Register File Customization for Embedded Multi-Threaded Pipelined Processors

Ran Zhang, Hui Guo, and Shivam Garg

***Abstract*—Multi-threading pipelined processor design enables high performance of a single processor core by exploiting both thread-level and instruction-level parallelism. However, when it is applied to embedded systems, its demanding for large register file, hence high resource overhead and energy consumption, becomes a big issue. In this paper, we propose a customization approach to reduce register file by maximally utilizing each of the registers used. The experiments on several applications demonstrate that with our approach, the register file for a single thread can be reduced by about 50%, based on which the multi-threading processor can achieve high performance while at low energy consumption - with 17-28% throughput improvement and 20% energy reduction when compared to the single-threading processor design.**

***Index Terms*—Register file customization, multithread pipeline, energy-aware design.**

I. Introduction

Pipelined processor design is an effective solution to exploit instruction-level parallelism. However, the pipelined processor usually cannot operate at its full speed due to instruction dependency in the pipeline. When a hazard happens, the pipeline has to be stalled until the dependency is resolved, which degrade the performance. Multi-threading processor design [1] improves the execution efficiency of the processor pipeline by exploiting thread level parallelism, and has been applied to high-end network processor systems.

With the multi-threading design, the threads are executed by the pipeline in an interleaved fashion. For each clock, the switching unit controls to fetch an instruction from a different thread. When the next instruction in a thread is executed, its preceding instruction has completed and exited from the pipeline. Therefore, there is no hazard, hence no pipeline stall is incurred and the maximum throughput (i.e., one instruction per clock cycle) can be achieved.

One issue with the design, however, is that the register file should be enlarged and must be big enough to hold the data for different threads, leading to high hardware cost and energy consumption, which may nullify or outweigh the gain from the multi-thread processing.

In this paper, we propose two customization techniques to reduce the register file size required by the multiple threading designs. The rest of the paper is organized as follows. Section 2 reviews existing works related to register file customization. Our customization techniques to reduce register file are explained in Section 3. Section 4 presents the experimental setup and simulation results. Finally, Section 5 draws a conclusion on the proposed approach.

Manuscript received October 9, 2012; revised December 18, 2012.

The authors are with the School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia (e-mail: {ranz, huig, shivam.garg}@cse.unsw.edu.au).

II. Related Work

A majority of the existing works on register file design are related to power/energy reduction.

Hu et al. [2] found that most register accesses occurred within a few cycles after the value was first produced. Based on this observation, they used a Value Aging Buffer (VAB) to store short-lived register values between Functional Units and Register File. When a functional unit reads a register, it first searches VAB; only when this register value is not in VAB, does it then access the register file. This exploitation of the short lived values, to reduce register accesses, can also be found in [3]-[4].

Rixner et al [5] investigated a set of different register architectures for media application processing, where a large number of arithmetic units are used to achieve high parallel operations. The architecture is formed by partitioning a conventional central register, based on the data/instruction level parallelism and memory hierarchy designs. The partitioned register architecture reduces the register file area, delay and power consumption with a small performance degradation, as compared to the centralized global register file where the area, delay and power dissipation are exponentially proportional to the number of arithmetic units.

Gonzalez et al. [6] proposed a content aware integer register file structure, where a traditional register file is partitioned into three small bit-portions. Their design is based on the belief that the access to register file has high temporal locality and the accessed data exhibit a high degree of similarity in their high-bit values. They utilized such data patterns in the register file design so that each register access only activates as small register portion as possible, hence the register file power consumption is reduced. Similarly, Nalluri et al. [7] proposed multi-bank register file architecture. They found that most of the register accesses occurred to the smaller bank. As the smaller bank has a relatively smaller bit-line switching capacitance, a significant reduction in the overall power consumption can be achieved.

Guan and Fei [8] partition the register file into two regions. The most frequently used registers are in one region, and the rarely accessed registers are in another region. They use a partitioning algorithm and register renaming to enhance the usage of the partitioned register file to reduce the register file power consumption.

In [9], authors looked into the issue of uneven power density distribution issue in the processor. They proposed a compiler-based register reassignment methodology to break such clusters and to uniformly distribute the accesses to the register file.

In terms of register file reduction, a few approaches exist. One related approach is based on register sharing [10]-[11]. Balakrishnan et al [10] found that a considerable number of register accesses were reading and writing some common values such as 1 and 0. They use a set of registers to store such common constant values. When a result of instruction is generated, it is compared with the values in the shared registers. If the result equals one of the common values, the destination logical register is re-mapped (on a hardware level) to the common value register.

The method in [12] reduces the register file size through instruction pre-execution and the register value prediction. The approach proposed in [13] enhances register utilization for the superscalar processors by allocating registers later and releasing them earlier than conventional schemes.

Zhou *et al*. in [14] proposed a hardware-level register file customization approach, where profiling is used to detect the unused registers and those unused registers are removed. The hardware customization is transparent to the software code.

The Register File Reduction process provided in this paper can be applied on top of the existing customization approaches to push the register usage for a thread execution to a minimum so that the growing register file size for multi-threading processor can be tamed.

III. Register File Customization

For a given thread (application), its register usage can be profiled. Based on the profiling information, we can build a time diagram to represent the engagement of each register, as shown in Fig. 1(a). The time diagram consists of a sequence of used sections during which the register is used. Each used section begins with a write operation (writing a value to the register) and ends with the last read operation for the register value. Some registers may be highly used, while others may be less used. Here, we define the usage of a register as the ratio of its total used sections over the whole execution time. The registers with zero usage are unused and can be removed from the register file.

Our approach to reducing the register file size is to merge registers so that each used register is maximally utilized and the registers required can be abated, which is detailed below.

A. *Register Merge*

Register merge reduces registers by shifting out the used-sections of less used registers to other registers, and renaming the original registers used in the code (instruction memory) to the target registers.

The used sections of different registers can be overlapped. Take Fig. 1 as an example. There are four registers *R*1-*R*4. Their used sections are given in Fig. 1(a). Between these registers, there exist overlapped used sections, as highlighted by the shaded areas. For example, the first shaded area shows the period during which both registers *R*1 and *R*2 are used. There is a largest register set where all registers have at least one common used-section. We call such register set, maximum overlapped register set. With the example given in Fig. 1(a), the maximum overlapped register set is {*R*1, *R*3, and *R*4}.

The overlapping of used sections prevents the merge of related registers. Therefore, we have the low bound for the number of required registers: the minimum number of registers required must not be less than the size of the maximum overlapped register set. For the given example, since the size of the maximum overlapped register set is three, at most one register can be reduced.

We want to reduce the less used registers by eliminating their used sections, which leads to our first register reduction method specified in Algorithm 1.

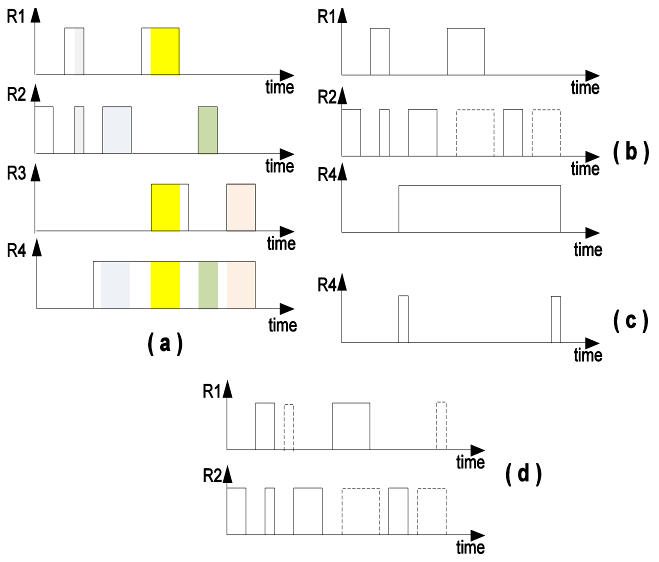

Fig. 1. Example of register file reduction.

The algorithm first finds the lower bound of the registers required and recursively reduce the less used registers. In the algorithm, set *S* holds the sorted registers generated from function Sort Reg1, which sorts the registers in *S* according to their usage *U*. The not-used-registers with no used sections in *S* are removed from *S* and stored in *S0* by function Not Used Regs (*S*). Function get Max Over Lapped Reg Set (*T*1, *S*) finds the maximum overlapped register set in *S* and returns the size of the overlapped register set, and function Most Used Regs (*S*, *m*) gets the m most used registers in register set *S*. Function Register Renaming (*r*, *S*1, temp*T*) attempts to shift all used sections of register *r* to registers in *S1*. For a used section shifted from register *r* to register *q*, the register *r* in the related instructions will be renamed as register *q*, which also leads to the change to the transcript temp *T*.

Continue with the example shown Fig. 1(a). With Algorithm 1, the four registers can be reduced to three registers as shown in Fig. 1(b), where the used section of register *R3* are shifted to *R2* as shown in dashed blocks and *R3* is renamed as *R2* in the program code.

B. *Enhancement with Memory Replacement*

We can further reduce registers by saving infrequently accessed register data in memory rather than leaving the data "sleeping" in registers. For example, *R4* holds a value that is read after a long period. The value can be transferred to the memory right after it is generated by the processor. The value is fetched back to the register only when it is used. The register is therefore can be freed for most of the time, as shown in Fig. 1(c). The reduced used-sections of register *R4* can now be shifted to registers *R1*. The four registers are, therefore, reduced to two, as shown in Fig. 1(d).

Algorithm 1 Register Reduction by Register Merge

```
/*RegMerging(R, T): merging registers by register renaming.
    Input:  R, register set in the register file, and
            T, the register usage transcript.
    Output: R1, the reduced register set, and
            T1, the usage transcript of all registers in R1. */
 /* calculating the usage of each register, U; */
   for (each register, r∈R)} do
     U(r)=getUsage(T,r)
   end for
   S = R; /* register set to be reduced */
   S = SortReg1(S, U); /* registers in R are sorted based on usage. */
/* put the registers that are not used to S0; */
   S0 = NotUsedRegs(S);
   S=S-S0; /* All registers initially used. */
    n=|S|; /* the number of registers in S */
/* Get the usage transcript for registers in S from T. */
    tempT=getUsageTranscript(T, S);
/* find the size of the maximum overlapped register set in S, m */
    m=getMaxOverlappedRegSet(tempT, S);
/* put m most used registers in S1, */
   S1 = MostUsedRegs(S, m);
/* Those registers are part of the resulting register set retained */
   R1 = S1;
/* Reduce the rest registers as much as possible */
    while (m != n) do
        /* put the less used registers in S to S2. */
        S2 = S-S1;
        for (each register, r, r∈S2, from the least used register) do
        /* try to shift its used sections to registers in $S1$ by register renaming
        */
        RegisterRenaming(r, S1, tempT);
        /* and recalculate its usage. */
       U(r) = getUsage(tempT, r);
        /* If all its used sections are removed, the register is freed */
            if (U(r)==0) then
                /* the register is removed from S2 and put into S0. */
                S0 <= {r};
                S2 = S2 - {r};
            end if;
         end for;
        /* next round reduction for registers in S2*/
       S = S2;
    n=|S|;
       S=SortReg1(S, U);
       m=getMaxOverLappedRegSet(tempT, S);
       S1 = MostUsedRegs(S, m);
       R1 <= S1; /* registers in S1 are added to the final register set */
    end while
    R1 <= S2;
    T1 = getUsageTranscript(tempt, R1);
```

However, using memory to save the register usage will incur overhead on instruction memory size and performance. With the memory replacement, for each register read/write, one memory read/write instruction should be inserted. For a used section, if there are *k* instructions accessing the register, *k* memory-access instructions should be inserted in the execution, thus increasing the instruction memory size and execution time. Take Fig. 2(a) as an example. There are one register write and two register read instructions, hence three memory access instructions will be inserted, as demonstrated in Fig. 2(b).

Moreover, since memory access instructions may take longer time than register-access instructions, the pipeline will often stall waiting for the memory data available, namely, there is a load delay. For the example shown in Fig. 2(c), if memory access takes two extra clock cycles, the performance overhead is seven clock cycles.

To reduce the overhead of the memory replacement, we save the less frequently-assessed register value in the on-chip scratchpad memory. For each scratch memory access, there is a small and fixed latency.

To further minimize the impact of the memory latency on the overall execution performance, we want to schedule the load instruction (load from the scratchpad memory) in such a way that the memory value arrives to the processor right before it is used. Assume the scratchpad memory access latency is *d* clock cycles and each instruction takes one clock cycle to finish (i.e., the throughput of the pipeline). To avoid the processor stall due to the memory access, we want the load instruction to be placed *d* instructions ahead of the instruction that uses the memory data.

An example of inserting load instruction is given in Fig. 3. The Fig shows a code section with three basic blocks: *B(i), B(j), B(k)*. (A basic block contains a sequence of instructions. If one instruction in the basic block is executed, all its other instructions in the block should be also executed.) In *B(i)*, instruction *I*1 calculates a+b and the result c is saved in register r3 which will be used by instruction *I*4 in block *B(k)*. To reduce the usage of register r3, its value is saved to memory by instruction *I*2 after it is generated and is fetched back from the memory by instruction *I*3 before instruction *I*4. Assume each instruction takes one clock cycle to execute. If memory access latency is d clock cycles, we want ideally to place the load instruction d instructions earlier so that the long memory access time can be hidden. But the location where the load instruction to be inserted can fall into a conditional block, like *B(j)*, which is not always executed.

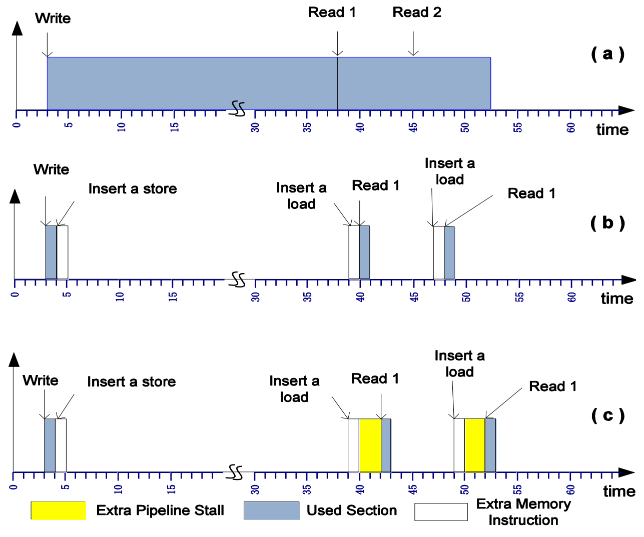

Fig. 2. Memory replacement: (a) initial used section (b) after memory replacement (c) after memory replacement (with extra access time)

Therefore, we only consider scheduling for the load instruction in the same block it is used; namely, both the load and the related register-read instructions are in the same basic block. Assume after scheduling, the distance of the two instructions is ζ. The performance overhead of the memory delay is $d - \zeta$ ($\zeta <= d$). The register merge enhanced with the memory replacement is specified in Algorithm 2.

In Algorithm 2, the access rate of a register is its access times over the total number of instructions executed. Function lessAccessedRegs(*S1*, α) finds a set of less accessed registers from the sorted register set *S1* such that the sum of access rates of the less accessed registers is close to but not larger than the parameter α. The value of α can be tuned for a better tradeoff between the register saving and the overhead on performance and instruction memory. Larger α value allows for more chances of register reduction but possibly incurs more instruction memory and performance overhead.

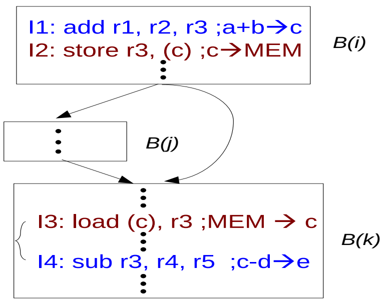
I1: add r1, r2, r3 ;a+b→c
I2: store r3, (c) ;c→MEM
B(i)
B(j)
I3: load (c), r3 ;MEM → c
I4: sub r3, r4, r5 ;c-d→e
B(k)

Fig. 3. Load instruction insertion example

Algorithm 2 Register Reduction with Memory Replacement

```
/*RegRedWithMemoryReplacement(S1,T1):Reducing registers in S1 by memory replacement.*/
Input: S1, used-register set to be applied by memory replacement
        T1, the register usage transcript of S1
        d, memory access time (load stall)
        α, the threshold of register access rate.
Output: S2: the reduced register set,
        T2: the usage transcript of registers in S2
        InstructionMemOH: the memory overhead, and
        PerfOH: the performance overhead.
/*Initialization:*/
InstructionMemOH=0;
PerfOH=0;
/* calculate the access rate of each register, A */
for (each register, r ∈S1) do
    A(r)=getAccessRate(T1,r)
end for;
S2 = SortReg2(S1, A); /* registers in S1 are sorted based
    on access rate */
/* Put less accessed registers in S3 */
S3 = lessAccessedRegs(S2, α);
 /* using memory replacement for each register in S3 */
for (each register, r, r∈S3), from the least accessed register do
     (I\_load, I\_store, T2) =usedSectionReduction(r,T1,d);
end for
/* Merging the registers in S2 by using Algorithm 1 */
(S3, T3)=RegMerging(S2, T2);
/* if there is saving, */
if |S3|<|S2| then
    /* scheduling inserted load instructions to reduce load delay*/
    T4=basicBlockScheduling(T3);
    /* calculate the instruction memory and performance overhead */
    InstructionMemOH=getMemoryOverhead(T2, T4);
    PerfOH=getPerfOverhead(T2, T4);
 /* save the result */
    S2=S3;
    T2=T4;
end if;
```

IV. Experimental Setup and Simulation Results

Fig. 4 shows the experimental setup. It consists of three parts: processor design for single thread execution, processor design for multi-thread execution, and the design evaluation. All processor designs are based on the PISA instruction set architecture. Without loss of generality, for multi-thread execution, we simply replicate a given application multiple times.

The experiment starts with the base pipeline processor for single thread. The PISA instruction set is implemented into a pipeline processor of six stages (two memory access stages). In the base processor design, there are 32 registers in the register file. ASIPMeister [15] is used to generate the VHDL model for the processor.

Given an application written in C, it is compiled with the Simplescalar [16] GCC. The execution of the application on the processor model is simulated with Modelsim [17], and its results are compared with the Simplescalar simulation results for functional verification of the processor design. The execution trace of the Modelsim is used in profiling the usage of registers for register file reduction, based on which the multi-threading processor design, and the related execution code are generated.

ModelSim and Synopsys Design Compiler [18] are used for design evaluation, both taking the processor VHDL model and execution code as the input. ModelSim provides the execution time (in clock cycles); Design Compiler estimates the chip area and power consumption of the processor.

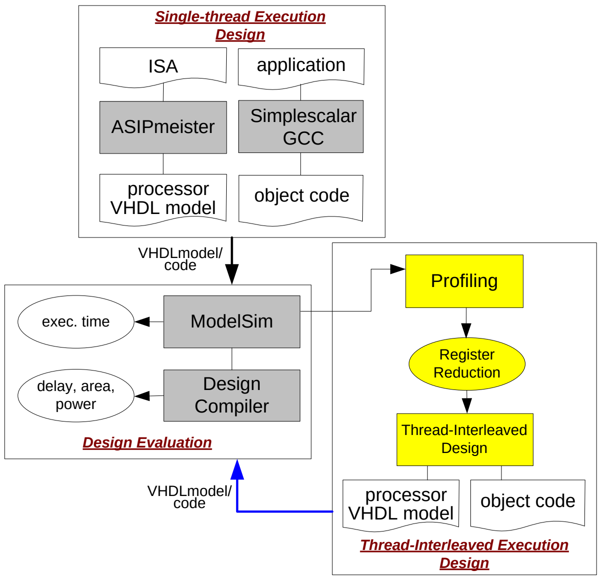

Fig. 4. Experimental setup

It is worth to point out that the power obtained from the Design Compiler is based on the used chip area with an assumed fixed clock frequency. Designs with a different number of threads may result in different clock frequencies, leading to different power consumption, which is, however, not covered in the measurement. To avoid the impact of the clock frequency change on the energy consumption, we introduce energy per instruction (EPI) to evaluate the energy consumption of each model.

Table I shows the results of register reduction for different applications, based on Algorithm 1 without memory replacement (given in rows 5-7), and Algorithm 2 with memory replacement when α = 0.5% (rows 8-13). The register file sizes, area, power consumption for the baseline design are given in rows 2-4. The relative area and power consumption of the two customized designs as compared to the baseline design are given in their corresponding data sections. For the designs with memory replacement, the performance overheads with two different scratchpad latencies (1 clock cycle and 3 clock cycles, respectively) are also provided in the table (rows 11-12), and the increase of code size (i.e. instruction memory overhead) is given in the last row. The average values for each data group are summarized in the last column. From the table, we can see that, on average, register merge achieves about 46% register file reduction and consumes 54% of the area/power as compared to the original design. If memory replacement is applied, a further 5% registers can be reduced with a performance overhead of 1.68% and 2.2% for the scratchpad memory latency of 1cc and 3cc, respectively. In addition, the memory replacement also incurs an average of 0.62% more instruction memory.

Table II shows simulation results of the processor designs for single-thread execution, two-thread execution, and three-thread execution. Each design for a given application is evaluated in their CPI (clock cycles per instruction), and energy consumption per instruction, EPI. As can be seen, the processor designs with multi-threading improve performance and energy - on average, 19% of CPI and 20% energy can be reduced for the two-thread designs. Further improvements (with CPI=1) can be observed for the three-thread designs.

TABLE I: REGISTER FILE CUSTOMIZATION RESULTS

		SHA	Quick sort	Bitcount	Dijkstra	String Search	AES	Bubble Sort	AVG
ori.RF	RF size	32	32	32	32	32	32	32	32
	area	2251540	2251540	2251540	2251540	2251540	2251540	2251540	2251540
	power(mw)	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42
red.RF w.o mem	RF size	21	22	14	17	13	22	12	17
	relative area	65.63%	68.75%	43.75%	53.13%	40.63%	68.75%	37.50%	54.02%
	relative power	65.63%	68.75%	43.75%	53.13%	40.63%	68.75%	37.50%	54.02%
red.RF w.t mem	RF size	21	21	12	14	12	20	11	16
	relative area	65.63%	65.63%	37.50%	43.75%	37.50%	62.50%	34.38%	49.56%
	relative power	65.63%	65.63%	37.50%	43.75%	37.50%	62.50%	34.38%	49.56%
	perfOH(1cc)	N/A	1.27%	1.22%	4.12%	2.59%	0.67%	0.22%	1.44%
	perfOH(3cc)	N/A	1.43%	1.62%	4.51%	4.95%	0.70%	0.02%	1.89%
	instrMemOH	N/A	0.11%	1.47%	1.33%	0.88%	0.32%	0.21%	0.62%

TABLE II: MULTI-THREADED PROCESSOR DESIGNS

		SHA	Quick sort	Bitcount	Dijkstra	String Search	AES	Bubble Sort	avg
1-thread	CPI	1.35	1.40	1.48	1.38	1.41	1.40	1.36	1.40
	EPI	2.55	2.65	2.80	2.61	2.66	2.65	2.57	2.64
2-thread	CPI	1.16	1.12	1.23	1.11	1.14	1.11	1.08	1.14
	EPI	2.36	2.28	2.14	1.97	1.98	2.22	1.84	2.11
3-thread	CPI	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	EPI	1.35	1.40	1.48	1.38	1.41	1.40	1.36	1.40

V. CONCLUSIONS

In this paper, we discussed two techniques to reduce register file for multi-threading processor designs: register merge and its enhancement with memory replacement. The register merge does not incur any instruction memory and performance overhead; while memory replacement provides a room for trading off the instruction memory and performance for more register reduction.

Experiment results show that the proposed customization techniques can greatly reduce registers required by a single application (near 50% on average). With the reduced register file requirement, the multi-threading processor can achieve high performance while at low energy consumption as compared to the single processor design – on average, for the two-threading processor, 19% of CPI and 20% energy can be reduced. By the three-thread design, the same energy saving (20%) as well as the highest pipeline throughput (CPI=1, or 28% improvement) can be achieved, with a slight cost of area and power. It must be pointed that the improvement may become less significant when the pipeline goes deeper and fewer independent threads are available. The techniques to handle such cases will be studied in the future.

REFERENCES

[1] T. Ungerer, B. Robic, and J. Silc, "A survey of processors with explicit multithreading," in *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29-63, March 2003.

[2] Z. Hu and M. Martonosi, "Reducing register file power consumption by exploiting value lifetime characteristics," in *Workshop on Complexity Effective Design (WCED)*, vol. 1, pp. 1829-1841, 2000.

[3] J. H. Tseng and K. Asanovic, "Energy-efficient register access," in *Proceedings of the 13th Symposium on Integrated Circuits and Systems Design, Manaus,* pp. 377-382, September 2000.

[4] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Peak, and E. Earlie. "Register file power reduction using bypass sensitive compiler," *IEEE Trans. on Computer-Aided Design of Intergrated Circuits and Systems,* vol. 27, pp.1155-1159, 2008.

[5] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *HPCA-6: Proceedings of the 16th International Symposium on High-Performance Computer Architecture,* pp. 375-386, 2000.

[6] R. Gonzalez, A. Cristal, D. Ortega, A. Veidenbaum, and M. Valero, "A content aware integer register file organization," *Proc. 31st Ann. Int'l Symp. Computer Architecture,* pp. 314-324, June 2004.

[7] R. Nalluri, R. Garg, and P. R. Panda, "Customization of register file banking architecture for low power," in *Proceedings of the 20th International Conference on VLSI Design*, pp. 239-244, 2007.
[8] X. Guan and Y. Fei, "Reducing power consumption of embedded processors through register file partitioning and compiler support," in *International Conference on Application-Specific Systems, Architectures and Processors*, pp. 269-274, 2008.
[9] X. Zhou, C. Yu, and P. Petrov, "Compiler-driven register reassignment for register file power-density and temperature reduction," in *Design Automation Conference*, pp. 750-753, 2008.
[10] S. Balakrishnan and G. S. Sohi, "Exploiting value locality in physical register files," in *36th Annual IEEE/ACM International Sympo-sium on Microarchitecture, IEEE*, pp. 265-276, 2003.
[11] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang, "Dynamically reducing pressure on the physical register file through simple register sharing," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS '04)*, 2004.
[12] Y. Tanaka and H. Ando, "Reducing register file size through instruction pre-execution enhanced by value prediction," in *IEEE International Conference on Computer Design*, pp. 238-245, 2009.
[13] T. Monreal, V. Vinals, J. Gonzalez, A. Gonzalez, and M. Valero, "Late allocation and early release of physical registers," *IEEE Trans. on Computers*, vol. 53, no. 10, 2004.
[14] Y. Zhou, H. Guo, and J. Gu, "Register File customization for low power embedded processors," in *2nd IEEE International Conference on Computer Science and Information Technology*, pp. 92-96, 2009.
[15] Asip-Meister. [Online]. Available: http://www.eda-meister.org/asip-meister/.
[16] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 12-67, 2002.
[17] Mentor Graphics Modelsim. [Online]. Available: http://www.model.com/.
[18] Synopsys. Synopsys Design Compiler. [Online]. Available: http://www.synopsys.com/.

Ran Zhang received BS degree from Beijing Information Science and Technology University. He is currently a Master candidate in Computer Science and Engineering at the University of New South Wales, Australia. His research interests include computer architecture for high performance and low power embedded system design.