Register Allocation for Simultaneous Reduction of Energy and Peak Temperature on Registers

Tiantian Liu
Department of Computer Science
City University of Hong Kong

Alex Orailoglu
Department of Computer
Science and Engineering
University of California, San Diego

Chun Jason Xue and Minming Li
Department of Computer Science
City University of Hong Kong

***Abstract*—In this paper, we focus on register allocation techniques to simultaneously reduce energy consumption and heat buildup of register accesses. The conflict between these two objectives is resolved through the introduction of a hardware rotator. A register allocation algorithm followed by a refinement method is proposed based on the access patterns and the effects of the rotator. Experimental results show that the proposed algorithms obtain notable improvements in energy consumption and temperature reduction for embedded applications.**

***Index Terms*—Register allocation, Bit transition activity, Heat buildup, Rotator**

I. Introduction

Energy and thermal issues are two important concerns for embedded system design. The instruction fetch logic of processors associated with register accesses makes a significant contribution towards energy consumption levels. Meanwhile, the register file has been shown to exhibit the highest temperature compared to the rest of the hardware components in an embedded processor [1]. In this paper, we use register allocation techniques to reduce the energy consumption and heat buildup on register accesses simultaneously.

Register allocation is usually performed at the end of global optimization, with the main focus being restricted to the minimization of the number of memory references. The stringent requirements for energy and thermal issues imposed by embedded systems force us to extend traditional register allocation so as to consider energy and heat requirements while preserving the minimization of memory spills. In most instruction set architecture (ISA) designs, the register fields reside in fixed positions within the instruction, thus forming streams of indices on the instruction bus and to the register file address decoder [2]. The number of bit transitions of these streams greatly affects the energy consumption on the instruction bus and the instruction decoder. A register allocation which minimizes bit transitions will reduce energy consumption. While the use of the same registers frequently and consecutively can reduce bit transitions [2], it leads to thermal buildup for these heavily used registers, thus inducing hotspots on the chip. Register allocation should also balance the access distribution to avoid hotspots [3]. However, this balanced access strategy may cause higher energy consumption.

This work is partially supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 123609, 117408].
978-3-9810801-7-9 / DATE11 / © 2011 EDAA

This paper provides a register allocation technique for a code block of variable accesses which simultaneously helps reduce bit transitions on the address bus while precluding heat buildup in register accesses. These conflicting goals can be resolved with the use of a small hardware, namely a **rotator**, in the instruction decoder. The register indices output by the compiler and transferred on the address bus are logical register indices. When decoding, the rotator will rotate the bit strings of the logical register indices several bits to the right each time to obtain the physical register indices. The rotator can spread out the accesses to distinct physical registers even when using the same logical register without any bit transition. As a result, both energy consumption and peak temperature are reduced. These novel improvements are attained through the following contributions of this paper:

1) The introduction of a rotator with a simple rotating rule to balance objectives of energy and heat buildup minimization.

2) A register allocation algorithm to simultaneously reduce energy consumption and heat buildup, based on a Live Range Successive Access Graph (LRSAG) to model the access patterns of variables and a classification of registers.

3) A refinement method to adjust the allocation for improved goal satisfaction.

Register allocation is an NP-complete problem [4] [5] which is often transformed to a graph coloring problem in a Conflict Graph (CG). The goal is to determine which live ranges should be brought into registers so that the number of loads and stores is minimized. The problem has been analyzed using several paradigms, including bin packing, priority-based graph coloring [6], and bottom-up graph coloring [4] [7]. Recent research work has focused frequently on alternative objectives, for example, the energy problem [8] [9]. **Minimizing transition activities** in transferring data across various parts of digital systems has been a major goal in recent research efforts for low energy. The Bus-Invert method [10] inverts the bits on bus lines when this reduces the Hamming distance. Petrov et al. [2] propose heuristics that utilize register pair frequencies and dead register re-assignment to identify an efficient register re-mapping and reduce bit transition activity. Several researchers investigate the **thermal problem** on registers. Zhou et al. [3] propose a register re-assignment method to uniformly distribute the register accesses to reduce heat buildup. Yang et al. [11] reduce the peak temperature

of registers by a register shuffling strategy. Hsieh et al. [12] propose a register binding method to balance the register temperature considering both spatial and temporal information.

The remainder of this paper is organized as follows. Section II presents some assumptions regarding the register allocation problem and the main strategy used in this paper. Section III outlines the model and objectives of the register allocation problem, and provides a motivational example. Section IV presents a novel register allocation algorithm followed by a refinement method. Section V presents the experimental results, with the paper being concluded in Section VI.

II. Problem Overview

Several assumptions of the graph coloring framework used in this paper are as follows:

1) Unit of allocation: identical to that of Chaitin's method [4]; we use machine-level instructions as the unit of coloring.

2) Use of registers: identical to that of Chaitin's method [4]; we assume all physical registers participate in coloring.

3) Splitting handling: not used.

The main strategy is to introduce a **rotator** within the address decoder. The register indices obtained by the compiler and transferred through the bus are the logical register indices. When decoding, the rotator will rotate their bit strings several bits to the right each time. The physical register indices are thus obtained and used to access the register file. By several well-defined rotation mechanics and register allocation methods, the identical logical register can be used as much as possible to reduce bit toggles on the bus, thus saving energy, while the physical accesses are scattered to different registers, thus avoiding heat buildup.

III. Problem Analysis

A. Register Allocation Framework

The register allocation problem is formulated based on the following input, output and architecture framework.

Input: a code block of n instructions, and a total of 2^k registers. In line with most ISA, we assume each instruction has three operands of variables: *Destination*, *Source1* and *Source2*. Different operand counts can be effectively accommodated with the algorithms proposed. An example of a code block is shown in Fig. 1 (a).

Output: to allocate registers for these variables so that the following two objectives are jointly considered:

Objective 1: The number of bit transitions on the bus is reduced, thus leading to savings in energy consumption.

Objective 2: Access density on registers is distributed evenly, thus avoiding register heat buildup.

The architecture framework is shown in Fig. 1 (c), with a **rotator** added within the decoder to translate logical register indices into physical indices. The complete process is shown in Fig. 1 (a) to (d) by an example with $k = 3, n = 10$. During the compilation stage, the techniques proposed in this paper provide register allocation as shown in Fig. 1 (b) for variables as shown in Fig. 1 (a). The allocation shown has taken into consideration both the energy and thermal objectives and the impact of the introduced **rotator**. The register indices shown

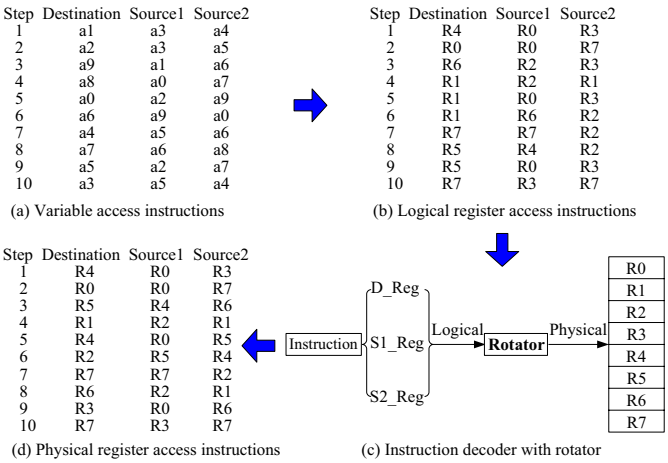

Fig. 1. The process of register allocation and access.

in Fig. 1 (b) represent the logical register indices which are transferred in the address bus. The physical register indices are subsequently extracted from the decoder with a rotator depicted in Fig. 1 (c). The rotator rotates each logical register index of an instruction $(i-1)\%k$ bits to the right, where $1 \leq i \leq n$ is the step index of the instruction and k is the total number of bits in a register index's bit string. The physical register indices, which denote the actual architectural register names, are shown in Fig. 1 (d).

B. Register Allocation Problem

A Conflict Graph (CG) is constructed based on the input instruction schedule, where each node is a live range of a variable. To define a CG, we provide the definition of live range first. We use V to represent the variable space.

Definition 1: **A Live Range** Vlr is a life period of a variable. It has four attributes: $Var(Vlr) = a_x \in V$ denotes the variable that this live range is associated with. $VBegin(Vlr)$ and $VEnd(Vlr)$ are the instruction steps which represent when this live range begins and ends. A live range begins at a new write of the variable $Var(Vlr)$ at $VBegin(Vlr)$, and ends at the last read of it at $VEnd(Vlr)$ before its next write. $VLRcount(Vlr)$ denotes the number of accesses to variable $Var(Vlr)$ within this variable live range Vlr.

For a code block of variable accesses, there may be several live ranges associated with an individual variable.

Definition 2: **Conflict Graph (CG)** is an undirected graph, where each node is a live range and each edge denotes the conflict relationship between nodes. An edge $e(u, v)$ indicates a use conflict between two live ranges. "Conflict" implies that the two live ranges cannot be assigned to the same physical register, because their instruction steps overlap. That is, for two nodes u and v, if

$[VBegin(u), VEnd(u)) \cap [VBegin(v), VEnd(v)) \neq \emptyset$

then the two nodes conflict with each other, thus introducing an edge between them.

The CG for the schedule in Fig. 1 (a) is shown in Fig. 2, where each node v is a live range with its attributes represented as $Var(v)(VBegin(v), VEnd(v)) : VLRcount(v)$. For a variable with some reads before its first write in a code block, we assume the variable is initialized at Step 0 and construct

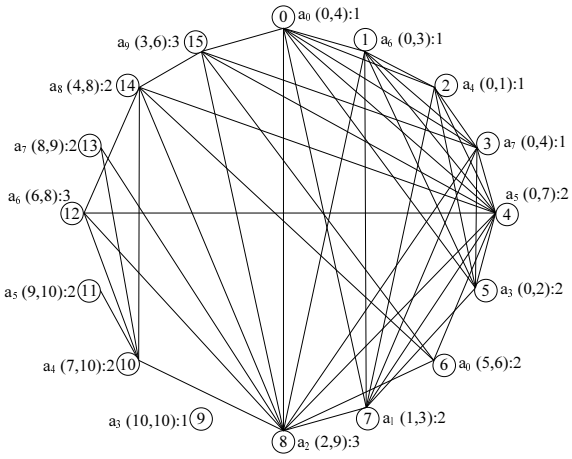

Fig. 2. The CG for the schedule shown in Fig. 1 (a).

a Vlr for it with $VBegin(Vlr) = 0$. The register allocation problem reduces to a graph coloring problem on the CG.

C. Objectives

We use three matrices to represent the instruction schedule. For the code block in Fig. 1 (a), its **Variable Access Matrix (VAMatrix)** $P \in V^{n\times 3}$ is shown in Fig. 3 (a). Each $p_{ij} \in P$ is a variable representing the jth access at instruction step i, where $j = 0$ denotes the *Destination* access, and $j = 1$ and $j = 2$ denote the *Source1* and *Source2* accesses respectively. Fig. 3 (b) shows the **Logical Register Access Matrix (LRAMatrix)** $P' \in R^{n\times 3}$ which corresponds to the logical register indices transferred on the bus, where R denotes the register space. Fig. 3 (c) depicts the **Physical Register Access Matrix (PRAMatrix)** $P'' \in R^{n\times 3}$ which corresponds to the actual register accesses. By using a rotator, each $p''_{ij} \in P''$ is the result of each $p'_{ij} \in P'$ rotating $(i-1)\%k$ bits to the right.

$$\begin{bmatrix}1&3&4\\2&3&5\\9&1&6\\8&0&7\\0&2&9\\6&9&0\\4&5&6\\7&6&8\\5&2&7\\3&5&4\end{bmatrix} \xrightarrow{\text{Allocator}} \begin{bmatrix}4&0&3\\0&0&7\\6&2&3\\1&2&1\\1&0&3\\1&6&2\\7&7&2\\5&4&2\\5&0&3\\7&3&7\end{bmatrix} \xrightarrow{\text{Rotator}} \begin{bmatrix}4&0&3\\0&0&7\\5&4&6\\1&2&1\\4&0&5\\2&5&4\\7&7&2\\6&2&1\\3&0&6\\7&3&7\end{bmatrix}$$

(a) VAMatrix: P (b) LRAMatrix: P' (c) PRAMatrix: P''

Fig. 3. Three matrices of the instruction schedule.

1) Objective 1: Minimize the number of bit transitions: The larger the difference in terms of bit positions in any two consecutive indices is, the higher the energy on the bus and the address decoder will be, as increased bit toggles will occur. The number of bit transitions between two logical registers is the Hamming Distance of the binary bit strings of the two register indices. For two registers R_i and R_j, we use $H(R_i, R_j)$ to denote this Hamming Distance. In a context of eight registers, $R_1(001)$ and $R_7(111)$ would result in $H(R_1, R_7) = 2$. The value of each $H(R_i, R_j)$ can be easily obtained. We would like to allocate the registers to obtain an LRAMatrix P' so that the following expression is minimized.

$$\sum_{j=0}^{2}\sum_{i=1}^{n-1} H(p'_{ij}, p'_{i+1j}) \tag{1}$$

LRAMatrix: P' = PRAMatrix: P" without rotator

$$\begin{bmatrix}6&0&4\\0&0&1\\5&6&5\\2&3&2\\3&0&5\\3&5&3\\4&1&3\\1&3&2\\0&0&1\\0&0&4\end{bmatrix}$$ BT=**42** HB=**41.5**

$$\begin{bmatrix}0&4&7\\5&4&3\\4&0&2\\1&1&6\\0&5&4\\0&4&0\\3&3&0\\1&0&1\\1&5&1\\1&1&3\end{bmatrix}$$ BT=**28** HB=**53.5**

$$\begin{bmatrix}1&2&4\\4&2&5\\3&1&6\\1&0&7\\0&4&3\\6&3&0\\2&5&6\\7&6&1\\5&4&7\\3&5&2\end{bmatrix}$$ BT=**44** HB=**1.5**

(a) Chaitin allocation[7] (b) Allocation for bit transitions[2] (c) Allocation for heat[3]

LRAMatrix: P' PRAMatrix: P"

$$\begin{bmatrix}2&0&3\\0&0&7\\6&1&6\\1&4&1\\1&0&3\\1&6&2\\4&7&2\\5&4&2\\5&0&3\\7&3&4\end{bmatrix} \quad \begin{bmatrix}2&0&3\\0&0&7\\5&2&5\\1&4&1\\4&0&5\\2&5&4\\4&7&2\\6&2&1\\3&0&6\\7&3&4\end{bmatrix}$$ BT=**33** HB=**9.5**

LRAMatrix: P' PRAMatrix: P"

$$\begin{bmatrix}4&0&3\\0&0&7\\6&2&3\\1&2&1\\1&0&3\\1&6&2\\7&7&2\\5&4&2\\5&0&3\\7&3&7\end{bmatrix} \quad \begin{bmatrix}4&0&3\\0&0&7\\5&4&6\\1&2&1\\4&0&5\\2&5&4\\7&7&2\\6&2&1\\3&0&6\\7&3&7\end{bmatrix}$$ BT=**27** HB=**5.5**

(d) The proposed allocation (e) The proposed allocation with refinement

Fig. 4. The motivational example.

This is the summation of the number of bit transitions considering the three operands. For example in Fig. 3 (b), the first column vector representing the logical register indices of the *Destination* forms the sequence <4, 0, 6, 1, 1, 1, 7, 5, 5, 7>, resulting in a total of 10 bit transitions. The second column vector results in 10 bit transitions as well, while the third column vector results in 7 bit transitions. In total, all the register indices are responsible for 27 bit transitions.

2) Objective 2: Balance heat buildup: The simulation results in [3] and [11] have confirmed that evenly distributing the register accesses can effectively reduce the chip-wide peak temperature. We use $Rcount(R_x)$ to indicate the total number of accesses to a physical register R_x. It can be obtained by scanning the PRAMatrix P''. The following expression is used to assess the evenness of an access distribution.

$$\sum_{R_x \in R} (\frac{3\times n}{2^k} - Rcount(R_x))^2 \tag{2}$$

This expression captures the variance of the numbers of accesses to different physical registers, where $\frac{3\times n}{2^k}$ is the expected average number of accesses to each physical register. For heat buildup, it is desirable that Expression (2) be minimized. Other more involved and accurate thermal models, such as those utilized in [3] [12], can be easily accommodated within the framework we propose.

D. Motivational Example

The VAMatrix P in Fig. 3 (a) is used as the code example. Suppose we have eight registers: R_0 to R_7. The register allocation techniques under comparison are: (a) Chaitin-style allocation [7], (b) allocation for bit transition activity [2], (c) allocation for heat buildup [3], (d) the proposed allocation method in Section IV-B, and (e) the proposed allocation algorithm combined with the refinement operations detailed in Section IV-C. The allocation results represented by LRAMatrix P' and PRAMatrix P'' are shown in Fig. 4, where "BT" represents the number of bit transitions and "HB" represents the value of Expression (2).

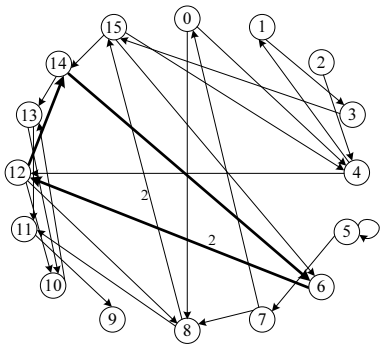

Fig. 5. LRSAG of the schedule in Fig. 1 (a).

From LRAMatrix P', we obtain the numbers of bit transitions. With the proposed allocation method, the number of bit transitions is 33 as shown in Fig. 4 (d). By applying the refinement operations, the number can be reduced to 27 as shown in Fig. 4 (e). The proposed method assigns the same logical register index to those live ranges frequently accessed successively, thus engendering a reduction in bit transitions. For example, in the red boxes in Fig. 4 (d) and (e), within each column, some identical logical registers consecutively repeat in several intervals. The real accesses are spread out to different physical registers via the rotator, which further balances the heat buildup of registers. Techniques in [7], [2], and [3] achieve 42, 28 and 44 toggles, respectively. The values of Expression (2) with the five methods, (a) to (e), are 41.5, 53.5, 1.5, 9.5, and 5.5. Even though the proposed methods deliver higher values than method (c) which optimizes solely for heat buildup, they achieve great improvements compared to the other solutions and have much better bit transition results than (c), thus balancing the two objectives.

IV. Register Allocation

In this section, a graph, named LRSAG, is outlined which embodies the successive access patterns of a code block. Then a register allocation algorithm is proposed based on this graph, the CG and the register rotation properties. Finally, a refinement method is proposed to further improve the solution.

A. LRSAG

Definition 3: **Live Range Successive Access Graph (LRSAG)** is a directed graph, where each node is a live range and each edge denotes the successive access relationship within the same operand between two live ranges. It is obtained by scanning the instructions based on operands from the beginning to the end. For each pair of successive accesses to two variables a_x and a_y, an edge is added from node u to node v where nodes u and v represent the two variable live ranges covering these two accesses to a_x and a_y. The weight is set to $w(e(u,v)) = 1$. If an edge from node u to node v already exists, its weight, $w(e(u,v))$, is increased by one.

The code block in Fig. 1 (a) corresponds to the LRSAG shown in Fig. 5. The index of each node corresponds to the index of each node of CG in Fig. 2. For example, for the first two steps on operand *Source2*, the accesses to a_4 and a_5 are within the live ranges ② and ④, resulting in an edge from node 2 to node 4.

Algorithm 1 Register Grouping

Require: A set of registers: R_0 to R_{2^k-1}.
Ensure: Group the registers into rotating groups.
1: $i = 0; j = 0$; // i, j is the index of a register and a group
2: **for** ($; i < 2^k - 1; i++$) **do**
3: **if** R_i is already contained by a group **then**
4: Continue;
5: **end if**
6: $Group_j$=$\{R_i\}$;R_x=R_i; // a new group forms
7: ***Rotating Step***: Rotate the bit string of R_x 1 bit to the right and obtain a new register index of R'_x;
8: **if** $R'_x \neq R_i$ **then**
9: $Group_j = Group_j \bigcup R'_x$; //a new register in this group
10: $R_x = R'_x$; goto ***Rotating Step***;
11: **else**
12: $j++$; Continue; // need to start a new group
13: **end if**
14: **end for**

The total number of bit transitions can be formulated as

$$\sum_{e(u,v)\in E} H(LR(u), LR(v)) \times w(e(u,v)) \qquad (3)$$

where $LR(v)$ represents the logical register index allocated to v. The register allocation problem for bit transition minimization now reduces to the assignment of nodes to registers so that Expression (3) is minimized.

B. Register Allocation Algorithm

The rotator we propose induces a strict partitioning on the name space of the registers. This strict partitioning decomposes the register space into groups of varying size. Within each group, each register transforms to other members of the group as bits are rotated. The effect can be illustrated by looking at the partitioning of an eight register name space as below.

Group 0: R_0 (000)
Group 1: R_1 R_4 R_2 (001 100 010)
Group 2: R_3 R_5 R_6 (011 101 110)
Group 3: R_7 (111)

For example, in Group 1, R_1 will transform to R_4, R_4 will transform to R_2, and R_2 will transform back to R_1. If we can find a loop in an LRSAG with three nodes and assign a register of Group 1 as the logical register to all of them, no bit transitions will occur in this loop. Furthermore, with the rotation effect, the real accesses will be spread to the three physical registers of Group 1. For example, the three nodes in the loop (6, 12, 14) of Fig. 5 can be assigned a logical index R_1. Then, the logical register sequence of the *Destination* operand from Step 4 to 6 is <1, 1, 1> with no bit transitions, while the physical sequence after rotating each logical index 0, 1, and 2 bits respectively to the right is transformed to <1, 4, 2>. Group 0 or 3 consists of only one register which is self-transformed. If we assign them to the nodes with a self-loop in an LRSAG, for example node 5 in Fig. 5, it will induce no bit transition for that self-loop edge. These groups can be constructed using Algorithm 1.

Algorithm 2 LRSAG Partition

Require: $LRSAG$.
Ensure: Partition LRSAG into clusters.
1: (a) Find nodes with self-loop edges whose weights exceed those of the other edges; order them in a descending order according to their self-loop edges' weights; set each node to be a new cluster and delete it from LRSAG;
2: (b) Find loops with k nodes; order them in a descending order according to their weights; every time set the first loop to be a new cluster, delete it from LRSAG and update the loop list;
3: (c) Find paths with $x = k, ..., 2$ nodes; order them in a descending order according to their weights; every time set the first path to be a new cluster, delete it from LRSAG and update the path list;
4: (d) Set each remaining isolated node to be a new cluster.

Based on the properties of the rotating groups, we partition an LRSAG into clusters using Algorithm 2, where we find loops or paths in a descending order of their weights so that we can assign the registers in one group to them. Then, Algorithm 3 is proposed to allocate registers based on the LRSAG clusters, CG conflicts and register groups.

In Algorithm 3, we try to allocate a register group or part of a register group to each cluster as its physical register indices. We consider the order of registers within one group to further reduce bit transitions. The access number of each register is bounded by a threshold α. For example, by setting the threshold to $\alpha = \frac{3\times n}{2^k} \times 20\%$, we allow a register to be accessed by no more than 20% over the expected average number of accesses. Using this α and Algorithm 3 for the motivational example, a solution is obtained as shown in Fig. 4 (d).

C. Register Allocation Refinement

To deliver further improvements, we utilize two refinement methods: Live Range Migration and Live Range Exchange. After allocation, each variable live range has been assigned a physical register. We use $Reg(Vlr) = R_x \in R$ to represent the physical register that Vlr is allocated to.

Definition 4: **Live Range Migration** denotes a move of a live range to another physical register. By "move", we mean changing $Reg(Vlr) = R_x$ of a Vlr to a physical register R_y, where $R_y \neq R_x$. This can occur only when Vlr does not conflict with any live range whose physical register is R_y.

Definition 5: **Live Range Exchange** denotes an exchange of two live ranges without causing conflicts with each other. By "exchange Vlr_1 with Vlr_2", we mean changing $Reg(Vlr_1) = R_x$ and $Reg(Vlr_2) = R_y$ to $Reg(Vlr_1) = R_y$ and $Reg(Vlr_2) = R_x$. This can occur only when Vlr_1 does not conflict with any other live range whose physical register is R_y, and Vlr_2 does not conflict with any other live range whose physical register is R_x.

A migration or exchange may have distinct impact on the two objectives. Our method attempts feasible migrations or exchanges of the obtained solution. For each possible operation, we calculate its impact on the number of bit transitions and heat buildup as $change\_ratio\_of\_bit\_transitions$ and $change\_ratio\_ of\_heat\_buildup$. These two ratios can be positive or negative, implying increasing or decreasing ratios of Expression (1) and (2), respectively. These two conflicting goals can be integrated into one objective function through a linear combination denoted by β, $0 \leq \beta \leq 1$, in the expression given below in Expression (4).

$$\beta \times change\_ratio\_of\_bit\_transitions \; + \; (1-\beta) \times change\_ratio\_of\_heat\_buildup \quad (4)$$

When a migration or an exchange is tried, we check the sign of Expression (4), as a negative result would imply a decrease in the value of the metric. If so, this refinement operation will be applied, while no change will take place otherwise.

Algorithm 3 Register Allocation

Require: $LRSAG$ with clusters, CG, Group information.
Ensure: Allocate registers according to LRSAG and CG.
1: **for** Each cluster of $LRSAG$ in index order **do**
2: **if** the cluster is a node with self-loop edges **then**
3: Try to use a register group to color it, where the register group has one available register, causes no conflicts and balances the distribution most;
4: **end if**
5: **if** the cluster is a loop or path **then**
6: Try to use a register group G_i to color it, where G_i has more than one available register, causes no conflicts and balances the distribution most; Use an order of the registers in G_i which can minimize the bit transitions with the already allocated nodes;
7: **end if**
8: **if** the cluster is an isolated node **then**
9: Try to use one available register R_i to color it, where R_i balances the distribution most; If there is more than one R_i, use the one which can minimize the bit transitions with the already allocated nodes;
10: **end if**
11: Any time if no feasible solution can be located, find one node to be spilled and try again;
12: After each allocation, update $Rcount(R_x)$. If $Rcount(R_x) - \frac{3\times n}{2^k} > \alpha$, set R_x to be unavailable.
13: **end for**
14: // Use the obtained PRAMatrix P'' to get LRAMatrix.
15: Rotate each element $p''_{ij} \in P''$ $(i-1)\%k$ bits left to obtain LRAMatrix P'.

V. Experimental Results

This section presents the experimental results to illustrate the effectiveness of the proposed algorithms. We use benchmarks from DSPstone and MediaBench [13], including 4-stage lattice filter, 8-stage lattice filter, elliptic filter, differential equation solver, adpcm encoder and adpcm decoder. An Itanium-like processor [14] is simulated with 8 registers. For each benchmark, we use GCC for compilation and obtain its variable access triplets. The techniques being compared are: (a) Chaitin-style allocation of [7] which considers neither of the two objectives; (b) the allocation algorithm of [2] which only minimizes bit transitions for energy; (c) the allocation

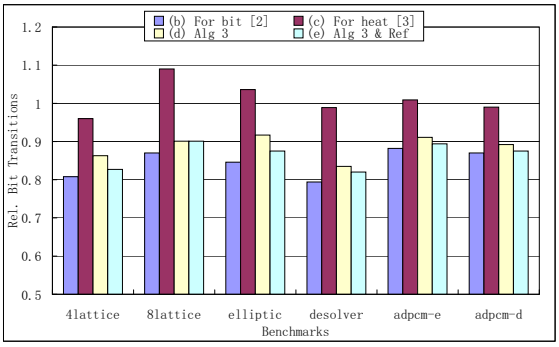

Fig. 6. Comparison of bit transition numbers.

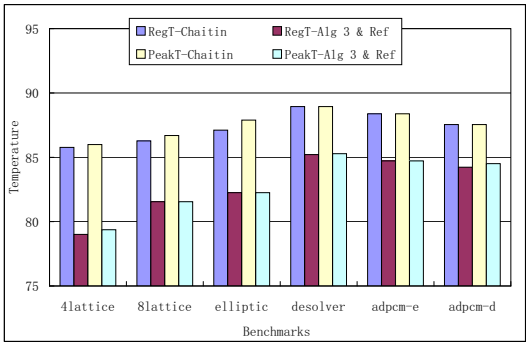

Fig. 7. Temperature Results.

algorithm of [3] which only balances the heat distribution; (d) the proposed allocation algorithm (Algorithm 3) only; and (e) the proposed allocation algorithm combined with the migration-exchange refinement. The input parameters in this set of experiments are given as $\alpha = \frac{3\times n}{2^k} \times 20\%$ and $\beta = 0.5$. The results for bit transitions are shown in Fig. 6, where the X-axis represents the benchmarks and the Y-axis represents the normalized results. The base line of comparison is set to technique (a), i. e., the Chaitin-style allocation of [7].

From Fig. 6, we can see that technique (c), that is the allocation obtained in [3], sometimes performs worse than Chaitin's method (a). The proposed allocation technique (d) achieves about 6%-13% and 7%-15% reduction of bit transitions compared to (a) and (c) respectively. The proposed technique (e) achieves about 8%-17% and 8%-18% reduction respectively, with only about 2%-4% overhead compared to technique (b) [2] which focuses solely on energy reduction.

We evaluate the temperature of registers and the processor chip to show the efficiency of our algorithms on heat. Hotspot [16] is employed to sample the temperature of each hardware resource using the processor [14] as the floorplan. The chip size is set to 8mm $\times$ 8mm, and the initial temperature is set to 60°C. The Chaitin-style allocation algorithm is used as a base line of comparison. In Fig. 7, "RegT-" denotes the peak temperature of the integer registers, while "PeakT-" denotes the peak temperature of the whole chip.

From Fig. 7, we can see that the proposed technique can achieve a reduction of 3.3-6.8°C in register peak temperature. The peak chip-wide temperature is also reduced by about 3.4-6.6°C. Previous studies have shown that the fault rate doubles for every 10°C increase [17], and a large number of delay violations would occur if the peak temperature exceeds 85°C [18]. For most benchmarks, the proposed algorithms can effectively reduce the peak temperature to below 85°C. Therefore, our improvements can effectively reduce the fault rate of the entire chip.

VI. Conclusion

In this paper, we have focused on register allocation techniques to reduce the energy consumption and heat buildup on register accesses. We have introduced a rotator to find a balanced solution for the two conflicting objectives. Together with an LRSAG which represents the successive access relationship of live ranges and register rotating group information, we have proposed a register allocation algorithm. A live range migration-exchange method is furthermore proposed to refine the allocation so as to deliver additional improvements. Experiments confirm that our methods can balance the two objectives while delivering notable improvements.

References

[1] J. Srinivasan, S. Adve, *Predictive dynamic thermal management for multimedia applications*. ICS: 109-120, 2003.

[2] P. Petrov, A. Orailoglu, *Compiler-Based Register Name Adjustment for Low-Power Embedded Processors*. ICCAD03: 523-527, 2003.

[3] X. Zhou, C. Yu, P. Petrov, *Compiler-driven Register Reassignment for Register File Power-density and Temperature Reduction*. DAC08: 750-753, 2008.

[4] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, P. W. Markstein, *Register Allocation via Coloring*. Computer Languages, 6: 47-57, 1981.

[5] F. M. Q. Pereira, J. Palsberg, *Register Allocation after Classical SSA Elimination is NP-complete*. Foundations of Software Science and Computation Structures: 79-93, 2006.

[6] F. C. Chow, J. L. Hennessy, *The Priority-Based Register Allocation*. ACM Transactions on Programming Languages and Systems, 12(4): 501-536, 1990.

[7] G. J. Chaitin, *Register Allocation and Spilling via Graph Coloring*. ACM SIGPLAN Notices, 39(4): 66-74, 1982.

[8] C. H. Gebotys, *Low Energy Memory and Register Allocation using Network Flow*. DAC97: 435-440, 1997.

[9] Y. Zhang, X. Hu, D. Z. Chen, *Efficient Global Register Allocation for Minimizing Energy Consumption*. ACM SIGPLAN Notices, 37(4): 42-53, 2002.

[10] M. R. Stan, W. P. Burleson, *Bus-invert Coding for Low-power I/O*. IEEE TVLSI, 3(1): 49-58, 1995.

[11] C. Yang, A. Orailoglu, *Processor Reliability Enhancement through Compiler-directed Register File Peak Temperature Reduction*. DSN09: 468-477, 2009.

[12] W. Hsieh, T. Hwang, *Thermal-aware Post Compilation for VLIW Architectures*. ASP-DAC09: 606-611, 2009.

[13] C. Lee, M. Potkonjak, W. H. Mangione-Smith, *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems*. In 30th MICRO: 330-335, 1997.

[14] S. Rusu, G. Singer, *The first IA-64 microprocessor*. IEEE Journal of Solid-State Circuits, 35(11): 1539-1544, 2000.

[15] SimpleScalar. http://www.simplescalar.com/.

[16] HotSpot. http://lava.cs.virginia.edu/HotSpot/.

[17] C. J. Lasance, *Thermally driven reliability issues in microelectronic systems: Status-quo and challenges*. Microelectronics Reliability, 43(12): 1969-1974, 2003.

[18] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, D. Tarjan, *Temperature-aware microarchitecture*. 30th ISCA: 2-12, 2003.