SCRF – A Hybrid Register File Architecture

Jer-Yu Hsu, Yan-Zu Wu, Xuan-Yi Lin, and Yeh-Ching Chung

Department of Computer Science, National Tsing Hua University,
Hsinchu, 30013, Taiwan, R.O.C.
{zysheu, ocean}@sslab.cs.nthu.edu.tw, {xylin, ychung}@cs.nthu.edu.tw

Abstract. In VLIW processor design, clustered architecture becomes a popular solution for better hardware efficiency. But the inter-cluster communication (ICC) will cause the execution cycles overhead. In this paper, we propose a shared cluster register file (SCRF) architecture and a SCRF register allocation algorithm to reduce the ICC overhead. The SCRF architecture is a hybrid register file (RF) organization composed of shared RF (SRF) and clustered RFs (CRFs). By putting the frequently used variables that need ICCs on SRF, we can reduce the number of data communication of clusters and thus reduce the ICC overhead. The SCRF register allocation algorithm exploits this architecture feature to perform optimization on ICC reduction and spill codes balancing. The SCRF register allocation algorithm is a heuristic based on graph coloring. To evaluate the performance of the proposed architecture and the SCRF register allocation algorithm, the frequently used two-cluster architecture with and without the SRF scheme are simulated on Trimaran. The simulation results show that the performance of the SCRF architecture is better than that of the clustered RF architecture for all test programs in all measured metrics.

Keywords: VLIW processor, cluster processor architecture, register architecture, register allocation algorithm

1 Introduction

The clustered RF architecture is one of the solutions for this scalability problem of the wide-issue architecture [1,2,3]. In the clustered RF architecture, the functional units and the RF are partitioned into clusters and functional units can only have intra-cluster accessibility to their local RFs. Therefore, the complexity of RFs and bypass network can be reduced significantly.

In this paper, we propose a shared clustered RF architecture, *SCRF*, to reduce the ICC overhead. The *SCRF* is a hybrid architecture by combining the clustered RF architecture and the RF replication scheme. In the *SCRF* architecture, the RF and functional units are divided into clusters. The RF in each cluster contains one shared RF and one local RF. The shared RFs (*SRF*s) act as the replicated RFs in the RF replication scheme and the local RFs (*CRF*s) act as the RFs in the clustered RF architecture. Any one of the ICC models mentioned above can be used as the ICC model of the *SCRF* architecture. When a functional unit in a cluster wants to access data in another cluster, it can access the data through either the ICC or the *SRF* in its

cluster. By putting the frequently used variables that need ICCs on the *SRF*, we can reduce the number of data communication of clusters and thus reduce the ICC overhead. In the clustered RF architecture, some registers will be used for calling convention. These registers are called macro registers [4]. The ICCs generated to access these macro registers cannot be optimized by the clustering algorithm. In the *SCRF* architecture, we can define these macro registers in the *SRF* and the ICC overhead can be reduced a lot.

In the *SCRF* architecture with macro registers in the *SRF*, the execution cycles, the ICC overhead, the spill codes overhead, and the code density can get 11.6%, 55.6%, 52.7%, and 18.2% reduction in average, respectively.

The rest of the paper is organized as follow. In Section 2, we will give brief descriptions for some related research work. Section 3 will describe the *SCRF* architecture and discuss some hardware design issues. In Section 4, we will give the details of the *SCRF* register allocation algorithm. Section 5 will give the experiment evaluation and analysis.

2 Related Work

The clustered RF architecture has advantage in hardware efficiency, but the drawback is the extra ICC overhead [5]. Many new RF organizations have been proposed in the literature to eliminate ICC overhead and remain hardware efficient [1,6,7,8].

Narayanasamy *et al.* propose a clustered superscalar architecture. The RF organization is similar as that of the *SCRF*. But their ICC is decided by hardware, and shared RF is only used for ICC. Zhang *et al.* [9] also proposed a similar *SCRF* architecture. They design a two destination write operation to write registers in shared RF and clustered RF simultaneously. This way can remove the anti-dependency to speed up the software pipelining. But they did not propose any compiler algorithms for this architecture.

Some researchers try to solve the problem of binding variables to clustered *RF*s. In our work, we want to bind variables to *SRF* and *CRFs*. Hiser *et al.* do variables binding before the instruction clustering. The authors proposed a heuristic to do the variables binding for ICC reduction. Terechko *et al.* [10] proposed several global values binding algorithms for the clustered *RFs* architecture. Since the global values are long live range variables, the binding of global values is more important. The authors proposed a feedback-directed two-pass assignment. The assignment does variable binding after initial assignment, clustering and scheduling for accurate ICC estimation. But it dose not take spill pressure into consideration.

3 The Architecture Models

3.1 The Clustered RF Architecture

The clustered RF architecture and its instruction set used in this paper are based on the EPIC processor architecture [4] (see Fig. 1).

In the EPIC processor architecture, each cluster contains an integer unit, a floating unit, a branch unit, a general purpose RF (GPR), a floating point RF (FPR), a branch target RF (BTR), and a memory unit. In this clustered RF architecture, it has the following disadvantages:

- Since the ICC uses communication units to move date between RFs in different clusters, it will lead to several performance overheads such as extra issue slots occupation, the schedule length increasing, and the code size increasing.
- Since the spill codes pressure will aggregate to one cluster under certain situations in the register allocation phase, this imbalance of the spill code pressure will cause more spill codes overhead.

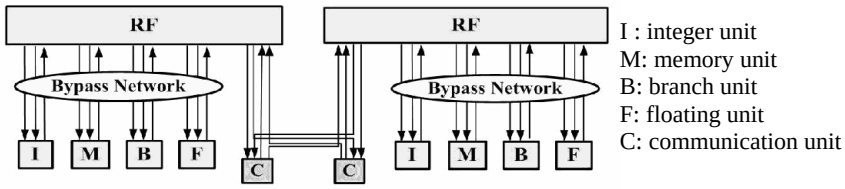

Fig. 1. The clustered RF architecture.

3.2 The *SCRF* Architecture

In the *SCRF* architecture, each cluster contains two types of RF, *SRF* and *CRF* (see Fig. 2). The *SRF* can be accessed by all functional units and the *CRF* can be accessed only by functional units in its local cluster.

The ICC is solved by using communication units to move date between *CRF*s in different clusters. The *SCRF* architecture has the following advantages:

- The *SRF* provides a shared storage to allocate variables. Functional units can access these variables on *SRF* without the ICC. An efficient *SRF* register allocation scheme is needed for variables binding in order to optimize the ICC reduction.
- In addition to the ICC reduction, the balanced spill codes pressure is another advantage of the *SCRF* architecture. In the *SCRF* architecture, we can allocate some variables from high spill codes pressure cluster to the *SRF*. Therefore the unbalanced situation of spill codes pressure can be eliminated.

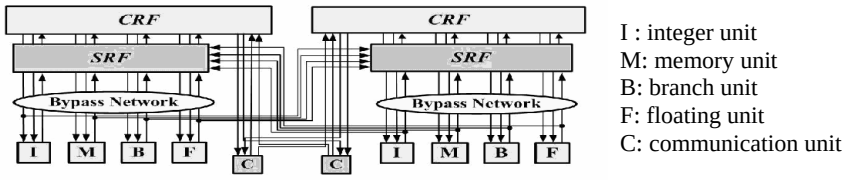

Fig. 2. The *SCRF* architecture.

4 The *SCRF* Register Allocation Algorithm

The notations used in the following sections are listed below:

- *CLR*: the set of all variables.
- *SLR*: the set of meta-variables.
- C_i: the *i*th cluster of the *SCRF* architecture.
- v_i: a variable in *CLR*.
- S_i: a variable in *SLR*.
- cr_{ij}: the *i*th register of the *CRF* in the *j*th cluster.
- sr_i: the *i*th register of the *SRF*.
- $C\_LR(v_i)$: the live range of variable v_i.
- $S\_LR(S_i)$: the live range of variable S_i.
- $icc\_cost(v_i)$: the ICC overhead of variable v_i.
- $icc\_cost(S_i)$: the ICC overhead of variable S_i.
- $sp\_code(v_i)$: the spill code overhead of variable v_i.
- $sp\_code(S_i)$: the spill code overhead of variable S_i.
- $icc\_relation(v_i)$: the set of variables that have an *ICC relation* with v_i.
- $in\_relation(v_i)$: the set of variables that have an *interference relation* with v_i.
- $node(v_i)$: the corresponding vertex of variable v_i in a variable graph.
- $(node(v_i), node(v_j))$: an edge of a variable graph.
- $W(node(v_i))$: the weight associated to vertex $node(v_i)$ in a variable graph.
- $W((node(v_i), node(v_j)))$: the weight associated to edge $(node(v_i), node(v_j))$ in a variable graph.
- $color(cr_{ij})$: the color of register cr_{ij}.
- $color(sr_i)$: the color of register sr_i.
- $color(v_i)$: the color of variable v_i.
- $color(S_i)$: the color of variable S_i.

Fig. 3 shows the compilation flow used in the *SCRF* architecture. In Fig. 3, the clustering phase performs clustering for instructions and variables. In this phase, each variable is assigned to one cluster. If ICC is needed, the clustering algorithm will generate new variables and insert communication instructions for data transfer. The new generated variables are assigned to the demand cluster.

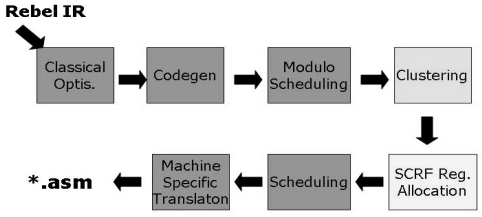

Fig. 3. The compilation flow of the *SCRF* architecture.

Before modeling the variable binding problem, we need the following definitions:

Definition 1: If two variables v_i and v_j in *CLR* are on different clusters and they are used in an ICC code, then $C\_LR(v_i)$ and $C\_LR(v_j)$ have an *ICC relation*.

Definition 2: If $C\_LR(v_i)$ and $C\_LR(j_i)$ have intersections, then $C\_LR(v_i)$ and $C\_LR(j_i)$ have an *interference relation*.

If $C\_LR(v_i)$ and $C\_LR(v_j)$ have an *ICC relation*, the ICC overhead occurred between variables v_i and v_j. If $C\_LR(v_i)$ and $C\_LR(v_j)$ have an *interference relation*, variables v_i and v_j cannot be allocated to the same register.

Definition 3: A variable graph $VG = (V, E)$ is defined as a weighted graph, where $V = \{node(v_i) \mid \forall\ v_i \in CLR\}$, $E = E_{icc} \cup E_{in} = \{(node(v_i), node(v_j)) \mid \forall\ v_i, v_j \in CLR, v_j \in icc\_relation(v_i)\} \cup \{(node(v_x), node(v_y)) \mid \forall\ v_x, v_y \in CLR, v_y \in in\_relation(v_x)\}$, $W(node(v_i)) = sp\_code(v_i)$ for all $v_i \in CLR$, $W((node(v_i), node(v_j))) = icc\_cost(v_i)$ for all $(node(v_i), node(v_j)) \in E_{icc}$, and $W((node(v_x), node(v_y))) = 0$ for all $(node(v_x), node(v_y)) \in E_{in}$.

Given a variable graph *VG* and the colors of *CRFs* and *SRF*, the variable binding problem can be modeled as a graph coloring problem as follows:

Input:

A variable graph $VG = (V, E)$ and the colors of *CRFs* and *SRF*

Constrains:

1. For all $(node(v_x), node(v_y)) \in E_{in}$, $\text{color}(v_x) \neq \text{color}(v_y)$.
2. If v_i is assigned to cluster C_j, v_i can only be colored by the colors of registers in C_j or the colors of registers in *SRF*

Goal:

Based on the constraints, do a graph coloring on VG using the colors of registers in CRFs and SRF such that the following cost function is minimized:

$$Cost(VG) = Cost(V) + Cost(E_{icc}) \quad (1)$$

where $Cost(V) = \sum_{v_i \in CLR} W(node(v_i))$ if v_i is not colored,

and $Cost(E_{icc}) = \sum_{(node(v_i), node(v_j)) \in E_{icc}} W((node(v_i), node(v_j)))$ if

$\text{color}(v_i) \neq \text{color}(v_j)$.

Since the graph coloring problem above is *NP*-complete, we propose a greedy algorithm, the *SRF* register allocation algorithm, to find a suboptimal solution. Given *CLR* and the variable graph *VG* of *CLR*, the proposed algorithm consists of the following phases:

4.1 Phase 1

Since our goals are to reduce the ICC and the spill code overheads, if variables used in ICCs can be binding to *SFR*, the ICCs and the spill code overheads of variables can be eliminated. Therefore, in this phase, we want to construct the set *SLF* from *CLF* by merging variables in *CLF* that have the same ICC relation as a variable in *SLF*. We called the variables in *SLF* as meta-variables. Note that a meta-variable in *SLF* contains at least one variable in *CLF*. The meta-variables are candidates for *SRF* binding. The construct of *SLF* from *CLF* is given as follows:

```
Algorithm SLR_gen(CLR)
1.  Let SLR = ∅;  for (i=1; i< |CLR|; i++) mark[i] = 0;
2.  for (i=1; i< |CLR|; i++) {
3.      if (mark[i] == 0) then {
4.          S_i = {v_i}; S_LR(S) = C_LR(v_i);
5.          for (j=i+1; j<= |CLR|; j++)
6.              if (mark[j] == 0 && (v_j ∈ icc_relation(v_i)) then
7.                  { S_i = S_i ∪ {v_j}; S_LR(S) = S_LR(S) ∪ C_LR(v_j); mark[j] = 1;}
8.          SLR = SLR ∪ S_i;
9.      }
10. }
End_of_SLR_gen
```

4.2 Phase 2

Given a colored or uncolored *VG*, in this phase, the *CRF-coloring* algorithm will color those uncolored vertices in *VG* such that *Cost*(*VG*) in Equation (1) is minimized. This problem is similar to the traditional Chaitin's style graph coloring register allocation problem [11]. In the *CRF-coloring* algorithm, the weights of vertices are used to decide the coloring order of vertices. Initially, all uncolored vertices are sorted as a list, *CAN*, according to their weights in descending order. Then, vertex *node*(v_i) in *CAN* is colored one by one according to the colors of neighbors of *node*(v_i). During the coloring process, a vertex may be colored or uncolored. However, the number of colors used should be the minimum. The *CRF-coloring* algorithm is given as follows:

```
Algorithm CRF_coloring(VG)
1.  Let CAN be a sorted uncolored vertices of VG according to the weights of vertices
    in descending order; /* CAN(k) denotes the kth element in CAN */
2.  for (k=1; k<= |CAN|; k++) {
3.      Let CAN(k) be the corresponding vertex v_i of variable in VG and v_i is a
        variable assign to cluster C_j;
4.      Let neighbor_color(v_i) = {color(v_l)| v_l ∈ in_relation(v_i)};
5.      m = 1; while (color(cr_mj) ∈ neighbor_color(v_i)) m++;
6.      if (m <= the number of register of CRF in C_j)
7.          then color(node(v_i)) = color(cr_mj)
8.  }
End_of_CRF_coloring
```

4.3 Phase 3

Given the colored *VG* obtained in the Phase 2, in this phase, we want to find the variable S_i in *SLF* such that the binding of S_i to *SRF* can lead to a maximum gain of the ICC and the spill code overheads reduction. After binding S_i to *SRF*, we change the colors of vertices in *VG* that corresponding to variables in S_i to the color of S_i, that is, for each variable v_i in S_i, set *color*(v_i) = *color*(S_i). Since the change of variable colors will release some colors for other uncolored variables, we call algorithm *CRF_coloring* to color those uncolored variables. Then, we continue the binding of

variables in *SLF* to *SRF* process until all variables in *SLR* are binding to *SRF* or no variable in *SLR* can be binding to *SRF*. The following algorithm performs the tasks mentioned above.

Algorithm SRF_coloring(*VG*, *SLF*)

```
1.  n = 1; for (k=1; k<= |SLF|; k++) mark[k] = 0; VG2 = VG;
2.  do {
3.      gain = 0; i = 0; VG1 = VG;
4.      for (k=1; k<= |SLF|; k++)
5.          if (mark[k] == 0) then {
6.              call color_Sk(VG, Sk);
7.              temp = Cost(VG1) – Cost(VG);
8.              if (temp > gain) then { gain = temp; VG2 = VG; i = k;}
9.              VG = VG1;
10.         }
11.     VG = VG2; n++; mark[i] = 1;
12. } until (n > |SLF|)
```

End_of_SRF_coloring

*Algorithm color_*S_k(*VG*, S_i)

1. $neighbor\_color(S_i) = \bigcup_{v_j \in S_i} \{color(v_l) | v_l \in in\_relation(v_j)\}$;
2. m=1;**while**($color(sr_m) \in neighbor\_color(S_i)$) m++;
3. **if** (m <= the number of register of *SRF*) **then** {
4. $color(S_i) = color(sr_m)$;
5. $\forall v_j \in S_i$, $color(v_j) = color(S_i)$;
6. **call** *CRF_coloring*(*VG*);
7. **}**

*End_of_color_*S_k

4.4 Phase 4

The coloring process will be terminated as one of the following conditions is satisfied;

- All variables in *SLR* are binding to *SRF*.
- No variable in *SLR* can be binding to *SRF*.

If the first case is satisfied, all variables can be binding to *SRF*, the ICC and the spill codes overheads of variables can be eliminated. For the second case, if all variables are colored, the spill code overheads of variables can be eliminated. Otherwise, some variables will be spilled to memory. The corresponding spill codes, callee-saved codes, and caller-saved codes will be generated.

4.5 An Example to Illustrate the SCRF Register Allocation

In the *SCRF* register allocation, we calculate the *CLR*, interference relations and ICC relations as the input data. In this example, these input data is listed as follow.

- CLR = { v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11} }
- Interference relations = {(v_0, v_1), (v_0, v_2), (v_0, v_5), (v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_6), (v_2, v_4), (v_3, v_4), (v_4, v_5), (v_5, v_7), (v_6, v_7), (v_8, v_9), (v_8, v_{10}), (v_8, v_{11}), (v_{10}, v_{11})}
- ICC relations = {(v_0, v_8), (v_3, v_9), (v_5, v_8), (v_7, v_{10})}

According to **Definition 3,** *VG* can be build as the Fig. 4. In Fig. 4, Fig. 5 and Fig. 6, the interference relations are show as the solid lines and the ICC relations are show as dash lines.

The goal of the *SCRF* register allocation is to minimize Equation (1). The following will show the state transition of this example in these four phases of the *SCRF* register allocation.

In phase 1, *SLR* is computed from *CLR* by merging variables in *CLR* based on the *ICC* relations of variables in *CLR*. Variables in *SLR* are candidates that can be binding to *SRF*. The following is the computed result by *SLR_gen* algorithm.

$SLR = \{ S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7\}$,
$S_0 = \{ v_0, v_5, v_8\}$, $S_1 = \{ v_1\}$, $S_2 = \{ v_2\}$, $S_3 = \{ v_3, v_9\}$,
$S_4 = \{ v_4\}$, $S_5 = \{ v_6\}$, $S_6 = \{ v_7, v_{10}\}$, $S_7 = \{ v_{11}\}$

In phase 2, the *CRF-coloring* algorithm is performed to color *VG* such that the *Cost*(*VG*) in Equation (1) is minimized. In this example, CRF_i denotes the *CRF* in cluster *i* and cr_i denotes the register *i* in *CRF*. In *SRF*, sr_i denotes the register *i* in *SRF*. In Fig. 5 and Fig. 6, if one node is colored by cr_i or sr_i, we color the node as green or blue respectively and mark the number *i* inside this node. In this example, there are two registers in each CRF_i and one register in *SRF*.

$CRF_0 = \{ cr_0, cr_1 \}$, $CRF_1 = \{ cr_2, cr_3 \}$, $SRF = \{ sr_0 \}$

After the *CRF-coloring* algorithm, the state of *VG* is show as Fig. 5.

In phase 3, the *SRF-coloring* algorithm is performed to bind variables in *SLF* to *SRF* such that the *Cost*(*VG*) is minimized. The *SRF-coloring* algorithm will terminate when all variables in *SLR* are binding to *SRF* or no variable in *SLR* can be binding to *SRF*.

In *SRF-coloring* algorithm, the variable S_{max} in *SLR* is found such that the binding of S_{max} to *SRF* can lead to a maximum gain based on the current colored *VG*. Each gain of S_i in *SLR* is calculated as follow and S_0 is found as S_{max}.

gain(S_0) = 155, gain(S_1) = 22, gain(S_2) = 34, gain(S_3) = 101,
gain(S_4) = 78, gain(S_5) = 0, gain(S_6) = 89, gain(S_7) = 25.
=> $S_{max} = S_0$

After the *SRF-coloring* algorithm, the state of *VG* is show as Fig. 6.

In phase 4, the termination condition of *SRF-coloring* is checked. If one of the termination conditions is occurred, the *SRF-coloring* is finished. Otherwise, *SRF-*

coloring is repeated until one of the termination conditions is occurred. In Fig. 6, the termination condition is occurred in no variable in *SLR* can be binding to *SRF*. The following lists each S_i state in Fig. 6.

S_0:colored, S_1:un-colorable, S_2: un-colorable, S_3: un-colorable,
S_4: un-colorable, S_5:gain<0, S_6: un-colorable, S_7: un-colorable.
=> No more variable in *SLR* binding to *SRF* is possible

After *SRF-coloring*, the un-colored node variables are spilled to memory. In this example, v_4 is spilled to memory and the corresponding spill codes are generated.

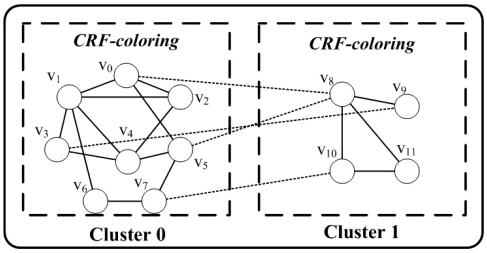

Fig. 4. The initial graph for *SCRF* register allocation.

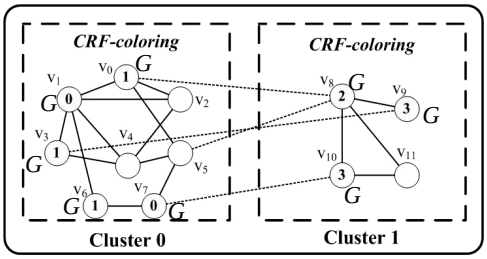

Fig. 5. The state of *VG* after *CRF-coloring*.

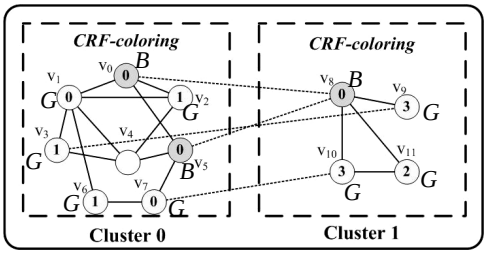

Fig. 6. The state of *VG* after *SRF-coloring*.

4.6 Macro Register Allocation

In the *SCRF* architecture, if the macro registers are defined in *CRFs*, variable v_i used for the dedicated functionalities cannot be merged with variables in *icc_relation*(v_i)

and the improvement of the ICC overhead reduction and the spill code balance is constrained. If we define some frequent used macro registers in *SRF*, the ICC and the spill code overheads of variables binding to these macro registers are eliminated. In the *SCRF* register allocation phase, we can define some frequent used macro registers in *SRF* and bind variables used for the dedicated functionalities to their macro registers before performing algorithm *SLR_gen*. From the simulation results, we can see that the proposed register allocation algorithm with macro register defined in *SRF* has better performance than that with macro register defined in *CRFs*.

5 Performance Comparisons

To evaluate the proposed the *SCRF* register allocation algorithm, we have implemented the *SCRF* architecture shown in Fig. 3 and the *SCRF* register allocation algorithm on a compiler framework, Trimaran [12], from CCCP project [13] along with the clustered RF architecture shown in Fig. 2. We use a set of multimedia research benchmarks, MediaBench [14], as test programs. The BUG [15] is used as the clustering algorithm for instructions and variables. We compare the performance of the benchmarks in terms of the execution cycles, the ICC overhead, the spill codes overhead, and the code density for the clustered RF architecture and the *SCRF* architecture under different architecture parameters. Table 1 shows the settings of these architecture parameters used in the performance evaluation. In Table 1, the *CRF* size field indicates the number of registers in a *CRF* (GPR, FPR, BTR). The *SRF* size field indicates the number of registers in the *SRF*. The Macro Reg. field dedicates what RF is the frequent used macro registers.

Fig. 7 to Fig. 10 shows the simulation results of execution cycles, ICC overhead, spill code overhead and code size, respectively.

Table 1. The architecture parameter settings.

Architecture	*CRF* size	*SRF* size	Macro Reg.
CRF{16, 16}	16	0	*CRF*
CRF{20, 20}	20	0	*CRF*
SCRF{16, 16, 8}	16	8	*CRF*
SCRF_m{16, 16, 8}	16	8	*SRF*

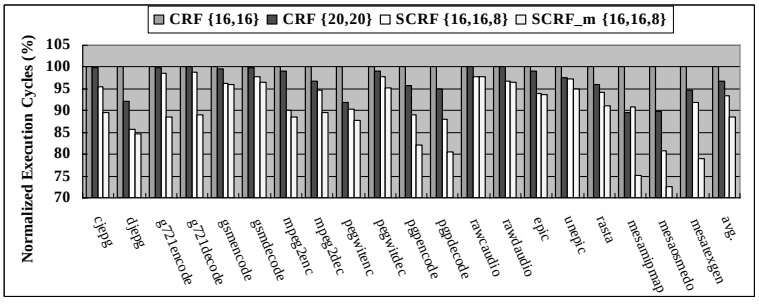

Fig. 7. The execution cycles benchmark result.

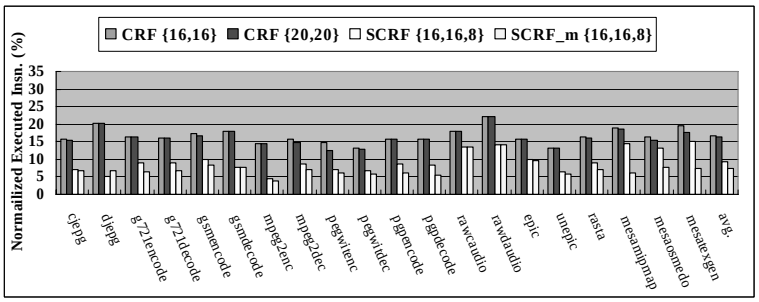

Fig. 8. The ICC overhead benchmark result.

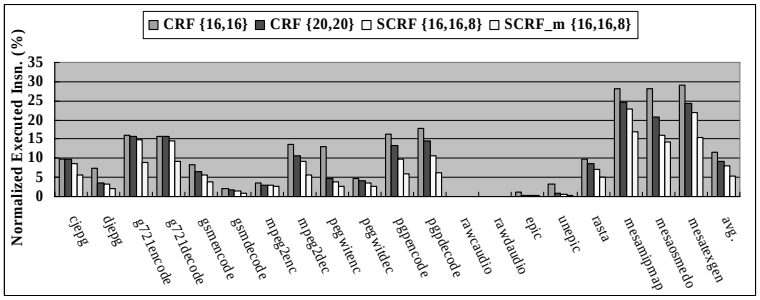

Fig. 9. The spill code overhead benchmark result.

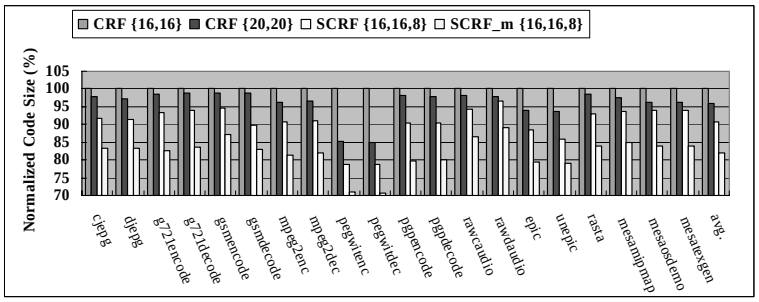

Fig. 10. The code size benchmark result.

6 Conclusions

In this paper, we propose the *SCRF* architecture and the *SCRF* register allocation algorithm. We use replication techniques in *SRF* for hardware efficiency. The *SCRF* register allocation algorithm is a heuristic and priority based algorithm. It not only considers the ICC reduction but also the spill code pressure balance. To evaluate the proposed the *SCRF* register allocation algorithm, we have implemented the *SCRF* architecture and the *SCRF* register allocation algorithm on a compiler framework, Trimaran, along with the clustered RF architecture. A set of multimedia research benchmarks, MediaBench, is used as test programs. We compare the performance of

the benchmarks in terms of the execution cycles, the ICC overhead, the spill codes overhead, and the code density for the clustered RF architecture and the *SCRF* architecture under different architecture parameters. The simulation results show that the performance of the *SCRF* architecture is better than that of the clustered RF architecture for all test programs. In the *SCRF* architecture with specific registers in the *SRF*, the execution cycles, the ICC overhead, the spill codes overhead, and the code density can get 11.6%, 55.6%, 52.7%, and 18.2% reduction in average, respectively.

References

1. Aleta, A., Condina, J. M., Gonzalez, A., Kaeli, D.: Removing Communications in Clustered Microarchitectures through Instruction Replication. ACM Trans. Arch. and Code Opt. 1 (2004) 127–151
2. Gibert, E., Sanchez, J., Gonzalez, A.: Distributed Data Cache Designs for Clustered VLIW Processors. IEEE Trans. Computers 54 (2005) 1227–1241
3. Parcerisa, J.M., Sahuquillo, J., Gonzalez, A., Duato, J.: On-chip Interconnects and Instruction Steering Schemes for Clustered Microarchitectures. IEEE Trans. Parallel and Distributed Systems 16 (2005) 130–144
4. Terechko, A., Garg, M., Corporaal, H.: Evaluation of Speed and Area of Clustered VLIW Processors. In Proc. 18th Int. Conf. VLSI Design (2005) 557–563
5. Gangwar, A., Balakrishnan, M., Kumar, A.: Impact of Inter-cluster Communication Mechanisms on ILP in Clustered VLIW Architectures. In 2nd Workshop on Application Specific Processors, in conj. 36th IEEE/ACM Annual Int. Symp. Microarchitecture (2003)
6. Lin, Y.-C., You, Y.-P., Lee, J.-K.: Register Allocation for VLIW DSP Processors with Irregular Register Files. In Proc. Compilers for Parallel Computers (2006) 45–59
7. Nagpal, R., Srikant, Y.N.: Integrated Temporal and Spatial Scheduling for Extended Operand Clustered VLIW Processors. In Proc. 1st Conf. Computing Frontiers (2004) 457–470.
8. Zalamea, J., Llosa, J., Ayguade, E., Valero, M.: Hierarchical Clustered Register File Organization for VLIW Processors. In Proc. 17th Int. Symp. Parallel and Distributed Processing (2003) 77.1
9. Zhang, Y., He, H., Sun, Y.: A New Register File Access Architecture for Software Pipelining in VLIW Processors. In Proc. Conf. Asia and South Pacific Design Automation 1 (2005) 627–630
10. Terechko, A., Le Thenaff, E., Corporaal, H.: Cluster Assignment of Global Values for Clustered VLIW Processors. In Proc. Int. Conf. Compilers, Architecture and Synthesis for Embedded Systems (2003) 32–40
11. Chaitin, G. J.: Register allocation and spilling via graph coloring. In Proc. ACM SIGPLAN Symp. Compiler Construction. (1982) 98–105
12. Trimaran Consortium: The Trimaran Compiler Infrastructure. (1998) http://www.trimaran.org
13. CCCP research group: Compilers Creating Custom Processors. http://cccp.eecs.umich.edu
14. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In Proc. 30th ACM/IEEE Int. Symp. Microarchitecture (1997) 330–350
15. Ellis, J.: Bulldog: A Compiler for VLIW Architectures. MIT Press, MA (1985)