Journal of Signal Processing Systems 51, 269–288, 2008

DOI: 10.1007/s11265-007-0059-4

Effective Code Generation for Distributed and Ping-Pong Register Files: A Case Study on PAC VLIW DSP Cores

YUNG-CHIA LIN, CHIA HAN LU, CHUNG-JU WU, CHUNG-LIN TANG, YI-PING YOU, YA-CHAIO MOO AND JENQ-KUEN LEE
Department of Computer Science, National Tsing-Hua University, Hsinchu, 30013 Taiwan

Received: 15 September 2006; Revised: 17 February 2007; Accepted: 20 February 2007

Abstract. The compiler is generally regarded as the most important software component that supports a processor design to achieve success. This paper describes our application of the open research compiler infrastructure to a novel VLIW DSP (known as the PAC DSP core) and the specific design of code generation for its register file architecture. The PAC DSP utilizes port-restricted, distributed, and partitioned register file structures in addition to a heterogeneous clustered data-path architecture to attain low power consumption and a smaller die. As part of an effort to overcome the new challenges of code generation for the PAC DSP, we have developed a new register allocation scheme and other retargeting optimization phases that allow the effective generation of high quality code. Our preliminary experimental results indicate that our developed compiler can efficiently utilize the features of the specific register file architectures in the PAC DSP. Our experiences in designing compiler support for the PAC VLIW DSP with irregular resource constraints may also be of interest to those involved in developing compilers for similar architectures.

Keywords: compiler, ping-pong register files, VLIW, DSP, clustering, parallel processing

1. Introduction

Optimizing compiler development has always been the key factor in building a productive environment for new embedded processors and SOC chips. High-end embedded processor design is moving toward the intensive exploitation of instruction-level parallelism (ILP) and the incorporation of many advanced application-specific features, with the resulting immense increase in the complexity of compilers for these advanced processors demanding increased long-term development efforts and manpower. Hence, designing code-generation supports and optimizations based on open-source compiler infrastructures—rather than developing everything from scratch—are becoming common in attempts to reduce the delivery time of the compiler for a newly designed processor.

Several open-source compiler infrastructures have been used in research and practical applications, such as SUIF [1], Impact [2]/Trimaran [3], Zephyr [4], the popular GNU GCC [5], and the recent Open Research Compiler (ORC) [6]. The SUIF compiler infrastructure forms the major part of the National Compiler Infrastructure (NCI) project, which is focused on providing a general compiler platform to support collaborative compiler research at the intermediate-representation (IR) level. The Zephyr infrastructure constitutes the other important part of the NCI project, which primarily aims to develop

This paper is being submitted to the *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*.

portable low-level optimizations at the instruction level for compiler research. The IMPACT/Trimaran compiler infrastructures have been developed mainly to support advanced ILP research, and also to support research related to architectural issues through HPL-PD architecture models and MDES machine description interfaces. In addition to the above compilers, which are used chiefly in research, GCC is also suitable for use in practical applications, and is available for more than 100 hardware platforms. However, GCC has great difficulty in coping with modern ILP features due to its IR design and compiler structure. The recent development of the ORC has increased the momentum for supporting ILP in modern architectures, and this compiler is highly capable in supporting stable code generation in both research and practical applications.

ORC is an open-source compiler infrastructure released by Intel that represents the successor of Pro64 [7], which is the open-source compiler project for the IA-64 processor created by SGI in May 2000. Since the Pro64 originally evolved from the commercial SGI MIPSPro compiler suite that had been developed over a long period by SGI as one of the best optimized development tools known for any platform, ORC has incorporated most of the industry-strength optimization techniques developed to date. The ORC is expected to act as a stable base infrastructure for further research works, including the support for new target processors in the future. In addition, the ORC has already achieved an excellent porting status for the IA-64 processor, enabling the compiler to generate codes with good performance by utilizing several advantages of the EPIC/VLIW architecture. As modern VLIW DSPs similarly incorporate many advanced architecture features, it is of considerable interest to explore the possible deployment of ORC in VLIW DSPs.

In this paper, we describe the issues involved in applying the ORC infrastructure to VLIW DSPs with port-restricted, distributed, and partitioned register file structures. We present our experiences in the development of code generation and optimization design for a novel 32-bit VLIW DSP designed with several new architecture features, in particular for the effective support for the distributed and so called *ping-pong* register files [8, 9]. The target processor, named the Parallel Architecture Core (PAC) DSP [10–13], is being developed from scratch by SOC Technology Center at Industrial Technology Research Institute in Taiwan. The PAC DSP is natively designed to meet multimedia high-performance computing requirements and the low power consumption demanded by mobile systems. We propose effective register allocation policies in the compiler framework to support the register file organizations that are specific to PAC architectures, peephole optimization for the architecture, and essential modeling for the architecture to support loop nest optimization. Moreover, we reveal the steps employed in our development work on top of the ORC infrastructure for the PAC DSP. This paper also presents essential compiler supports for heterogeneous clustered VLIW architectures with port-restricted, distinct partitioned register file structures, which may also be of interested to those involved in developing compilers for novel VLIW DSPs with similar architectures.

The remainder of this paper is organized as follows. Section 2 introduces the target architecture of the PAC DSP, and Section 3 describes the associated compilation challenges. The development of code generation and preliminary optimizations for the PAC DSP, including the specific design for the architecture, are presented in Section 4. Experimental results from preliminary evaluations are then illustrated in Section 5. Finally, Section 6 draws conclusions related to this work.

2. PAC DSP Architectures

The PAC DSP is a 32-bit, fixed-point, VLIW DSP core that can be used as a core-component in a multicore SOC platform (such as the DaVanci system solutions of Texas Instruments [14]) to support high-performance multimedia processing, or employed as stand-alone solutions for any DSP system. The PAC DSP features an original clustered VLIW architecture that boosts scalability, a feature-rich instruction set with SIMD operation support, a variable-length-instruction encoding scheme, and a large number of registers that are innovatively heterogeneously arranged in the highly distributed register file structures.

Being unlike symmetric architectures of most current DSPs, the PAC DSP core is constructed as a heterogeneous five-way-issue VLIW architecture, comprising two integer ALUs (I-unit), two memory load/store units (M-unit), and a program sequence control unit/scalar unit (B-unit) that is mainly in charge of control flow instructions such as branch and jump. The M- and I-units are organized in pairs,

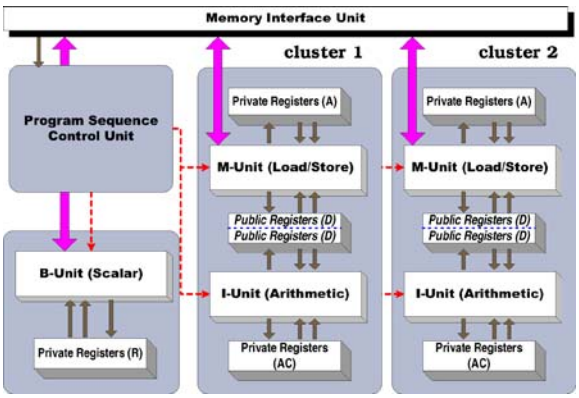

Figure 1. PAC DSP architecture.

with each pair containing 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 processing a single data stream, and the current design of the PAC DSP consists of two clusters that therefore support a maximum workload capacity of two concurrent data streams. However, the scalability of the cluster design makes it easy for future PAC DSP processors to involve more clusters to handle larger data processing workloads. The B-unit consists of two subcomponents—the program sequence control unit and the scalar unit—due to the hierarchical decoder design for variable-length-instruction encoding in the PAC DSP. The program sequence control unit is primarily responsible for control flow instructions, and the scalar unit, which is capable of simple load/store and address arithmetic, is separated from data stream processing clusters, and has its own register file. The overall architecture is illustrated in Fig. 1.

Figure 1 shows that registers in the PAC DSP are organized into four distinct partitioned register files and are arranged as cluster structures, which reduces the wire connections between functional units and registers and thereby decreases the chip area and power consumption. The A, AC, and R register files are private registers that are directly attached to and only accessible by the M-, I-, and B-units, respectively; the D register files are shared within a cluster and can be used to communicate between the paired M- and I-units. The internal structure of the D register file is further designed to utilize a special port-switching technology that further reduces the wire connections between the shared functional units. The technology—ping-pong register file structure—involves dividing a single register file into two banks, where each bank can only be accessed by one of the functional units at any one time. The instruction bundle encoding contains the information that could be set to direct which bank is to be accessed by each functional unit, so that the hardware can perform port switching between register file banks and functional units to implement data sharing within a cluster. The access states of read ports and write ports in a D register file are additionally designed to use the separate port-switching settings to increase the flexibility. The advantage of the ping-pong register file structure design is that the area size and access time could be decreased due to the reduced read/write ports [15, 16] while retaining an effective data communication capability. The register file structure inside a data stream cluster is illustrated in Fig. 2. Furthermore, a unique design is employed in the PAC DSP to allow the intercluster communication to be processed by the internal data-routing paths in the memory interface unit which connects with all B- and M-units. To transfer data between the two clusters, or between a cluster and the B-unit, programmers need to use a paired instructions ("bdt" and "bdr") in the same bundle to inform one of the units to send the data and the other to receive them. This mechanism simplifies the implementation of intercluster communication com-

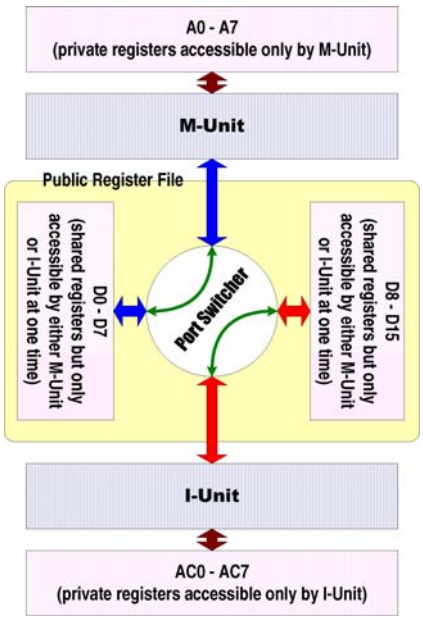

Figure 2. Distinct partitioned register file organization in a cluster.

pared to other existing schemes [17], providing further reduction of area size and access time [9]. The overall design of the distributed and ping-pong register file organizations adopted by the PAC DSP could decrease 76.8% of silicon area and 46.9% of access time compared to a centralized architecture with the equivalent number of registers.

3. Code-Generation Issues with PAC DSP Architectures

The PAC DSP incorporates various leading-edge architecture features in an attempt to increase the performance and reduce the hardware cost and power consumption. However, this design increase the interference between valid code generation, instruction scheduling, and register allocation than typical VLIW architectures, which impacts on optimizing the code for 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 heterogeneous and distributed register file design with irregular port access constraints (see Figs. 1 and 2). Each cluster inside the architecture contains A and AC register files, which is directly connected to the M- and I-units, respectively, and one D register file. Each D register file is divided into two banks that share a single set of access ports connecting to the 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 functional units 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 in the same cycle, and vice versa—accesses from two functional units to the same D register bank are mutually exclusive in a cycle. In addition, each functional unit in the PAC DSP has a different set of instructions that could be executed, and each instruction has its own register-access constraints. All of these irregular designs increase the challenge of generating effective and optimized code. Conventional instruction scheduling policies and register allocation strategies are seldom applicable to 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 performs an arithmetic operation on them. The first two instructions can be scheduled in parallel only if TN1 and TN2 are registers allocated to distinct D register banks; if both are assigned to the same D register bank, they can only be scheduled and issued sequentially. But the ping-pong register file structure more than simply limits the parallelism in the instruction scheduling. The above example is complicated by the third instruction: since this refers to both 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 Fig. 3, without considering other hazards, parallelizing the first two instructions requires a copy instruction to be inserted before the last instruction if TN1 and TN2 are allocated to different D register banks. Therefore, the advantage of parallelizing the first two instructions is counteracted by the insertion of the additional copy

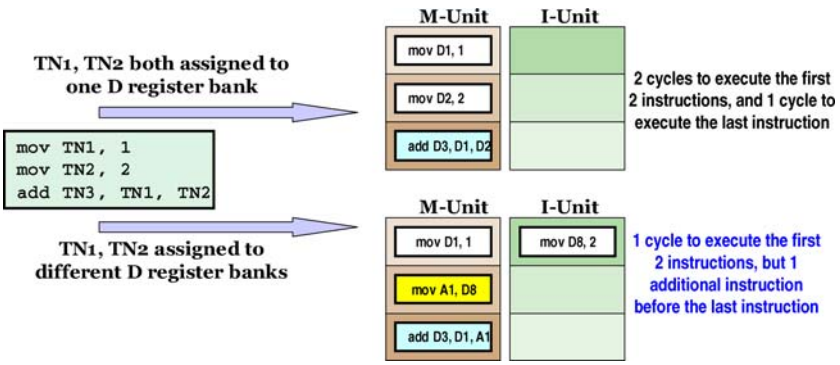

Figure 3. Interference caused by the ping-pong register file structure.

instruction and the associated increase in the size of the generated code compared to when both TN1 and TN2 are allocated to the same D register bank. But allocating virtual registers to the same D register bank will always increase register pressure of that bank, and register spilling from different register files in the PAC DSP architecture will alter the cost incurred because the memory access is restricted to only B- and M-units. These various code-generation issues have unpredictable combined effects. We know of no previous method that can be applied to the PAC DSP to obtain the optimal result before finalizing the instruction scheduling and the register allocation of all codes, making it imperative to develop new compiler schemes that can effectively handle the issues caused by the innovative architecture of the PAC DSP.

Register allocation also critically interferes with both instruction scheduling and code generation during the implementation of data communication across clusters in the PAC DSP architecture. The current version of the PAC DSP requires the code to explicitly issue a pair of instructions to complete the data communication between clusters. The paired communication instructions not only need to be issued by two of the M- and B-units with occupying two slots in the same instruction bundle, but also introduce penalties from the additional data-dependency and data-available latency for any scheduled code that is distributed into multiple clusters. Figure 4 illustrates two possible code distributions with the same performance for two clusters (considering all major constraints for scheduling), which each have their own benefits. The example shows the common code sequence generated from a dot-product operation of two vectors. To schedule the code in parallel, it is typical for compilers trying to utilize all available functional units; hence the two critical multiply instructions in this example should be scheduled into the two different I-units since there is no data dependence between them. The result of optimizing such code scheduling with register allocation is showed in the upper-right part of Fig. 4. Considering the various functional-unit capabilities and hardware constraints in the PAC DSP, the performance of this parallel scheduling is limited to ten cycles, which is the same as the result from sequentially scheduling the two multiply instructions into the same I-unit, as shown in the bottom-right part of Fig. 4. While it requires issuing two more instructions for extra data communication between B- and M-units in the latter case, the result may still benefit the power consumption because the PAC DSP could shutdown the I-unit using power-gating/clock-gating technologies. This kind of trade-off increases the difficulty of optimizing code generation since the design of PAC DSP mainly targets the embedded-system products. Furthermore, it appears that the compiler for the PAC DSP needs to perform a thorough evaluation before distributing the generated code into two clusters to avoid the penalty

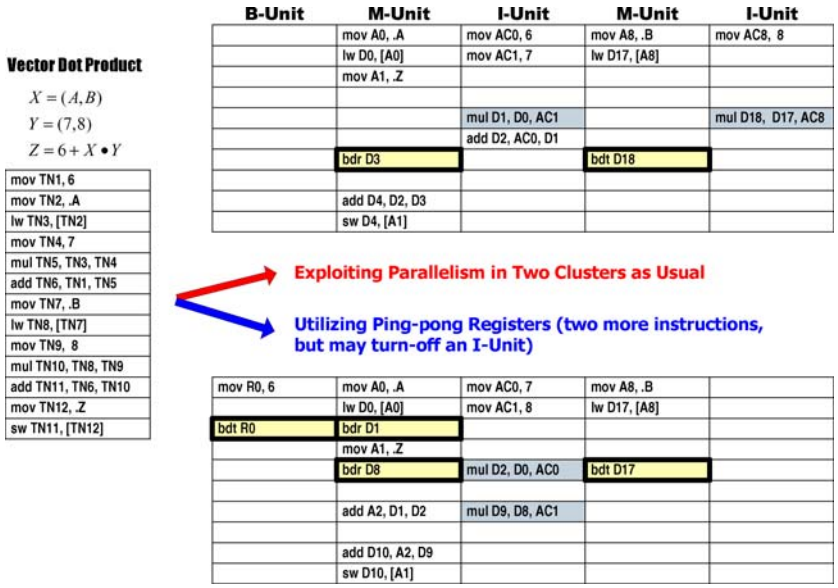

Figure 4. Example of generating optimal scheduled codes across clusters.

of cross-cluster communication undermining the advantages of the parallelism of two clusters; however, the evaluation becomes more complicated and nondeterministic with the interference associated with the ping-pong register file structure. This increases the challenge if constructing a good compiler for the PAC DSP architecture. Table 1 summarize the currently considered interference inherent in the PAC DSP compiler design.

4. Effective Compiler Supports for PAC DSP Cores

We now describe our work in developing 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 incorporating incremental development achievements and optimization improvements into the compiler framework. In brief, the ORC compilation procedure begins with front-end processing, generating a machine-independent intermediate representation (WHIRL [18] IR) of the source program, and then feeding this into the back-end. Since the WHIRL IR has five levels of representation, the back-end will invokes several components to perform a series of lowering processes and optimizations on the WHIRL IR before transforming it into the CGIR, which is a target-specific low-level IR that is near to the real instruction representation. The components developed for optimizations that could optionally be activated at the WHIRL IR level include the interprocedural analysis/optimizer (IPA), loop nest optimizer (LNO), and WHIRL optimizer (WOPT). The IPA in ORC analyzes the program information across several source files, and performs the following optimizations: dead function elimination, interprocedural constant propagation, and memory disambiguation for precise alias analysis. The LNO, which is one of our prioritized working items, is based on a cost model of code generation in the Instruction Set Architecture (ISA) of the PAC DSP. It is designed to perform optimizations related to locality, parallelization, and loop transformation, including the most well known loop optimizations, such as loop peeling, loop tiling, vector data prefetching, loop fission, loop fusion, loop unrolling, and loop interchange. The WOPT performs the major classical optimizations: common subexpression elimination, loop invariant code motion, strength reduction, code hoisting, redundancy elimination (partial and full), register promotion, and partial dead store elimination.

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 assignment
Communication insertion	Register access constraints	Register pressure
	Instruction latency	Spill code variations
	Bundling constraints	

After the WHIRL-level processing, the back-end invokes the code generator to transform the WHIRL IR into the CGIR. In addition to register allocation, compilation modules may be activated to process the CGIR depending on the code optimization level before the final codes are emitted. Figure 5 illustrates how we have extended the PAC compiler phases to include several new optimization/analysis modules that may benefit PAC DSPs. Many of these new phases are based on our previous research, and we are currently in the process of integrating those technologies into this infrastructure, including low-power optimizations [19–21] advanced pointer analysis/optimizations [22, 23], and DSP-specific optimizations [24]. Since the design of the PAC DSP architecture is still being improved progressively, our development of compiler support and optimization for the PAC DSP represents an ongoing effort, with this paper focusing on supporting basic ORC infrastructures for PAC VLIW DSPs. We mainly present the phases (see the thick bordered blocks in Fig. 5) required to generate effective code with essential optimizations at the basic-block level, and the target-dependent modifications on the LNO.

4.1. Code Generation with Target Information Extension

The target information table (TARG_INFO) in ORC is crucial to supporting the code generation, by providing parameterized data about the architecture and the ISA of the target processor. In this section we present our adaptation from the IA-64 processor

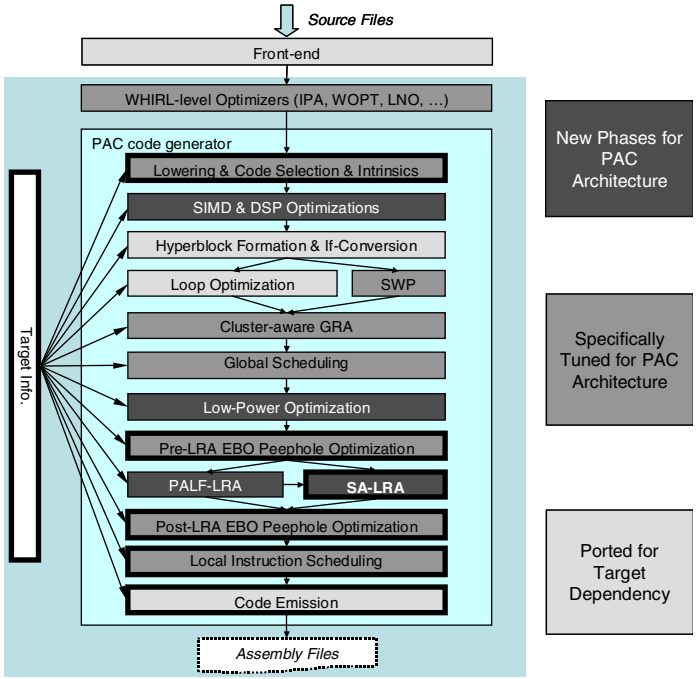

Figure 5. The refinement of compiler code generation phases for PAC DSP processors.

to the PAC DSP and the improvement of the TARG_INFO to support more flexible CGIR-level processing and optimizations for the PAC DSP architecture. The original TARG_INFO is written in the constructs of the C programming language, and then they are used to generate the actual source files of the TARG_INFO library. The machine parameters described in the TARG_INFO library are referred to 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 so as to reduce the effort of constructing compilers for different target machines. There are three categories in TARG_INFO: ISA, Application Binary Interface (ABI), and miscellaneous processor-related descriptions (PROC).The ISA descriptions comprise the following:

– Registers: sizes, classes, supported types, and usages of special registers.
– Literals: sizes, ranges, and excluded bits.
– Instructions: opcodes, operands, attributes, bundles, assembly format, and object code.
– Resources: functional units and busses.

To conform to the PAC architecture and minimize the complexity of instruction scheduling and register allocation, those instructions available to more than one functional unit are defined as being distinct in different function units in the ISA descriptions. That is, the register allocation range can be determined by the instruction used so as to clarify the register file accessibility in the implementation of register allocators. Since the PAC DSP processor has two clusters with no shared register files, special-purpose registers that should be treated as always available to all operations (e.g., stack pointer, frame pointer, and global pointer) are defined in both clusters, and the code generation must allow for duplication of the content of these registers to meet calling conventions. This is not possible by altering the machine descriptions, instead requiring some hard-coded modifications to the core routines of code generator. Moreover, to overcome the disadvantage of the functional-unit-bound instruction definition, we design new descriptions that can assist the CGIR-level phases in choosing the appropriate instruction in different units to implement the same semantics. The hazard descriptions and handler functions in original TARG_INFO are also fully redesigned to manipulate multiple

hazards of multiple single instructions, because the constraints of the PAC DSP are more complicated than the original IA-64 processor architecture.

The main code-generation phase begins with transforming the codes from the "very-low WHIRL" form into CGIR operations, which are then mapped into instructions for the target processor. This is achieved using a set of programmer-provided callback functions to select the target-dependent CGIR operations. The style of the interface is like 'Exp_OP', which expands an inputted WHIRL operation into a list of CGIR operations that are appended to the provided operation list. Thus, when the code generator locates a particular WHIRL operation, it invokes the corresponding code-expansion function and then builds the CGIR operation lists as the WHIRL IR is traversed. The code generator generates program control structures as separate basic blocks. By combining the code-expansion functions and basic blocks, the generated CGIR operation lists can be further optimized by machine-dependent optimizers.

The further adaptation of the WHIRL-to-CGIR code-generation functions includes designing the selection of optimal instructions, which depends on the optimization policies to produce preferred CGIR operations for the PAC DSP architecture, and implementing the specific handler for the PAC DSP architecture deficiency in generating correct code to follow C language conventions. For example, parameters are typically passed to functions through a register stack or rotating registers, and since the PAC DSP does not support a shared register stack and convenient register-passing mechanisms, the parameter passing mechanism in the code-generation part must be redesigned to employ a run-time memory stack. Furthermore, other features in the IA-64 processor not found in the PAC DSP, such as control and data speculation, need to be identified and dealt with. Another example is floating-point operations: the PAC DSP has no hardware floating-point support, and so we adopted the SoftFloat library [25] to simulate IEEE binary floating-point arithmetic through the intrinsic-call interface in the ORC infrastructure.

4.2. *Optimizing Register Allocation and Instruction Scheduling*

The rationale of the highly partitioned register file design of the PAC DSP is, of course, to reduce the register file port counts in order to avoid the slow access speed and high power consumption of a unified register file, although this is at the expense of an irregular architecture. This design results in phase interaction between register allocation and instruction scheduling becoming a critical problem in the code generation. Not only does the clustered design also make register access across clusters an issue, but the switched-access nature of the ping-pong register files makes register-file assignment (RFA) and instruction scheduling interdependent, as shown in Section 3.

Our current proposed solution to this problem is to add a new RFA phase before register allocation. In the current compilation flow, three kinds of RFA/register allocation schemes are developed to provide more opportunity for optimizing code than the primitive design of code generation [26]. The first scheme proposed is to optimize RFA using *simulated annealing* (SA) [27, 28]. The design extends that of Leupers [29] and our initial implementation [26], using a combined instruction scheduling/cluster assignment algorithm to iteratively approach the near-optimal result. In brief, the algorithm operates by first generating a random cluster partitioning of instructions, and a modified list scheduler (LS) then schedules the partitioned instructions whilst inserting/managing cross-cluster communications.The subsequent iterations involve random changes to the partitioning state and rerunning of the LS. The LS returns the obtained schedule length of the instructions as the "energy" value used in a usual SA optimization process, representing an evaluation of the current partitioning state. Depending on whether a random change results in improvement or deterioration, it will be retained or discarded. This process is iterated until the energy/evaluation falls to below a threshold at which we are confident that the obtained optimization state is of sufficient quality.

Adapting this SA solution for the PAC DSP involves changes to the formulation of optimized state: our search is for RFAs in the chosen schematic placement (as the search space) for virtual registers, instead of the original bipartitioning of the instructions. Figure 6 gives the high-level SA algorithm, which controls the scheduler, performs 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 allow the compiler to dynamically control the iteration scale and limit the register file usage in

Coupled Instruction Scheduling/RFA via Simulated Annealing

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

1. Choose a schematic register file placement(e.g., 1 cluster, 2 clusters, ...).
2. Make initial RFAs: randomly assign each virtual register to any of the wanted register files, and record in RFA_map.
3. Given RFA_map, run **PAC_Scheduler** (see Fig. 7), setting $sched\_len$ to the computed total schedule length in cycles.
4. Set the following initial values:
   $threshold$: threshold value for the SA process.
   $energy$: initial energy ($> threshold$).
   p_test: a probability test value ($0 < p\_test < 1$).
5. Repeat the following steps while $energy > threshold$:
   5a. Change 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 change, rerun **PAC_Scheduler**, setting new_sched_len to the new total schedule length.
   5c. Adjust $energy$, $sched\_len$, and RFA_map according to 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, repeat steps 2–5, and to select the best results.
7. Retain the final schedule and RFA_map as the output.

Figure 6. High-level SA algorithm for optimizing register allocation.

accordance with other optimizations, and may also increase the overall speed of register allocation.

Figure 7 provides 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 output of the RFA will improve progressively during the SA iterations, in terms of the schedulable length of the instructions. Lastly, a final register allocator is run to allocate and assign hardware registers, guided by the RFAs (i.e., *RFA_map*).

The second scheme developed for optimizing RFA/register allocation is using a heuristic named as PALF (ping-pong aware local favorable), which is proposed in our previous work [30]. This heuristic determines RFA using the associated data-dependency graphs and graph-partitioning methods, with several assignment policies to better utilize the distributed and ping-pong register file architectures. This scheme could provide a comparable result of code generation to the SA-based approach.

The last one—a hybrid optimization scheme with both the PALF and SA heuristics - is proposed to further improve the performance of the generated code. In contrast to the pure SA-based scheme in which we make the initial RFA based on a random assignment, this method instead uses the PALF heuristic to obtain a better RFA as the initial one, providing more chances to result in the most improvement in the end. Since the SA requires to be processed within a limited iterations (controlled by *threshold*), an appropriate initial RFA usually ensures a good result.

4.3. PAC-aware Peephole Optimizers

The Extended Block Optimizer (EBO) is a peephole optimizer that performs simple optimizations within the scope of extended basic blocks at the CGIR level.

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

Figure 7. Schedule/evaluation algorithm in the SA approach.

Extended blocks are constructed by choosing a sequence of blocks that may contain branch instructions before the end of the last block, 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 its listed successors. New blocks are processed until a branch-to label is encountered, at which point the processing backs up and attempts to take a different path down another successor on the list. The EBO is used to perform peephole optimizations immediately after instruction translation, during unrolling and pipelining, after unrolling and pipelining, and after register allocation. The 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. Performing these peephole optimizations improves the performance and quality of the generated code.

There remain many situations in which the core routines inside the EBO would need to be rewritten given that the machine-dependent implementation in influenced by the PAC DSP architecture. Hence, our work not only involves refinement of the basic peephole optimizations, but also aims to employ techniques for supporting the PAC DSP architecture. Table 2 lists the design features included in the EBO for the original ORC that supports the IA-64 processor and for the PAC DSP compiler.

Both compilers have implemented the basic peephole optimizations (forward propagation, common expression elimination, constant folding, and dead code elimination). However, due to the PAC DSP using irregular register files and clustered architectures, illegal propagation may occur when

Table 2. EBO refinement from the original ORC to the PAC DSP compiler.

EBO optimization	ORC for IA64	PAC DSP compiler
Forward propagation	×	×
Common expression Elimination	×	×
Constant folding	×	×
Dead code elimination	×	×
Conditional branch resolving	×	–
Redundant condition elimination	×	–
Memory offset merging	–	×
Compound operation conversion	×	×
Subword calculation	–	×
Dual load/store operation	–	×

multiple virtual registers are allocated to different register files. A major problem when applying such basic peephole optimizations to the PAC DSP is that we cannot take all the virtual registers as registers in a unified register file to analyze their correlation. Instead, we have to develop a strategy with cost models to enhance the extended block optimizations.

Among those basic optimizations, constant folding is calculated with constant variables and dead code elimination is analyzed with liveness of registers. They are less affected by restricted register access and instruction insertion in the generation of valid code. But forward propagation and common-expression elimination may be greatly affected by the specific PAC DSP architecture, and require the supplementary analysis of the information about clustering and ping-pong setting. Their behavior should be carefully analyzed for the possibility of illegal propagation of data flow. Here are some examples to motivate the need of our optimization schemes. Consider the code fragment below:

Code fragment 1
1. x:=t1;
2. a[t2]:=t5;
3. a[t3]:=x+t6;
4. a[t4]:=x+t7;

The technique for compilers to optimize the above code is to use t1 instead of x, wherever possible after the copy statement x:=t1 [31]. Following the common data-flow analysis and copy propagation applied to the code fragment 1, we have the optimized code below:

Code fragment 2
1. x:=t1;
2. a[t2]:=t5;
3. a[t3] :=t1+t6;
4. a[t4]:=t1+t7;

This propagation can remove all the data dependency produced by x:=t1, providing the compiler with possibility to eliminate the assignment of x:=t1. However, the simple scheme above is not appropriate for the design of the PAC DSP architecture. Due to the specific-architecture design with clustering and heterogeneous distributed register files, extra inter-cluster-communication code needs to be inserted if there occurs the data flow across clusters. Suppose t1 is allocated to a different cluster from t6, t7 and x, the insertion of intercluster-communication code will then need to be done if applying copy propagation. Such overhead of communication code increases the total cycles of the optimized code compared with the non-optimized one.

Another example below presents the issue of private-access nature of A and AC registers. For the convenience to trace the properties of private-register access, Code Fragment 3 lists assembly code generated from code fragment 1. Assume that D register d2, and private registers a1, ac1, and ac2 are allocated to the variables x, t1, t6, and t7, respectively.

Code fragment 3
1. MOV d2, a1
2. MOV d3, a3
3. ADD d4, d2, ac1
4. SW d4, d0, 24
5. ADD d6, d2, ac2
6. SW d6, d0, 28

Note that the operation MOV d2, a1 reaches the use of d2 in lines 3 and 5. However, it is impossible to replace all the uses of d2 with a1 directly, for the reason that A register files are only attached to M-unit and AC register files are also only attached to I-unit. If d2 is replaced with a1, the compiler must insert extra copy instructions for indirectly private-register access. This insertion of extra copy instructions also brings the penalty and occupies additional computing resources, and therefore needs to be considered for performing copy propagations.

For handling the issues above, we build a cost model to evaluate the extra communication cost and the benefit gainable from the variable n being replaced by m. The equation is defined in the following:

$$Cost(n,\ m) = Gain(n,\ m) - (CPAC(n,\ m) + RP(n,\ m)), \quad (1)$$

where $CPAC(n,\ m)$ represents the cost from propagating across clusters, and $RP(n,\ m)$ is the cost from the increase of register pressure due to data duplication between two different private register files. Furthermore, $Gain(n,\ m)$ is calculated using the reduced size of communication code and reduced number of copy assignments after optimization. If

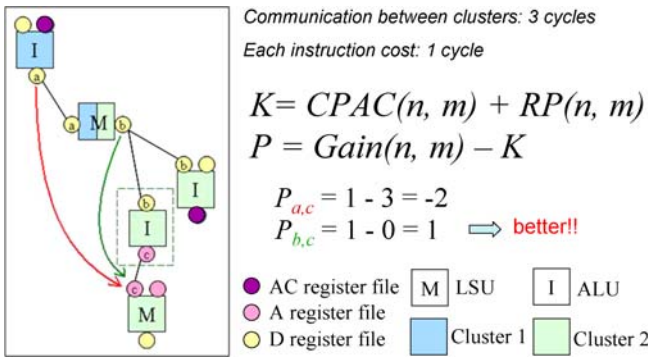

Figure 8. Example 1 of enhanced EBO cost model.

$Cost(n, m)$ reveals a positive result, it implies that the optimizations can be applied with no anomalous effect. On the contrary, the optimizations related to propagation are suspended for the variables n and m in case of $Cost(n, m)$ showing a negative result.

We further explain the employment of the cost model by using some examples for clarity. Consider the examples with the data-flow graph given in Figs. 8 and 9, in which every rectangular node is represented as an operation and its attached small circles stand for the corresponding register-type data used/defined. Different register file types and clusters are illustrated as the legend shown in the figures. Assume that it takes three cycles for intercluster communication (as such nodes associated with both clusters) and one cycle for executing every other operation. In the data-flow graph of Fig. 8, there exist two choices to do the copy propagations; the first is to propagate a to c, and the other is to propagate b to c. If we choose to propagate a to c, the code block surrounded by the dotted lines could be eliminated; hence we would gain one-cycle performance improvement by eliminating the code block but three-cycle degradation of additional intercluster communication because a and c are set in different clusters. On the other hand, if we propagate b to c, one-cycle performance improvement could be done without any penalty. By using the proposed cost model, the better choice, "propagation from b to c", should be made evidently. Another example is shown in Fig. 9. Originally, the data flow going through the right path in the figure has the availability of copy propagation from a to c. However, once we propagate a to c, the register pressure would increase oddly since the AC and A register files are assigned to the operands concurrently. To meet the register-access constraints in PAC DSP architectures, we must issue one more instruction to copy a to the D register files for the simultaneous data access. The evaluation result of our proposed cost model shows the profit of the propagation; we earn six-cycle performance improvement because of eliminating the two nodes in the right path but one-cycle degradation due to the extra copy operation needed.

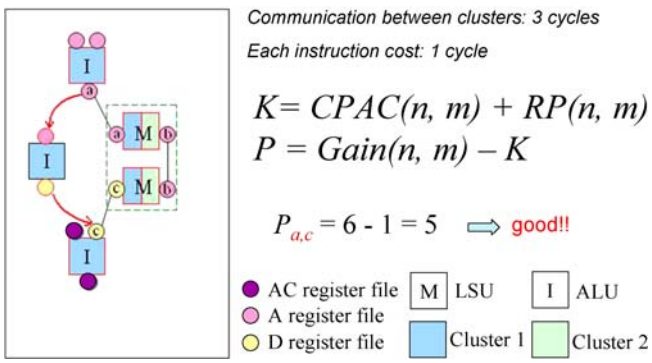

Figure 9. Example 2 of enhanced EBO cost model.

The positive result indicates that the propagation could be done properly.

To take further advantage of the architecture, we propose additional machine-dependent optimizations on the EBO phases of the PAC DSP compiler, including memory offset merging, subword calculation, and dual load/store operation. The memory offset merging utilizes the compound forms of load/store instructions. Rather than wasting two instructions to perform an actual load/store operation after the computation of the entire "*base* + *offset*" address, we calculate the final address of the memory and access the data using a single instruction.In addition, the ISA of the PAC DSP includes a rich and general set of subword instructions to accelerate lower precision operations. A subword in the PAC DSP can be 8 or 16 bits long, so that quad or dual subwords can be accommodated in a single register, which comprises 32 bits. The challenge is to find a set of data-parallel computations that operate on lower precision data, and to map them onto packed or unpacked subword instructions (the PAC DSP provides instructions that operate on two sets of 16-bit data or four sets of 8-bit data residing in a single register). A basic technique is to divide a loop into multiple loops with lower precision data. We first need to extend the data structure that describes virtual register (e.g., to add a new field for data precision) to enable the determination of which virtual registers—and thus operations—are candidates for subword operations. Moreover, a phase for packing subword operations into a single compound instruction before the process of register allocation is required to integrate subword optimizations into the PAC DSP compiler.

Finally, dual load/store instructions are powerful operations for accessing data from different memory address and then combining/separating the values into/from two 32-bit registers simultaneously. When processing optimizations of dual operations, we need to refer the precision field mentioned above, and have to examine the operand width of the processing data. Thus, we are able to select the most suitable instructions for dual load/store operations.

The EBO of the PAC DSP compiler is supposed to improve performance with minimum overhead. But in practice, we experienced the propagation anomaly under certain circumstance issued by the distributed, ping-pong register files and clustered architectures. The proposed cost model is helpful to determine the feasibility of each possible propagation, plus further machine-dependent optimizations, resulting in a positive improvement in the code generation.

4.4. *Architecture-Dependent Loop Nest Optimizer*

The major optimizations held in the LNO phase are based on restructuring loops to optimize data accesses. This phase is considered to be one of the most important optimizations affecting the performance of the code generated for DSPs since DSP programs typically include many loop constructs with intensive data accesses. The LNO phase employs traditional loop transformation techniques such as fusion, fission, tiling, unrolling, and unimodular transformation. These transformations are designed to make the optimized forms consistent with machine features, code generation, and low-level optimizations [32]. Three target-specific models—resource, latency, and register pressure—are constructed for the PAC DSP to estimate the best unrolling factor and tiling size for candidate loops at the WHIRL level. Depending on the rate at which instructions are issued (the issue rate), the number of memory units and ALUs in the PAC DSP architecture, we first determine the essential information (as in Table 3) to model the basic processor parameters. Using resource models, LNO estimates the resource usage in each iteration of a loop from tables mapping the equivalence between WHIRL operations and PAC DSP instructions. Next, in the phase of estimating the latency constraint, LNO builds a dependence graph for the loops in order to generate code that is suitable for software pipelining. This graph can help to calculate the total latencies by observing each load and store instructions. For the PAC DSP architectures, new modeling of integer operations is used rather than the original ORC floating-point considerations to calculate the opera-

Table 3. Basic parameters used to model the PAC DSP.

Parameter	Description
_issue_rate=4.5	We use an issue rate of 4.5 because there are two M-units, two I-units, and a single B-unit
_num_mem_units=2.5	The B-unit is used mainly for control, so it is estimated at only 0.5. Similarly, the number of memory units is estimated at 2.5

```
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
```

Figure 10. Register-pressure cost model for loop transformation.

tion latencies more accurately. The latency value is then used in the scheduling of software pipelining optionally enabled in the later phase to optimize the performance of the generated code.

Finally, register-pressure estimation policies are elaborated to include the effects of the irregular register file structures in the PAC DSP. The clustered architecture characteristics of the PAC DSP are also considered, with the register pressure for a single cluster (neglecting the possible intercluster interference) being appraised initially. If the register pressure of any cluster is too high, interference between two clusters is assessed for possible register resource usage and communication penalty while accessing cross-cluster content. Therefore, a cost model adapting for PAC cluster feature is proposed in Fig. 10, which shows that register-pressure estimation affects decisions regarding loop transformation.

5. Experiment and Discussion

Preliminary experiments were performed using DSPstone benchmarks [33]. We evaluated the stable optimization combinations of our designs mentioned in this paper and examined the performance of the three register allocation schemes described in Section 4.2 in particular. All benchmark programs were compiled with the following register allocation schemes, with or without the combination of the EBO and LNO optimization phases (disabling all other optimizations): initial ORC adaptation (primitive code generation), register allocation using the PALF approach, register allocation using the SA approach, and register allocation using the hybrid approach (PALF-LRA+SA-LRA). The primitive code generation, which is a modification of the original ORC register allocation that assumes that

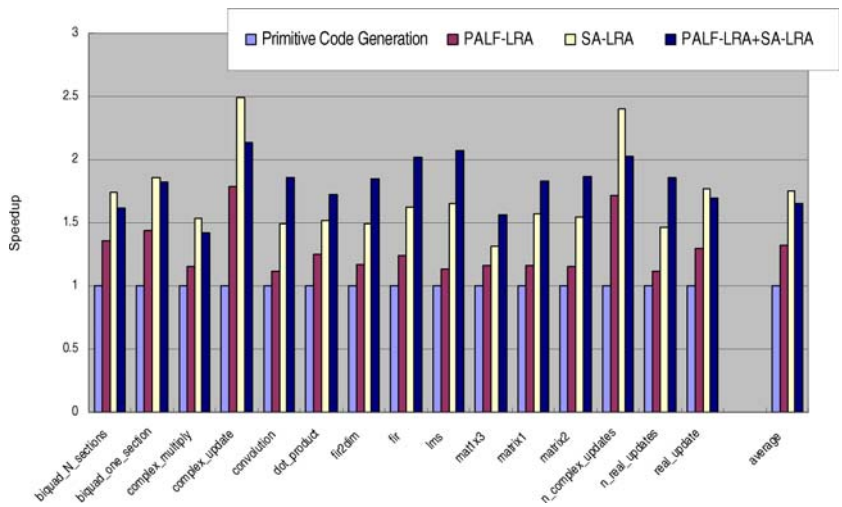

Figure 11. Speedup comparison between different register allocation schemes.

the PAC DSP has only one unified register file containing all registers and inserts the codes required to make the register allocation result executable, is treated as the base reference in the comparison. We first compare the speedup of DSP benchmarks for various register file assignment methods, which are labeled as the "PALF-LRA", "SA-LRA", and "PALF-LRA+SA-LRA" in Fig. 11, relative to those for the primitive code generation. As shown in the figure, the performance gain for the speedup for the PAC DSP varies widely across the different benchmarks, and with averages 1.32, 1.75, and 1.66 relative to the "PALF-LRA", "SA-LRA" and "PALF-LRA+SA-LRA", respectively. Due to the property of randomization, the SA provides a locally exhaustive exploration on RFA and thereby in most cases obtains the better results than the PALF, as revealed in the comparison. Also, the hybrid approach produces the best results against the other individual assignment methods for many benchmark programs, as what we expect. There are some abnormal cases (e.g. *complex_update* and *n_complex_updates*) in which we think that the additional code insertion made by the PALF heuristic for the initial RFA would sometimes impair the evaluation procedure in the SA phase of the hybrid scheme, resulting in less optimized code than the pure PALF manner. Such an anomaly mostly results from that all available M-unit slots for inserting intercluster communication while scheduling certain basic blocks in which the PALF assigns the A register files to the most TNs, are jammed by the initial RFA so that the searching space of SA is limited since the attempting in modifying the cluster assignment of the operations becomes difficult due to the jam. This anomaly occurrence may be eliminated with a specific handler in between the initial PALF and SA procedures to assure that the initial RFA using PALF would not restrict the SA process heavily.

Figure 12 shows the speedup comparison with enabling the EBO optimization phase. The results present that our development of the PAC-aware EBO optimizer obtains significant performance improvement while using all the three register allocation schemes. The anomaly between "SA-LRA" and "PALF-LRA+SA-LRA" still exists in a few cases but it has less effect than without the EBO optimization because the elimination of operations in the processing of EBO provides more the searching space of SA. Figure 13 exhibits the performance gain with enabling the LNO optimization phase additionally. The effectiveness of the LNO optimization phase in the ORC infrastructure (only enabled in the "-O3" level by default) is highly depend on the processing of other optimization phases, and enabling the LNO will also turn on the EBO phase and many other optimizations by default in the original ORC design. Therefore, we only examined the combination of LNO and EBO phases but not test the LNO solely to make the experiments match the real-compilation process. It appears that the results suggest that our current approaches employing the LNO, EBO, and various schemes of register alloca-

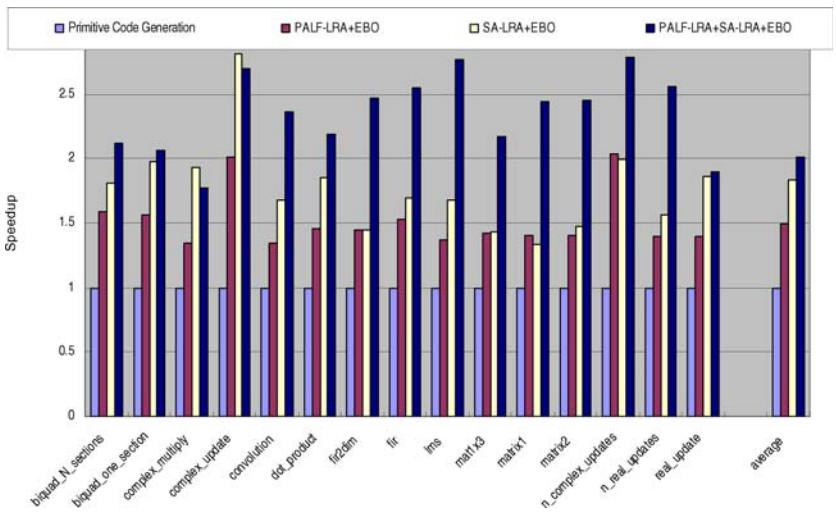

Figure 12. Speedup comparison while activating the EBO optimization phase.

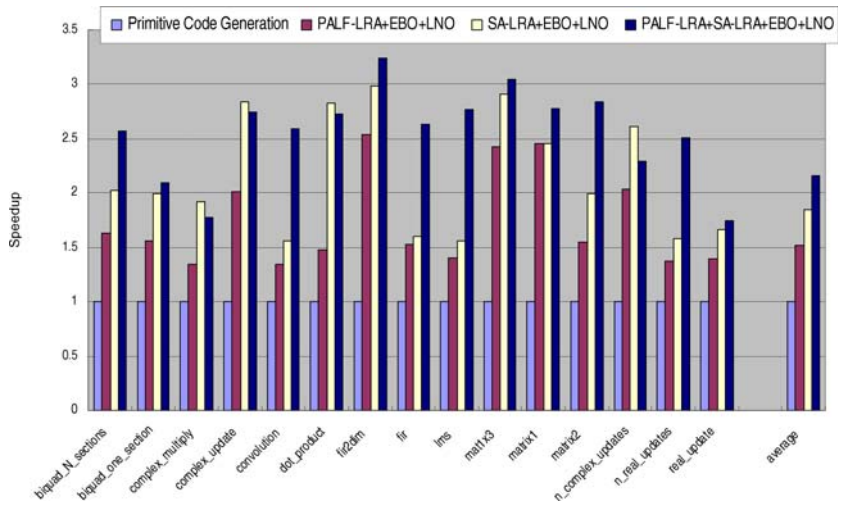

Figure 13. Speedup comparison while activating the LNO optimization phase.

tion could achieve significant performance improvement for code compilation in most cases. In general, while the "PALF-LRA" providing the fine initial RFAs for the "SA-LRA", the hybrid method of "PALF-LRA" and "SA-LRA" could boost the most performance consequently, resulting in the total speedup from 1.75 to 2.83 times as displayed in the figure. Unsurprisingly, the massive hazards of the PAC DSP impacts on the exploitation of ILP in all functional units, because an increase in ILP will often introduce further hazards, resulting in some of the benchmark codes (e.g., *real_update*) being less affected by our optimizations. These evaluations also revealed to the DSP designers how the architecture support could be enhanced for better compiler code generation.

We next explore the feasibility and effectiveness of the SA process used in both the "SA-LRA" and "PALF-LRA+SA-LRA" schemes. Obviously, the number of the SA iterations required to achieve a near-optimal result is highly proportional to the length of the basic block processed because the number of the instructions available for changing their RFAs is actually the most significant factor affecting the searching space. Therefore, in the development of our current SA procedure for "SA-LRA" and "PALF-LRA+SA-LRA" schemes, we determine to set the value of *threshold* to the length of the basic block processed and reasonably set the initial value of *energy* by a constant ratio of *threshold*, to make the searching convergent to about the equivalent status for each basic block with

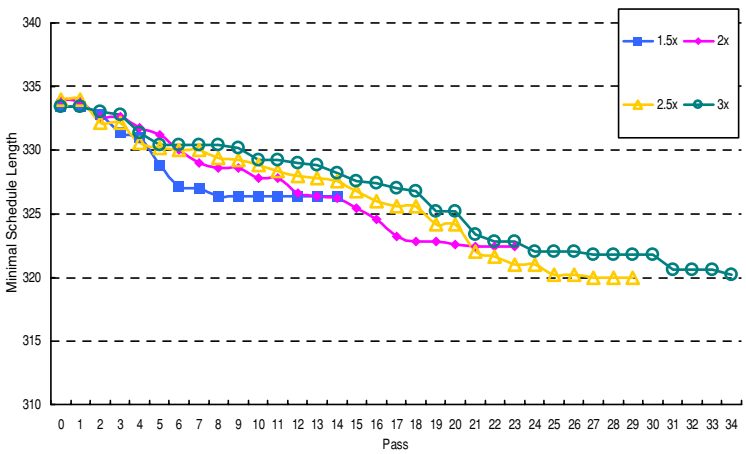

Figure 14. The relation between the iteration passes and the minimal schedule length found while using different values of the initial energy.

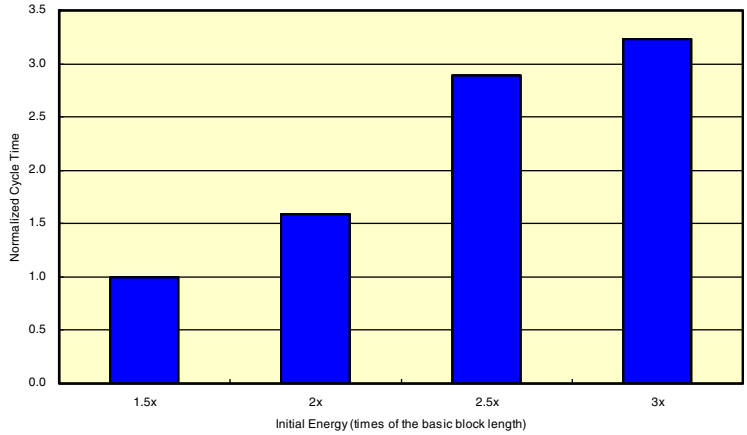

Figure 15. The comparison of the compilation time while using different values of the initial energy.

different length. Although the lengths of basic blocks in the programs that we tested were almost smaller than 200, we picked an uncommonly large basic block ($threshold = 451$) found in the *fir2dim* program, which is produced by the unrolling in the LNO phase, for the better coverage of the SA exploration. Figure 14 shows the experimental results of the SA iteration tests in average with different values of initial energy. We used $1.5 \times threshold$, $2.0 \times threshold$, $2.5 \times threshold$, and $3.0 \times threshold$, respectively, as the initial energy, to evaluate the minimal schedule length in each iteration of SA. The normalized compilation-time comparison is also provided in Fig. 15. Apparently the larger initial energy gives the more probability to find the lower value of schedule length before the searching iteration stops. By referring to both Figs. 14 and 15, it reveals that in our experiments $2.0 \times threshold$ should be an appropriate initial energy that will produce a sufficiently good result within the limited iterations practically. Moreover, Fig. 16 exhibits the compilation-time comparison of the experiments referred in Fig. 13, in which the initial value of *energy* is set to $2.0 \times threshold$. The compilation time for most benchmark programs was less than 5 s in our measurement, assuring the feasibility of our proposed schemes with the SA approach. As shown in the figure, the hybrid method takes much less process time than the pure SA approach for most programs, especially for the programs which need mass compilation time, but delivers the better or comparable results. Some exceptional results, like

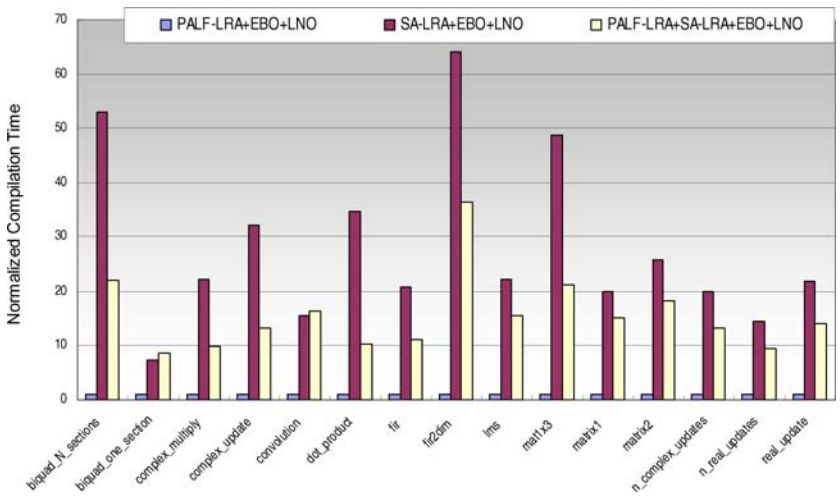

Figure 16. The comparison of the compilation time while using different register allocation schemes.

biquad_one_section, which takes a little more time for the hybrid method than the pure SA, are mainly caused by the penalty of the initial RFA procedure using "PALF-LRA". However, the compilation using the hybrid approach for these programs still bring some performance improvement over the "SA-LRA" (referring to Fig. 13).

6. Conclusions

In this paper we present the design and implementation of compilers for the PAC DSP, which is a novel high-end DSP with a clustered architecture design and distributed and ping-pong register files. The compiler was based on the ORC infrastructure, consisting of PAC-DSP-specific code compilation schemes, register allocation for the particular register file architectures, and various optimization phases tuned to specific processors, to achieve effective code generation. We have demonstrated the viability of our approaches to the PAC DSP via several preliminary experiments performed on the PAC DSP prototype. Experience gained in the design of compilers for the PAC DSP helps in elucidating the effects of applying various compiler technologies to the novel architectures. We believe that ORC infrastructures could also be adapted to other similar type of VLIW DSPs, thereby yielding effective code generation.

Acknowledgment

This research work was supported in part by the NSC under grant nos. 95-2220-E-007-001 and 95-2220-E-007-002, and by the MOEA research project under grant nos. 95-EC-17-A-01-S1-034 and 96-EC-17-A-01-S1-034 in Taiwan.

References

1. The SUIF 2 compiler system, http://suif.stanford.edu/suif/suif2.
2. P.P. Chang et al., "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," in *Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada*, vol. 28, no. 5 1991, pp. 266–275.
3. ReaCT-ILP Laboratory, "Trimaran: An Infrastructure for Research in Instruction-Level Parallelism," http://www.trimaran.org.
4. A. Andrew et al., "The Zephyr Compiler Infrastructure," http://www.cs.virginia.edu/zephyr/.
5. The GNU Compiler Collection, http://gcc.gnu.org.
6. R. Ju, S. Chan and C. Wu, "Open Research Compiler for the Itanium Family," Tutorial at the 34th Annual International Symposium on Microarchitecture, Dec. 2001.
7. G.R. Gao, J.N. Amaral, J. Dehnert and R. Towle, "The SGI Pro64 compiler infrastructure: A tutorial," in Tutorial at the International Conference on Parallel Architecture and Compilation Techniques, Oct. 2000.
8. T.-J. Lin, C.-C. Lee, C.-W. Liu and C.-W. Jen, "A Novel Register Organization for VLIW Digital Signal Processors," in *Proc. of 2005 IEEE Int. Symp. on VLSI Design, Automation, and Test*, 2005, pp. 335–338.
9. T.-J. Lin, P.-C. Hsiao, C.-W. Liu and C.-W. Jen, "Area-Efficient Register Organization for Fully-Synthesizable VLIW DSP Cores", *International Journal of Electrical Engineering*, vol. 13, May 2006.
10. D. Chang and M. Baron, "Taiwan's Roadmap to Leadership in Design," Microprocessor Report, In-Stat/MDR, Dec. 2004. http://www.mdronline.com/mpr/archive/mpr_2004.html.
11. D.C.-W. Chang, C.-W. Jen, I-T. Liao, J.-K. Lee, W.-F. Chen and S.-Y. Tseng, " PAC DSP Core and Application Processors," in *Proc. of the IEEE Int. Conf. on Multimedia & Expo*, Toronto, July 9–12, 2006.
12. T.-J. Lin, C.-C. Chang, C.-C. Lee and C.-W. Jen, "An Efficient VLIW DSP Architecture for Baseband Processing," in *Proceedings of the 21th International Conference on Computer Design*, 2003.
13. T.-J. Lin, C.-M. Chao, C.-H. Liu, P.-C. Hsiao, S.-K. Chen, L.-C. Lin, C.-W. Liu, C.-W. Jen, "Computer Architecture: A Unified Processor Architecture for RISC & VLIW DSP," in *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, April 2005.
14. TMS320DM6443 Digital Media System-on-Chip Datasheet, Texas Instruments, 2006.
15. S. Rixner, W.J. Dally, B. Khailany, P. Mattson, U.J. Kapasi and J.D. Owens, "Register organization for media processing," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 375–386, 2000.
16. A. Capitanio, N. Dutt and A. Nicolau, "Partitioned register files for VLIW's: A preliminary analysis of tradeoffs," in *Procs. of the 25th Int. Symp. on Microarchitecture: Portland, OR*, December 1–4, 1992, pp. 292–300.
17. A. Terechko, E.L. Thenaff, M. Garg, Eijndhoven and H. Corporaal, "Inter-cluster communication models for clustered VLIW processors," in *Procs. HPCA*, 2003, pp. 354–364.
18. WHIRL Intermediate Language Specification, "*SGI*," 2000.
19. Y.-P. You, C.-R. Lee and J.K. Lee, "Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors," in *LCPC'02*, USA, July 2002.
20. C.-R. Lee, J.-K. Lee, T.-T. Hwang and S.-C. Tsai, "Compiler Optimizations on VLIW Instruction Scheduling for Low Power," *ACM Transact. Des. Automat. Electron. Syst.*, vol. 8, no. 2, 2003, pp. 252–268.
21. Y.-P. You, C.-W. Huang and J.-K. Lee, A Sink-N-Hoist Framework for Leakage Power Reduction," in *Proceedings of ACM EMSOFT 2005*, September 2005.
22. P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju and J.K. Lee, "Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-To Analysis," in *Proceed-*

ings of ACM Principles and Practices of Parallel Programming (ACM PPoPP), San Diego, 2003.
23. P.-S. Chen, Y.-S. Hwang, D.-C. Ju and J.K. Lee, "Interprocedural Probabilistic Pointer Analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 10, Oct. 2004, pp. 893–907.
24. Y.-C. Lin, Y.-S. Hwang and J.K. Lee, "Compiler Optimizations with DSP-Specific Semantic Descriptions," in *LCPC'02*, USA, July 2002.
25. John R. Hauser. SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html.
26. C.-W. Chen, C.-L. Tang, Y.-C. Lin and J.-K. Lee, "ORC2DSP: Compiler Infrastructure Supports for VLIW DSP Processors," in *Proceedings of 2005 IEEE International Symposium on VLSI Design, Automation, and Test*, 2005, pp. 224–227.
27. S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, 1983, pp. 671–680.
28. P. Salamon, P. Sibani and R. Frost, "Facts, Conjectures, and Improvements for Simulated Annealing. ser. Monographs on Mathematical Modeling and Computation," Society for Industrial and Applied Mathematics, no. 7, 2002.
29. R. Leupers, "Instruction scheduling for clustered VLIW DSPs," in *Proc. Int'l Conference on Parallel Architecture and Compilation Techniques*, Oct. 2000, pp. 291–300.
30. Y.-C. Lin, Y.-P. You and J.-K. Lee, "Register Allocation for VLIW DSP Processors with Irregular Register Files," in *CPC 2006*, Spain, Jan. 2006.
31. A.V. Aho, R. Sethi and J.D. Ullman, "*Compilers: Principles, Techniques and Tools*," Addison-Wesley, November 1985.
32. M.E. Wolf, D.E. Maydan and D.-K. Chen, "Combining loop transformations considering caches and scheduling," *International Journal of Parallel Programming*, vol. 26, no. 4, 1998.
33. V. Zivojnovic, J. Martinez, C. SchlÌger and H. Meyr, "DSPstone: A DSP-Oriented Benchmarking Methodology," *Proc. of ICSPAT, Dallas*, 1994.

Yung-Chia Lin received his B.S. degree in Physics from National Tsing Hua University, Taiwan in 1997. He is working toward the Ph.D. degree in the Department of Computer Science, National Tsing Hua University. His current research interests include optimizing compilers, computer architectures, and system software and operating system for embedded SOC environments.

Chia-Han Lu received his B.S. degree in Computer Science and Information Engineering from Feng Chia University, Taiwan in 2003 and his M.S. degree in Computer Science from National Tsing Hua University, Taiwan in 2005, where he is currently working towards a Ph.D. His researches include optimizing compilers and system software for embedded SOC environments.

Chung-Ju Wu received his B.S. degree in Computer Information and Science from National Chiao Tung University, Taiwan in 2001 and the M.S. degree in Computer Science from National Tsing Hua University, Taiwan in 2003, where he is currently working towards the Ph.D. degree. His current researches include GCC compiler porting and system software for embedded SOC environments.

Chung-Lin Tang received his B.S. degree from the Department of Computer and Information Science, National Chiao Tung University, Taiwan in 2003 and the M.S. degree at the Department of Computer Science, National Tsing Hua University, Taiwan in

2005. His research interests include compilers, computer architecture, operating systems, and other systems related topics.

Yi-Ping You received his B.S. degree in Computer Science and Information Engineering from National Chi Nan University, Taiwan in 2000 and his Ph.D. degree in Computer Science from National Tsing Hua University, Taiwan in 2007, where he also received his M.S. degree in 2002. His researches include optimizing compilers and software power management.

Ya-Chiao Moo received his B.S. degree in Computer Science from National Tsing Hua University, Taiwan in 2004, where he also received his M.S. degree in 2006. His researches include loop-nest optimizations and interprocedural analysis in compilers.

Jenq-Kuen Lee received his B.S. degree in Computer Science from National Taiwan University in 1984. He received the Ph.D. degree in Computer Science from Indiana University in 1992, where he also received the M.S. degree (1991) in Computer Science. He is now a Professor in the Department of Computer Science at National Tsing-Hua University, Taiwan where he joined the department in 1992. He was a key member of the team who developed the first version of the pC++ language and SIGMA system while at Indiana University. He was also a recipient of the most original paper award in ICPP '97 with the paper entitled "Data Distribution Analysis and Optimization for Pointer-Based Distributed Programs.'' His supervised Ph.D. student received the distinguished dissertation award as honorable mention by IICM, 1999. He also received an achievement award from Ministry of Education of Taiwan for the University and Industrial joint research, 2001 and a Microsoft research award for embedded systems, 2003. His current research interests include optimizing compilers, compilers for low-power and embedded systems, parallel object-oriented languages and systems, and computer architectures.