Expression Rematerialization for VLIW DSP Processors with Distributed Register Files [⋆]

Chung-Ju Wu, Chia-Han Lu, and Jenq-Kuen Lee

Department of Computer Science,
National Tsing-Hua University,
Hsinchu 30013, Taiwan
`{jasonwu,chlu}@pllab.cs.nthu.edu.tw,jklee@cs.nthu.edu.tw`

Abstract. Spill code is the overhead of memory load/store behavior if the available registers are not sufficient to map live ranges during the process of register allocation. Previously, works have been proposed to reduce spill code for the unified register file. For reducing power and cost in design of VLIW DSP processors, distributed register files and multi-bank register architectures are being adopted to eliminate the amount of read/write ports between functional units and registers. This presents new challenges for devising compiler optimization schemes for such architectures. This paper aims at addressing the issues of reducing spill code via rematerialization for a VLIW DSP processor with distributed register files. Rematerialization is a strategy for register allocator to determine if it is cheaper to recompute the value than to use memory load/store. In the paper, we propose a solution to exploit the characteristics of distributed register files where there is the chance to balance or split live ranges. By heuristically estimating register pressure for each register file, we are going to treat them as optional spilled locations rather than spilling to memory. The choice of spilled location might preserve an expression result and keep the value alive in different register file. It increases the possibility to do expression rematerialization which is effectively able to reduce spill code. Experiments were done for the PAC VLIW DSP processor and based on Open64 compiler infrastructures. Early experimental results show that our approach can reduce memory access operations due to the well-partitioned live ranges and well-rematerialized expression values.

1 Introduction

Register allocation is known as one of the most important phases in advanced compilers. Generally, compilers use intermediate form to represent code sequence

[⋆] This paper is submitted to *CPC 2009*. The correspondence author is **Jenq Kuen Lee**. His e-mail is **jklee@cs.nthu.edu.tw**, phone number is **+886-3-5715131 EXT. 33519**, FAX number is **+886-3-5723694**. His postal address is **Prof. Jenq-Kuen Lee, Department of Computer Science, National Tsing-Hua University, Hsinchu 30013, Taiwan**.

corresponding to original source code, in which infinite temporary names are used as data representations for variables. Before emitting assembly codes for specific target machine, the register allocator is responsible to create a map from all live ranges of those data representations to hardware physical registers. Such live-range-mapping problem can be modeled as a graph-coloring problem.

Chaitin's work [6, 7] is the first to implement register allocator based on a graph coloring algorithm. It attempts to color graph that illustrates the interference relation among live ranges using K colors, where K is the number of physical registers. Once the graph is not K-colorable, which means the physical registers are not sufficient to map live ranges, the spill code must be inserted into code sequence to split particular live ranges in order to release register pressure so that register allocator is able to map all live ranges with limited number of registers. However, Chaitin's naive method may produce lots of redundant spill code which issue too many unnecessary memory access. Since the relative cost of memory access is increasing in present day, well-utilized registers are helpful to avoid spilling code. Several approaches have been proposed [8–10] to reduce spill code and have shown successful result. But all those works are dedicated to unified register file.

Nowadays, for the design of embedded processing with greater parallelism, power consumption and chip die size are always to be the significant concerns on top of processor performance. In designing VLIW DSP processors, the large number of registers that are accessible by all functional units require large number of ports. This restricts the possible processor cycle duration and raises the difficulty of the design [4]. To solve this weakness, a variety of distributed register file architectures have been developed for embedded processors in recent years, promoting hardware design to lower power dissipation and reduce die size over traditional unified register file structures. The appearances of distributed register files architectures on embedded VLIW DSP processors present a great challenge for compilers to generate efficient codes for multimedia applications. In the literature, current research results in compiler optimizations for such problems include the work on partitioning register files to work with instruction scheduling [24], loop partitions for clustered register files [25], copy propagation method for distributed register files [26], and global register allocation method [27, 28].

In this paper, we propose a new technique which exploits characteristics of distributed register files to reduce spill code. Our method is a form of rematerialization, which tries to re-compute the values in the case of spill for memories. Previous work for rematerialization has been mainly emphasizing on re-computing constant expression, while we attempt to re-compute expression still in the live range. In addition, we deal with architecture with distributed register banks. The difficulty in trying to re-compute the expression in the case of register spill is that the expression we want to compute might no longer be alive in the live range while we attempt to re-compute the expression. Our proposed solution is to perform a pre-calculation ahead of time when the expression to be re-computed is still in the live range. We then try to find a free register in different register banks to save the value. This also provides a code motion opportunity for the generation of spill codes to different register banks. In the de-generalized case, our method works as if spill codes are done to other register banks instead of memory. We propose methods to estimate register pressures in each register bank, models to measure if the method is profitable for distributed register file architectures, and a software framework for incorporating our proposed scheme. Experiments were done with a developing compiler for the PAC DSP architecture and based on Open64 compiler and with our proposed schemes. PAC (Parallel Architecture Core) DSP [1–3], is designed with distinctively banked register files where port access is highly restricted. The results indicate that our approach delivers significant performance improvement over spill code cases.

The remainder of this paper are organized as follows. In Section 2, we will introduce the processor architecture and register file organizations as target example. Section 3 will brief the motivation for expression rematerialization concept, and then describe our proposed approach in detail. The experimental results are shown in Section 4 and Section 5 finally concludes this paper.

2 Background

Throughout this paper we take PAC DSP processor as our target machine. This section brief the overview of PAC DSP VLIW architecture and its distributed register files structure, plus a little register allocation concept and register file assignment effort prior to our work.

2.1 PAC DSP Architecture

The Parallel Architecture Core (PAC) is a 32-bit, fixed-point, and five-way issue VLIW digital signal processor (DSP). PAC DSP is comprised of two Load/Store Units (LSU, memory access, M-unit), two Arithmetic Logic Units (ALU, instructions for powerful arithmetic, I-unit), and one Scalar Unit (Scalar, branch operations, B-unit). LSU and ALU are organized into two clusters, each containing its own private register file and being capable to access public register file. The Scalar unit individually takes charge of operations of control flow instructions, plus issue some load/store and arithmetic operations with its own private register file. The overview architecture is illustrated in Figure 1.

As in Figure 1, the register file structure in each cluster is highly partitioned and distributed. Among this architecture, PAC DSP contains four distinct register files. The A, AC, and R register files are private registers, directly attached to and only accessible by M-unit, I-unit, and B-unit, respectively. The D register file is public within one cluster where M-unit and I-unit are both able to access it. Furthermore, the internal of the D register file is partitioned into two banks to utilize the instructional port switching technology in order to reduce more wire connections between M- and I-unit. During each cycle, the two functional units can only access two different banks. We believe this special register file design

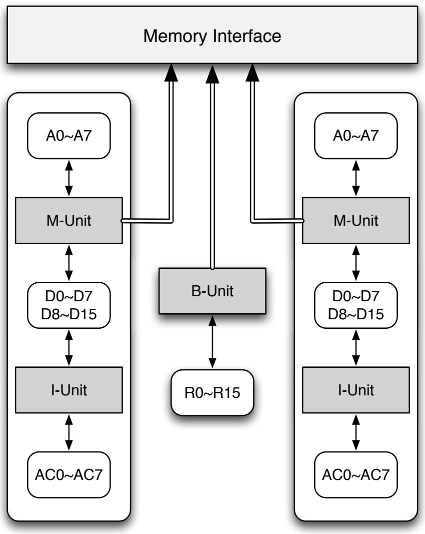

Fig. 1. The PAC DSP architecture

can help us achieve low-power consumption because it retains an effective way of data communication with less wire connections between functional unit and registers. But it also introduces several issues to be dealt with register allocation. These issues would be described in Section 3.

PAC DSP processor [1] is currently developed at ITRI STC, and our laboratory is collaborating with ITRI STC under MOEA projects for the challenging work to develop high-performance and low-power toolkits for embedded systems under PAC platforms [11–16, 27, 28]

2.2 Chaitin-Style Register Allocator

Previous literature on register allocation has proved that the problem of optimal register allocation is NP-complete [5]. Therefore, compilers always develop their own heuristic techniques to approximate its solution. The commonly used heuristic technique is the graph coloring algorithm which was originally developed by Chaitin et al. It constructs an interference graph in which there are nodes and edges. Nodes in the graph represent the live ranges which need to be allocated to machine registers. Edges represent the interference between two connected nodes (i.e. live ranges) which cannot occupy same register. Besides, for a live range l_i, its neighbors are the live ranges that interference with it and the degree of l_i is the number of its neighbors. Chaitin operates these information and keep changing the interference graph phase by phase. Figure 2 illustrates the Briggs' register allocation flow [9] which is the improvement of the Chaitin-style coloring algorithm.

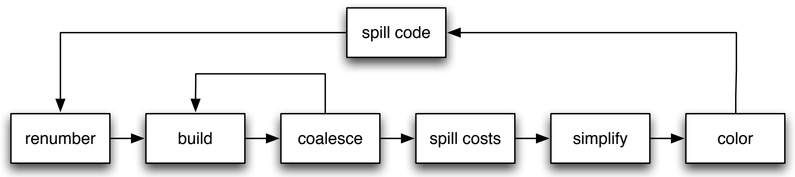

Fig. 2. The Chaitin-Briggs Allocator

To find an allocation for interference graph with K colors, all nodes with degree < K will be removed continuously from interference graph during the *simplify* phase. If there remains nodes in the graph eventually, it tells K-coloring is not discovered and some live ranges will be chosen to do spilling. Chaitin's spilling roughly insert a memory store after each *def* of the live range and a memory load before every *use*. This spilling technique would produce memory access everywhere for the entire live range. So he also mentioned that several optimizations can reduce amount of unnecessary spill code [7] and there were many researches working on it [8, 9, 20–22].

2.3 Register File Assignment

Because the PAC DSP has a clustered organization with distributed register files, the communication cost is a considerable issue to do register allocation. For supporting instruction level parallelism in a VLIW architecture, effective register allocation with scheduling stands in crucial position to optimize code performance. Typically, register allocation and instruction scheduling are performed in separate phases for most compiler infrastructures to decrease the complexity of combining these two optimization problems. If we can dispatch the register usage on register files as more parallelism as possible, the instruction scheduler is able to bundle more than one operations in single instruction word. Unfortunately, the dispatch job always requires new code sequence to exchange data between different register files as long as they have dependencies. Those essential communication cannot be eliminated. Instead, it is supposed to be minimized.

Minimizing the communication cost in PAC DSP makes it desirable to use a new phase, Register File Assignment (RFA), to handle communications [11, 12, 27, 28]. With this heuristic approach assistance, we then apply general graph coloring techniques for each register file.

3 Expression Rematerialization

Expression Rematerialization is the idea for improving the quality of spill codes. Our proposed methodology analyzes register pressure for each register file, picking up a live range and then move it into the destination with low pressure. We are going to talk about the motivation of our work followed by the details and algorithm in this section.

3.1 Motivation

As it is mentioned in Section 2, we apply register file assignment for every live range, and then invoke register allocator on each register file. If the register allocator determine the interference graph is not K-colorable, the spill code is inserted into code sequence. In general, *Register Pressure* is the major factor that leads the effect of K-coloring process. It comes from the number of physical registers (K colors), the amounts of live ranges (Nodes), the interferences of live ranges (Edges), and other potential hardware constraints. The higher register pressure it is in the graph, the less possibility to make coloring succeed. This also implies a simple idea: High register pressure causes more spill code than low register pressure.

In distributed register file design, although it reduces the number of hardware ports and decreases die size area, it leaves another impact to compiler that there are only few registers can be used for each register file. Hence, the register pressure is easily getting high and spill code is raised frequently because of the low number of registers. Based on Chaitin's algorithm, following improvements [8, 9] develop spilling heuristic and utilize rematerialization to reduce spill code. Later work, Kolt and Bergner [20, 21] futher narrowed the interference live ranges into interference regions. Also, with precise knowledge of registers availability, spill code motion [22] partially allocating registers by their precise availability.

While existing methods have made significant strides in reducing spill code, they still basically address the case of a single bank of unified registers. Their allocation algorithms are not sufficient to handle distributed live ranges on different register files where the register allocator can do nothing for those unallocated live ranges but spilling. However, under observation, it reveals that the register file assignment is local-favorable to pick up register file class because its purpose is to minimize the communication cost [11, 12]. It always lacks of precise knowledge of register pressure so that it sometimes causes too many live ranges crowded in one register file.

The situation encourages us to think about a new thought for register file architecture. The idea is to spill (shuffle) a live range into other register files which are estimated to still have sufficient free registers to hold. Figure 3 shows the basic idea.

Suppose the live range in the middle of Figure 3 is the one which is chosen to do spilling, in which the black circle denotes *def* point and the white circle denotes *use* in the operations. Traditionally, as it is shown at left-hand-side in Figure 3, the spilling process would do spill-out (i.e. memory store) to memory after the definition and get the value by doing spill-in (i.e. memory load) from memory before every use. On the contrary, rather than spilling out/in to/from memory, the idea at right-hand-side employs data communication instructions to move live range from D Register File to A Register File. It is obvious that the second strategy is much better since the cost of memory access is always expensive.

However, to move live range into A Register File may cause it to produce new spill code due to the increasing of register pressure of it. There are supposed

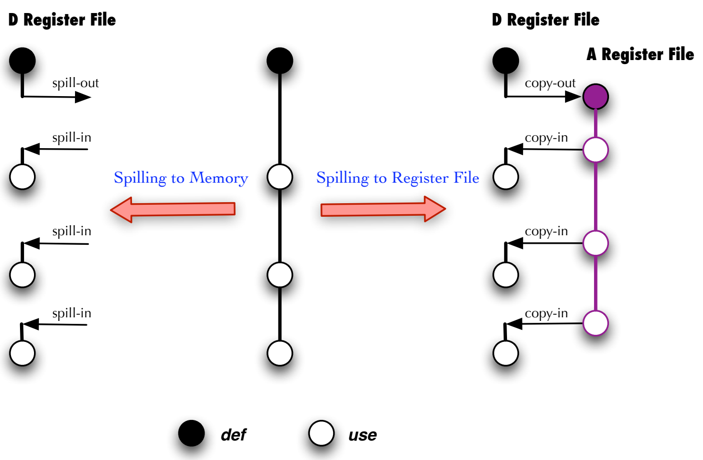

Fig. 3. Memory spilling and register file spilling

to have some cost model or algorithm to estimate which live range should be picked up and which register file could be used as spilled destination. Those cost model and algorithm are described in the following subsections.

In addition, the example in Figure 3 can be considered as a de-generalized case from the viewpoints of expression rematerialization. In the more general term, one can further perform a pre-calculation ahead of time when the expression to be re-computed is still in the live range. We then try to find a free register in different register banks to save the value. This can provide further opportunity for a code motion opportunity for the generation of spill codes, while attempting to find the most profitable candidate in another bank for a spill location.

3.2 Register File Spilling

In the subsection, we would like to introduce *register file spilling*, an alternative spilling for the architecture of distributed register files. In the Chaintin-style allocator, when coloring fails, it would spill virtual registers to memory. We call such spilling *memory spilling*. For the architecture with distributed register files, since the register pressure of register files varies, its spilling can employs the natrual of distributed feature and spill virtual registers to register files instead.

One of the essential issues in the spilling is to estimate the *register pressure* of each register files since register file spilling would decrease the register pressure of one register file and increase that of another. In the paper, there are two methods of the register pressure estimation for register file spilling. Both the methods are based on the *global interference graph*. For the register allocator of distributed register files, it builds the interference graphs for each register file. That is, every register file has its interference graph built on interference between

Algorithm 1 Estimate the Register Pressure

```
/* Higher number means lower register pressure. */
procedure numFreeRegs($liveRange, regFile, method$)
    /* Initialization */
    for all $regFile$ do
        // $regMembers[regFile]$ is a set of the registers
        // belongs to $regFile$.
        $regs[regFile] \leftarrow regMembers[regFile]$
    end for
    /* The first method: the global interference degree (GID).
        The second method: the global interference degree
        and register availability (GID/RA). */
    if $method =$ GID then
        for all $regFile$ do
            $regs[regFile] \leftarrow regs[regFile]$
        end for
    else if $method =$ GID_RA then
        /* Check the register availability. */
        $itfLiveRanges \leftarrow liveRange.globalItfLiveRanges$
        for all $itfLiveRange \in itfLiveRanges$ do
            $reg \leftarrow itfLiveRange.reg$
            if $reg \neq$ NONE then
                $regs[regFile] \leftarrow$ differElement($regs[regFile], reg$)
            end if
        end for
    end if
    for all $regFile$ do
        $free[regFile] \leftarrow$ size($regs[regFile]$)
    end for
    /* Check the global interference degree. */
    for all $itfLiveRange \in itfLiveRanges$ do
        $reg \leftarrow itfLiveRange.reg$
        if $reg =$ NONE then
            $free[regFile] \leftarrow free[regFile] - 1$
        end if
    end for
    $num \leftarrow free[regFile]$
    if $num \leq 0$ then
        $num \leftarrow 0$
    end if
    return $num$
end procedure
```

Algorithm 2 Code Motion of the First Definition and Last Use

```
procedure moveFirstDefLastUse($liveRange$)
    $liveRange \leftarrow$ moveFirstDef($liveRange$)
    if $liveRange \neq liveRange$ then
        $liveRange \leftarrow liveRange$
    else
        $liveRange \leftarrow$ moveLastUse($liveRange$)
    end if
    return $liveRange$
end procedure

procedure moveFirstDef($liveRange$)
    $firstDef \leftarrow liveRange.firstDef$
    $nextRef \leftarrow$ nextReference($liveRange$)
    if $(firstDef + 1) \neq nextRef$ then
        $firstDef = firstDef + 1$
    end if
    $liveRange \leftarrow$ updateFirstDef($liveRange, firstDef$)
    return $liveRange$
end procedure

procedure moveLastUse($liveRange$)
    $lastUse \leftarrow liveRange.lastUse$
    $prevRef \leftarrow$ prevReference($liveRange$)
    if $(lastUse - 1) \neq prevRef$ then
        $lastUse = lastUse - 1$
    end if
    $liveRange \leftarrow$ updateLastUse($liveRange, lastUse$)
    return $liveRange$
end procedure

procedure beforeMove($liveRange$)
    $dupLiveRange \leftarrow liveRange$
    removeLiveRange($liveRange$)
    addLiveRange($dupLiveRange$)
    return $dupLiveRange$
end procedure

procedure afterMove($liveRange, dupLiveRange$)
    removeLiveRange($dupLiveRange$)
    addLiveRange($liveRange$)
end procedure
```

Algorithm 3 Register File Spilling

```
/* Determine memory spilling or register file spilling.
   If register file spilling does, determine which register file
   to be spilling position. */
procedure chooseSpill($liveRange, method$)
    $spillCand \leftarrow$ NONE
    $minCost \leftarrow$ MAX
    $bestLiveRange \leftarrow$ NONE
    $dupLiveRange \leftarrow$ beforeMove($liveRange$)
    $changed \leftarrow$ TRUE
    while $changed =$ TRUE do
        for all $regFile$ do
            $free \leftarrow$ numFreeRegs($regFile, dupLiveRange, method$)
            $cost \leftarrow$ regFileSpillCost($dupLiveRange, regFile$)
            if $free > 0$ and $cost < minCost$ then
                $spillCand \leftarrow regFile$
                $minCost \leftarrow cost$
                $bestLiveRange \leftarrow dupLiveRange$
            end if
        end for
        $dupLiveRange \leftarrow$ moveFirstDefLastUse($dupLiveRange$)
        if $dupLiveRange = dupLiveRange$ then
            $changed \leftarrow$ FALSE
        end if
    end while
    if $bestLiveRange \neq$ NONE then
        $liveRange \leftarrow bestLiveRange$
    else
        $liveRange \leftarrow liveRange$
    end if
    afterMove($liveRange, dupLiveRange$)
    if $spillCand \neq$ NONE then
        regFileSpill(liveRange, spillCand)
    else
        memSpill(liveRange)
    end if
end procedure

/* Get the communication cost between the register files. */
procedure regFileSpillCost($liveRange, regFile$)
    $src \leftarrow liveRange.regFile$
    $dst \leftarrow regfile$
    $cost \leftarrow commCost[src][dst]$
    return $cost$
end procedure
procedure regFileSpill($liveRange, regFile$)
    <Similar to spilling in the Chaitin-style allocator,
     but replace all loads/stores with copy-ins and copy-outs.>
end procedure

procedure memFileSpill($liveRange$)
    <Same as spilling in the Chaitin-style allocator>
end procedure
```

live ranges with same register file. We call the graph the *local interference graph.* However, the local interference graph is not sufficient for register file spilling, which employs inter-regster-file communication to act as the spill-in and spill-out. To overcome this issue, we would like to include the *global interference graph* into the allocator. Unlike the local interference graph built on interference within a register file, the global graph build on interference across register files.

With the global interference graph, the *global interference degree* can provide rough estimation of the register pressure. Though, the first estimation, called the GID method, of the global interference degree seems conservative for register file spilling. Therefore, we propose the second estimation in which we enhance the estimation with the *register availability*, called the GID/RA method. Algorithm 1 reveals the details the two methods of register pressure estimation. The procedure numFreeRegs returns the number of free registers. If the number is higher, the register pressure is lower. At first, the procedure retrieves the register set for each register file. While both methods checks the global interference degree later, only the GID/RA method checks the register availability, which is computed on the *colored* global live ranges and presents coloring outcome in this iteration.

With the register pressure estimation, register file spilling can use it to determine whether it is worthy to utilize other regsiter files. Algorithm 3 describes the main flow of register file spilling. The first loop calls moveFirstDefLastUse in Algorith 2 to sink the first definition or hoist the last use into all the possible positions. For each of the potential live ranges, the procedure uses chooseSpill, numFreeRegs and regFileSpillCost to estimate the register pressure and spill cost of each register file, and records which live range can be the most benifitial in these estimation. Here, the higher value returned by numFreeRegs means lower register pressure. If numFreeRegs's of all the register files are zero, memory spilling is more suitable for the live range than register file spilling. Otherwise, there is one register file candidate at least for spilling. When register file spilling is the choice, the procedure selects the suitable register file for spilling, whose regFileSpillCost is minimum among the register files with positive numFreeRegs in the previous estimation.

4 Experiment

This section describes our preliminary experiments on register file spilling. All the experiments were performed with Open64-based compiler and on the cycle-accurate instruction set simulator, provided by STC/ITRI (SoC Technology Center of Industrial Technology Research Institute) in Taiwan, of the PAC DSP.

4.1 Infrastructure Design

Our compiler has many tuned optimizations for the PAC DSP. Some are introduced for the complicated communication. Because the PAC DSP has a clustered organization with distributed register files, the data movement between register files in PAC DSP can be classified into intra-cluster and inter-cluster communications. In that case, the common cluster-assignment problem for VLIW processors [17–19] does not directly apply to this design. Complicated communication scheme in the PAC DSP makes it desirable to have RFA in handling communications. including PALF-LRFA [11, 12], LC-GRFA [27, 28], and SA-RFA [29].

In addition to RFA, the copy propagation in the original Open64 could also be modified [26]. Due to the non-uniform distributed register file structure in the PAC DSP, conventional copy propagation might degrade the performance. In the modified optimization, the communication cost model was derived for copy propagation, which was based on the cluster distance, register port pressure, and movement type of register sets. The model was used to guide data flow analysis for better performance on the PAC DSP architecture. Moreover, since the PAC DSP has other architecuture features, other optimizations like software pipelining [30], loop nest optimization, and subword optimization, have been also improved for the processor.

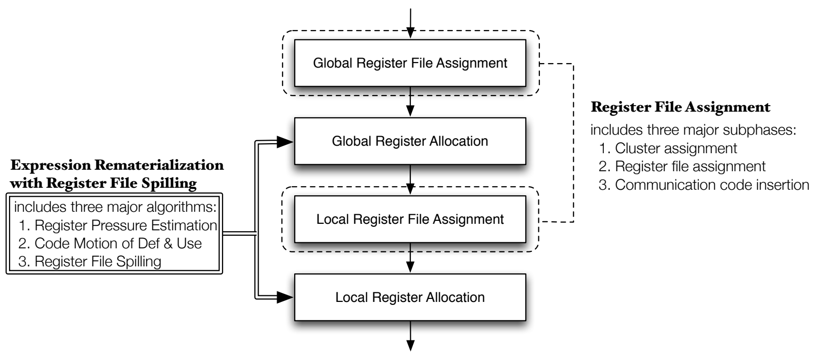
Global Register File Assignment
Global Register Allocation
Local Register File Assignment
Local Register Allocation
Register File Assignment
includes three major subphases:
1. Cluster assignment
2. Register file assignment
3. Communication code insertion
Expression Rematerialization with Register File Spilling
includes three major algorithms:
1. Register Pressure Estimation
2. Code Motion of Def & Use
3. Register File Spilling

Fig. 4. Expression Rematerialization in Open64 compiler infrastructure

Since this compiler is derived from Open64 compiler, it divides register allocation into Global Register Allocation (GRA) and Local Register Allocation (LRA) as well. The mentioned RFA has to be extended into GRFA and LRFA for GRA and LRA, respectively. Then our work is an extension built in GRA and LRA, as shown in Figure 4

4.2 Experimental Result

The spilling methods are implemented in LRA, and we use Figure 5 and Figure 6 to illustrate our experimetal result on the DSPStone [23] benchmark suite. In the figures, it compares the speedup and the numbers of memory loads/stores and register copies under the three methods: memory spilling, register file spilling with the GID method, and register file spilling with the GID/RA method. They were mentioned in Section 3.2 and the last two were estimated on global interference graph. In Figure 5, the fisrt column of Figure 5 specify the programs compiled. The second and third columns give the number of memory loads/stores and register copies for memory spilling. The fourth column shows the speedup of the program, also for memory spilling. Note that in the results we use memory spilling as the baseline for the speedup. The columns from the fifth to the tenth give the same information as column 2, 3, and 4 for register file spilling with the GID method and register file spilling with GID/RA method respetively. As to Figure 6, we use the figure to emphasize the speedup under different methods.

For example, the resuls of *fir2dim* is shown in the seventh row. With memory spilling, register allocation in compiling *fir2dim* needs 464 memory loads/stores and none of register copies. When register file spilling with GID is used, the number of loads/stores is reduced to 342 and that of copies is increased to 118. Compared to memory spilling, *fir2dim* get 109% speedup. Then, when register file spilling with GID/RA is used, the number of loads/stores is reduced to 334 and that of copies is increased to 126, and the speedup is increased to 109%.

Figure 5 shows that register file spilling tranfers part of loads/stores into copies. Thus, due to less lateny required for copies, register file spilling causes less cycles in total and higher speedup. For *n_complex_updates*, the GID/RA method transfers more loads/stores into copies than the GID method. As it transfer more into copies, the GID/RA method can provide better speedup. Register file spilling could improve lesser performance caused by unbalanced register file assignment. Some have unbalanced register file assignment among register files, such as *n_complex_updates*. Since the unbalanced assignment would centralize live ranges into one or two register files and make them of high pressure, these programs are suitable for register file spilling. For *n_complex_updates*, the GID and GID/RA methods adjust the unbalance and boosts the speedup of *n_complex_updates* to 150% and 160% respectively.

Program	Memory Spilling			Reg File Spilling (GID)			Reg File Spilling (GID/RA)		
	memory	register	speedup	memory	register	speedup	memory	register	speedup
biquad_N_sections	32	0	100%	18	14	107%	18	14	107%
biquad_one_section	13	0	100%	0	13	101%	0	13	101%
complex_multiply	23	0	100%	0	23	110%	0	23	110%
complex_update	9	0	100%	0	9	101%	0	9	101%
convolution	0	0	100%	0	0	100%	0	0	100%
dot_product	0	0	100%	0	0	100%	0	0	100%
fir2dim	464	0	100%	342	118	109%	334	126	109%
fir	2	0	100%	0	2	100%	0	2	100%
lms	0	0	100%	0	0	100%	0	0	100%
mat1x3	8	0	100%	0	8	110%	0	8	110%
matrix1	0	0	100%	0	0	100%	0	0	100%
matrix2	0	0	100%	0	0	100%	0	0	100%
n_complex_updates	63	0	100%	21	43	150%	11	53	160%
n_real_updates	0	0	100%	0	0	100%	0	0	100%
real_update	0	0	100%	0	0	100%	0	0	100%

Fig. 5. Speedup and numbers of memory loads/stores and register copies

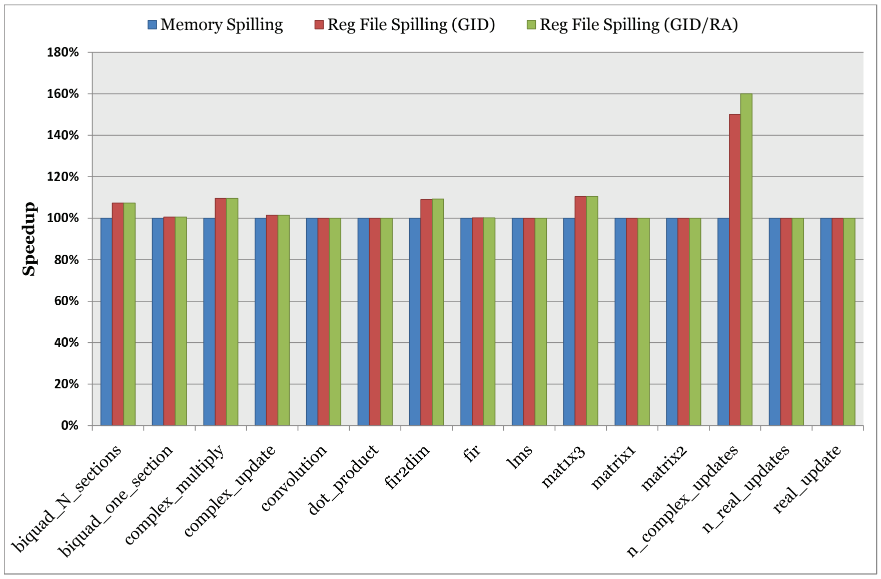

Fig. 6. Speedup chart

5 Conclusion

Embedded DSP processors are currently designed for high instruction level parallelism. The techniques used in their designs commonly tend to include a clustered/distributed register files architecture. Although this raises impact for devising compiler optimizations, it provides an optional destination for spill code in the register allocation phase.

In this work, we developed and implemented a new heuristic approach of spill code reduction for PAC VLIW DSP processors with highly-distributed register files. The approach called *register file spilling* utilizes differences of register pressure between register files. *Global interference graph* and *register availibility* are used to assist register file spilling in register file spilling decision. As it transfers memory loads/stores into register copies and causes less cycles in spilling, register file spilling increases performance up to 125%. The experimental evaluation of the benchmark programs indicates that our methodology is able to achieve expression rematerialization which effectively reduces spill code to improve the performance. Since the method is designed for LRA, our future work would focus on an extened method for GRA.

References

1. David Chang and Max Baron: Taiwan's Roadmap to Leadership in Design. *Microprocessor Report, In-Stat/MDR, December 2004.*

2. T.J. Lin, C.C. Chang. C.C. Lee, and C.W. Jen: An Efficient VLIW DSP Architecture for Baseband Processing. *Proceedings of the 21th International Conference on Computer Design, 2003.*
3. Tay-Jyi Lin, Chie-Min Chao, Chia-Hsien Liu, Pi-Chen Hsiao, Shin-Kai Chen, Li-Chun Lin, Chih-Wei Liu, Chein-Wei Jen: Computer architecture: A unified processor architecture for RISC & VLIW DSP. *Proceedings of the 15th ACM Great Lakes symposium on VLSI, April 2005.*
4. A. Capitanio, N. Dutt, and A. Nicolau: Partitioned Register Files for VLIW's: A Preliminary Analysis of Tradeoffs. *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25), pages 292–300, Portland, OR, December 1–4 1992.*
5. Ravi Sethi: Complete register allocation problems. *SIAM Journal on Computing, 1975.*
6. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein: Register allocation via coloring. *Computer Languages, 6:47-57, 1981.*
7. G.J. Chaitin: Register allocation and spilling via graph coloring. *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, pages 201-207, 1982.*
8. D. Bernstein, D.Q. Goldin, M.C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R.Y. Pinter: Spill code minimization techniques for optimizing compilers. *Conference on Programming Language Design and Implementation, 1989.*
9. P. Briggs, K.D. Cooper, and L. Torczon: Rematerialization. *Conference on Programming Language Design and Implementation, 1992.*
10. S. Kurlander and C. Fisher: Zero-cost range splitting. *Conference on Programming Language Design and Implementation, 1994.*
11. Yung-Chia Lin, Yi-Ping You, Jenq-Kuen Lee: Register Allocation for VLIW DSP Processors with Irregular Register Files. *International Workshop on Compilers for Parallel Computing, January 2006.*
12. Yung-Chia Lin, Yi-Ping You, Jenq-Kuen Lee: PALF: Compiler Supports for Irregular Register Files in Clustered VLIW DSP Processors. *Concurrency and Computation: Practice and Experience, 2007:19:1-16, Wiley, 2007.*
13. Yung-Chia Lin, Chung-Lin Tang, Chung-Ju Wu, Jenq-Kuen Lee: Compiler Supports and Optimizations for PAC VLIW DSP Processors. *Languages and Compilers for Parallel Computing, 2005.*
14. Yi-Ping You, Ching-Ren Lee, Jenq-Kuen Lee: Compilers for Leakage Power Reductions. *ACM Transactions on Design Automation of Electronic Systems, Volume 11, Issue 1, pp.147-166, January 2006.*
15. Yi-Ping You, Chung-Wen Huang, Jenq-Kuen Lee: A Sink-N-Hoist Framework for Leakage Power Reduction. *ACM EMSOFT, September 2005.*
16. Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, Jenq-Kuen Lee: Interprocedural Probabilistic Pointer Analysis. *IEEE Transactions on Parallel and Distributed Systems, Volume 15, Issue 10, pp.893-907, October 2004.*
17. Ellis JR: Bulldog: A compiler for VLIW Architectures. *MIT Press: Cambridge, MA, 1986.*
18. Capitanio A, Dutt N, Nicolau A: Design considerations for limited connectivity VLIW architectures. *Technical Report TR59-92, 1993.*
19. Ozer E, Banerjia S, Conte TM: Unified assign and schedule: A new approach to scheduling for clustered register files micro architectures. *Proceedings of the 31st Annual International Symposium on Microarchitecture, November 1998.*

20. P. Kolte and M.J. Harrold: Load/Store range analysis for global register allocation. *Proceedings of Programming Language Design and Implementation, 1993.*
21. P. Bergner, P. Dahl, D. Engebretsen, and M.O'Keefe: Spill code minimization via interference region spilling. *Proceedings of Programming Language Design and Implementation, 1997.*
22. Akira Koseki, Hideaki Komatsu, and Toshio Nakitani: Spill Code Minimization by Spill Code Motion *Proceedings of Parallel Architectures and Compilation Techniques, 2003.*
23. V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr: DSPstone: A DSP-oriented benchmarking methodology. *Proceedings of the International Conference on Signal Processing and Technology, 715–720, 1995.*
24. R. Leupers: Instruction scheduling for clustered VLIW DSPs. *Proceedings of International Conference on Parallel Architecture and Compilation Techniques, pp.291-300, October 2000.*
25. Yi Qian, Steve Carr, Philip H. Sweany: Optimizing Loop Performance for Clustered VLIW Architectures. *International Conference on Parallel Architectures and Compilation Techniques, September 2002.*
26. Chung-Ju Wu, Sheng-Yuan Chen, and Jenq-Kuen Lee: Copy Propagation Optimizations for VLIW DSP Processors with Distributed Register Files *Languages and Compilers for Parallel Computing, 2006.*
27. Chia-Han Lu, Yung-Chia Lin, Yi-Ping You, and Jenq-Kuen Lee: A Local-Conscious Global Register Allocator for VLIW DSP Processors with Distributed Register Files *International Workshop on Compilers for Parallel Computing, January 2007.*
28. Chia-Han Lu, Yung-Chia Lin, Yi-Ping You, and Jenq-Kuen Lee: LC-GRFA: Global Register File Assignment with Local Consciousness for VLIW DSP Processors with Non-uniform Register Files. Accepted, *Concurrency and Computation: Practice and Experience, Wiley.*
29. Yung-Chia Lin, Chung-Lin Tang, Chung-Ju Wu, Ming-Yu Hung, Yi-Ping You, Ya-Chiao Moo, Sheng-Yuan Chen, and Jenq Kuen Lee: Compiler Supports and Optimizations for PAC VLIW DSP Processors. *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing, 2005.*
30. Chung-Kai Chen, Ling-Hua Tseng, Shih-Chang Chen, Young-Jia Lin, Yi-Ping You, Chia-Han Lu, Jenq-Kuen Lee: Enabling Compiler Flow for Embedded VLIW DSP Processors with Distributed Register Files. *ACM SIGPLAN Notices, Volume 42, Issue 7, Pages: 146 - 148, 2007. (ACM LCTES 2007 Issue)*