ORC2DSP: Compiler Infrastructure Supports for VLIW DSP Processors

Cheng-Wei Chen, Chung-Lin Tang, Young-Chia Lin, Jenq-Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 300, Taiwan
Email: {cwchen, cltang, yclin, jklee}@pllab.cs.nthu.edu.tw

***Abstract*— In this paper, we describe our experiences in deploying ORC infrastructures for a novel 32-bit VLIW DSP processor (known as PAC core), which equips with new architectural features, such as distributed and 'ping-pong' register files. We also present methods in retargeting ORC compilers for PAC VLIW DSP processors. In addition, mechanisms are proposed to incorporate register allocation policies in the compiler framework for distributed register files in PAC architectures. In the early design stage, several iterations of tuning are needed between architecture and software designs. Our work gives an early estimation of architecture performance so that refinements of architectures are possible with the software feedbacks.**

I. Introduction

Modern optimizing compilers for VLIW processors are complex programs that require from tens to hundreds of people-years to be developed. To deliver the compiler in time, open-source compiler infrastructures instead of developing everything from scratch are an interesting direction for rapid developments of optimizing compilers and toolkits. There are several open-source compilers available for research purposes, for example, SUIF [12], Impact/Trimaran [11], and gcc [4]. Recently the usage of the Open Research Compiler (ORC) [6] is also gotten momentum. ORC is an open-source compiler infrastructure released by Intel. The predecessor of the ORC is Pro64 [3], the open-source compiler project for IA-64 by SGI in May 2000. The popularity is mainly because that the Pro64 evolved from the SGI MIPSPro compiler suite; it has been developed by SGI as the production compiler for a long period, so it incorporates most of the optimization techniques of industry strength. In addition, ORC/Pro64 has now achieved a porting for IA-64 codes with awesome performance. Key optimization features for EPIC architectures include explicit parallelism at the machine code level, instruction bundle, predication, control and data speculation, and hardware supported software pipelining. Because modern VLIW DSP processors also incorporate many of the advanced architecture features, it looks interesting and promising to explore possible ORC deployments for VLIW DSP processors.

In this paper, we describe our experiences in devising ORC infrastructures for a novel 32-bit VLIW DSP processor, which equips with new architectural features, such as distributed and 'ping-pong' register files. The processor, a.k.a. Parallel Architecture Core (PAC) DSP, is being developed from scratch by SOC Technology Center/ITRI at Taiwan. It is designed to meet the future requirements for multimedia and communication services on mobile devices, e.g., portable media players and cellular phones. However, the applications on these mobile devices, such as H.264 decoding and encoding for video conference, demand high computation power and low power consumption, the design of PAC DSP needs to fulfill the conflicting goals simultaneously. In the early design stage, several iterations of tuning are needed between architecture and software designs. Our work gives an early estimation of architecture performance so that refinements of architectures are possible with the software feedbacks. We also report experiences in retargeting ORC compilers for VLIW DSP processors. In addition, mechanisms are proposed to incorporate register allocation policies in the compiler framework for distributed register files in PAC architectures.

The remainder of this paper is organized as follows. Section II gives a brief description of ORC infrastructures and our DSP architecture. Next, Section III presents our experiences for retargeting ORC to PAC DSP processors. Next, Section IV proposes a mechanism to deal with distributed register files. Section V then presents experimental results and discussions. Finally, Section VI concludes this paper.

II. ORC and PAC VLIW DSP

We brief the ORC infrastructure and PAC DSP architecture in the following:

A. Open Research Compiler Infrastructure

The Open Research Compiler is a suite of optimizing compilers for Intel Itanium platforms running Linux. It is an extension of the Pro64 compiler developed jointly by Intel and Chinese Academy of Sciences, Institute of Computing Technology (ICT). The latest compiler suite includes compilers for C, C++, and Fortran 90 for the IA-64 Linux ABI and API standards. Intel and ICT have added many enhancements into ORC departing from the original Pro64: they include region-based compilation [5], [8], if-conversion and predicate analysis, control and data speculation with recovery code generation, and global instruction scheduling with resource management.

ORC is made up of modularized components, with C and C++ front-ends from GCC. The compilation by ORC starts with processing by the front-ends, generating an intermediate representation (IR) of the source program, and feeding it in the back-end; the IR , called WHIRL, which is a part of the Pro64 compiler released by SGI, includes five representation levels from "very high" to "very low". The back-end invokes several components to perform a series of lowering processes and optimizations on the WHIRL IR. Each component is organized as a dynamically-shared library, loaded and executed on demand by the back-end depending on whether the optimization phase is enabled. The components for optimizations in these levels include the inter-procedural analysis/optimizer, loop nest optimizer, global optimizer, and code generator.

The inter-procedural analysis/optimizer in ORC analyzes the program information across several source files and performs the following optimizations: dead function elimination, inter-procedural constant propagation, and memory disambiguation for precise alias analysis.

The ORC loop nest optimizer is based on an unified cost model and a model of the target cache. This phase performs locality optimization, parallelization, and array privatization. It also includes

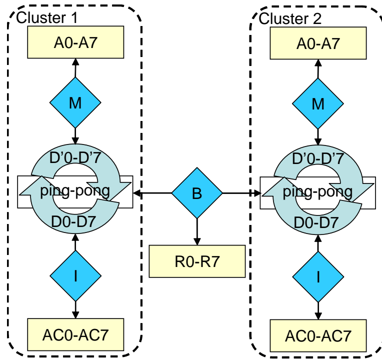

Fig. 1. The PAC DSP architecture from compiler's perspective: all register files except 'D' files are private registers.

most well known loop optimizations, such as loop peeling, loop tiling, vector data prefetching, loop fission, loop fusion, loop unrolling, and loop interchange.

The scalar global optimizer operates on static single assignment (SSA) [1], [2] form. Because WHIRL is not an SSA language, it is translated into the SSA form inside the optimizer. The optimizer subsumes most 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.

After the lowering and optimization phases listed above, the code generator takes over, translating the WHIRL IR into CGIR (Code Generation Intermediate Representation), a low level IR reflecting the instruction set architecture of the target processor. Global and local register allocation, and final assembly codes emitting are performed on this IR. Most optimization phases in the code generator are dependent on target processor characteristics: they include control flow optimization, extended block (peephole) optimizer, integrated global/local scheduling (IGLS) [10], hyperblock formation [9], CG loop analysis and transformation, and software pipelining.

B. PAC DSP Architecture

The PAC DSP processor is a five-way issue VLIW, comprised of two ALUs (I-unit), two load/store units (M-unit), and a single scalar unit (B-unit). The M- and I- units are organized into two clusters, each containing a pair of both functional unit (FU) types. The B-unit is placed independently, and is in charge of branch operations; it is also capable of simple load/store and address arithmetic. The architecture is illustrated in Fig. 1.

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 used to communicate across clusters; only the B-unit, being able to access all D registers, is capable of executing such copy operations across clusters. Aside from being partitioned across two clusters, the D register files use a so-called '*ping-pong*' register file design, which is believed to achieve low-power consumption. This will be detailed in Section IV.

III. ORC2DSP: Design and Implementation

In this section, we describe our experiences in retargeting ORC to the PAC DSP architecture. The project is an on-going research effort; up to present, we are done with the porting of the compiler code generator and supporting toolchain, e.g., assembler and linker. Our compiler can now compile C/C++ source files and generate PAC DSP assembly codes. After processed through the assembler/linker, the generated binaries can be executed on the Instruction Set Simulator to evaluate the correctness and performance of both hardware and software components. Currently, the retargeted compiler works without optimization phases. And we are making efforts in both adapting the available ORC optimization phases, and discovering new optimization methods for the PAC architecture: especially important for tackling the peculiar distributed and 'ping-pong' register file design discussed in Section IV.

To initiate the compiler retargeting, we needed to study ORC internals and the IA-64 architecture. Since the original target of ORC is IA-64, knowledge of the IA-64 architecture is necessary for the understanding of ORC internals. After careful study of the PAC DSP architecture, we decided that the first step is to change the default word size of compiler conventions from 64-bit to 32-bit, because the PAC DSP is a 32-bit processor. This includes changing the register size description in the front-end to 32-bit, modifying 64-bit WHIRL lowering process, as well as a portion of 64-bit CG.

A. Target Information Table

In the next step, we started to retarget the Target Information Table (Targ_info), which is a kind of machine descriptions written in C constructs, and then they are compiled with the utility functions to generate the Targ_info library. The machine descriptions in the Targ_info library are used everywhere in almost all of the back-end CG components after the code expansion phase. There are three categories in Targ_info: Instruction Set Architecture (ISA), Application Binary Interface (ABI), and Processor description (PROC). Most of the system dependent descriptions are abstracted into Targ_info. The objectives of Targ_info is to provide the compiler a parameterized description of the target machine and system architecture, which separates architectural details from the compiler's algorithms, and to minimize compiler changes when targeting a new architecture.

We redesigned the machine descriptions to conform to the architecture of PAC DSP; all descriptions in Targ_info were rewritten. In this stage, we faced a difficulty in that the PAC DSP processor has two clusters with no shared register files. Some special purpose registers usually treated as always available to all operations (e.g., stack pointer and frame pointer) had to be duplicated in both clusters. This could not be done only by altering the machine descriptions, hence we required some small, sprinkled modifications of the CG code to implement our conventions.

B. Code Expansion Functions in CG

As we mentioned in Section II, after the multiple levels of WHIRL lowering, IR transforms into the "very low WHIRL" form; this very low WHIRL is then translated to CGIR operations, which are mapped into the target processor instruction set. This is done by a set of programmer provided callback functions that do the target-dependent selection of CG instructions. This is the main retargeting task required to do for this phase. The style of the interface is like 'Exp_OP', which expands an inputted OP into a list of CGIR operations that are appended to the operation list passed in. 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 traversed. For program control structures, the code generator will generate them 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 work we have done so far are the adaption of the code expansion functions to generate CGIR operations for the PAC architecture.

These PAC DSP instructions and their operand formats are specified in Targ_info. While implementing the code expansion functions, we found the lack of some instructions to complete these functions, and then made our suggestions to the DSP design team. This is the advantage gained by having the compiler team participating in processor design early on.

Besides the above issues, we need to bridge many architectural gaps between IA-64 and PAC DSP. For example, IA-64 passes parameters to functions through a register stack; the PAC DSP, not supporting such fancy features, requires modifying of the parameter passing mechanism in CG to use a runtime memory stack instead. Furthermore, other features in IA-64 not found in PAC DSP, such as control and data speculation, are needed to identify and to deal with.

IV. Exertion of Distributed Register Files

The PAC DSP features a highly partitioned register file design (referring to Fig. 1) that each cluster inside the architecture contains: A and AC register files, which is directly connected to the M-unit and I-unit respectively, and two D register files. Each D register file has only a single set of access ports, shared by both M- and I- units; each VLIW instruction word contains two bits field that controls the access ports to be switched between the files and the two FUs in each cluster. Hence, in each cycle, each FU can only access its dedicated D register file which is assigned by each instruction; accesses from two different FUs to one same D register file are mutually exclusive at the same time.

The rationale of this 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, but at the expense of an irregular architecture. With this design, the cross-interference between register allocation and instruction scheduling extremely increases, 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 file makes the details of register assignment and instruction scheduling highly dependent on each other. For example, the short code sequence:

```
mov TN1, 1
mov TN2, 2
```

moves two constants into two virtual registers, TN1 and TN2. These two instructions can be scheduled in parallel only if TN1 and TN2 are assigned registers from distinct D register files; if both are assigned to the same D file, they can only be scheduled and issued sequentially.

Our current proposed solution to this scenario, is to add a new pre-register allocation instruction scheduling phase by *simulated annealing* (SA). The design originates from Leupers' work [7], where a combined instruction scheduling/cluster assignment algorithm for the TI C6 VLIW DSP was proposed.

Leupers' original algorithm 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 a 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.

Adapting this simulated annealing solution for the PAC DSP involves changes in the formulation of optimized state: our search is for register file assignments for virtual registers, instead of the original bi-partitioning of the instructions. The overall algorithm is shown in Fig. 2.

Combined Instruction Scheduling/Register File Assignment by Simulated Annealing

Input: n unscheduled instructions without register allocation
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. Build initial register file assignments: randomly assign each virtual register to one of any register file, and record in RFA_map.
2. Do *modified list scheduling*† of the instructions with regard to RFA of operands, and set $sched_len$ to the computed total schedule length in cycles.
3. 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 ($0 < p_test < 1$).
4. Repeat the following sub-steps while $energy > threshold$:
   4a. Make change in RFA_map:
   randomly choose a virtual register, and assign it to a different register file.
   4b. With the new RFA assignment change, re-do modified list scheduling, and set new_sched_len to the new count of total schedule length.
   4c. Adjust $energy$, $sched_len$, and RFA_map by the following rules:
   - If $new_sched_len < sched_len$ then
     decrease $energy$ by 1, set $sched_len$ to new_sched_len,
     and keep the new RFA changes made in step 4a.
   - If $new_sched_len \geq sched_len$, get random number $0 \leq R \leq 1$:
     - If $R > p_test$,
       decrease $energy$ by 1, set $sched_len$ to new_sched_len,
       and keep the changes made in step 4a.
     - If $R \leq p_test$,
       increase $energy$ by 1 and revert changes made in step 4a.
5. Retain the final schedule and RFA_map as the output results.

†Regular list scheduling with management of cross cluster communication and 'ping-pong' register file access constraints.

Fig. 2. Proposed distributed register file assignment algorithm.

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 hardware registers officially, which is guided by the register file assignments (RFA_map).

V. Experiments and Architecture/Compiler Co-Exploration

Experiments are conducted on the DSPstone DSP benchmarks [13]. Table I shows the description of each DSPstone benchmark program. Three DSP processor/compiler combinations are evaluated, namely PAC DSP ORC compiler (version 20040703) with Instruction Set Simulator from ITRI, Blackfin's IDEVisualDSP++ 2.0 for Blackfin SP1, and TiC62x's IDECode Composer Studio v1.0. All benchmark programs are compiled with the -O0 option, i.e., disabling all optimizations.

We present the simulated execution time and code size numbers of the benchmark programs. We are currently trying to incorporate several optimization phases into the compiler; experimental results with these enhancements will be reported in future works. Note that the algorithm described in Section IV is not yet incorporated in the current compiler.

Fig. 3 compares the execution time in cycles of DSP benchmarks on the three DSP platforms. All data are normalized to the numbers of TiC62x. As shown in Fig. 3, the execution time results of PAC DSP

TABLE I

THE DESCRIPTIONS OF DSPSTONE BENCHMARK PROGRAMS.

Benchmark	Description	Line Num.
Convolution	convolution filter	36
Complex_multiply	DSP-kernel	31
IIR	an iir biquad (one section)	37
Matrix1*3	matrix computation	38
Real_update	DSP-kernel	36
N_complex_updates	DSP-kernel	41
Complex_update	DSP-kernel	41
N_real_updates	DSP-kernel	28
FIRDIM	filter	87
Matrix1	matrix computation	65
Matrix2	matrix computation	65
FIR	filter	54

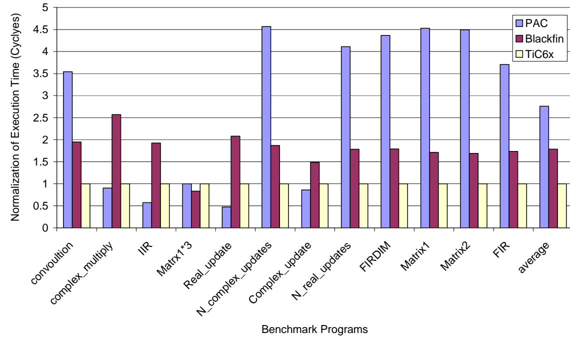

Fig. 3. The normalized execution time measured among PAC DSP, Blackfin, and TiC62x.

varies widely across different benchmarks. The best result of PAC DSP reaches 47% of TiC62x (Real_update) in the cycle count, in contrast with that the worst case approaches 456% (N_complex_updates). Comparing the average results, the cycle count of PAC DSP is 275% of TiC62x's.

We conjecture that the results are caused from two reasons: the first is the numerous hardware constraints in the current version of the PAC DSP architecture. One RAW hazard among different clusters needs 3 cycles, for instance, even the RAW hazard between the M-unit and I-unit in the same cluster takes 3 cycles due to the lack of bypass paths between the two FUs. There are other control and data hazards as well. The basic method dealing with such hazards is the insertion of nops to conform to latency requirements. Fig. 4 shows, in terms of code size, the percentage of nops can reach a maximum of 32% (N_complex_updates). As we can see, the hardware constraints existing on the current architecture can affect both performance and code size. We have proposed suggestions to the DSP design team for eliminating most hazards. The revision process is on-going and we expect the problems of hazards to be eased in the next PAC DSP architecture version. As for the compiler, we are also expecting more sophisticated instruction scheduling, which is being implemented for PAC DSP by our team. The second reason is the problem of cluster utilization, which our compiler currently does not do well at all. We expect substantial improvements in code quality as the method outlined in Section IV is eventually implemented.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduce the design and the implementation of our PAC DSP compiler based on the ORC infrastructure. The experiments are done with the PAC DSP prototype so that the co-exploration could drive the architecture of PAC to be improved. From this compiler infrastructure for the PAC DSP, we could take the experiences of porting to move on to other VLIW DSP processors, providing the qualified code generation beyond the hand-coded assembly. We are currently bringing the WHIRL-level optimizers and several CGIR optimization phases online, which will complete our compiler for the PAC DSP and also determine the effects of various compiler technologies upon the architecture design.

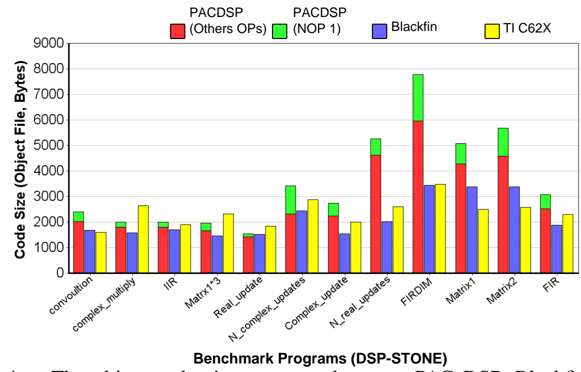

Fig. 4. The object code sizes measured among PAC DSP, Blackfin, and TiC62x.

ACKNOWLEDGMENT

We would like to thank all people that work for the PAC DSP compiler toolkits in NTHU, Taiwan, including P. S. Chen, Y. P. You, B. S. Hsu, etc. We would also like to thank everyone that worked on the development of the PAC DSP hardware in STC/ITRI, Taiwan.

This research work was supported in part by NSC-92-2213-E-007-074, NSC-92-2213-E-007-075, MOE research excellent project under grant no. 91-E-FA04-1-4, and MOEA research project under grant no. 91A0096EA of Taiwan.

REFERENCES

[1] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1990.

[2] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proc. of SIGPLAN 97*, pages 273–286, May 1997.

[3] 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.

[4] The GNU Compiler Collection. http://gcc.gnu.org.

[5] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-Based Compilation: An Introduction and Motivation. In *Proc. of the 28th Annual Int'l Symposium on Microarchitecture*, Dec. 1995, pp. 158–168

[6] 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.

[7] R. Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proc. Int'l Conference on Parallel Architecture and Compilation Techniques*, pages 291–300, Oct. 2000.

[8] Y. Liu, Z. Zhang, R. Qiao, and R. Ju. A Region-Based Compilation Infrastructure. In *Proc. of the 7th Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2003.

[9] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. of the 25st Int'l Symposium on Microarchitecture*, pages 45–54, 1992.

[10] S. Mantripragada, S. Jain, J. Dehnert. A New Framework for Integrated Global Local Scheduling In *Proc. 1998 Int'l Conference on Parallel Architecture and Compilation Techniques*, page 167, 1998.

[11] ReaCT-ILP Laboratory. Trimaran: An infrastructure for research in instruction-level parallelism. http://www.trimaran.org.

[12] The SUIF 2 compiler system. http://suif.stanford.edu/suif/suif2.

[13] V. Zivojnovic, J. Martinez, C. Schläger and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. *Proc. of ICSPAT*, Dallas, 1994.