Compiler Supports and Optimizations for PAC VLIW DSP Processors *

Yung-Chia Lin Chung-Lin Tang Chung-Ju Wu Ming-Yu Hung
Yi-Ping You Ya-Chiao Moo Sheng-Yuan Chen Jenq-Kuen Lee

Department of Computer Science
National Tsing-Hua University
Hsinchu 300, Taiwan
Email: {yclin, cltang, jasonwu, myhung, ypyou, ycmo, sychen, jklee}@pllab.cs.nthu.edu.tw

Abstract. Compiler is substantially regarded as the most essential component in the software toolchain to promote a successful processor design. This paper describes our preliminary employment of the Open Research Compiler (ORC) infrastructure on a novel VLIW DSP processor (known as PAC DSP core) and its specific compilation and optimization design. The PAC DSP processor exceedingly utilized port-restricted, distinct partitioned register file structures in addition to the heterogeneous clustered datapath architecture to attain low power consumption and reduced die size; however, these architectural features lend new challenges to the compiler construction. As part of an effort to deal with the challenges of efficient code generation for PAC DSP, the register allocation scheme developed in this work and other retargeting optimization phases are also presented. Results indicated that our compiler development for PAC DSP could gives an early estimation of architecture performance so that refinements of architectures are possible with the software feedbacks. Our experiences in designing the compiler support for heterogeneous VLIW DSP processors with irregular resource constraints may benefit those who have interests in the compiler construction for the similar architectures.

1 Introduction

Optimizing compiler development has always been the key factor of building a productive environment for new embedded processors and SOC chips. Since high-end embedded processor design is moving towards exploiting intensively instruction level parallelism (ILP) and incorporating many advanced application specific features, the complexity of compilers for these advanced processors grows into immensity, which demands more long-term development efforts and

* This paper is submitted to *LCPC 2005*. The correspondence author is Jenq Kuen Lee. His e-mail is jklee@cs.nthu.edu.tw, phone number is 886-3-5715131 EXT. 3519, FAX number is 886-3-5723694. His postal address is Prof. Jenq- Kuen Lee, Department of Computer Science, National Tsing-Hua Univ., Hsinchu, Taiwan.

extremely larger manpower than before. Hence, designing code generation supports and optimizations based on open-source compiler infrastructures instead of developing everything from scratch are the alluring trend to shrink the delivery time of the compiler for a newly designed processor.

ORC [1] is an open-source compiler infrastructure released from Intel, which is the successor of Pro64 [2], the open-source compiler project for IA-64 by SGI in May 2000. Since the Pro64 was originally evolved from the commercial SGI MIPSPro compiler suite which had been developed by SGI as the production compiler for a long period, ORC has incorporated most of the optimization techniques of industry strength so far. Undoubtedly, it is expected to provide as a well-stabilized base infrastructure for further research works and as the satisfying foundation for any new target porting works. In addition, ORC/Pro64 has already achieved an excellent porting status for IA-64, enabling the compiler to generate codes with good performance by utilizing numbers of EPIC/VLIW architectural advantages. As modern VLIW DSP processors incorporate many of the advanced architecture features likewise, it looks interesting and promising to explore possible ORC employments for VLIW DSP processors.

In this paper, we study the issue of supporting ORC/Pro64 platforms for VLIW DSP Processors. We present our experiences in the development of code generation support and preliminary optimization design for a novel 32-bit VLIW DSP processor designed with several new architectural features, such as distinct partitioned register files with significant port restriction [3]. The target processor, named as Parallel Architecture Core (PAC) DSP [4], is being developed from scratch by SOC Technology Center at Industrial Technology Research Institute in Taiwan with several joint efforts of academic research works [5,6]. PAC DSP is natively designed to meet the high-performance computing requirement of multimedia and the low power consumption demand of mobile system. In the early design stage in developing PAC DSP, several tuning iterations may be needed between architecture and software designs by co-exploration, to attain the finest result with satisfactory in the end. As a result, our work gave a preliminary estimation of architecture performance so that refinements of architectures can be established according to the software feedbacks. We proposed effective register allocation policies in the compiler framework to support the specific register file organizations in PAC architectures, the peephole optimization for the architecture, and the essential modeling for the architecture to support loop-nest optimizer. Moreover, we revealed evident steps in employing our development works on top on the ORC infrastructure for PAC DSP, which is a series of our research work to develop high-performance and low-power compiler toolkit for VLIW DSP processors and SOC platforms [7,8]. This paper provides the feasible information to develop essential compiler supports for heterogeneous clustered VLIW architectures with port-restricted, distinct partitioned register file structures, which may benefit anyone who has interests in developing compilers for novel VLIW DSP processors with similar architectures.

The remainder of this paper is organized as follows. Section 2 first introduces the target architecture of PAC DSP. Section 3 then describes the compilation

challenges for the architecture. Next, the development of code generation and preliminary optimizations for PAC DSP, including the specific design for the architecture, are presented in Section 4. Experimental results of the early stage evaluation are then illustrated in Section 5. Finally, Section 6 concludes this paper.

2 An Insight into PAC DSP Architectures

PAC DSP is a 32bit, fixed-point, VLIW digital signal processor core which can be used as a co-processor in a multi-core SOC platform (like TI's OMAP platform [9]) or employed as stand-alone solutions for any DSP system. The PAC DSP originally features a clustered VLIW architecture which boosts scalability, a feature-rich instruction set with SIMD operation support, a variable-length instruction encoding scheme, large number of registers which are arranged as innovative heterogeneous and distinct partitioned register file structures.

Being unlike symmetric architectures of most DSP processors available nowadays, the PAC DSP processor is constructed as a heterogeneous five-way issue VLIW architecture, comprised of two integer ALUs (I-unit), two memory load/store units (M-unit), and the program sequence control unit/scalar unit (B-unit) which is mainly in charge of control flow instructions like branch and jump. The M- and I- units are organized in pairs, and each pair contains exactly one M-unit and one I-unit to form a cluster arrangement with associated register files. It is apparent that each cluster is logically appropriate for one data stream processing, and the current design of PAC DSP consists of two clusters to support maximum workload capacity of two concurrent data stream. But the scalability of the cluster design in PAC DSP could allow the processor to easily involve more clusters to handle larger data processing workload demand. The B-unit consists of two sub-components, the program sequence control unit, and the scalar unit, due to the hierarchical decoder design for variable-length instruction encoding in PAC DSP. The program sequence control unit, which primarily takes charge of operations of control flow instructions. The scalar unit, which is capable of simple load/store and address arithmetic, is placed separatedly from data stream processing clusters, with its own register file. The overall architecture is illustrated in Fig. 1.

As shown in Fig. 1, registers in PAC DSP are organized into four distinct partitioned register files and placed as cluster structures, to reduce wire connections between functional units and registers so that chip area and power consumption may be decreased. The A, AC, and R register files are private registers, directly attached to and only accessible by the M-, I-, and B-unit, respectively; D register files are shared within a cluster and can be used to communicate across clusters; only the B-unit, being able to access all D registers, is capable of executing such cross-copy operations to move data between clusters. The internal of the D register file is further designed to utilize the instructional port switching technology in order that reducing more wire connections between the shared functional units. The technology, being referred to the name as '*ping-*

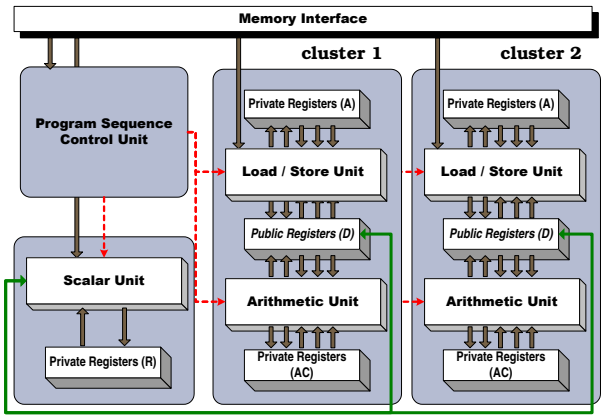

Fig. 1. The PAC DSP architecture illustration

pong register file structure', is that dividing one register file into two banks, and each bank can only be accessed mutual-exclusively by one functional unit at the same time. The instruction bundle encoding contains the information of which bank to be accessed for each functional unit so that the hardware can do port switching between register file banks and functional units, to attain the purpose of data sharing within a cluster. The advantage of such a '*ping-pong register file structure*' design is believed to consume less power due to its reduced read/write ports [10] while retaining an effective way of data communication capability. The illustration of the register file structure inside a data stream cluster is shown as Fig. 2.

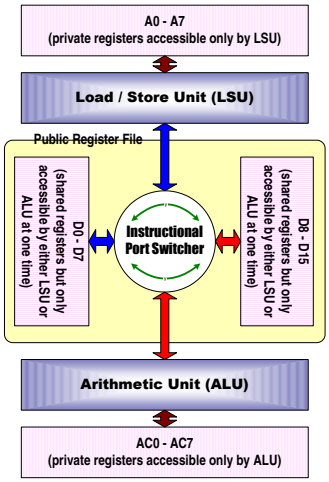

Fig. 2. The distinct partitioned register file organizations in PAC DSP processors

3 Code Generation Issues with PAC DSP Architectures

The PAC DSP incorporates various leading edge architectural features, attempting to take more opportunities for both high-performance and low-power; nev-

ertheless, this design introduces interference between valid code generation, instruction scheduling, and register allocation than typical VLIW architectures. Furthermore, these compilation issues impact the code optimizations between performance, size, and power consumption.

One of the most significant issues is caused by the '*ping-pong register file structure*'. As mentioned in Section 2, the PAC DSP features a heterogenous, distinct partitioned register file design with irregular port access constraints (referring to Fig. 1 and Fig. 2). Each cluster inside the architecture contains: A and AC register files, which is directly connected to the M-unit and I-unit respectively, and one D register files. Each D register file is divided into two banks which share a single set of access ports connecting to M- and I- units; in each VLIW instruction bundle, there is a bit-field that controls the access ports to be switched between the D register banks and the two FUs in each cluster. In other words, if the M-unit is accessing the first bank of the D register file, then the I-unit can only access the second bank at the cycle, and vice versa; accesses from two different FUs to the same D register bank are mutually exclusive in a cycle. In addition, each FU in the PAC DSP has different set of instructions that could be executed and each instruction has its own register access constraints. All of these irregular designs make more challenges in generating effective code while considering optimization issues. Conventional instruction scheduling policies and register allocation strategies are seldom applicable to the code generation for the PAC DSP architecture. For example, the short code sequence:

```
mov TN1, 1
mov TN2, 2
add TN3, TN1, TN2
```

moves two constants into two virtual registers, TN1 and TN2 and then takes an arithmetic operation on them. While observing the first two instructions, these two can be scheduled in parallel only if TN1 and TN2 are assigned registers from distinct D register bank; if both are assigned to the same D register bank, they can only be scheduled and issued sequentially. But '*ping-pong register file structure*' affects more than limiting the parallelism in the instruction scheduling. While further observing the third instruction, the instance becomes complicated. Since the last instruction in the code sequence refers TN1 and TN2, which are the results of the first two instructions, TN1 and TN2 must be in the register access range of the last instruction. Referring to the Fig. 3, without considering other hazards, there must be a copy instruction insertion before the last instruction if allocating TN1 and TN2 to different D register banks for parallelizing the first two instruction. Therefore, the advantage of parallelizing the first two instruction is counteracted by the insertion of the additional copy instruction and the generated code may be worse because the code size is larger than the case of allocating both TN1 and TN2 to the same D register bank. But allocating the same D register bank will always raise the register pressure of that bank when the compiler process the register allocation, and spilling from different register file will make different cost in the PAC DSP architecture, these cause more unpredictability of the combined effects of all code generation issues.

Currently, no past method can be used for PAC DSP to effectively produce the finest result before finalizing the instruction scheduling and the register allocation of all codes, to the best of our knowledge; the development of new compiler schemes to handle the issues caused by the innovative architecture of PAC DSP is undoubtedly necessary.

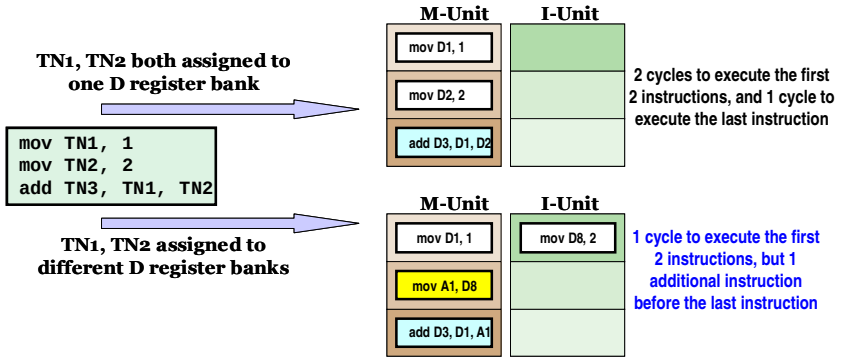

Fig. 3. The Illustration of interference caused by Ping-Pong register file structures.

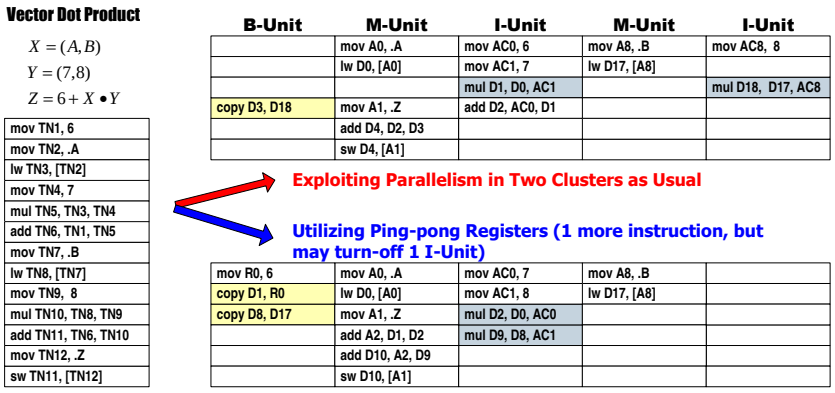

Fig. 4. An example of generating optimal scheduled codes across clusters

Another critical subject of how the register allocation interferences with both the instruction scheduling and the code generation is issued by the implementation of data communication across clusters in the PAC DSP architecture. The current version of PAC DSP require the code to explicitly issue a cross-cluster copy instruction to complete the data communication between clusters. Although the cross-cluster copy instruction is designed to be issued by the stand-alone scalar unit without occupying a slot in the clusters, the additional instruction insertion introduces additional data-dependency and data available latency for any code which is scheduled and distributed into two clusters. Fig. 4 gives an illustration of the two possible scheduling of code distributed on the two clusters (considering only major constrainsts for easier understanding), which both have their own benefit. As a result, it seems that the compiler for PAC DSP needs a well evaluation before generates code distributed into two clusters to

avoid the penalty of cross-cluster communication disadvantaging the parallelism of two clusters; however, the evaluation becomes more complicated and non-deterministic with the interference of the '*ping-pong register file structure*' issue. This makes more challenges to construct an good compiler for the PAC DSP architecture. The Table 1 summaries the current considered interference in PAC DSP compiler design, but not limits to these.

Table 1. Major interferences in the code compilation for the PAC DSP architecture

Code Generation	**Code Scheduling**	**Register Allocation**
Instruction Selection	Execution Unit Constraints	Register Bank Selection
Insert Copy to Cluster Communications	Register Access Constraints	Register Pressure
Insert Copy to Generate Valid Code	Instruction Latency	Vary Spill Cost for Different Register Bank
	Hardware Hazard	
Trade-Off between Performance, Code Size, and Power Consumption		

4 Compiler Supports for PAC DSP Processors

In this section, we describe our development works of applying compiler supports for the PAC DSP architecture. Our compiler prototype is based on the ORC infrastructure, which is constructed by modularized components that are ideal for putting incremental development achievements and optimization improvements on the compiler framework. Roughly speaking, the compilation procedure by ORC starts with processing by the front-ends, generating an intermediate representation (WHIRL IR) of the source program, and then feeding it in the back-end. Since WHIRL IR has five levels of representation forms, the back-end will invokes several components to perform a series of lowering processes and optimizations on the WHIRL IR before transforming WHIRL IR into CGIR which is a target specific low-level IR near the real instruction representation. The developing components for optimizations could optionally be activated on the WHIRL IR level include the inter-procedural analysis/optimizer, loop nest optimizer, global optimizer. The loop nest optimizer, which is one of our prioritized working items, is based on a cost model of code generation in ISA of PAC DSP. It is designed to perform optimizations related to locality, parallelization, and loop transformation.

After the WHIRL level processing, the back-end will invoke the code generator to transform the WHIRL IR into the CGIR. Besides register allocation, compilation modules may be activated to process the CGIR depending on the code optimization level before emitting the final codes. Fig. 5 illustrates PAC compiler phases, as it is extended based on our research innovations to include several new optimization/analysis modules that may benefit more for PAC DSP processors. These include probabilistic point-to analysis schemes, allocations distributed register clusters, low-power optimizations, and DSP-specific optimizations. Many of these new phases are based on our previous research innovations and we are in the process of integrating those technologies into this infrastructures. These schemes include low power optimizations [11–13] advanced pointer

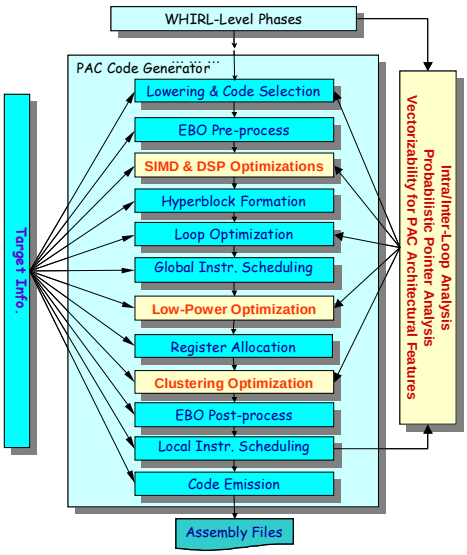

Fig. 5. The refinement of compiler code generation phases for PAC DSP processors

analysis/optimizations [14, 15], and DSP-specific optimizations [7]. Til now, our development of compiler support for PAC DSP is still an on-going effort. In this paper, we will first focus on the studies of supporting basic ORC infrastructures for PAC VLIW DSP processors.

4.1 Code Generation with Target Information Extension

The Target Information Table (Targ_info) in ORC is the most essential part to support the code generation by providing the parameterized data about the architecture and the ISA of the target processor. In this subsection, we present our employment from IA64 to PAC DSP and the improvement of the Target Information Table to support more flexible CGIR level processing and optimizations for the PAC DSP architecture. The original Targ_info is written in the constructs of C language, and then they are processed by the generator utilities to generate the actual source files of Targ_info library. The machine parameters described in the Targ_info library are referred everywhere in the codes of almost all CGIR-level components after the WHIRL-to-CGIR expansion phase; they are used to abstract the target machine dependent information and are distinguished from the compiler's algorithms to reduce the effort of compiler construction when changing the target machine.

To conform to the PAC architecture and minimize the complexity of instruction scheduling and register allocation, the same instruction for different functional units is defined to be distinct in Targ_info, i.e. the register allocation range can be determined by the instruction used so that the management of the register file usage may be much clear for the implementation of register allocators. As PAC DSP processor has two clusters with no shared register files, some special purpose registers that are treated as always available to all operations (e.g., stack pointer and frame pointer) need to be defined in both clusters

and the code generation must implement the duplication of these register content to meet calling conventions. Moreover, to overcome the disadvantage of the unit-binded instruction definition, we design new descriptions that can assist the CGIR-level phases to choose the appropriate instruction in different units to complete the same semantics. The hazard descriptions and handler functions in original Targ_info are also fully redesigned to manipulate multiple hazards of multi-type for single instruction because the constraints of PAC DSP are more complicated than the original IA-64 architecture.

The further adaptation of WHIRL-to-CGIR code generation functions includes designing the optimal instruction selection which depends on the optimization policies to produce preferable CGIR operations for the PAC DSP architecture, and implementing the specific handler for the PAC DSP architectural deficiency in generating correct code to follow C language conventions. For example, the typical passing parameters to functions is through a register stack or rotating registers; the PAC DSP, not supporting a shared register stack and convenient register passing mechanisms, requires redesign of the parameter passing mechanism in the code generation part to employ a runtime memory stack instead.

4.2 Register Allocation Scheme for PAC DSP Processors

The rationale of PAC's highly-partitioned register file design is, of course, to lower register file port counts in order to avoid the slow access speed and high power consumption of an unified register file, though at the expense of an irregular architecture. With this design, the phase-interaction between register allocation and instruction scheduling becomes a critical problem, elevating this classical phase ordering issue in compiler code generation. Not only does the clustered design make register access across clusters an additional issue, but the switched access nature of the 'ping-pong' register files makes the details of register assignment and instruction scheduling dependent on each other, as shown earlier in Section 3.

Our current proposed solution to this problem, is to add a new instruction scheduling phase before register allocation/assignment by *simulated annealing* (SA). The design is extended from Leupers' work [16] and our initial implementation [8], using a hybrid instruction scheduling/cluster assignment algorithm to iteratively approach the near-optimal result. The algorithm roughly operates by first generating a random cluster partitioning of instructions; a modified List-Scheduler (LS) then schedules the partitioned instructions, inserting/managing cross cluster communications along the way.

The following iterations then make a random change to the partitioning state, and re-run the LS to schedule again. The LS returns the obtained schedule length of the instructions as the 'energy' value used in an usual simulated annealing optimization process, representing an evaluation of the current partitioning state. Depending on that improvement is gained or not, the random change may be retained or discarded. This process is iterated until the energy/evaluation falls to

be under some thresholds, where we are confident that the obtained optimization state is of sufficient quality.

Hybrid Instruction Scheduling/Register File Assignment by Simulated Annealing

Input: n operations to be scheduled
Output: Schedule of the n instructions and a register file assignment (RFA) map:
VR: set of all virtual registers, RF: set of register files
$RFA\_map = \{(v_1, f_1), (v_2, f_2), ...\}$ $v_i \in VR, f_i \in RF$

1. Choose a schematic register file placement(e.g. 1 cluster, 2 clusters, ...).
2. Make initial register file assignments: randomly assign each virtual register to any of the wanted register files, and record in RFA_map.
3. Given RFA_map, run **PAC_Scheduler**, and set $sched\_len$ to the computed total schedule length in cycles.
4. Set initial values for:
   $threshold$: threshold value for the simulated annealing process.
   $energy$: initial energy, larger than $threshold$.
   p_test: a probability test value p_test $(0 < p\_test < 1)$.
5. Repeat the following steps while $energy > threshold$:
   5a. Make change in RFA_map:
   randomly choose a virtual register, and assign it to a different register file.
   optionally change the schematic register file placement.
   5b. With the new RFA assignment change, run **PAC_Scheduler** again, and set new_sched_len to the new count of total schedule length.
   5c. Adjust $energy$, $sched\_len$, and RFA_map by the following rules:
   If $new\_sched\_len < sched\_len$ then
   decrease $energy$, set $sched\_len$ to new_sched_len,
   and keep the new RFA changes made in step 5a.
   If $new\_sched\_len \geq sched\_len$, get random number $0 \leq R \leq 1$:
   If $R > p\_test$ then
   decrease $energy$, set $sched\_len$ to new_sched_len,
   and keep the changes made in step 5a.
   If $R \leq p\_test$ then
   increase $energy$ and revert changes made in step 5a.
6. Optionally choose another schematic register file placement, and repeat steps 2–5 to select the better results.
7. Retain the final schedule and RFA_map as the output results.

Fig. 6. The high-level simulated annealing algorithm

Adapting this simulated annealing solution for the PAC DSP involves changes in the formulation of optimized state: our search is for register file assignments in the chosen schematic placement (as the search space) for virtual registers, instead of the original bi-partitioning of the instructions. The above algorithm in Fig. 6 is the high-level simulated annealing algorithm. it controls the scheduler, which does fine-grain sequencing of operations, and returns the schedule length as the evaluation of the current optimization state. The two optional procedures in the algorithm could let the compiler dynamically control the iterative scale and limit the register file usage to coordinate with other optimizations; they may also improve the overall register allocation speed.

Fig. 7 illustrates more details of the scheduler algorithms. In general, the overall operation of the algorithm is to proceed through the state space, making changes according to the feedback obtained from the LS. The assignment of register files will improve progressively throughout the SA iterations, with respect to the schedulable length of the instructions. A final register allocator is then run to allocate and assign hardware registers, which is guided by the register file assignments (RFA_map).

The *PAC_Scheduler* Algorithm

Input: $ReadyList$ of operations to be scheduled
RFA_map, in the form of a function $RegisterFile : VR \rightarrow RF$
Output: Schedule of the n instructions, and the schedule length

```
While ReadyList is not empty:
  Select operation Op from ReadyList
  Find earliest cycle Cycle we can schedule Op
  While Op is not scheduled:
    Examine available resources in Cycle, and:
    For each register operand oi of Op:
      If no resources available to access RegisterFile(oi):
        Enumerate possible copy sequences to transfer oi to an accessible register file
        For each copy sequence cpseq
          If cpseq is schedulable in prior cycles:
            Feasible(cpseq) = true
    If for each register operand oi of Op we have resources to access RegisterFile(oi),
    or we have some c such that Feasible(c):
      Schedule Op into Cycle, advance ReadyList
      Break from inner loop
    else
      Increment Cycle
Return length of schedule
```

Fig. 7. The scheduler/evaluation algorithm

4.3 Peephole Optimizer for PAC DSP Processors

The Extended Block Optimizer (EBO) is a peephole optimizer which performs some simple optimizations on the scope of extended basic blocks at CGIR level. Extended blocks are constructed by choosing a sequence of blocks that may contains branch-out, but can only be executed from the start of the first block in the sequence. Instructions are processed in the forward direction through each block and the block's successor list. New blocks are processed until a branch-to label is encountered, at which time processing backs up and attempts to take a different path down another successor list. The entry points to invoke EBO are, respectively, performing optimizations right after instruction translation, during unrolling and pipelining, after unrolling and pipelining, performing peephole optimizations on a region, and after register assignment. EBO performs optimizations such as forward propagation, common expression elimination, constant folding, dead code elimination and a host of special case transformations that are unique to the architecture of a particular machine. By doing these peephole optimizations, we can improve the performance and the code quality of the program.

In our case, there are still many situations that should be taken as consideration to rewrite routine as long as the implementation has something to do with machine-dependent architecture. Hence, our work not only does the refinement of basic peephole optimizations, but also intends to employ some techniques for supporting PAC DSP architecture. Table 2 shows the the designs for EBO on ORC for IA64 and on PAC DSP compiler, respectively.

Both compilers have implemented the basic peephole optimizations. However, due to that the PAC DSP uses the irregular register files and clustered architectures, illegal propagation may occur on several *TNs (Temporary Name)* which reside in different register files. Hence, as applying such basic peephole optimizations for PAC DSP, the major problem is that we cannot take all the

Table 2. EBO Refinement from ORC to PACDSP Compiler

EBO optimization	ORC for IA64	PAC DSP compiler
Forward Propagation	×	×
Common Expression Elimination	×	×
Constant Folding	×	×
Dead Code Elimination	×	×
Resolve Conditional Branch	×	-
Condition Redundant	×	-
Merge Memory Offset	-	×
Compound Operation Conversion	×	×
Subword Calculation	-	×
Dual Operation	-	×

TNs as registers in an unified register file to analyze their relationship. Among those basic optimizations (forward propagation, common expression elimination, constant folding, and dead code elimination), constant folding and dead code elimination are less affected by the impact of restricted register accessing and instruction inserting for valid code. But forward propagation and common expression elimination may highly relate to the speciality of the PAC DSP architecture, and require the analysis of the cluster/ping-pong information. For example, propagation between different clusters should be carefully marked as the may-illegal propagation of data flow. We setup some flags for a TN to handle such condition; each time we find a pair of a replaced TN and the TN being propagated, then store the cluster/ping-pong relationships as flags. Before the instruction scheduling, the flags can provide the information whether EBO should take propagation related optimizations for the extended basic blocks or not.

For taking more advantage of the architecture, we propose further machine-dependent optimizations on EBO phases of the PAC DSP compiler: they include Merge Memory Offset, Subword Calculation, and Dual Load/Store Operation. Merge Memory Offset is the one which utilizes the convenience of load/store instructions. Rather than wasting two instructions to do an actual load/store operation after the computation of the whole address of *base+offset*, we calculate the final address of memory and access data just by one instruction.

In addition, the ISA of PAC DSP includes a rich and general set of subword instructions to accelerate the process of lower precision operations. A subword in PAC DSP can be 8 or 16 bits long so that quad or dual subwords can be accommodated in a single register, which is 32 bits long. The challenge is to find a set of data-parallel computations that operate on lower precision data and map them onto packed or unpacked — PAC DSP provides instructions that operate on two 16-bit data which reside in two registers — subword instructions. A basic technique is to divide a loop into multiple loops with lower precision data. The first thing is that we need to extend the TN structure, e.g., to add a new field for data precision, so that we have the ability to determine which TNs, and thus operations, are primitives for subword operations. Moreover, a phase for packing subword operations into one compound instruction before the process of register allocation is required, so as to integrate subword optimizations into PAC DSP Compiler.

Finally, Dual LOAD/STORE instructions are powerful operations for accessing data from different memory address and then combine/separate the values

into/from two 32-bit registers simultaneously. When processing optimizations of dual operations, we need to reference the precision field mentioned previously, and have to examine the operand width of the processing data supposed to be. Thus, we are able to select the most suitable instructions for dual load/store operations.

4.4 Loop Optimization Phase for PAC DSP Processors

The loop-nest optimization (LNO) phase development inherits the traditional loop transformation techniques, for example, fusion, fission, tiling, unrolling, and unimodular transformation. The purposes of transformations are to make the optimized forms suitable with machine features, code generation, and low level optimizations. Three target-specific models, resource, latency, and register pressure are constructed for PAC DSP to estimate the best unrolling factor and tiling size for candidate loops at WHIRL-level. Depending on the issue rate, memory units and the amount of ALU in the PAC DSP architecture, we first determine the essential information, as Table 3 to model the basic processor parameters. By resource models, LNO estimates resource usages in each iteration of a loop from mapping tables about the equivalence between WHIRL operations and PAC DSP instructions.

_issue_rate = 4.5; *_num_mem_units = 2.5;*	We describe issue rate 4.5, because there are 2 ld/st units, 2 ALU units and one scalar unit. And the scalar unit is used mainly for control, so it is estimated only 0.5. For the same reason, we estimated memory unit at 2.5.

Table 3. Basic parameters to model PAC DSP processor

Next, in the estimating latency constraint phase, LNO builds a dependence graph for the loops in order to generate codes that are suitable for software pipelining. This graph can help calculate total latencies by observing each load and each store. For PAC DSP architectures, new modeling of integer operations is designed instead of the original ORC floating point considerations, to calculate the more accurate operation latencies. The latency value is then used in the scheduling of software pipelining optionally enabled in the later phase to optimize code for performance.

Finally, register pressure estimation policies are elaborated to well formula the effects of the irregular register file structures in PAC DSP. The clustered architecture characteristics of PAC DSP are also considered; the register pressure for a single cluster neglecting the possible inter-cluster interference is appraised at the initial. If the register pressure of any one cluster is too high, the interference of the two clusters are deliberated, to count the possible register resource usage and communication penalty while accessing cross-cluster content. Therefore, a cost model adapting for PAC cluster feature was proposed in Fig. 8. It showed that register pressure estimation affects the decision of loop transformation.

```
cycle = cycle_estimation(one cluster resource)
if( (RA+RD/2)>ERA && (RAC+RD/2)>ERAC )
      do no fission for candidate loop // can be scheduled in one cluster
else
      if( (2*RA+RD)>ERA && (2*RAC+RD)>ERAC )
            //cannot be scheduled in one cluster, so extra overhead should be considered.
            new cycle = cycle_estimation(two cluster resources) * r
                  if(new cycle > cycle)
                        do fission
                  else
                        do no fission for candidate loop
      else
            do fission

RA= number of A register in one cluster
RD= number of D register in one cluster
RAC= number of AC register in one cluster
ERA= estimated register count for addressing usage
ERAC= estimated register count for data usage
cycle= estimated executing cycles in one cluster
new cycle= estimated executing cycles in two clusters
r= cluster interference coefficient
```

Fig. 8. the register pressure cost model for loop transformation

5 Experimental Results

Preliminary experiments were done with the DSPstone benchmarks [17]. Since the PAC DSP compiler is still in progress, we only evaluated some stable optimization combinations for early stage performance evaluations our designs. All benchmark programs are compiled with three types of option combinations and disabling all other optimizations; they are the traditional-approach-based register allocation (TRA), the traditional-approach-based register allocation plus LNO and EBO (LNO+EBO), and the register allocation using the simulated-annealing approach (SARA), respectively. The TRA, which is a modification of the original ORC register allocation that assumes PAC DSP has only one unified register file containing all registers and inserts necessary codes to make register allocation result executable, is treated as the base reference in the comparison. Fig. 9 compares the speedup of DSP benchmarks on the later two options, LNO+EBO and SARA, with the numbers of -O0 (with the traditional approach based register allocation). As shown in Fig. 9, the performance gain for PAC DSP varies widely across different benchmarks with the average 1.78 speedup for LNO+EBO and the average 1.58 speedup for SARA. Though the integrated test of LNO+EBO plus SARA has not yet stable enough to exhibit the overall advantage, the results shows that our approaches in LNO, EBO, and register allocation could achieve significant performance improvement for code compilation in most cases. Also, the simulated-annealing approach gives a locally exhaustive exploration on how the register usage affects PAC DSP and investigate the flaws of the architecture. Currently, there is a fateful hazard among any data that has dependency across different functional units and need 3 cycle delay slots. This hazard makes a contradictive impact on exploiting ILP on all functional units because the increase of ILP will often introduce more hazards, causing some of the benchmark codes, like *biquad_one_section*, less affected by our optimizations. By our evaluations, several suggestions have also been proposed to the DSP design team, to enhance the architecture support for better compiler code generation. The revision process is on-going for the next generation of PAC DSP.

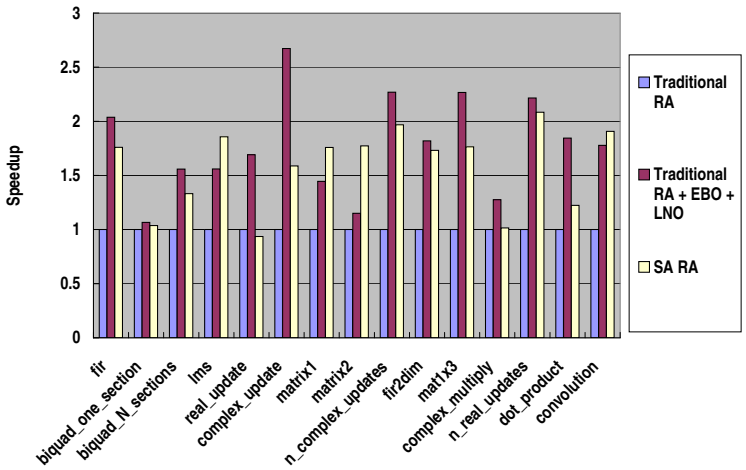

Fig. 9. The Speedup comparison while activating various optimization options

6 Conclusion

In this paper we present the design and implementation of compilers for PAC DSP – a novel high-end DSP processor with clustered architecture design and distinct partitioned register files. The compiler was based on the ORC infrastructure, consisting of PAC DSP specific code generation schemes, register allocation, peephole optimizer, and loop-nested optimizer. We demonstrated the viability of our approaches to PAC DSP via several preliminary experiments which are done with the PAC DSP prototype. By means of the experiences of the compiler design for the PAC DSP, the effects of various compiler technologies upon the novel architecture design could be validated. We believe that the experiences of employing the ORC infrastructures could also be taken to apply to other similar VLIW DSP processors, providing the qualified code generation beyond the hand-coded assembly.

Since some drawbacks of the first generation of PAC DSP architecture were revealed by the evaluation, we are currently referring to the experiences and reforming the development of compilers for the next generation of PAC DSP architecture, which will further extend our current works. Moreover, additional optimization phases and refinements of the code generation have been developing and may be integrated in the future to get more opportunities of compiler advantages on PAC DSP architectures.

References

1. Roy Ju, Sun Chan, and Chengyong Wu: Open Research Compiler for the Itanium Family. Tutorial at the 34th Annual Int'l Symposium on Microarchitecture, Dec. 2001
2. G. R. Gao, J. N. Amaral, J. Dehnert, and R. Towle: The SGI Pro64 compiler infrastructure: A tutorial. Tutorial at the Int'l Conference on Parallel Architecture and Compilation Techniques, Oct. 2000

3. Tay-Jyi Lin, Chen-Chia Lee, Chih-Wei Liu, and Chein-Wei Jen: A Novel Register Organization for VLIW Digital Signal Processors. Proceedings of 2005 IEEE International Symposium on VLSI Design, Automation, and Test, pages 335–338, 2005
4. David Chang and Max Baron: Taiwan's Roadmap to Leadership in Design. Microprocessor Report, In-Stat/MDR, Dec. 2004. http://www.mdronline.com/mpr/archive/mpr_2004.html
5. Tay-Jyi Lin, Chin-Chi Chang. Chen-Chia Lee, and Chein-Wei Jen: An Efficient VLIW DSP Architecture for Baseband Processing. Proceedings of the 21th International Conference on Computer Design, 2003
6. 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
7. Yung-Chia Lin, Yuan-Shin Hwang, and Jenq Kuen Lee: Compiler Optimizations with DSP-Specific Semantic Descriptions. LCPC'02, USA, July 2002
8. Cheng-Wei Chen, Chung-Lin Tang, Yung-Chia Lin, and Jenq-Kuen Lee: ORC2DSP: Compiler Infrastructure Supports for VLIW DSP Processors. Proceedings of 2005 IEEE International Symposium on VLSI Design, Automation, and Test, pages 224-227, 2005
9. OMAP5910 Dual Core Processor - Tecnhincal Reference Manual, Texas Instrucments, Jan, 2003.
10. S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens: Register organization for media processing. International Symposium on High Performance Computer Architecture (HPCA), pp.375-386, 2000
11. Yi-Ping You, Ching-Ren Lee, and Jenq Kuen Lee: Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors. LCPC'02, USA, July 2002
12. Ching-Ren Lee, Jenq-Kuen Lee, Ting-Ting Hwang and Shih-Chun Tsai. Compiler optimizations on VLIW instruction scheduling for low power. ACM Transactions on Design Automation of Electronic Systems, 8(2): 252–268 (2003).
13. Yi-Ping You, Chung-Wen Huang, and Jenq-Kuen Lee: A Sink-N-Hoist Framework for Leakage Power Reduction. Proceedings of ACM EMSOFT 2005, September 2005.
14. Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee: Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-To Analysis. Proceedings of ACM Principles and Practices of Parallel Programming (ACM PPoPP), San Diego, 2003.
15. Peng-Sheng Chen, Yuan-Shin Hwang, Dz-Ching Ju and Jenq Kuen Lee. Interprocedural Probabilistic Pointer Analysis IEEE Transactions on Parallel and Distributed Systems, Volume 15, Issue 10, pp. 893–907, Oct. 2004.
16. R. Leupers: Instruction scheduling for clustered VLIW DSPs. In Proc. Int'l Conference on Parallel Architecture and Compilation Techniques, pages 291–300, Oct. 2000
17. V. Zivojnovic, J. Martinez, C. Schläger and H. Meyr: DSPstone: A DSP-Oriented Benchmarking Methodology. Proc. of ICSPAT, Dallas, 1994