Load scheduling: Reducing Pressure on Distributed Register files for free

Mei Wen, Nan Wu, Maolin Guan, Chunyuan Zhang
National Laboratory for Parallel & Distributed Processing
Chang Sha, Hu Nan, P. R. of China, 410073
meiwen@nudt.edu.cn

Abstract
In this paper we describe load scheduling, a novel method that balances load among register files by residual resources. Load scheduling can reduce register pressure for clustered VLIW processors with distributed register files while not increasing VLIW scheduling length. We have implemented load scheduling in compiler for Imagine and FT64 stream processors. The result shows that the proposed technique effectively reduces the number of variables spilled to memory, and can even eliminate it. The algorithm presented in this paper is extremely efficient in embedded processor with limited register resource because it can improve registers utilization instead of increasing the requirement for the number of registers.

1 Introduction

Distributed register files are better than traditional central register file in area, delay and power dissipation [1], and are widely used in clustered VLIW processors including: high performance DSP, such as TMS320C6x [4] and Storm [2] , stream processors such as Imagine [5], Merrimac [6], FT64 [7] and some specialized processor like Equator's MAP1000 [8].

These processors are mostly applied in compute-intensive applications. The much shorter access time of physical registers compared to that of memory, has always made them a critical processor resource [9]. In some processors, such as Imagine, multi-hierarchy memory managed by software is used to optimize access memory [10]. Once register allocation fails, programmer is needed to alter programs. On the other hand, optimizations that increase the size of the working set such as software pipelining increase register pressure during the scheduling process [9]. The pressure reduction on distributed register files in VLIW processor becomes a new problem [10].

Obviously, the difference between distributed register files and central register file is that data will spill out of the central register file if it exceeds the capacity of the register file. On the other hand, since the capacity of the distributed register files are normally tens and even hundreds times bigger than the central ones [3], plus the imbalance in load distribution, it is very rare that all register files are filled up at the same time. It is more likely that one or some of the register files overflow on some cycles. (See data analysis in section 2 and section 4)

Based on the above analysis, this paper suggests a pressure reduction method on distributed register files, in terms of clustered VLIW processor, which called ***load scheduling***. The method works as follows: consider the overall load of distributed register files after VLIW scheduling, fully utilize the imbalance among registers files, spilling some data from full register files to ones with free register, and moved back to source register file when it allows. The cost is inserting some *copy* operations into the unoccupied instruction slot, without altering the compiling result of other operations. Therefore, it is free. This way not only prevents data spills which may cause lower performance, but also increase the usage of resources. The experiment result shows that this method decreases the average peak value of register pressure by 45%, shortens average program execution time by 16%. It largely reduces the pressure on register files and memory accesses caused by spills.

The rest of the paper is organized as follows: Section 2 introduces related backgrounds that motivated our research and study; Section 3 describes related researches; Section 4 describes load scheduling; a performance evaluation is given in Section 5; and the last Section we conclude with a summary and a discussion of issues with possible enhancements to load scheduling.

2 BACKGROUND AND MOTIVATION

In this section we discuss the two aspects that motivated load scheduling: distributed register file architecture and VLIW scheduling, with description of the architecture of our experiment platform.

2.1 Distributed Register file architecture

In a distributed register file architecture each functional unit input is connected to the single read port of a dedicated register file and all functional unit outputs are connected by shared buses to the single shared write port into each register file (See Figure 1). The load scheduling presented in this paper is for this kind of distributed register file structure.

This structure is different from traditional structures, such as the one is that every functional unit input or output is connected by a dedicated bus to a dedicated register file port of a central register file, and the other type is when a clustered register file architecture, which divides the functional units into clusters and provides each cluster with its own register file.

Compared to these traditional structures, distributed register file architecture offers better performance, smaller size and power consumption, and reduces register file access delay [1].

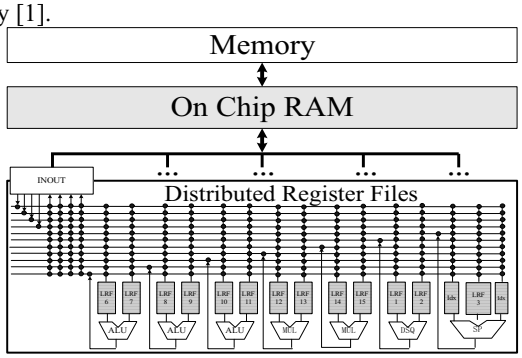

Figure1 Distributed register file architecture

978-1-4244-1922-7/08/$25.00 ©2008 IEEE

2.2 VLIW Scheduling

A VLIW scheduler takes a set of operations and produces a schedule that specifies which operations to issue to which functional unit on a given cycle. The key problems in VLIW scheduling are finding enough parallel operations, and scheduling those operations to occur on a particular functional unit on a particular cycle in a way that effectively utilizes this parallelism. Compared to traditional VLIW scheduling for central register file, the VLIW scheduling for distributed register files also includes the communication scheduling for data transmission among register files [10]. When all the instruction assignment are finished, graph coloring method is used for register allocation [14, 15].

2.3 Experimental platform

Imagine and FT64 [5, 7] is examples of the combination of distributed register files/VLIW. Based on these practical platforms [1], it is easy to find new challenges and opportunities for effectively managing distributed register files pressure, enabling us to carry out research on load scheduling.

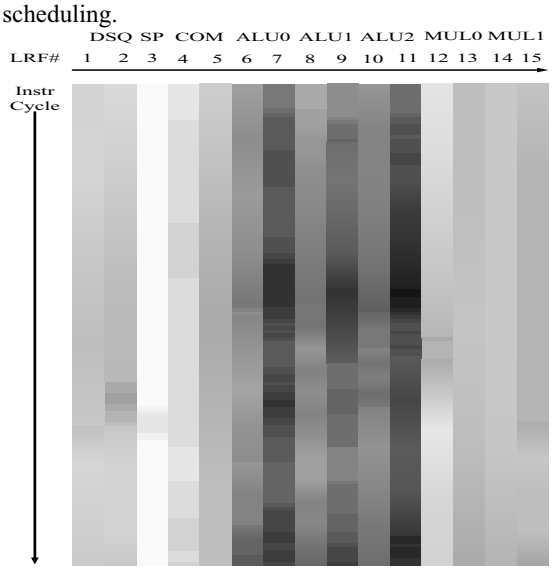

Figure 2 Register pressure of kernel *Blocksearch*

We can see in Figure 1 that every ALU has several local register files, the spill of register files normally is not the spill of the entire register files but the spill of one local register file, since it is partial spilling caused by imbalanced register load. Figure 2 shows the segment of register allocation of the Kernel *Blocksearch* that is running on Imagine. Each column represents one register file, and vertical axis represents the instruction schedule length. Depth of the color shows the pressure on the register, the deeper the color represents the more variables assigned to the register file on the cycle. We can see that the distribution in the register load is extremely imbalanced.

Figure 3 is a study of an Imagine that contains 15 distributed register files, it shows the average demand for register capacity in each register file. 1~15 is the index of register file. We have studied 9 Kernels, suppose that the size of each register file is 16 entries, since the requirement between each register files are imbalanced, to each kernel, some of the register file requirement are over 100% and some can not even reach 20%.

[1] All data are acquired from ISIM [10], the cycle accurate simulator of Imagine, and FT64.

3 RELATED WORK

For VLIW architecture with distributed register files，ISCD compiler [10] tries to introduce heuristic to ease register press during communication scheduling. But since the heuristic is added onto a single operation, it can not consider the whole system. Besides, the heuristic may increase the chances of communication scheduling failures. Experiment result shows that register files still overflow severely (see Table 1). Nan et al. [13] suggests the cutting strategy, re-cut

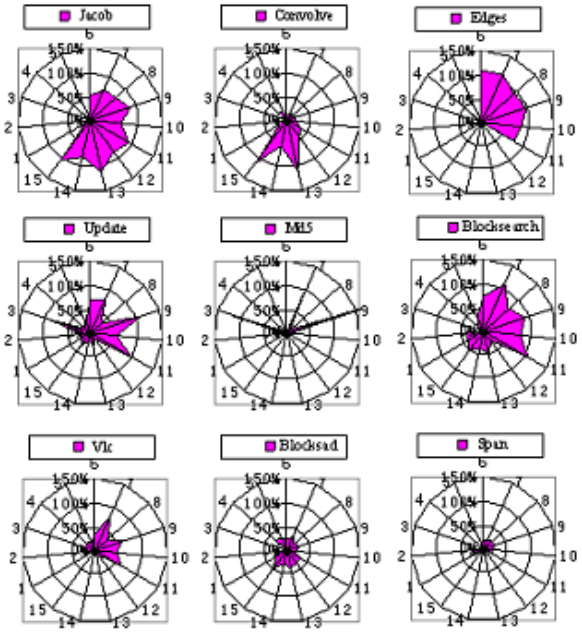

Figure 3 Average demands for register capacity

the basic segments to reduce register pressure, avoid spilling, but the cost is huge deduction in performance.

In the literature we can find many works dealing with register pressure for some other architecture. One opinion is that register allocation and instruction scheduling are two separate phases, this introduces the problem of phase coupling. It is suggested to consider register allocation and operation scheduling at the same time [12, 16, 18]. Also for load imbalance problem between cluster register files, Zalamea [12] suggests executing early or delay some of the operations in the full loaded register files based on cycle-driven. Cycle-driven is no better than operation-driven for the VLIW scheduling that needs communication scheduling, since it can not ensure that communication between operations on the critical path are scheduled first. However, operation-driven mode will make it impossible for operation to execute earlier or later, after operation scheduling is finished.

Spilling variable from full register file to other register file with free register is not a new idea. Some scheduling algorithms target specific architectures that take incremental steps beyond a clustered register file architecture. Compiler [11] targets architecture with multiple cluster register files that includes a small number of cross-cluster buses, it still provides each functional unit input and output with a dedicated bus and register file port to access its cluster register file. The Cydra5 compiler [3] targets an architecture in which each functional unit input can read from multiple

register files, but provides each input with a dedicated bus and a dedicated register file port to access each register file.

The contribution of this paper is applying the idea to the distributed register file architecture as shown in Figure 1. It is different from shared multiple register file architecture. Compared to published papers, there are differences in architecture or method. Some new problems, such as communication network scheduling, limited Load/Store ports and register ports, application domains, need to be considered.

4 Load scheduling

This section describes the algorithmic details of load scheduling. Pseudo-code of load scheduling implemented in compiler is shown. We describe the following terms for easy interpretation:

T_{up}, T_{down}: For a register file, the period of time that the number of registers needed exceeds the number of registers available is called *overflow zone*, its beginning and end are called the upper limit and lower limit of the overflow zone. T_{up} is its upper limit and T_{down} is its lower limit.

$T_{overflow}$: any cycle during the overflow zone.

$T_{produce}$: the beginning of the live range of a variable.

$T_{lastuse}$: the end of the live range of a variable.

T_{out}: the cycle when the spilled variable is moved out from the source register file.

T_{early}: the cycle when the spilled variable was last used before $T_{overflow}$, if it was never used before $T_{overflow}$, T_{early}= $T_{produce}$

T_{sbegin}, T_{send}: The beginning and the end of split live range of the spilled variable.

T_{use}: the cycle of the first use of the spilled variable, after $T_{overflow}$.

T_{back}: the cycle of moving spilled variable back to the source register file.

After the accomplishment of the VLIW operation assignment, communication scheduling and register pre-allocation, load scheduling that we have suggested have the following five main steps:

Step 1. Construct residual network

After all the processor resources, including interconnect buses, registers and functional units, are allocated, the schedule result as Figure 4a can be derived.

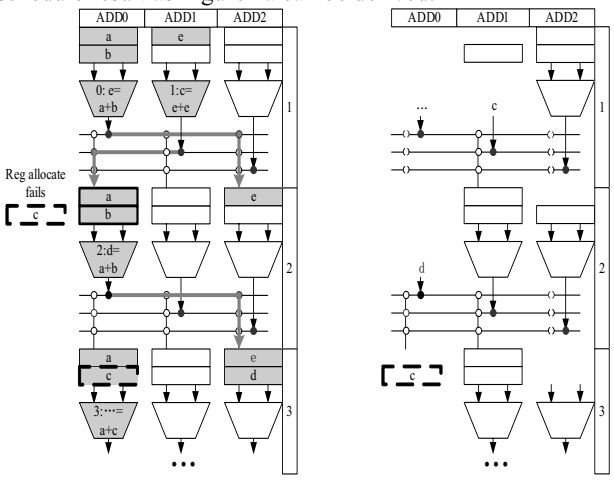

Figure 4a (left) Schedule for shared interconnect architecture Figure 4b (right) Residual network

Figure 4a illustrates some functional units, interconnect, and register file activity on each cycle. If we eliminate all the grey area, which stands for the used resource, in the Figure4a, we would get the residual network as shown in Figure 4b. The residual network indicates all the available resource after VLIW scheduling, including functional unit (FU) interconnect and local register file. The load scheduling is going to be carried out in the residual network. Experiment shows that in most cases, residual network contains adequate available resource. The idle rate of the main resources when running kernels on Imagine is usually exceeds 40%.

Step 2. Choosing spilled variable

To overflowing register file, it can reduce its register pressure by spilling any of the variables which existing in the *overflow zone*, multiple variables spilled can keep register from overflow (if route is available). Each *overflow zone* contains one or more $T_{overflow}$. As to any $T_{overflow}$, as long as $T_{produce} \leq T_{overflow}$ and $T_{overflow} < T_{lastuse}$, the variable that is not used on cycle $T_{overflow}$ can be used as spilled variable.

Theorem 1.1: Register files that has ***n*** writing ports and reading ports, suppose that the latency of *copy* operation, that is used for moving variable, is L, when the capability of register file is no less than n*L, it is always possible to find spilled variables.

Proof: when register file overflows, there are at least n*L+1 variables whose live ranges contain $T_{overflow}$, as to these n*L+1 variables, their $T_{produce}$ are not later than $T_{overflow}$. Since the writing ports and reading ports of the register file is n, it only allows a maximum of n variables used at one cycle. From $T_{overflow}$, choose the last used variable from n*L+1 variables, so called *v*, when *v* is used, the distance between this use and the last use, or this use and its produce cycle is at least L, we can use *copy* operation to move them out or back, reducing the pressure on register file on cycle $T_{overflow}$. As long as the route is available, multiple variables spilled can avoid the overflows of the register file. The choice of spilled variable affects the difficulty of following route assignment and influences other register files. In this paper the following rules are used to choose spilled variables and there are two definitions to be explained as follows:

The distance before recently usage **(D1)**: Before the cycle $T_{overflow}$, the distance between T_{early} and $T_{overflow}$. The wider the distance before recently usage is, the bigger the probability is to spill such variable out.

The distance after recently usage **(D2)**: After the $T_{overflow}$ cycle, the distance between the first use and $T_{overflow}$, the wider the distance of the distance after recently usage is, the bigger the probability is to move such variable back after $T_{overflow}$.

Suppose that the weighted value P = C1×D1 + C2×D2 {D2 ≠ 0, C1 and C2 are experience values}, the bigger P is, the bigger the probability is to find route. Then we select the variable with the biggest P to spill, if the biggest one couldn't find route, then select the second biggest variable and likewise.

Step 3. Assign route

After choosing the spilled variable, the route needs to be assigned to move such variable. The route is a circular route; if one spilled variable is spilled out from a register file, it has to be moved back before the next usage. Suppose the live range of the chosen spilled variable is [$T_{produce}$, $T_{lastuse}$]. A route consists of the following elements:

1) T_{out}: T_{out} has to satisfy $T_{out} \leq T_{overflow}$ to make the moving

valid. Any cycle between [$T_{produce}$, $T_{overflow}$] can be chosen as T_{out}, the closer to $T_{produce}$, the more effect it is to the destination register. Besides, the effect on the source register is also decided by T_{early}. In the interference graph, T_{sbegin}= max (T_{out}, T_{early}).

2) Port of moving out: The output port of the register file and the output port of the functional unit, which are needed for moving out spilled variable, are bound in VLIW scheduling (fixed operation latency), and can be seen as one port resource. On cycle T_{out}, the chosen spilled variable is output from the output port: OutputPort [T_{out}] = *v*. There are two scenarios that the output port is available. One is when OutputPort [T_{out}] =IDLE, at this point a v=copy(v) operation can be inserted into the respective functional unit, making OutputPort [T_{out}]= *v*. The other one is OutputPort [T_{out}] = *v*, now no operation is needed to be inserted.

3) Functional unit of moving out: The variable *v* is moved from the functional unit to interconnect on T_{out}. There are two scenarios that the functional unit is available. One is that functional unit of moving out is the functional unit whose output is connected to the source register file, also when FU.OutputPort[T_{out}]= *v*. The second one is that there is other functional unit outputting variable *v* on T_{out} (at this moment the destination register file can get the variable from such unit, not from the functional unit connected to the source register file). There is an important special situation, which is when T_{out} = $T_{produce}$, then always ∃ FU → FU.OutputPort[T_{out}]=v. This is to say that when time of moving out T_{out} equals time of produce, there must be a functional unit whose output variable is *v*.

4) Destination register file: For the register files for temporary storing of the spilled variable that is moved out, the suitable destination register has to meet two requirements: free register and free writing port. It's best that there is free register in this destination register file during the *overflow zone* of the source register file. However it is also acceptable if there is free register on cycle $T_{overflow}$. If such moving caused the destination register file to overflow, load scheduling algorithm can be called repetitively to start the next movement. The free register and writing port can be found through searching the residual network. There may be several register files that match the requirements, the following rules apply to rank them, e.g., Reg.InputPort=IDEL & Reg.ResidualCapacity=MAX (choosing the register file with free register during [T_{out} , T_{use}] and free writing port on T_{out}, can avoid recursion movement, ranking them by the number of free registers).

5) Interconnect of moving out: The interconnect path between the spilling-out functional unit and the destination register file, depends on the hardware topology. The load scheduling algorithm applies to all kinds of topology of full connected. For the interconnect topology as Figure 1 shows, the communication scheduling is used to assign the interconnect path of the moves. As long as the source functional unit is available, the destination functional unit is available, and the structure is full interconnect one, there must be moving out interconnect available to use.

6) T_{back}: T_{back} has to satisfy $T_{overflow} < T_{back} \le T_{use}$. T_{back} can be any cycle between [$T_{overflow}$+1, T_{use}]. In the interference graph, T_{send}=T_{back}.

7) The functional unit of moving back, interconnect of moving back and port of moving back are similar to the ones for moving out, they are just the opposite of the other ones. For such reason they will not be described here in detail.

The pseudo-code used in assigning route is as follows:

```
Function RouteAssign
Imput:
   RN /* the residula networks constructed by step1*/
   v /*the var need to be spilled, which is chosen by step2*/
   T_overflow /* the instr cycle on which the register file overflows*/
   latency /* the latency of copy operation*/
Output
   route /* the route assigned to move v*/

route.exist=fasle;
for (cycle T_out= v.T_produce; T_out<= T_overflow; T_out++){
/*look for the cycle on which v will be spilled out*/
   route.OutFU=NULL

 for ( i=0, i < Number of FUs in RN, i++)
    if (RN->Fus[i]->OutPort[T_out] = v){
       route.OutFU=RN->Fus[i]; break;}
     /* look for the available outport to spill v out*/
    if (route.OutFU!=NULL | (v.FU->OutPort[T_out]=IDLE &
       v .FU->OP[T_out-latency+1]=NOP)){
        /* v can be spilled out by a copy operation*/
        List <Reg > FreeRegs = NULL;
         for ( j=0, j < Number of RFs in RN, j++) {
         /*look for free register file, ordered by the free capacity*/
           if (Reg.inPort[T_out]=IDLE){
               Add Reg to FreeRegs ordered by the free capacity
              of each RFs in RN during [T_out,T_use];}
           while (!FreeRegs .empty){
               MaxFreeRegs=FreeRegs .pop();
               for ( cycle T_back= v.T_use; T_back>T_overflow; T_back--){
              /* look for he cycle to move v back */
                  if (MaxFreeRegs .FU->OutPort[T_back]=IDLE &
                   Max FreeRegs.FU->OP[T_back-latency+1]=NOP &
                     v.Regs->InPort[T_back] = IDLE &
                     (route.OutFU != NULL |(T_back -T_out) >=2*latency)){
                     /*v can be moved back by copy operation*/
                     route.exist=true;
                   break out all ;}
}
if (route.exist) {
   route.T_out=T_out;
   route.T_back=T_back;
   route.DesReg=MaxFreeReg;
   /*construct route for spilling v out*/
  /*function ConstructTansform construct route connected FU, interconnet and
registers*/
   if (route.OutFU!=NULL){
     route.OutConnect=ConstructTansform( route.OutFU, DesReg);
     route.OutCopyFU=NULL;}
   else{
     route.OutConnect=ConstructTansform( v.FU, DesReg);
     route.OutCopyFU=v.FU;}
    /*construct route for moving v back*/
     route.BackConnect=ConstructTansform( DesReg.FU, v.Reg);}
return route.exist;
```

Step 4. Inserting copy operation

After route is assigned, inserting *copy* operations [17] in the route, and assign the output and input of such *copy* operation.

Copy operation is going to acquire the data directly on the cycle of the production of the variable, and store it in other register file with free register. Then after $T_{overflow}$ and before the time when variable is needed, move it back to the source register file to use. The live range of the spilled variable in the source register file is [T_{back}, $T_{lastuse}$]. If at the time of variable production, the input port of the destination register file is not available, a second *copy* operation can be inserted between $T_{overflow}$ and time of production, and executed by the functional unit connect to the source register file. This *copy* operation acquires data from the port of the production of variable, and such variable is stored in the same register file. The result of the basic register allocation method shows that such *copy* operation would not increase pressure on the source register file. The first *copy* operation gets variable from the output of the second *copy* operation. Suppose the inserting time of the second *copy* operation is $T_{incopy2}$ and

divide the live range of the variable in the source register file into two parts: [$T_{produce}$, $T_{incopy2}$] and [T_{back}, $T_{lastuse}$], which can also reduce pressure on source register file on $T_{overflow}$.

For residual network as Figure 5a shows, the result of load scheduling is shown in Figure 5b. After the route is assigned, the resource such as the part marked X in Figure 5b, which falsely allocated because of register allocation failure, can be released. At the same time, the live range of the variable in the coloring graph is changed according to T_{sbegin} and T_{send}

Step 5. Re-allocate register using graph coloring

Using graph coloring to re-allocate registers, if it still fails, go back to step one for another variable spilling, till all the registers are successfully allocated or workload is unable to moved.

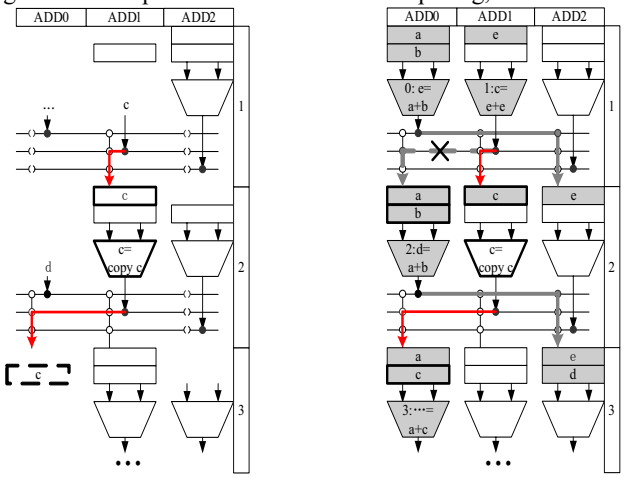

Figure 5a (left) New route in residual network
Figure 5b (right) Schedule result

The algorithm presented in this paper is suitable for several times of spilling or recursion movement. The former refers to be spilled out again after variable moved back. The latter is saying when destination register file overflows then it has to be moved to other register file. Recursion movement can be avoided through the heuristic when choosing destination register.

5 EXPERIMENT EVALUATION

In this section we use media stream processor Imagine and the stream processor FT64 that is for scientific computation as experimental platform. Both have 15 distributed register files, each register file in Imagine and FT64 have 16 entries. We perform our evaluations using Kernels, which press register file with heavy load. These kernels are chosen from SPEC2000 benchmarks, media benchmarks, Mibenchmarks. These application domains represent the typical applications of Imagine and FT64. Kernel *Jacob* is run on FT64. Others are run on Imagine. All kernels were written in a limited subset of C. There are three scheduling strategies in the evaluation:

-*default*: normal ISCD schedule [10], VLIW scheduling and communication scheduling, then register allocation.

-*rf3*: VLIW scheduling and communication scheduling with the heuristic, considering functional unit and register pressure during VLIW scheduling, then allocate register [10].

-*ls*: VLIW scheduling, communication scheduling, register allocation, and then load scheduling.

Firstly, define a evaluating metric: the maximum need of registers: N[idx], the maximum number of variables that are assigned to the number idx register file during scheduling length. This number implies pressure on such register file, and overflows would happen if such number exceeds the size of register file storage.

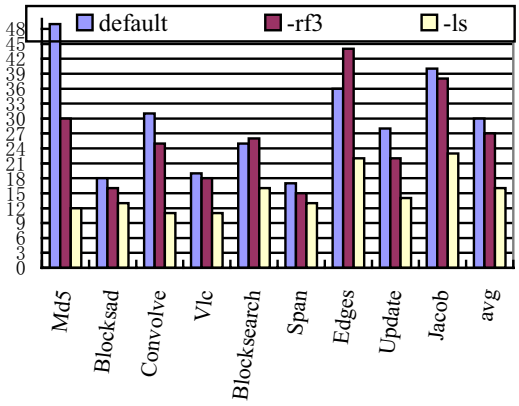

Figure 6 Peak register pressure by different strategies

Figure 6 shows the peak value of register demand (Max(N[1..15])) [2], such value determines the size of each register file on the condition of no overflows). Normally ISCD schedule does not consider register allocation fail. If let 16 being the size of each register file, it can be seen that the peak register demand of kernel *Md5* is more than three times of its actual storage. After optimizing by –rf3, it reduces the peak register demand of some Kernels. However, it is only 9% on average. Some of the peak value of register demand of Kernels is not decreased but increased, which means that the heuristic estimation of register pressure is not accurate, and it can not ensure reducing the maximum need of registers. Load scheduling algorithm can carry out schedule according to the final result, grasping the load distribution more accurately, and reducing maximum need of registers more effectively. Figure 6 shows that load scheduling reduces 45% more of the peak register pressure than normal ISCD schedule.

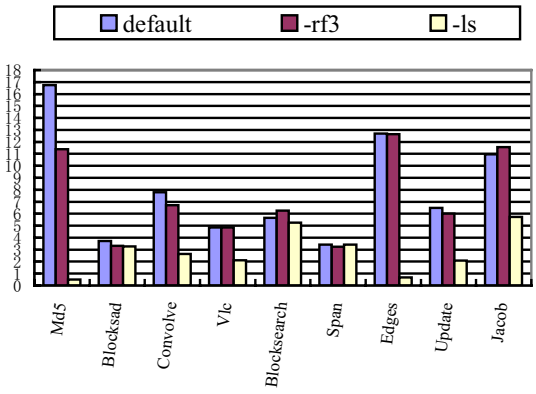

Figure 7 Standard deviation of register demand

Figure 7 reflects the difference of register demands among distributed register files (N[1] ~ N[15]), shown by way of standard deviation. Obviously, the need difference between register files, before load scheduling, is very dramatic, average standard deviation across kernels is 7.7, after -rf3

[2] To acquire the register demand limit by load scheduling, we make the algorithm continually spills variable out from register file which assigned the most variables to until no available route, in despite of register file's capacity.

optimazation, the value is 7.2. Compare to that without optimazation, it is reduced by 7%, after load scheduling, the value is 2.9, reduced by 62% and 60% compared to the previous two respectively. This demonstrates the load scheduling effectively balances the load among distributed register files.Table 1 shows the effect of three scheduling strategies on kernel execution time. Colomn 'spilled vars' indicates the number of variables spilled out to memory. Colomn 'spill time' indicates time caused by spilling out to memory (cycle). Colomn 'total run time' indicates kernel execution time (cycle). The load\store intruction latency is set to be 3 cycle. Table 1 shows that the number of variables spilled out to memory is decreased effectively, the running time is reduced by 16% on average, average speedup reached 1.1 by load scheduling (compared to –default). Generally speaking, load scheduling can balance between distributed register files better, it can ensure reducing the peak register pressure on distributed register file. There are three aspects of its meanings to this technology. First is that in processors whose capacity of register file is fixed, load scheduling can reduce the circumstances when maximum register demand exceed register file storage, avoiding access memory. Second is reducing data spilling efficiently can effectively support loop optimization and other compiler optimization technology, on the condition of no changes to the number of registers. Third is that for processor designers, load scheduling can save the number of registers in each register file, it is especially useful in embedded processor.

Table 1 Scheduling result by three scheduling strategies

Kernel	spilled vars	spill time	total run time	speedup	Kernel	spilled vars	spill time	total run time	speedup
Md5 default	91	4368	11232	1	Span default	1	1536	30976	1
Md5 –rf3	60	2880	9776	1.14	Span -rf3	0	0	29440	1.05
Md5 -ls	0	0	6864	1.63	Span –ls	0	0	29440	1.05
Blocksad default	5	120	1370	1	Edges default	92	210864	394224	1
Blocksad–rf3	0	0	1312	1.04	Edges –rf3	93	213156	396516	0.99
Blocksad –ls	0	0	1250	1.09	Edges –ls	41	93972	277332	1.42
Convolve default	25	375	1474	1	Update default	22	1056	4976	1
Convolve–rf3	17	255	1380	1.06	Update -rf3	20	960	4880	1.01
Convolve –ls	0	0	1099	1.34	Update –ls	0	0	3920	1.26
Vlc default	5	690	17006	1	Jacob default	96	2880	4680	1
Vlc -rf3	3	414	16279	1.04	Jacob -rf3	97	2910	4710	0.99
Vlc –ls	0	0	16316	1.04	Jacob –ls	82	2460	4260	1.09
Blocksearch default	29	4002	37473	1	Blocksearch -rf3	20	2760	36927	1.01
					Blocksearch -ls	16	2208	35679	1.05

6 CONCLUSIONS

We have presented a new post register allocation scheduling method, balancing load among distributed register files by load scheduling. It can largely avoid single register file overflows and helps the success of register allocation. One register file is stored in other un-filled register files, without repetitive computation and avoids more burdens. Since load scheduling uses the residual network to apply spills, it does not increase scheduling length and hardware cost.

Although the experiment is conducted under Imagine and FT64, this algorithm is also applicable to other VLIW compiler, which is for distributed register files. Given that embedded VLIW processors usually have fewer registers than our experimental assumptions, it would be particularly interesting to evaluate the impact of load scheduling in such an environment.

Acknowledgements. This research was supported by NSFC No. 60673148, 60703073, SRFDP No. 20069998025

REFERENCE

[1] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson et al. Register Organization for Media Processing. In Proceedings of the Sixth International Symposium on High Performance Computer Architecture, pages 375-387, January 2000.

[2] Brucek Khailany et al. A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing, ISSCC 2007

[3] Brucek Khailany. the VLSI Implementation and Evaluation of Area and Energy Efficient Streaming Media Processors. Ph.D. Thesis, Dept. of Electrical Engineering, Stanford University 2003

[4]T.I.Inc. TMS320C62x/67x CPU and Instruction Set Reference Guide. 1998

[5] S.Rixner,W.J.Dally et al. A Bandwidth-Efficient Architecture for Media Processing, proceedings of the 31 st annual ACM/IEEE international symposium on microarchitecture,1998 .

[6] William J.Dally, Mattan Erez et al. Merrimac: Supercomputing with Streams, SC'03, November 15-21, Phoenix, Arizona, USA.

[7]Xuejun Yang et al. A 64-bit Stream Processor for Scientific Applications, ISCA2007

[8]P.N.Glaskowsky. MAP1000 unfolds at Equator. Microprocessor Report. 12(16) Dec. 1998

[9]Ivan D. Baev. Prematerialization: Reducing Register Pressure for Free. PACT2006

[10] Peter Mattson. A Programming System for the Imagine Media Processor, Dept. of Electrical Engineering. Ph.D. Thesis ,Stanford University.2001

[11]David J.Kolson. A Method for Register Allocation to Loops in Multiple Register File Architecture In proceedings of IPPS'1996.

[12]Javier Zalamea. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW architectures. IEEE 2001.

[13]Nan Wu et al. Register Allocation on Stream Processor with Local Register File, ACSAC 2006.

[14] Muchnick, S. Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.

[15] Preston Briggs. Register Allocation via Graph Coloring. PhD thesis, Rice University, Houston, TX, USA, 1992.

[16] Bart Mesman et al. Efficient Scheduling of DSP Code on Processor with Distributed Register Files, In proceedings of the 12th International Symposium on System Synthesis, 1999

[17] Peter Mattson. Communication Scheduling, Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, Nov. 12-15, 2000, Cambridge, MA, pp. 82-92.

[18] Josep M. Codina, Jesus Sanchez and Antonio Gonzalez. A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors, IEEE 2001

[19] James C.Dehnert, Ross A. Towle. Compiling for the Cydra 5, Journal of Supercomputing 7(1/2), 1993, pp. 218-219.