Compiler-Based Register Name Adjustment for Low-Power Embedded Processors*

Peter Petrov
University of California at San Diego
CSE Department
ppetrov@cs.ucsd.edu

Alex Orailoglu
University of California at San Diego
CSE Department
alex@cs.ucsd.edu

ABSTRACT

We present an algorithm for compiler-driven register name adjustment with the main goal of power minimization on instruction fetch and register file access. In most instruction set architecture (ISA) designs, the register fields reside in fixed positions within the instruction encoding, hence forming streams of indices on the instruction bus and to the register file address decoder. The number of bit transitions in these streams greatly determines the power consumption on the address bus and the register file decoder. While general-purpose registers are semantically indistinguishable and hence interchangeable, the particular register indices do have a direct impact on power consumption. The algorithms presented in this paper address this power minimization problem by reassigning/encoding the registers so that the bit transitions within the register index streams are minimized.

1. INTRODUCTION

System-on-a-chip (SOC) design approaches, typically embedding a number of processor cores, have become a generally accepted implementation paradigm for numerous VLSI products. While power consumption is an increasingly important characteristic in general purpose processor architectures, it assumes a much heightened importance when embedded processor architectures are considered. In mobile applications, such as hand-held and wireless devices, not only does it impact directly cost of ownership, but furthermore may severely undermine the usability and acceptance of the product. Less energy dissipation leads not only to longer battery life, but also enables larger die sizes. Consequently, techniques for minimizing system power are of paramount importance for achieving high product quality.

Programming languages like C, C++ and Esterel are frequently utilized during the design process for quick time-to-market, low design cost, and portable function implementation. Consequently, it is of paramount importance for the compiling system to be able to generate performance and, even more importantly in some cases, power efficient code. Nonetheless, most of the current research effort has been directed towards performance oriented and retargetable compilers with reduced emphasis on power driven compiler optimizations. In this paper we present a technique which aims at minimizing power while retaining full orthogonality to all other compiler optimizations.

The instruction fetch logic of processor cores has a significant contribution towards the total power consumption [1]. In a typical ISA design based on RISC architecture concepts, the register fields are in *fixed positions* within the instruction encoding and constitute a significant part of the instruction word. The register fields form *streams of indices* being transferred on the instruction bus. Furthermore, the indices of the source registers are transferred to the address decoders on the read ports of the register file. The larger the difference in terms of bit positions in any two consecutive indices, the higher the power in the address decoder will be, as increased bit toggles will occur.

The number of bit transitions can be ameliorated by exploiting the fact that no fixed, preordained correspondence exists between program variables and register names. Nonetheless, revisiting the problem of register allocation is impractical as compiler technologies attempt optimizations that aim at satisfying multiple other objectives. A permutation of the register indices though can still be utilized while retaining intact all the prior optimizations achieved by the compiler. We outline such a technique that identifies a permutation of register names in order to minimize bit transitions. The reductions attained by such techniques are proportional to the skew in the distribution of adjacent pairs in the original program. We therefore complement our technique with a preprocessing step that relies on commutativity and dead register exploitation in order to increase the skew. This skew enhancing technique, applied right after register allocation and prior to register permutation, delivers exceptional bit transition reductions, which we illustrate in the experimental results section. The proposed methodology can be utilized within any system for embedded software compilation/synthesis or as a stand-alone tool that operates on the binary code produced either automatically or manually by the system designer.

Minimizing transition activities in communicating data across various parts of digital systems has been a major goal in recent research efforts for low power. The *Bus-Invert* method has been proposed in [2]. The bus lines are inverted if this leads to a smaller Hamming distance from the previous bus value. A power optimization technique applied during behavioral synthesis for memory intensive applications has been presented in [3]. The behavior of the memory access patterns is utilized to minimize the number of transitions on the address bus and decoder.

The objectives in some approaches for low-power FSM state encoding bear resemblance to the optimization goals of the register name adjustment methodology that we propose in this paper. Low-power state encoding algorithms aim at encoding the FSM states, so that the number of transitions within the state register are minimized. In [4] a heuristic based on a probabilistic model to describe the FSM is presented. In [5] the problem of low-power state encoding is formulated as a graph embedding problem.

2. GENERAL FRAMEWORK

Let's consider a code fragment shown in Figure 1a. The first register field in the instruction format represents the target, while the next two registers correspond to instruction sources. The target register indices form the sequence $3, 6, 3, 4$, resulting in a total number of bit transitions of 7. The first register source field sequence of $2, 3, 2, 4$ results in 4 transitions, while the second source register field sequence of $4, 5, 6, 5$ results in 5 transitions. In sum, all the register indices are responsible for 16 transitions in total.

It is evident that the total number of transitions depends on the particular register indices. Let's analyze how this number changes if registers $r3$ and $r6$ are interchanged with $r6$ and $r7$, respectively, while the rest are left intact. The resulting code sequence is shown in Figure 1b. Counting the number of bit transitions for the three new streams of indices shows that the first column has 3 transitions, the second 4, and the third 3 transitions, resulting in a total of 10 transitions. This example illustrates the power of register name reassign-

*This work is supported by NSF Grant 0082325.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICCAD '03, November 11-13, 2003, San Jose, California, USA.
Copyright 2003 ACM 1-58113-762-1/03/0011 ...$5.00.

```
add r3, r2, r4          add r6, r2, r4
sub r6, r3, r5          sub r7, r6, r5
sub r3, r2, r6          sub r6, r2, r7
mul r4, r4, r5          mul r4, r4, r5
```

a) b)

Figure 1: Example code fragment

ment in terms of achieving a significant reduction in bit transitions. Such register index permutations are referred to as *Register Name Adjustment (RNA)* throughout the paper.

The interchangeability of the general-purpose registers leads to the observation that any permutation of the register indices within a program fragment leads to a valid register assignment with no performance penalties but possibly with drastically differing power characteristics. Consequently, such transformations which reassign register indices can be considered with the optimization goal of achieving reduced bit transition activity on the register index streams. For instance, the commutativity property of a large number of instructions provides an opportunity for swapping register operands in order to minimize bit transitions from the register indices. Evidently, this transformation does not have any side effects on code performance and will be shown to be quite efficient to implement within the proposed methodology. Another transformation that we explore in this paper utilizes the observation that within basic blocks the particular register used to hold a temporary value is of no consequence. It is possible to reassign therefore target registers to currently dead registers so that an objective function with regard to bit transitions is optimized.

The typical register allocation algorithm performed by any compiler identifies a mapping between the program working set and the register file with minimal amount of spill/fill code. The optimal register allocation solution is not unique, even when we keep the instruction sequence intact by disallowing the introduction of new code or the reorganization of the structure of the program. Fundamentally, the transformations that we outline in this paper are the vehicle for traversing the space of equivalent register allocation solutions under the constraint of minimized bit transitions on the register indices.

The example presented in Figure 1 contains only arithmetic instructions with three register operands. It is possible of course to have a mixture of various types of instructions, including loads and stores. In such a case, the part of the instruction that corresponds to a non-utilized register field contains an immutable value, for example an op-code extension or an immediate field, forcing the *RNA* algorithm to take into account the existence of immutable constants within the stream of register indices.

One can notice that some of the registers in Figure 1, such as $r3$ and $r6$, are defined within the code fragment, other registers are used within the code ($r2$, $r4$, $r5$), while yet others are not referred to at all. Modification of the destination register of an instruction necessitates reflection of the change upon the subsequent code fragments which utilize this register as a source register. More formally, the proposed algorithm should keep intact the define-use register chains.

The optimization we propose is applied primarily on the major application loops, with special care taken in between them so that possible define-use register chains are preserved through the loops. This special support is needed in the case of renaming a register in the first loop, whose value is needed in the second loop; the insertion of a register transfer instruction helps to preserve the define-use chain in that case. The few such register move instructions between the hot spots introduce no overhead in practice, neither in terms of performance nor in terms of power, since they are executed quite rarely.

The proposed algorithm targets the application loops as independent and atomic blocks. It aims at finding an efficient register renaming with the objective of minimizing the total amount of bit transitions for the streams of register indices. As most of the transitions occur in executing instructions inside the basic blocks, the register renaming algorithm needs first to take into account the register utilization information within the basic blocks. Furthermore, a profile information may be utilized to impose a priority order to the basic blocks and to combine them into code fragments for considering the inter-block register index transitions. The algorithms that we present utilize information regarding the order of the register indices, the frequency of occurrence of register pairs, and the register live intervals within the basic blocks. Consequently, as no analysis across basic blocks is required, the algorithms need not be restricted to a single block but can be directly applied on a set of basic blocks.

3. MATHEMATICAL FORMULATION

The Register Name Adjustment (RNA) problem presented above can be defined formally as:

Given: Integers $n \in N$, $k \in N$, $C \in N$; a range $V = [0 \ldots 2^k - 1]$ of integers, a matrix $P \in V^{n\times 3}$, and a distance function $f_d \in V^2 \rightarrow N$.

Problem: Find a bijective mapping $M: V \rightarrow V$, such that:

$$Cost \stackrel{\text{def}}{=} \sum_{l=1}^{3} \sum_{i=1}^{n-1} f_c(M(P_{i,l}), M(P_{i+1,l})) \;\; \leq \;\; C.$$

In the problem formulation above, the integer number n corresponds to the number of instructions within the code fragment being considered, while the matrix P contains the initial register indices assigned by the register allocator. The number k represents the register bit-width. The distance function measures the distance between two register indices; in the case of bit transitions, f_d is defined as the Hamming distance between the two integers. For the code fragment in Figure 1, $n = 4$, $k = 4$, and matrix P with columns $p_1 = (3, 6, 3, 4)^t$, $p_2 = (2, 3, 2, 4)^t$, and $p_3 = (4, 5, 6, 5)^t$. The problem consists of identifying a bijective mapping of the current register indices to some new indices, i.e. renaming the registers, so that in summing up the Hamming distance between all consecutive pairs of indices, the total does not exceed a given threshold C. It is evident that if a polynomial time algorithm exists for solving this problem, then it is straightforward to solve the corresponding minimization problem in polynomial time as well, by iteratively decrementing the constant C, by starting, for example, from the total number of transitions in the original register assignment, until no mapping is feasible, with the last successful step constituting the optimization solution.

In the case of instructions utilizing not all three register fields, such as loads or stores, the bit positions within their encoding that coincide with an unused register field would contain an immutable value possibly as part of an opcode or immediate operand. We refer to such constants as *literals*. These literals would be intermixed within the streams of register indices and can have a significant impact on the total bit transition number. The problem definition is easily extended to handle such literals by simply treating them as special registers, which cannot be renamed. More formally, an additional binary matrix $L \in \mathbf{B}^{n\times 3}$ is introduced, which models the literal positions. A value of 1 in entry $L_{i,j}$ indicates that the jth register field of the ith instruction is not used and that a literal resides there instead. The computation of the $Cost$ value is consequently updated to the following mapping function.

$$M' = \begin{cases} M(P_{i,j}) & \text{if } L_{i,j} = 0 \\ P_{i,j} & \text{if } L_{i,j} = 1 \end{cases}$$

Furthermore, if there are multiple independent basic blocks to be considered, then multiple n's and P's would be needed. The final cost of the solution needs to be computed by summing the costs over all the individual blocks.

It is evident that the formulation we have presented is an instance of the general problem with multiple basic blocks and literals. Subsequently, we briefly discuss and show the NP-completeness of this

a)
```
ld  r5, (r1)0
add r3, r2, r5
add r4, r3, r2
mul r3, r4, r3
st  r3, r7(10)
```

$$P=\begin{bmatrix} v1 & v1 & 0 \\ v3 & v2 & v5 \\ v4 & v3 & v2 \\ v3 & v4 & v3 \\ v3 & v7 & 10 \end{bmatrix} \quad L=\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

b)
```
(v3,v4)-3    (v4,v7)-1
(v2,v3)-2    (v5,v2)-1
(v5,v3)-1    (0, v5)-1
(v3,v3)-1    (10,v3)-1
```

Figure 2: Frequency distribution of the register pairs

problem. The NP-completeness of the general problem is a trivial consequence of the NP-completeness of its simpler instance. Subsequently, we present an efficient heuristic for solving the general problem with support for literals and multiple basic blocks.

The NP-completeness of the formulated problem can be easily shown by mapping the *Travelling Salesman Problem (TSP)* to an instance of our problem. The cities in the TSP problem are mapped to register indices in the first column of the matrix P, while the second and third columns contain an arbitrary but fixed register. The distance function is defined as the distance between any two cities. Now, finding a sequence of cities for which the total distance when travelling through these cities is below a given threshold is equivalent to finding the mapping function for renaming the registers. Register renaming is a permutation of the register indices, such that the $Cost$ inequality is satisfied with the constant C defined as being the constant in the TSP formulation. This is a polynomial, in fact linear, mapping from the TSP problem to an instance of the RNA, which shows the NP-completeness of the RNA problem.

4. HEURISTIC SOLUTIONS FOR RNA

In this section we show an efficient RNA heuristic. The fundamentals of the RNA heuristic are based on performing a statistical analysis and applying efficient encoding on the most frequently occurring pairs of registers in terms of the number of bit transitions. A possible interpretation of the RNA problem is as an encoding problem. The registers assigned by the register allocator can be thought of as a set of variables being utilized throughout the generated code. Now the problem of finding the register permutation becomes one of assigning indices to the "register variables" so that the total number of transitions between register pairs is minimized. The *Register PermuTation (RPT)* algorithm utilizes the frequencies of the various register pairs and permutes the register indices in such a way so that the frequent register pairs are mapped to indices with minimal Hamming distance. As the impact of the RPT algorithm is directly proportional to the register pair distribution skew, we augment the RNA heuristic by adding the *Register PerturBation (RPB)* step, executed prior to the RPT, in order to maximize the distribution skew of register pair occurrences. The RPB algorithm utilizes two degrees of freedom in assigning registers to instruction operands. The first one relies on the commutativity property, a property common to many instructions. The second transformation, on the other hand, slightly modifies the current register allocation by swapping certain pairs of live intervals within the basic blocks, without perturbing the optimality of the register allocation or introducing any additional instruction.

4.1 Register PermuTation (RPT)

Figure 2a shows an example code fragment with literals and its corresponding representation with matrix P and literal matrix L. We have used indexed letters to represent the registers in the matrix P in order to emphasize the fact that one can consider the registers simply as independent variables for which an efficient power encoding is sought. It can be observed that certain pairs of registers appear more frequently than others, while certain register pair elements never occur successively in the code fragment. Figure 2b shows all pairs of registers and literal-register pairs, which appear in the code and the quantity of each pair. Consequently, the major goal of the register renaming heuristic is to try to find an encoding such that the frequently occurring pairs of registers are mapped to the closest code words.

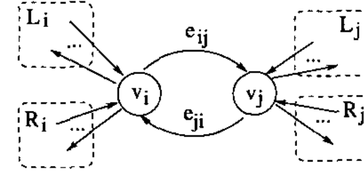

Figure 3: RPT algorithm iterative step

In order to capture the utilization frequency of the register/literal pairs, we construct the *Register Histogram Graph (RHG)*. The RHG is a directed graph with nodes corresponding to the registers and literals from the program fragment. An edge $e_{ij} = (v_i, v_j)$ exists between nodes v_i and v_j only if the registers associated with them occur consecutively within the code in this order. Each edge is annotated with a positive number weight denoted by $f(e_{ij})$, representing the total number of occurrences of the particular register pair. The nodes corresponding to literals are marked as such and no encoding is to be assigned to them. They are treated as nodes with already assigned encoding and their particular value influences the codes assigned to their neighboring nodes.

The RPT heuristic iteratively selects a pair of nodes v_i and v_j with the property that the sum of the weight of edges (v_i, v_j) and (v_j, v_i) is maximal and that at least one of the nodes remains still unassigned. Figure 3 depicts such a pair of nodes and shows the type of incoming and outgoing edges of these nodes. This structure is fundamental in assigning optimal codes to the corresponding nodes. The sets L_i and L_j denote all the literal nodes which are adjacent to the nodes v_i and v_j, respectively. Similarly, the sets R_i and R_j represent the register nodes adjacent to v_i and v_j, respectively. If the nodes v_i and v_j are such that $f(e_{ij}) + f(e_{ji})$ is maximum and at least one of the nodes has not as of yet been assigned a code, the RPT algorithm selects the pair and finds the optimal encoding. The implication is that the RPT algorithm step needs to find indices, such that the total number of transitions between the nodes v_i and v_j and all other adjacent nodes, which have already been assigned indices, is minimized. Therefore, all indices that remain unassigned are searched through and the ones that minimize the total cost are chosen. If both nodes are still unassigned, the total cost C_{ij} is computed using the following formula.

$$\begin{aligned} C_{ij} &= \sum_{k\in L_i} H(c(k),c(i)) + \sum_{\substack{k\in R_i,\\ k\in I}} H(c(i),c(k)) + \\ &+ \sum_{k\in L_j} H(c(k),c(j)) + \sum_{\substack{k\in R_j,\\ k\in I}} H(c(j),c(k)) + \\ &+ (f(e_{ij}) + f(e_{ji}))H(c(i)),c(j)) \end{aligned}$$

The set I in the above formula denotes the set of RHG register nodes, for which indices have been assigned in previous iterations; $c(v)$ denotes the index assigned to the node v, and the function $H(x, y)$ is the Hamming distance between x and y. If one of the nodes i or j has already been assigned an index, C_{ij} is computed by summing the bit transition on the edges of the unassigned node only. Since the set of available indices is, of course, no larger than the total number of registers, this step is computationally effective and its time complexity is $O(|E||R|^2)$, where R is the set of all registers.

4.2 Register PerturBation (RPB)

The fundamental principle of the basic RPT heuristic has been the assignment of efficient indices to the register pairs which have high occurrence frequency in the initial program. If, for example, the RHG contains fewer number of edges yet with higher utilization frequency, then it is evident that the achieved improvement in terms of bit transitions would be higher. If we think of the utilization frequency numbers as simply a distribution histogram of all possible register pairs, and since the total number of register pairs is determined by the particular program and hence immutable, then the above property is precisely reflected by the standard deviation of the histogram. Any transformation that does not change the structure of the program code but only reassigns the register working set, leaves the histogram average intact, but succeeds in altering the standard deviation, enabling the exploitation of the degrees of freedom which commutativity and dead register intervals provide us. The registers can be reassigned in such a way, so that the deviation of the distribution histogram is increased, i.e. by having more edges in the RHG with higher utilization.

It is desirable as well that the RHG have as many as possible self-edges, as no bit transitions across such a pair of registers exist, independent of the particular index assignment. Therefore, in order to further increase the utility of the RNA heuristic, one needs to apply certain transformations on the initial register assignment so that the aforementioned RHG properties are intensified, while performance and, of course, semantic properties of the initial program are left intact. The RPB heuristic based on this conceptual foundation consists of two register transformations applied as a preprocessing step so that both RHG characteristics described above are maximized.

In order to be able to evaluate the transformations quantitatively, a formal definition about both objectives and a way to combine their effect needs to be given. The distribution histogram deviation is computed by utilizing the well known statistical formula $\sigma^2 = \frac{\sum(x-\mu)^2}{N}$, where μ is the average and N is the total number of samples, which in our case corresponds to all possible register pairs. Since the maximal possible deviation can be easily computed, a normalized version of the deviation $\hat{\sigma}$ is introduced for ease of combining both objectives. The second objective of the initial program transformation, as discussed above, is the maximization of the number of pairs, which have the form (v_i, v_i). This quantity can be as well represented in a normalized way using the formula $\hat{D} = \frac{D}{N_p}$, where D is the number of single register pairs, while N_p is the total number of register pairs for the initial program. The combined objective function can be consequently formulated as a convex combination of $\hat{\sigma}$ and $\hat{D}$, i.e. $C_o = \lambda\hat{D}+(1-\lambda)\hat{\sigma}$, where $0 \leq \lambda \leq 1$. By adjusting λ, the priorities of the two objectives can be varied.

4.2.1 *Commutativity transformation*

A large number of instructions, such as addition, multiplication, and logic operations, are insensitive to the order of their operands. Consequently, the commutativity of certain instructions provides a degree of freedom that can be exploited in order to maximize the objective function C_o. It is evident that the *commutativity transformation* introduces no side effects in terms of code performance.

If for a sequence of instructions the commutativity transformation has been applied and an optimal solution has been identified, then the iterative algorithmic step of handling the next instruction while still preserving optimality consists of checking both possibilities for the operands and selecting the one which maximizes the objective function. This observation stems from the fact that the commutativity transformation has only a local impact on the objective function and no changes in the previously optimal instruction sequence are needed. Consequently, the commutativity transformation can be applied in linear time by traversing the instruction sequence and by identifying for each commutative instruction one of the two operand combinations that maximizes the current value of the objective function.

```
r1 <- r2,r3             r1 <- r2,r3
r4 <- r1,r2   --->      r2 <- r1,r2
r2 <- r3,r4             r2 <- r3,r2
```

Figure 4: Dead register reassignment

4.2.2 *Dead register reassignment*

When a register is defined by an instruction, its previous value is overwritten and the new value is utilized by subsequent instructions that have this register as a source operand. The interval between the last usage of a register and its subsequent redefinition is known as the *dead range* for the register, as the value which the register holds is no longer of any importance. Consequently, such a register can be directly assigned a live interval of another register as long as that live interval fits within the dead range. Furthermore, even if the live interval exceeds the dead range, the dead register can still accommodate that new live interval, as long as its own live interval can be reassigned to the other register, effectively swapping both live intervals.

Figure 4 shows an example of dead register reassignment. Register $r2$ is used for the last time in the second instruction. Therefore, it is possible to substitute the target of the second instruction $r4$ with $r2$ and propagate this change throughout the use chain of register $r4$. This transformation evidently improves the objective function for the particular code fragment. Of course, in this particular case such a replacement is possible if the use of the value of $r4$ extends no further than the third instruction, i.e. if $r4$ is dead after the third instruction.

Fundamentally, the dead register reassignment technique tries to reassign the live intervals within the basic blocks to different registers so that the objective function C_o is maximized. The example in Figure 5 shows the live intervals of three registers for a given basic block. The vertical axis corresponds to a sequence of instructions that utilize these registers. Each vertical line corresponds to the definition and last use of a certain register. Each of the live intervals is annotated with an integer for ease of identification. One can observe that register $r2$ is alive when the basic block is completed, i.e. its value is used by at least one of its successor basic blocks from the control flow graph. Note that this information regarding the live intervals is only obtained by performing a global live-in/live-out analysis across the basic blocks.

The right part of Figure 5 shows the live intervals after performing a dead register reassignment. It is evident that this transformation reassigns the live intervals of any temporary values generated within the basic block. For example, live interval 2 is now handled by register $r3$ instead of $r2$, live interval 4 by $r2$ instead of $r1$ and so on. As this transformation is applied within an application fragment with high execution frequency, the introduction of register transfer instructions needs to be avoided. The retainment of any register alive at the basic block exit fixes the live interval 8 to its original register $r2$.

In order to apply this transformation, the instruction sequence needs to be traversed one instruction at a time. At each step all dead registers need to be considered as possible "holders" of the live interval being defined by the current instruction. It is important that no live intervals assigned to the currently dead registers remain unassigned due to

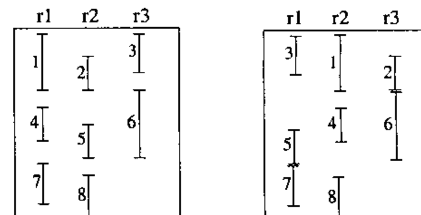

Figure 5: Dead register reassignment applied on a basic block

		RPT		RPB					
	Total	RPT	Impr%	$\lambda(0.0)$	$\lambda(0.25)$	$\lambda(0.5)$	$\lambda(0.75)$	$\lambda(1.0)$	Impr%
fdct	70	58	18.09	47	46	46	46	46	34.55
ej	73,837	63,169	14.45	49,203	48,933	48,934	48,934	45,224	38.75
mmul	7,613	6,463	15.11	4,710	4,460	4,460	4,460	4,593	41.41
tri	5,929	5,400	8.92	3,490	3,489	3,489	3,489	3,335	43.76
sor	1,440	1,142	20.69	1,004	1,003	1,043	1,043	1,004	30.30
adpcm_e	20,513	15,338	25.23	15,897	15,144	15,144	15,144	14,750	28.10
adpcm_d	17,212	13,689	20.46	13,393	12,655	12,655	12,655	11,404	33.74

Figure 6: Bit transition reductions for RPT and RPB

the utilization of their register for "holding" the current live interval. Instead, all such affected live intervals need to be reassigned among the set of dead registers. The evaluation is performed by computing the objective function C_o for each such replacement. At the end, the dead register reassignment that leads to a maximal increase in C_o is selected. As no live intervals are left unassigned, no danger of reaching an instruction and being unable to accommodate the live interval formed by its target register exists. The time complexity of the dead register reassignment is therefore linear in terms of the number of instructions of the program fragment, as no backtracking is necessitated.

5. EXPERIMENTAL RESULTS

In this section we quantitatively evaluate the proposed methodology for register name adjustment. The RPT and the RPB techniques have been implemented and applied on a set of applications. The algorithms have been coded as a stand-alone module, which is independent of the particular instruction set architecture. The input to this module is the control-flow graph for each application hot-spot, where the basic blocks contain a sequence of register/literal triplets with additional flags indicating the target/source fields, literal positions, and whether the instruction corresponding to the triplet is commutative. When the control-flow graph is loaded, a live register analysis is performed across and within the basic blocks, a fundamental part of our implementation for the proposed techniques.

The basic blocks' execution frequency and the control flow graph itself are generated by a specially modified version of the functional simulator from the SimpleScalar toolset [6]. SimpleScalar supports a MIPS-like instruction set architecture which utilizes 5 bit register fields. The simulated programs are instrumented at assembly level by inserting a specially designed instruction at the beginning of every basic block. Execution of this special instruction informs the modified simulator that a new basic block has been entered and that the appropriate actions for analyzing the basic block need to be performed.

In our experimental study we have used the following benchmarks: *Fast discrete cosine transform (fdct)* DSP kernel; *Extrapolated Jacobi-iterative method (ej)* on a 128x128 grid; *Matrix multiplication (mmul)* on a matrix of size 50x50; *A tridiagonal linear system solver (tri)* on a matrix of size 128x128. *Successive over-relaxation (sor)* on a matrix with size 128x128; The *adpcm_e* and *adpcm_d* benchmarks (the encoder and the decoder part of the ADPCM speech coding algorithm, part of the Mediabench [7] set).

Figure 6 shows the achieved results in terms of bit transitions in the register fields and their reduction. The second column presents in thousands the total number of bit transitions from the three register fields. The third column shows the bit transitions after applying the RPT heuristic, while the next column presents the percentage reduction of the RPT heuristic. The numbers of bit transitions after applying the RPB algorithm are shown in columns five through nine. These columns correspond to five different values of the optimization parameter λ. The cost function as explained in the previous section is represented as a convex combination of two objectives. While the two objectives have a distinct nature, they are also intertwined and can impact each other. Therefore, different problem instances necessitate distinct values of λ for attaining the best solution. Consequently, for our experimental setup we have performed iteratively the RPB algorithm with λ taking values in the range $[0 : 1]$, stepping by 0.25. Finally, the last column shows the percentage reduction of the RPB heuristic with respect to the initial number of bit transitions. The percentage in this column is computed by using the minimal number of bit transitions from the five previous columns. The improvements achieved by the RPT algorithm range up to 25%, while the RPB algorithm achieves even higher improvements up to 44% compared to the total bit transitions from the register fields of the original program.

6. CONCLUSION

In this paper, we have presented a compiler driven, power aware register name adjustment algorithm. The *RNA* problem for low-power has been formally defined and its NP-completeness has been shown. Subsequently, we have presented two efficient heuristics for attacking this problem. The achieved significant reductions in bit transitions within the instruction register fields result in reducing the power consumption on the instruction bus and the register file address decoders. The RPT algorithm utilizes register pair frequencies and identifies an efficient register remapping, while the RPB algorithm employs commutativity and dead register reassignment transformations for further reductions in bit transitions. The presented significant power improvements are achieved as a result of a compiler based optimization and consequently necessitate no additional hardware support whatsoever.

The proposed register name adjustment methodology is completely independent of particular instruction set architectures and can be easily integrated within any compilation framework or embedded software synthesis for achieving the ever so important goals of designing power efficient electronic systems.

7. REFERENCES

[1] N. Bellas, I. Hajj and C. Polychronopoulos, "Using dynamic cache management techniques to reduce energy in a high-performance processor", in *ISLPED*, pp. 64–69, August 1999.

[2] M. R. Stan and W. P. Burleson, "Bus-invert coding for low-power I/O", *IEEE TVLSI*, vol. 3, n. 1, pp. 49–58, March 1995.

[3] P. R. Panda and N. D. Dutt, "Low-power memory mapping through reducing address bus activity", *IEEE TVLSI*, vol. 7, n. 3, pp. 309–320, September 1999.

[4] L. Daldos, D. Sciuto and C. Silvano, "State encoding for low power embedded controllers", in *ISCAS*, pp. 421–424, 1998.

[5] W. Noth and R. Kolla, "Spanning tree based state encoding for low power dissipation", in *DATE*, pp. 168–174, 1999.

[6] T. Austin, E. Larson and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling", *IEEE Computer*, vol. 35, n. 2, pp. 59–67, February 2002.

[7] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in *MICRO*, pp. 330–335, December 1997.