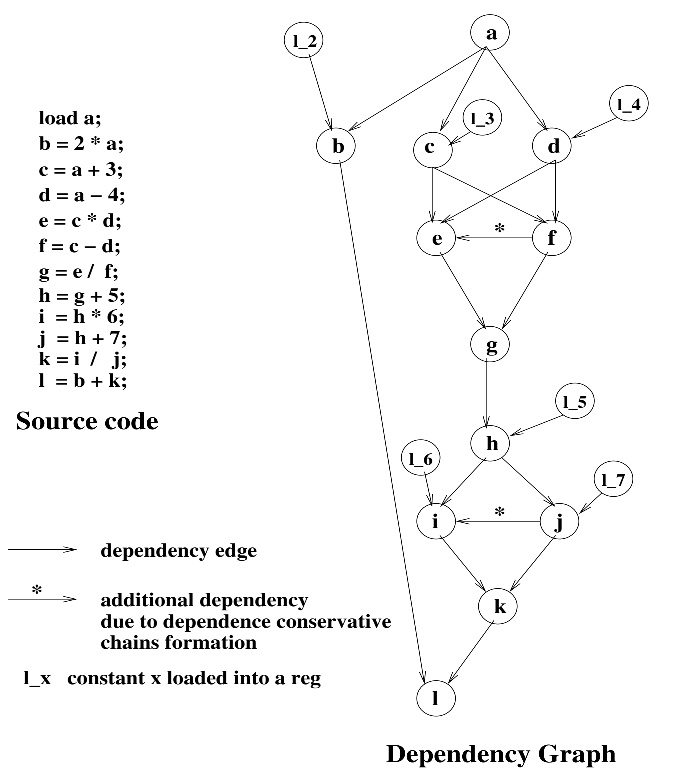



**Figure 4: Source code and dependency graph**

life time of the variable/ value residing in the node is guaranteed to be over [4]. EUK of the 'l_2' is 'b' in the considered example. To find the EUK of 'a' we have to find the earliest node in the graph where all the paths starting from node 'a' merges, 'l' is such a node, so EUK of 'a' is 'l' and neither of 'b', 'c' or 'd' which are directly dependent on 'a'. Because without knowing the schedule we don't know the order in which the dependent nodes will use the value of 'a', so only at 'l' we are sure that now definitely the value of 'a' is not required. Similarly EUK node of all the nodes is identified and shown in table 1.

Now, the first unvisited node in BFS order is picked up and will be the heading node of a new chain. Then recursively find the next EUK and if the identified node is unvisited then add this node into the chain otherwise add first next unvisited node into this chain. Once we cannot find any such node further, again find the first unvisited node (in BFS order) which will be heading another new chain and the process of forming the chain is repeated. This process is iterated till all the nodes are visited. Based on this method the initial chains formed for the illustrative examples are shown in figure 5.

### *4.1.2 Converting initial chains into dependence conservative Chains*

The way initial register reuse chains are formed, no additional dependencies would be generated and it is possible to schedule the dependency graph without any loss in terms of the performance. If the number of initial chains are more than the registers made available for local allocation ($k$), then it is not possible to allocate separate registers to each RRC and several chains may have to share a register. Hence, these chains are first converted into dependence

| Node | a | l_2 | b | l_3 | c | l_4 | d | e | f | g | l_5 | h | l_6 | i | l_7 | j | k | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EUK Node | l | b | l | c | g | d | g | g | g | h | h | k | i | k | j | k | l | l |

**Table 1: EUK nodes of all the nodes**

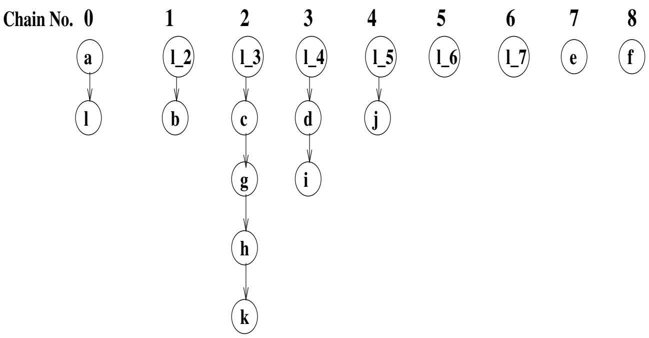



**Figure 5: Initial register reuse chains**

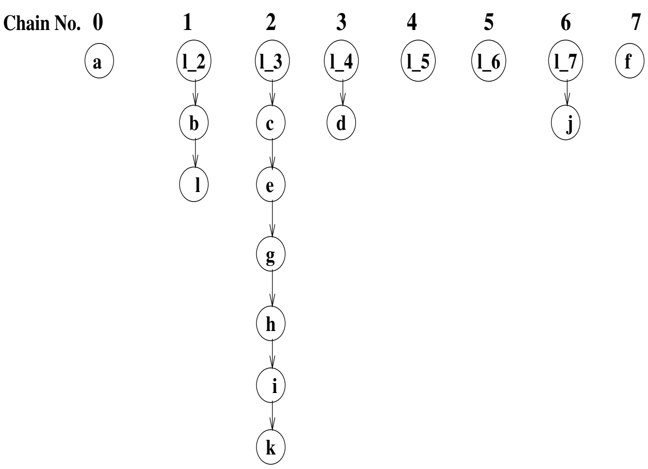



**Figure 6: Dependence conservative chains**

conservative chains, in which a node and its dependent node will be in the same chain [4]. In this process, if a node and its dependent node are in different chains, they are brought into one chain. If a node has more than one dependent nodes, then one of them will be brought into the chain and additional dependencies will be created from other dependent nodes to that node. This process can lead a certain chain to be a null chain. These chains are removed and thus reducing the number of chains. Reduced chains for illustrative example are shown in figure 6.

In this example, ‘l’ depends on ‘b’ and which were in different chains, are brought into the same chain. Similarly ‘e’ depends on ‘c’ and they are also brought into the same chain, here the chain which was containing ‘e’ becomes null and is removed. But when ‘c’ and ‘e’ are brought into the same chain, additional dependency is created from ‘f’ to ‘e’ since ‘f’ also needs the value of ‘c’. Though this additional dependency is slightly different from the true data dependencies, if the other resources allows, they may be scheduled together. It is important to note here that ‘f’ depends on ‘d’, but they are not allowed to go into the same RRC. The reason is that if we allow this, then another additional dependency is to be generated from ‘e’ to ‘f’ since ‘e’ also uses the value of ‘d’ and adding such a dependency may cause a situation where schedule is not feasible.

#### *4.1.3 Computing Merging Costs*

Once the initial chains are formed and they are converted into dependence conservative chains, if the number of chains are larger than the number of registers available ($k$), then we have to merge these chains so that the total number of chains is reduced to the desired level. To assist this merging process, compatibility analysis is performed and a directed weighted compatibility graph is generated. Nodes in the compatibility graph are the register reuse chains and the edges represents compatibility of merging two chains. An edge from node i to node j will indicate the compatibility of merging node j after node i (i.e. operation nodes of the chain j will reuse the register assigned to the chain i) and the cost $M_{i,j}$ associated with edge $< i, j >$ is the measure of loads and stores required for the said merging. Based on the dependency information, these costs are easily computed. It depends on how many values we have to store in memory and load back when they are required. Computation of merging costs between some pairs of chains of illustrative example is shown in fig. 7.

#### *4.1.4 Set Formation, Actual Merging and Performance Estimation*

After the compatibility graph is generated, the edges of this graph are sorted in a non-decreasing order of the associated costs. Sets of RRCs are formed. Each set will contain the RRCs which can share a register (of course by incurring cost in terms of loads and stores or by generating additional dependencies). Ordered compatibility edges helps in the set formation. Chains of each set are actually merged in the same order as they appear in the set. In this merging, spill instructions (loads and stores) are inserted. When chains are merged in a set apart from spills, dependencies may get modified. A priority based resource constrained scheduler is used to estimate computation time. This estimate is multiplied with the execution frequency of the block (obtained by profiling on typical data sets) to get the local estimate ($LE_{B,k}$) for that block. With varying number of registers for local allocation ($k$) the sets and the schedule estimates are shown in figure 8. Additional schedule time overhead because of global needs ($GE_{B,n-k}$) are added to get complete schedule estimate for that block. Next subsection explains how this additional overhead is estimated.

### 4.2 Addressing Global Register Needs

The additional schedule overhead due to global needs ($GE_{B,n-k}$) for basic block B, when out of 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 below.

While doing control and data flow graph generation following additional sets are generated.

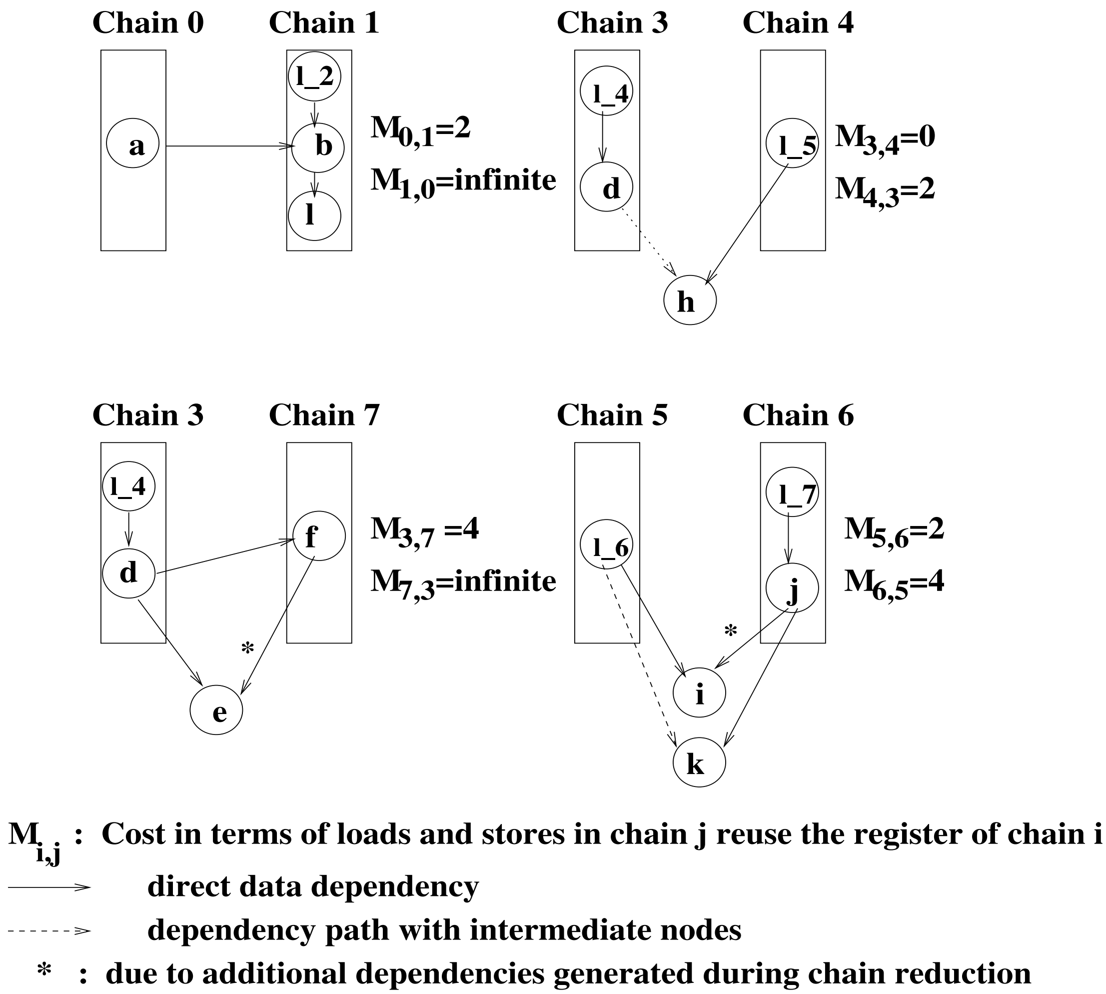



**Figure 7: Computing merging costs**

| Number of Registers (k) | Sets | Schedule estimates (# cycles) $LE_{B,k}$ |
|---|---|---|
| >= 8 | <{a}>, <{l_2,b,l}>, <{l_3,c,e,g,h,i,k}>, <{l_4,d}>, <{l_5}>, <{l_6}>, <{l_7,j}>, <{f}> | 29 |
| 7 | <{a}>, <{l_2,b,l}>, <{l_3,c,e,g,h,i,k}>, <{l_4,d},{l_5},{l_6}>, <{l_7,j}>, <{f}> | 35 |
| 6 | <{a}>, <{l_3,c,e,g,h,i,k}>, <{l_4,d},{l_5},{l_6},{l_2,b,l}>, <{l_7,j}>, <{f}> | 38 |
| 5 | <{a}>, <{l_3,c,e,g,h,i,k}>, <{l_4,d},{l_5},{l_6},{l_7,j},{l_2,b,l}>, <{f}> | 43 |
| 4 | <{a},{l_4,d},{l_5},{l_6},{l_7,j},{l_2,b,l}>, <{l_3,c,e,g,h,i,k}>, <{f}> | 50 |
| 3 | <{a},{l_4,d},{f},{l_5},{l_6},{l_7,j},{l_2,b,l}>, <{l_3,c,e,g,h,i,k}> | 55 |

**Figure 8: Sets and schedule estimates with diff. $k$**

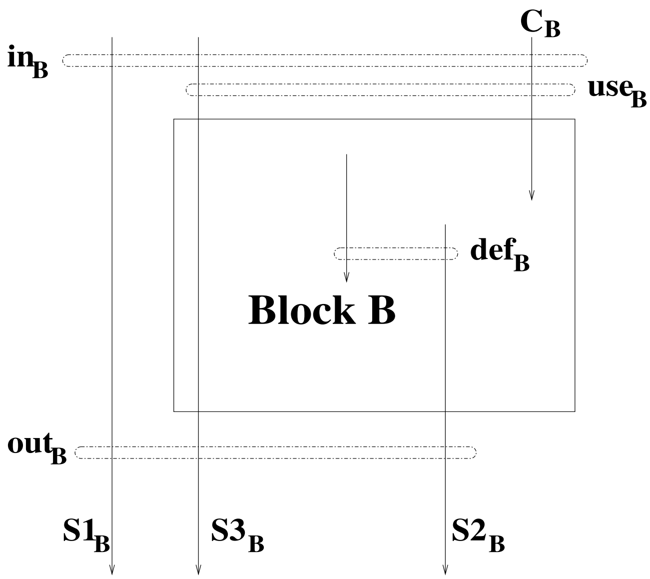



**Figure 9: Variable Use Sets**

$use_B$ : Set of variables used in block B prior to any definition in that block.
$def_B$ : Set of variables defined in block B prior to any use in that block.

From the control flow graph, predecessor and successor blocks of each block are known. Using this information along with the sets $use_B$ and $def_B$, two sets, namely $in_B$ and $out_B$ are computed with the help of the algorithm given in [11].
$in_B$ : Set of variables live at the time of entry into block B.
$out_B$ : Set of variables live at the time of exit from block B.

Taking $use_B$, $def_B$, $in_B$ and $out_B$ as inputs we compute the following three sets for each basic block B (figure 9).
$C_B$ : Set of variables used in block B, but not used in any of the successor blocks.
$S1_B$ : Set of variables that need to be kept live across block B but are not accessed in block B.
$S2_B$ : Set of variables generated in block B and required by successor blocks.
$S3_B$ : Set of variables used in block B and required by successor blocks.

Depending on the availability of registers, some variables will be allocated register, whereas the remaining are stored in memory. Similarly, variables to be loaded from memory needs memory access.

Some additional notations used are as follows.
$||S||$ : cardianility of set $S$.
$S^R$ : subset of S which gets registers.
$prof_B$ : total execution count of block B.

Upper bound on total register needs will be

$$Total\ needs = ||S1_B^R|| + max(||S2_B|| + ||S3_B||, k)$$

The reason for taking this as the total needs is based on the fact that out of the variables being used in the block and required by successor blocks (i.e. $||S2_B|| + ||S3_B||$), only $k$ could be preserved in the registers made available for local allocation. If we have sufficient number of registers to handle this need then additional store is not required. If any variable of $S3_B$ cannot be preserved in register then that should be spilled $prof_B$ times, because it is used in block B and may be modified in this block, otherwise the values which are not allocated register needs to be spilled $prof\_s_B$ times, where $prof\_s_B$ would depend on execution count of successor blocks.

$$\begin{aligned} GS_{B,n-k} &= 0 \\ &\quad if\ (\| S1_B^R \| + max(\| S2_B \| + \| S3_B \|, k)) \leq n \\ &= (\| S1_B^R \| + max(\| S2_B \| + \| S3_B \|, k) - n) * \\ &\quad prof\_s_B \qquad else\ if\ \| S3_B \| \leq n \\ &= (\| S3_B \| - n) * prof_B + (\| S1_B^R \| + \| S2_B \|) * \\ &\quad prof\_s_B \qquad otherwise \end{aligned}$$

If total number of outgoing values is greater than total registers ($n$) then only $n$ such values can get registers. While computing $||out_P^R||$, branch probabilities are also considered.

$$||out_B^R|| = min(n, ||S1_B^R|| + ||S2_B|| + ||S3_B||)$$

$||S1_B^R||$ will be non zero only if $||in_B^R||$ is larger than $||use_B||$ and the value will be the difference of two. This is based on the assumption that elements of $S1_B$ (values flowing across the block but not being accessed in the block) will be allocated registers only if registers are spare even after allocating registers to all the elements of $in_B \cap use_B$ that is those variables which are required in block B.

$$\begin{aligned} \| S1_B^R &= \| in_B^R \| - \| use_B \| \quad if\ \| in_B^R \| > \| use_B \| \\ &= 0 \qquad\qquad otherwise \end{aligned}$$

$$\| in_B^R \| = min(\| in_B \|, n, min_P \| out_P^R \|)$$

Where P is any immediate predecessor block of block B. Since $||in_B^R||$ depends on $||out_P^R||$ of its predecessor blocks and control flow graph may have loops, a few iterations may be required to get stable $||in_B^R||$ and $||out_B^R||$ for each block.

Loads are estimated based on how many times the required variable is to be loaded from the memory which would not be available in register.

$$\begin{aligned} GL_{B,n-k} &= 0 \quad if\ \| in_B^R \| \geq \| use_B \| \\ &= (\| use_B \| - \| in_B^R \|) * prof\_p_B \\ &\quad else\ if\ \| in_B^R \| > \| S3_B \| \\ &= (\| use_B \| - \| in_B^R \| - \| S3_B \|) * prof\_p_B + \\ &\quad \| S3_B \| * prof_B \qquad otherwise \end{aligned}$$

Where $prof\_p_B$ would depend on execution count of predecessor blocks. While estimating loads and stores we also considered minimizing them by possibly copy operation in case we have surplus registers after handling global needs.

# 5. RESULTS

Performance estimations with varying register file sizes for selected benchmarks applications were done. Two processors namely ARM (*ARM7TDMI*) [16] and Trimedia (*TM-1000*) [17] were chosen for experimentation and validation.

The *ARM7TDMI* by ARM Ltd [16] is a 32-bit RISC processor. It supports 16-bit instructions in *THUMB* mode which operates on the same 32-bit register set so it achieves better performance compared to traditional 16-bit processors using 16-bit registers and consumes less power than traditional 32-bit processors.

Trimedia (*TM-1000*) CPU is based on a VLIW architecture with an instruction set optimized for system-on-chip (SoC) solutions requiring real-time processing of video, audio, graphics, and communications data streams. Its five-issue-slot instruction length enables up to five operations to be scheduled in parallel into a single VLIW instruction.

## 5.1 Benchmark Suite

The benchmark 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.

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

## 5.2 Processor description

A simple description is written for the processor. Our description involves latency of *SUIF* operations for this processor core. We also specify the maximum range of register file size to be explored. Nature of pipelining (number of pipeline stages and specification of stages) and buses is specified. Processor can have more than one different type of functional units as well as multiple FUs of each type is permissible. For VLIW processors slotting information is also captured. Flat memory is considered and the read/ write latencies are captured in the description.

## 5.3 Performance Estimates with Varying Register File Size

Our technique generates performance estimates for the complete range of register file sizes. i.e. between 1 and the maximum limit as defined in the input description file. Estimates for all the functions as well as complete application are generated. Where register allocation is not feasible, even with the consideration of loads and stores, an error is reported.

Maximum register file size was defined 16 for *ARM7TDMI* and 128 for *TM-1000*. Results show the performance improvement with larger number of registers (we had ignored the impact of register file size on instruction width). 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. Observation for matrix multiplication benchmark for *ARM7TDMI* processor are shown in figure 10. In this case the performance is saturated at 12 registers.

Since this technique allows one to vary register file size and generate performance estimates, it can guide an ASIP designer in choosing a suitable register file size. This choice will be driven by the specified performance constraints, as well as a suitable cost model for register file.

## 5.4 Validation

To verify correctness of the estimates produced by our technique, it was validated for two processors, namely, ARM (*ARM7TDMI*) and Trimedia (*TM-1000*). Specific register file size is chosen for validation considering the size supported by their standard tool sets. Suitable estimates are considered for library function calls except *printf*. For this purpose, the schedule estimates for library function calls are generated off line using standard tool sets.

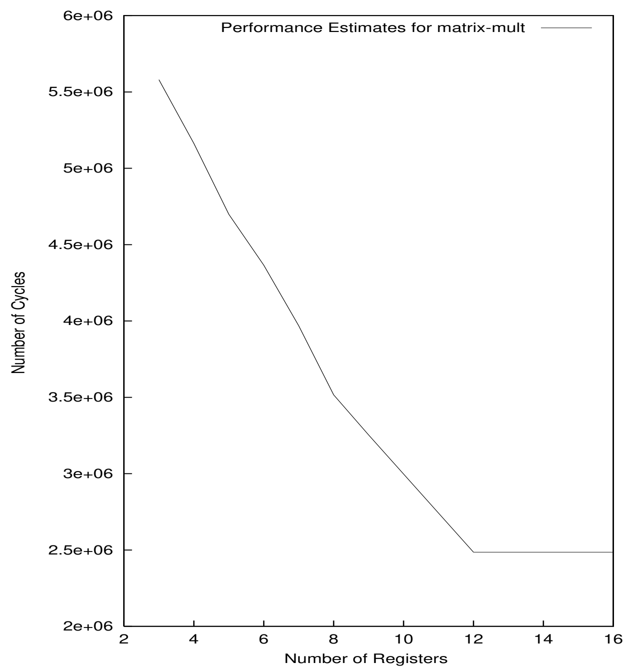



**Figure 10: Execution time estimates for matrix-mult for *ARM7TDMI***

### *5.4.1 Validation for ARM7TDMI*

Estimates for *ARM7TDMI* are generated for 8 registers using the following tool sets.

**1. ARM :** armulator tool set consists of ARM code generator and ARM simulator.
**2. ASSIST :** Our technique.
**3. encc :** *encc* code generator and ARM simulator [3].

Results are shown in figure 11. We observe that our technique produced estimates within 9.6% compared to the performance numbers produced by standard compiler and simulator available in armulator tool set. On the other hand the simulator based technique *encc* is produced estimates with an average error of about 29.6%.

We have compared the run time elapsed in our technique with *encc* technique. Both techniques provides facility of getting performance estimates with different register file sizes, though the register file size that could be explored in *encc* technique is limited to 8. This is because the ARM instruction set simulator used for simulation could not support larger register files. Both tools (ours and *encc*) are run on a SUN blade 1000 server having 64-bit dual processors, clock frequency of 750 MHz with 4 GB RAM. 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. It uses ARM simulator for simulation.

### *5.4.2 Validation for TM-1000*

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.

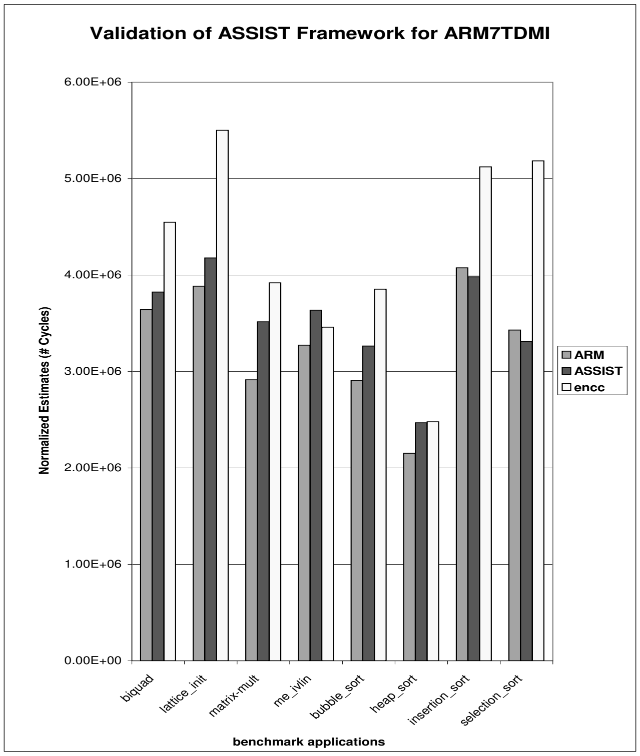



**Figure 11: Validation for *ARM7TDMI***

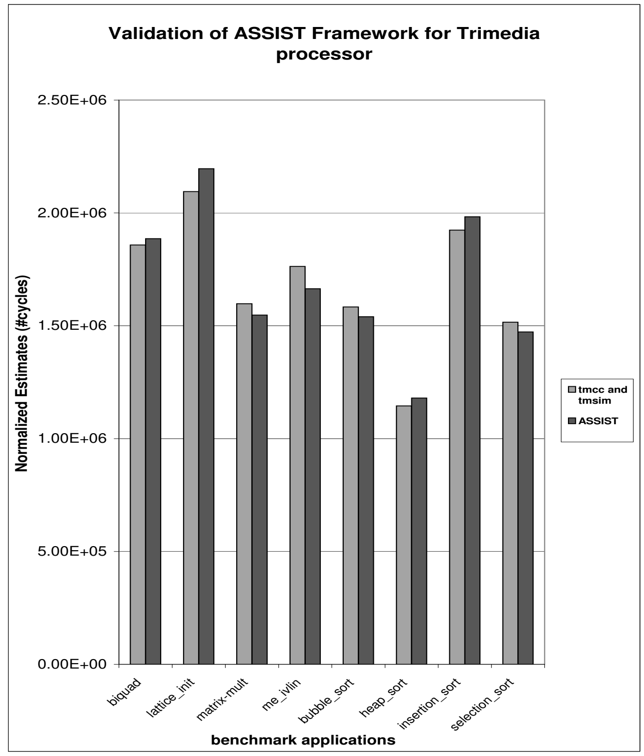



**Figure 12: Validation for *TM-1000***

Estimates are generated for 128 registers using the following tool sets.

**1. tmcc and tmsim :** Trimedia tool set consists of *tmcc* cross compiler and *tmsim* instruction set simulator.
**2. ASSIST :** Our technique.

Results are shown in figure 12. 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.

# 6. CONCLUSIONS AND FUTURE WORK

We have developed a complete strategy to decide suitable register file size using a scheduler based technique. We profile the application using *gprof* to get execution counts of various blocks. Our technique uses *SUIF* intermediate format to extract dependency and control information. Using the concepts of register reuse chains, we are able to do register allocation before scheduling. Using scheduler outputs and profiler outputs, local estimates for all the blocks are generated. Overheads to handle global needs are also considered at the time of estimating the performance.

Performance estimates for register file sizes between 1 to 8 for ARM *ARM7TDMI* and 1 to 128 for Trimedia TM-1000 were generated for selected benchmarks. The technique was validated for these two processors for specific number of registers. Results show that our estimates are within 9.6% for *ARM7TDMI* and 3.3% for *TM-1000* compared to the actual performance produced by standard tool sets. Further, this technique was nearly 77 times faster compared to the simulator based technique *encc*. Our technique is also capable of generating performance estimates with varying number of FUs of various types.

In summery, main features of the proposed technique are listed here.

- Neither machine specific code is generated, nor simulation is done. Using a simulator usually restricts the register file size which can be explored.
- Register allocation is performed before scheduling, which is more appropriate for ASIPs.
- If only spill overhead estimates can guide the register file size evaluation, then scheduling is not required.
- Option of spilling a variable to memory and loading back when it is required, is considered (which is not considered by Zhang et al [4]).
- Global needs are considered, which are ignored by the most of the DAG driven techniques including [4].
- Very simple parameterized processor description is taken as input, making it retargetable to a large architecture design space. As an example, the technique is capable of generating performance estimates with varying number of FUs of various types.
- Performance is estimated using a scheduler and profile values.
- Validated for a RISC (*ARM7TDMI*) and a VLIW (*TM-1000*) processor.
- Significantly faster compared to compiler-simulator based framework.

Our future work will involve exploring register file size and register windows in an integrated manner. This will be integrated in a comprehensive memory exploration tool.

## 7. ACKNOWLEDGMENTS

This research work is supported by DST-DAAD cooperation project (Project code MCS-216) and NRB, government of India sponsored embedded systems project.

## 8. 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] Y. Honma, M. Imai, A. Shiomi, and N. Hikichi. Register Count Optimization in Application Specific Integrated Processors. In *Proceedings of the second Asia Pacific Conference on Hardware Description Languages (APCHDL 1994)*, pages 263–266, October 1994.

[9] 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.

[10] 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.

[11] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison- Wesley, 1977.

[12] 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.

[13] C. Norris and L.L. Pollock. Register allocation over the program dependence graph. In *Proceedings of the Conference on Programming Language Design and Implementation, (SIGPLAN 1993*, June 1994.

[14] 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.

[15] SUIF Homepage. “http://www.stanford.edu/suif”.

[16] ARM Ltd. Homepage. “http://www.arm.com”.

[17] Trimedia Homepage. “http://www.trimedia.com”.