PALF: Compiler Supports for Irregular Register Files in Clustered VLIW DSP Processors

Yung-Chia Lin, Yi-Ping You, Jenq-Kuen Lee*,†

Department of Computer Science, National Tsing Hua University
Hsinchu 30013, Taiwan

SUMMARY

Wide varieties of register file architectures — developed for embedded processors — have turned to aim at reducing the power dissipation and die size these years, by contrast with the traditional unified register file structures. This article presents a novel register allocation scheme for a clustered VLIW DSP, which is designed with distinctively banked register files in which port access is highly restricted. Whilst the organization of the register files is designed to decrease the power consumption by using fewer port connections, the cluster-based design makes register access across clusters an additional issue, and the switched-access nature of the register file demands further investigations into optimizing register assignment for increasing the instruction-level parallelism. We propose a heuristic algorithm, named *ping-pong aware local favorable* (PALF) register allocation, to obtain advantageous register allocation that is expected to better utilize irregular register file architectures. The results of experiments performed using a compiler based on the Open Research Compiler (ORC), showed significant performance improvement over the original ORC's approach, which is considered to be an optimized approach for common register file architectures.

KEY WORDS: register allocation; ping-pong register file; DSP; VLIW

*Correspondence to: Jenq-Kuen Lee. Postal address: is Professor Jenq-Kuen Lee, Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan.
†E-mail: jklee@cs.nthu.edu.tw, Tel.: +886-3-5715131 ext. 3519, Fax: +886-3-5723694.
Contract/grant sponsor: The work was supported in part by the NSC under grant nos. 94-2220-E-007-019 and 94-2220-E-007-020, by the Ministry of Economic Affairs under grant no. 94-EC-17-A-01-S1-034, and by the NSC program for promoting academic excellence of universities under grant no. 94-2752-E-007-004-PAE in Taiwan.

1. Introduction

The large computing power required by today's numerous embedded applications cannot be met by typical RISC embedded processors and low-end DSPs with little parallelism, which is driving investigations into DSPs with efficient parallel architectures. In addition to processor performance, the power consumption and chip die size of such DSPs are always significant concern. Exploiting instruction-level parallelism using VLIW architectures typically requires a large number of registers in a unified file to optimize resource utilization in the instruction scheduling whilst minimizing processor-memory traffic. This approach is not feasible for embedded DSP processors due to the design constraints of power dissipation and chip die size. Furthermore, the ability of all functional units to access larger number of registers demands more ports, which greatly increases the access time to register files, restricting the possible processor cycle duration and adding to the difficulty of the design [5]. To solve this weakness, the variety of decentralized register file architectures that have been developed for embedded processors in recent years have aimed at reducing the power dissipation and die size compared with traditional unified register file structures.

One of the techniques for decentralizing a unified register file is clustering, whereby register files are partitioned for different groups of functional units. For example, members of the Texas Instruments TMSC6x DSP [17] series use homogeneous clustered architectures with partitioned register banks, and the CEVA CEVA-X [2] architectures utilize heterogeneous clustered architectures with partitioned register files. Another technique for reducing power dissipation without performance degradation is restricting accessing of register file structures, such as in windowed register files [15] and hierarchical register files [16]. Unfortunately, the more specific accessing features and irregular register constraints usually apply to processors incorporating such partitioned register files. Accordingly, compilers require improved code generation, register allocation, and instruction scheduling schemes for attaining optimal performance with these processors.

This article describes a novel register allocation scheme for a clustered VLIW DSP, known as a Parallel Architecture Core (PAC) DSP [3, 4, 12, 13], which is designed with distinctively banked register files in which port access is highly restricted. The PAC DSP employs a heterogeneous design comprising a single scalar unit (for simple arithmetic, address calculation, and program flow control), plus two data-stream processing clusters, each containing a pair of load/store units and ALU/MAC units with powerful SIMD (Single Instruction stream, Multiple Data stream) capabilities; each unit in the clusters can utilize three types of register file, providing different accessing methods and constraints, and the scalar unit has its own accessible register file. The main feature of the register file architecture of the PAC DSP processor is that it incorporates a so-called *ping-pong register file structure* [9, 10], which is divided into two banks that can only be accessed in a mutually-exclusive manner — as a semicentralized register file among clusters and functional units within a cluster. This cluster-based design reduces the silicon area and power consumption due to fewer port connections being required; nevertheless, not only the new design makes register access across clusters an additional issue, but the switched-access nature of the ping-pong register file demands further investigations into register assignment for increasing the instruction-level parallelism.

We propose a heuristic algorithm, named *ping-pong aware local favorable* (PALF) register allocation, to improve the register allocation by efficiently utilizing the irregular register file architectures in the PAC DSP. The algorithm appropriately considers various characteristics in accessing different register files, and attempts to minimize the penalty associated with the interference between register allocation and instruction scheduling, while retaining desirable parallelism despite ping-pong register constraints and intercluster overheads. Experiments were performed with a compiler for the PAC DSP based on the Open Research Compiler (ORC), with the results showing a significant performance improvement over the original ORC's approach. Moreover, our proposed PALF scheme greatly reduces the compilation time compared with a simulated-annealing (SA) method [11], which is known as an effective but time-consuming search heuristic for optimization problems based on randomization techniques.

The remainder of this paper is organized as follows. In section 2 we introduce the processor architecture and register file organization of the PAC VLIW DSP. Section 3 briefly describes the complicated issues caused by the strong correlation between code generation, register allocation, and instruction scheduling in PAC architectures. The proposed PALF register allocation scheme with the presentation of an illustrative example is addressed in Section 4. The discussion of our evaluation and experimental results are provided in Section 5. Section 6 reviews related works. Finally, Section 7 concludes this paper.

2. Ping-pong Register Files with Clustered Architectures

This section overviews the VLIW architecture of the PAC DSP and its design of irregular register files.

2.1. PAC DSP Architectures

The PAC DSP features a clustered VLIW architecture that boosts scalability, and a large number of registers that are arranged as innovative heterogeneous and distinct partitioned register file structures. In contrast to the symmetric architectures of most DSPs available nowadays, the PAC DSP is constructed as a heterogeneous five-way issue VLIW architecture, comprising two integer ALUs (I-unit), two memory load/store units (M-unit), and the program sequence control unit/scalar unit (B-unit) that mainly executes control flow instructions such as "branch" and "jump". Each unit has its own executable subset of the instruction set, and each executable instruction has its own register access and constraints. The M- and I-units are organized in pairs, with each pair containing exactly one M-unit and one I-unit to form a cluster with associated register files. It is apparent that each cluster is logically appropriate for processing a single data stream, and in its current design the PAC DSP consists of two clusters to support a maximum workload capacity of two concurrent data streams. However, the scalability of the cluster design in the PAC DSP is such that extra clusters could be easily added to handle a larger data processing workload. 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 primarily takes charge of control flow instructions. The scalar unit, which is capable of

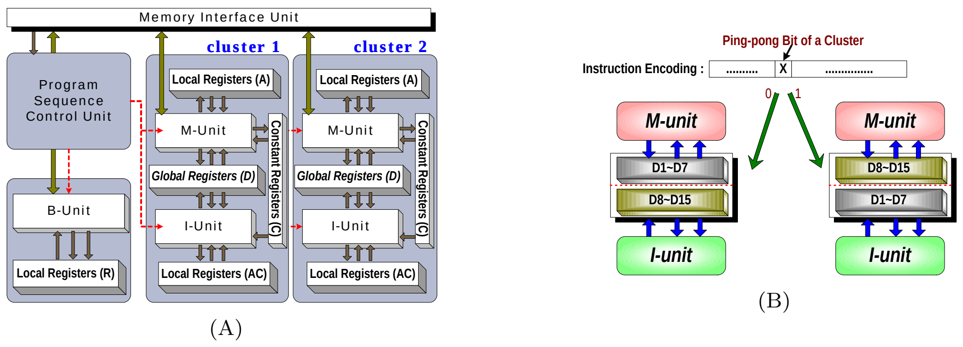


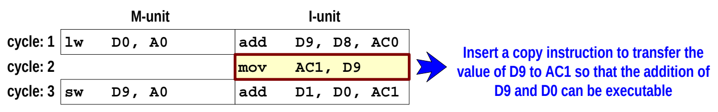
Figure 1. (A) The PAC DSP architecture (B) The ping-pong constraint for a single cluster

simple load/store and address arithmetic, is placed separately from the data-stream processing clusters, with its own register file. The overall architecture is illustrated in Fig. 1(A).

2.2. Irregular Register Files and Access Constraints

As shown in Fig. 1(A), registers in the PAC DSP are organized into several distinct partitioned register files (A, AC, D, and R) and organized into clusters. This reduces the wire connections between functional units and registers, and thereby decreases the chip area and power consumption. Dedicated local register files are attached to each unit in the processor: they include an R-register file, AC-register file, and A-register file, which are only accessible by the B-, I-, and M-units, respectively. A global register file named D-register file is designed to be shared by the pair of M- and I-units in each cluster. The D-register file is further partitioned into two banks to utilize instructional port switching technology in order to reduce the wire connections between the M- and I-units. This technology (i.e., the ping-pong register file structure), decreases the register-bank port connections that limit the accessibility of the two banks; in each cycle, the two functional units can only access different banks. Each instruction packet encodes the information of which bank is to be accessed for each functional unit in the cycle, so that the hardware can perform port switching between the D-register file banks and the functional units so as to implement data sharing within a cluster. By overlapping two different data-stream operations in a single cluster, we minimize the occurrence of the M- and I-units accessing the same data simultaneously; therefore, the access constraints of the ping-pong register file structure should have little impact on the performance. The assumed advantage of such a ping-pong register file structure design is that it consumes less power (due to its reduced number of read/write ports [16]) while retaining the similar data communication capability. Fig. 1(B) illustrates the constraints of the ping-pong register file. Besides local and global register files, each cluster contains an additional constant register file that is shared by both the M- and I-units as one of the read-only operand sources allowable by certain instructions. Only M-units can initialize the data in the constant register file.

The distributed and ping-pong register file organization used in PAC DSP are demonstrated to have 76.8% silicon area and 46.9% access time improvement comparing with its equivalent centralized register files [10]. It was reported that the DSP with such register file organization



	M-unit	I-unit
cycle: 1	`lw D0, A0`	`add D9, D8, AC0`
cycle: 2		`mov AC1, D9`
cycle: 3	`sw D9, A0`	`add D1, D0, AC1`

Insert a copy instruction to transfer the value of D9 to AC1 so that the addition of D9 and D0 can be executable

Figure 2. An example of inserting intracluster communications

could achieve comparable performance with state-of-the-art DSPs for popular DSP kernels [4, 10], which implies that lower power consumption is expectable due to the much less silicon area. This also reveals the importance of a good code generation strategy.

3. Optimizing Register Allocation in the Presence of Irregularity

For architectures supporting instruction-level parallelism, effective register allocation with scheduling is always one of the crucial issues to optimizing the code performance. Register allocation and instruction scheduling are typically performed in separate phases by most compilers so as to decrease the complexity of these two combinatorial optimization problems and interference between these two processes should be considered when determining a suitable execution sequence based on the architectural features of the target machine. If register allocation is performed after instruction scheduling, we may always obtain an infeasible register allocation for a certain schedule, particularly on architectures with a heterogeneous design and irregular constraints. Therefore, ensuring that the compiler performs register allocation before instruction scheduling is more favorable for the PAC architectures than for other target machines. Since register allocation may create additional dependencies and restrictions that impact subsequent scheduling, due to the intensive usage and overlapping liveness of registers, register allocation must be optimized such that the instruction-level parallelism could still be achieved thereafter.

Compared with other platforms, PAC architectures introduce severe problems in register allocation. First, the access constraints of ping-pong register file structures restrict the scheduling of two instructions that use the global D-register files in the same cluster in a cycle, irrespective of whether dependencies are present. Second, inserting additional code related to data communication into the original program will frequently be required while exploiting instruction-level parallelism because of the highly partitioned register files and their accessibility. A pair of instructions (implying send and receive) must be explicitly issued in the same instruction packet, for example, to transfer data from one cluster to another cluster, which uses the internal routing data path (between B- and M-units) of the memory interface unit [10] (as shown in Fig. 1(A)). This intercluster-communication mechanism used in PAC DSP is exhibited to have the lower hardware layout area and timing, comparing with the existing mechanisms in other clustered DSP architectures [10]. But dealing with this kind of intercluster communication in compilers is complicated, since both the occupation of issuing slots and the execution latency will affect the scheduling of the clustered programs that involve the pairs of explicit intercluster-communication instructions. Inserting code for intracluster data communication is also common after register allocation that utilizes more register files to

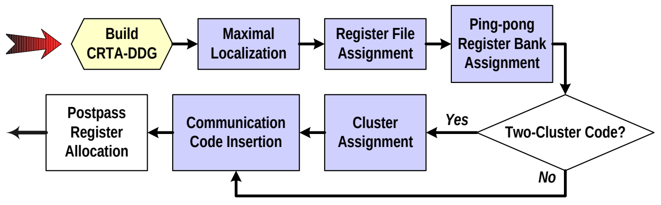


Figure 3. The flowchart of the PALF register allocation scheme

optimize the scheduling, as in the example shown in Fig. 2. To properly handle these issues with register allocation, we propose a heuristic approach to adapting general graph-coloring techniques for each register file and attaining an advantageous solution for PAC compilers.

4. PALF Register Allocation

In this section we present a register allocation algorithm that, given a dependency DAG (directed acyclic graph) [1] that describes the compilation regions, heuristically determines the appropriate register file/bank assignment and employs state-of-the-art graph-coloring register allocation for each assigned register file/bank in PAC architectures. An overall flowchart of the proposed register allocation algorithm is shown in Fig. 3. Our approach requires building an extended data-dependency DAG, called the component/register-type associated data-dependency graph (CRTA-DDG), which preserves the information of the execution and storage relationship for irregular constraint analysis in addition to the original partial order imposed by instruction-precedence constraints. An example code sequence and its initial CRTA-DDG are shown in Fig. 4. In the machine-level intermediate representation (CGIR) used by the ORC, an operand or result used in an instruction is represented as a TN (TemporaryName); a TN of register type is a virtual register required to be allocated to a physical register, and a TN of immediate type is converted to a literal value with an assembly format. Nodes in the CRTA-DDG represent instructions of the input code block, with the component-type association (which indicates the preferred functional unit to be scheduled for this node) and the register-type association (which annotates the favorite physical register file/bank, to where the operands/results will be allocated); the directed edges between the TNs represent data dependency that serializes the execution order to be followed in the scheduled code sequence. The advantage of using CRTA-DDG is that it clarifies the allocation and schedule restrictions for each node whilst considering the complex constraints in PAC architectures, while well-developed graph partitioning methods may still be readily applied to our register allocation algorithms. The PALF scheme can be organized into the following five phases:

1. Build the CRTA-DDG and apply the preferred functional-unit assignment to the default execution type of each instruction by "maximal localization" (see the next section).
2. Assign operands/results (required to be allocated to physical registers) of each node in the CRTA-DDG to the optimal register files.
3. Partition the operands/results assigned to the global ping-pong register files to the preferred register banks according to the strategy of optimizing ping-pong parallelism.

```
1: movi TN1, 5

2: movi TN2, 6

3: add  TN3, TN1, TN2
```

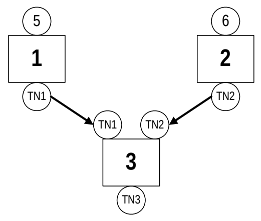


Figure 4. A simple code with its initial CTRA-DDG

4. Partition the nodes in the CRTA-DDG into two clusters if the "compiling for two clusters" option is set.
5. Insert nodes of the required communication code to avoid invalidities caused by the register file/bank assignment and cluster partitioning, followed by the physical register allocation for each register file.

For explaining the detail steps in Sec. 4.1–4.5, we also provide a simple example to illustrate how the PALF register allocation works. Fig. 5(a) shows the CRTA-DDG of an input program fragment: each rectangle labeled with its component-type association represents an operator, each circle represents a TN, and each edge presents a data dependency between two TNs; each type of circles represents one type of TNs as shown in the legend of Fig. 5. In order to avoid confusion between pre- and unassigned TNs, since our focus is on distributing unassigned TNs to register files, we simplify the representation by removing preassigned TNs (immediate values and dedicated registers) from the graphs in Fig. 5(b)–5(l).

4.1. Maximal Localization

Assume that a set of v nodes $V = \{n_1, n_2, \ldots, n_v\}$ with r TNs $R = \{t_1, t_2, \ldots, t_r\}$ are in a given CRTA-DDG, $G = (V, R, E)$, and the dependencies of these nodes are represented by e directed edges, each of which is denoted by (t_i, t_i), where $1 \leq i \leq r$ and $t_i \in R$. Assigning a node n to use functional units of type u is denoted by $n^u, u \in \{M, I, B\}$, and assigning a TN t to the register file f is denoted by $t^f, f \in \{D, A, AC, R, C\}$. If we define $TN(k)$ as the TNs of the node n_k, we can obtain a functional-unit assignment of v nodes that utilizes as many local register files as possible using the following strategies:

- We prefer to utilize M- and I-units more than the B-unit due to the fact that more instruction-level-parallelism may be exploited between instructions of M- and I-units by intra- or intercluster methods. Moreover, compared with the B-unit, M- and I-units have more register resources that are beneficial to optimizing the scheduling.
- Whereas the most concurrency may occur between M- and I-units, the utilization of M- and I-units should be balanced, so as to maximize the parallelism between instructions.
- Instructions using the same local register file in the same cluster cannot ever be executed in parallel, so that if two instructions have a data dependency, allocating the data to local register files will never be worse than allocating it to global register files. Also,

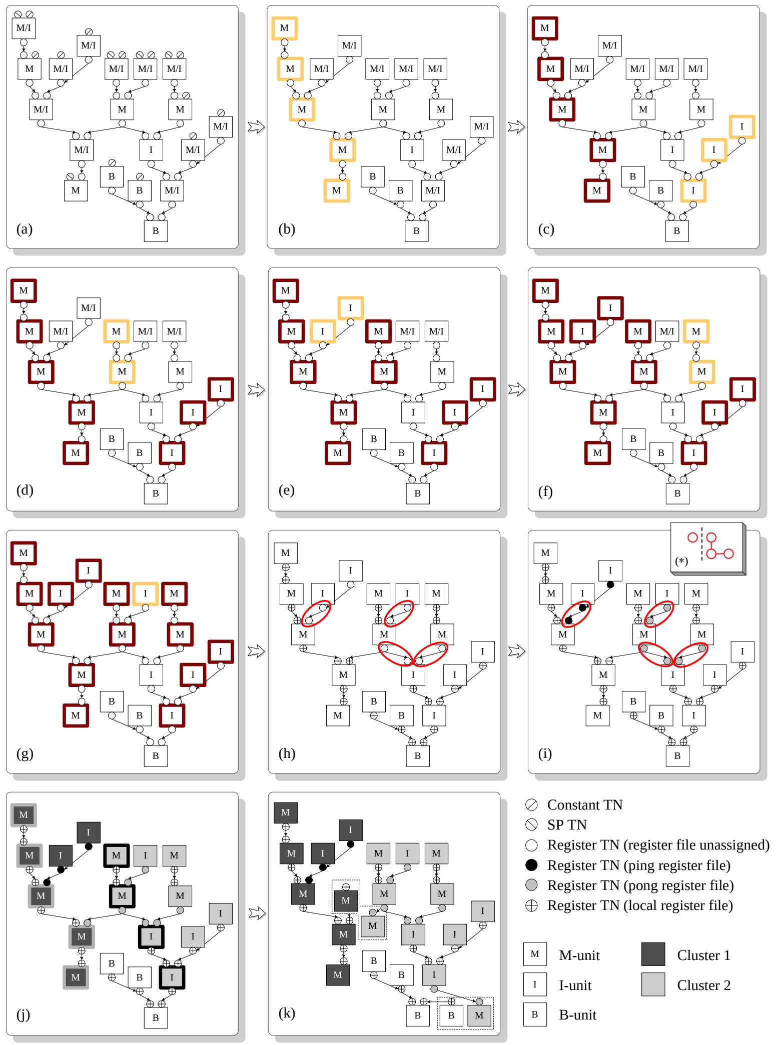

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(*)
(j)
(k)
Constant TN
SP TN
Register TN (register file unassigned)
Register TN (ping register file)
Register TN (pong register file)
Register TN (local register file)
M-unit
I-unit
B-unit
Cluster 1
Cluster 2

Figure 5. A running example for PALF register allocation

instructions with less global register file usage should result in less interference between scheduling (due to the constraints of the ping-pong structure) and register allocation.

By the strategies stated above, we greedily assign the v nodes to the appropriate functional unit denoted as u on a path-by-path basis; in each iteration we assign the nodes on a selected path to the same unit to maximize the possible utilization of local register files; the assignment follows the order of $u \equiv M$, $u \equiv I$, $u \equiv M$, $u \equiv I$, $\ldots$, and finally $u \equiv B$:

1. Let Ψ be the set of nodes unassigned to any functional unit.
2. Select a maximal set of s nodes $S = \{n_{p_1}, n_{p_2}, \ldots, n_{p_s}\} \subseteq \Psi$, where $\forall n_{p_q}$, $1 \leq q < s$, $\exists (t_i, t_i)$, $1 \leq i \leq r$, and $t_i \in TN(p_q) \cap TN(p_{q+1})$. These s nodes have a precedence

ordering (which means they are not parallelizable) so that they are preferred to use the same functional unit denoted as u.

3. Assign all s nodes to u, denoted as $S = \{{n_{p_1}}^u, {n_{p_2}}^u, \ldots, {n_{p_s}}^u\}$, and remove S from Ψ.
4. Repeat steps 1–3 until $\Psi = \emptyset$.
5. Annotate each edge connecting between one node assigned to B-unit and the other assigned to M- or I-unit, with the mark that indicates pending intercluster communication code insertion.

Since an M-unit has direct access to memory and intercluster communication, we prioritize the assigning order that M-unit is prior to I-unit so that it may result in less communication code to be inserted in the final phase of PALF register allocation. Another order of I-unit prior to M-unit could also be used for some cases, and the related experiment and discussion could be found in Sec. 5. The data dependency between the B-unit and other units always need to be fixed by the insertion of intercluster communication instructions because this is the only method to share data between these units in the architecture.

Fig. 5(b)–5(g) show the processing scenario; it is preferable that all nodes on a critical path (i.e., the path with the maximum number of nodes) in the graph operate on the same functional unit so that their operands can be stored in local register files. Rectangles with a light thick-border represent the nodes in the longest data-flow path, and those with a dark thick-border represent the nodes that have been assigned a functional unit.

4.2. Register File Assignment

After determining the functional-unit type of all the instructions, we are ready to assign the register file used for each TN in G. We first locate the TNs that must be allocated to global D-register files to avoid unnecessary communication caused by data sharing between different functional units, and then let other TNs be allocated to the appropriate local registers associated with their functional-unit assignments. While assigning all TNs to either global or local register files, we may optionally try to instead use constant register files where possible so as to aggressively reduce the possible pressure on global and local register files if the compiler option to utilize the constant registers is set by software developers.

The assignment steps are detailed as follows:

1. Let Ω be the set of TNs that are not assigned to any register file.
2. $\forall (t_i, t_i), 1 \le i \le r$, where $t_i \in TN(l) \cap TN(m), 1 \le l \ne m \le v$ with either $\exists {n_l}^I {n_m}^M$ or $\exists {n_l}^M {n_m}^I$ (e.g. the selected edges in Fig. 5(h)). Assign t_i to global D-register files, denoted as ${t_i}^D$.
3. Remove all ${t_i}^D, 1 \le i \le r$, from Ω.
4. Assign $t_i \in \Omega, 1 \le i \le r$ to the associated local register file of u, where $t_i \in TN(z), \forall {n_z}^u, 1 \le z \le v$, and $u \in \{A, AC, R\}$.

The practical approach for the example code is as follows. We first determine which edges connect two TNs that operate on different functional units — the TNs must be allocated to a global register file so that they can be accessed for the two operations. The remaining unassigned TNs of register type are then allocated to the corresponding local register files.

Fig. 5(h) shows the results of register file assignments: TNs with the mark of $\oplus$ are allocated to local register files, and other TNs are allocated to global register files.

4.3. Node Partitioning for Ping-pong Bank Assignment

To optimize the register allocation for ping-pong register files, we employ a partitioning procedure to determine which bank of ping-pong register file structures should be used for each TN assigned to D-register allocation. The partitioning for ping-pong bank assignment is developed to increase the opportunity of parallelizing ping-pong bank access in the schedule; we will assign TNs whose associated instructions may interfere with each other to different banks of D-register files. Let Δ be the set of TNs that are assigned to D-register files in Sec. 4.2, and w be the number of these TNs. We use the following methods to partition Δ into two groups, X (using the ping bank) and Y (using the pong bank), according to the threshold number λ of w:

- Build a new graph G^* based on the inverse of the original subgraph that contains only the edges implying $({t_i}^D, {t_i}^D), \forall t_i \in \Delta, 1 \leq i \leq r$ in G where $t_i \in TN(l) \cap TN(m), 1 \leq l \neq m \leq v$ with either $\exists {n_l}^I {n_m}^M$ or $\exists {n_l}^M {n_m}^I$, and the corresponding nodes connected with these edges: $G^* = (V^*, E^*)$; each node of V^* maps to an edge $({t_i}^D, {t_i}^D)$ selected in G and each edge of E^* represents a node n_z in the original subgraph, where $1 \leq z \leq v$ and $TN(z) \supseteq \{{t_i}^D, {t_j}^D\}, 1 \leq i \neq j \leq r$ (which means the nodes accessing two different TNs that are both allocated to global register files).
- If w is larger than λ, we apply multilevel k-way graph partitioning algorithms [8] to G^* to obtain two balanced partitions (in number of nodes) with the minimal number of cut-edges; one partition produces X and the other produces Y. We need to annotate all the cut-edges after the partitioning with the mark that indicates pending intracluster communication code insertion, since a cut-edge implies an illegal scenario that an instruction requires to access both the ping-pong banks simultaneously, and the insertion of the additional copy code is needed to fix this scenario; the additional code is added prior to this instruction and copy one available operand from the global register file into the local register file that is accessible.
- If w is smaller than or equal to λ, we partition Δ into the two groups that are subgraphs without any edge connecting between X and Y, thereby reducing the amount of communication between different ping-pong banks because the benefit of potential parallelism between too few instructions may be more than offset by any additional communication.

Currently the threshold λ is set equal to the total number of D-register banks, by reason of that the cost of cut-edges is obviously infeasible in the case with w smaller than or equal to the total number of D-register banks. Fig. 5(i) and the small diagram (Fig. 5(*)) on it illustrate how the graph is built for partitioning and show the possible result in this phase: TNs with black-filled and gray-filled circles are assigned to ping and pong register banks, respectively.

4.4. Node Partitioning for Cluster Assignment

Depending on the compilation options set by software developers, compilers may generate code that utilizes either two clusters to improve the performance or one cluster to reduce the power consumption. In generating code that will be scheduled onto two clusters, we employ an iterative partitioning method based on cost models to obtain two sets of the total graph G by the following scheme:

1. Given a CRTA-DDG, $G = (V, R, E)$, which is assumed to be a disjoint union of k subgraphs $(1 \leq k)$, we could always natively partition G into k pairwise disjoint parts (that each part has disjoint set of nodes from one another and no edge exists between parts): ${G^\dagger}_i, 1 \leq i \leq k$, where $G = \bigcup {G^\dagger}_i, 1 \leq i \leq k$, and ${G^\dagger}_i \cap {G^\dagger}_j = \emptyset, \forall 1 \leq i \neq j \leq k$. This partitioning could be easily done by traversing all the edges once in the linear time complexity.
2. We group ${G^\dagger}_1, \ldots, {G^\dagger}_k$ into two separate sets, G° and $G^\bullet$, where $G^\circ \supset {G^\dagger}_1$. The two group sets should be formed to approximately balance the overall schedule length by estimating a cost model; the current cost model is using the number of instructions, divided by 1.4 (that could be adjusted to the potential parallelism in a cluster), as the schedule length for a single cluster; the overall schedule length should be dominated by the longest length between the two sets. This model could be further revised to use a pseudo scheduler to calculate the more precise schedule length than the current one. The grouping procedure is that we first sort ${G^\dagger}_1, \ldots, {G^\dagger}_k$ in the order of the schedule length (by the cost model), and then greedily insert the longest one from the ungrouped set into one of the two group sets that results in the better overall schedule length.
3. If $G^\bullet$ is the empty set (which means $k = 1$), or the difference of estimations from the cost model by step 2 for G° and $G^\bullet$ is larger than a threshold, we attempt to balance the two group sets again, moving appropriate nodes from the set with the longer predicted schedule length into the other set by iteratively evaluating the cost model with the addition of the occurred cross-cluster communication-latency cycles.
4. We annotate all edges in G across G° and $G^\bullet$ with the mark that indicates pending intercluster communication-code insertion.

Based on the approach described above, two critical paths (whose nodes are thick-bordered in Fig. 5(j)) and their adjacent nodes are discovered. Fig. 5(j) presents the possible result of the cluster assignment, where dark gray, medium gray, and light gray nodes represent cluster 1, cluster 2, and the scalar unit, respectively.

4.5. Communication-Insertion/Postpass Register Allocation

If there is any pending intercluster or intracluster communication code insertion generated in the previous phases, we will insert the corresponding instructions. The pair instructions mentioned in Sec. 3 are inserted for intercluster communications. The processing of intracluster communications are mentioned in Sec. 4.3. Finally, according to the register file assignments for all r TNs of v nodes in G, we apply register allocation based on graph-coloring heuristics for each register file to allocate each TN of register type to a physical register.

Table I. Average benchmark performance (normalized in cycles) for various choices in PALF scheme

	local-preferred	*global*-preferred	PALF (*METIS*)	PALF (*CHACO*)
M-preferred	1.096	1.066	1.067	1.067
I-preferred	1.479	1.065	1.063	1.063
M/**I**-alternated	1.278	1.019	1	1.016
I/**M**-alternated	1.294	1.018	1.007	1.024

Fig. 5(k) shows the final result after inserting communication operations among cluster 1, cluster 2, and the scalar unit; the pair instructions added for intercluster communications are marked with the dashed frames. By using list scheduling, this example code could perform in 11 cycles with our PALF approach in the PAC DSP architecture, while a naive single-cluster register allocation that always assigns an operation to an M-unit, if possible, and always allocates the operand(s) of an M- or I-unit to the ping or pong register file, if possible, could only produce the result in 20 cycles.

5. Experiment and Discussion

The PALF register allocation scheme was implemented on ORC infrastructure, and the performance was evaluated on the PAC DSP with a cycle-accurate instruction set simulator [18], by running the DSPstone benchmark suite [19]. We first inspected the proposed PALF scheme and validated the effect of several phases. Table I shows the average performance normalized in cycles for various combinations of preferences, during the phases described in Sections 4.1-4.3. The four rows list the results with different policies of unit assignment in Section 4.1: "M-preferred" — to assign as many instructions to M-unit as possible, "I-preferred" — to assign as many instructions to I-unit as possible, "M/I-alternated" — to use the default "maximal localization" rule, and "I/M-alternated" — to use the "maximal localization" with I-unit prior to M-unit. The four columns list the results with different rules to manage register file assignment: "local-preferred" — to allocate as many TNs to local register files as possible, "global-preferred" — to allocate as many TNs to global register files as possible, "PALF (METIS)" — to process as described in Sections 4.2–4.3 by using the METIS package [8] to implement the multilevel k-way partitioning (for ping-pong bank assignment), and "PALF (CHACO)" — to process as described in Sections 4.2–4.3 by using the CHACO package [6] to implement the randomized partitioning instead for comparison. While all the results were evaluated with the processing of clustering in Section 4.4, we also tested with the single clustering and acquired the similar results about the relationship among the 16 configurations.

In general, "M/I-alternated" + "PALF (METIS)", the default process of PALF, produces the best result as we expected. The results of "PALF (METIS)" are better than or equal to those of "PALF (CHACO)" since the multilevel k-way partitioning tends to minimize the number of edge cuts, which result in additional intracluster communication. The randomized partitioning may only obtain the same results when the input size is small, as shown in the results using "M-preferred" and "I-preferred". The results of "global-preferred", which seems

Table II. Average benchmark performance (normalized in cycles) for various policies in clustering

single	two (PALF)	two (naive)	two (random)
1.02025	1	1.00044	1.39597

only a little worse than PALF, present that there are often too much data-dependency existing in a basic block of our programs, limiting the parallelism achievable; the little parallelism also explains why "M/I-alternated" is usually better than "I/M-alternated" but with only slight difference. Besides, the results of "local-preferred" reflects the penalty of intra-/intercluster communication, which also give strong evidence that "M/I alternated" is the best choice for our scheme.

Table II shows the average performance normalized in cycles for four different clustering manners: "single" — the non-clustering scheme, "two (PALF)" — the clustering scheme of PALF, "two (naive)" — the same as "two (PALF)" without the third step in Section 4.4, and "two (random)" — the randomized clustering scheme. The results reveal that our clustering policy in PALF profits the performance only a little in our tests; there are two reasons to explain this: first, only low parallelism could be exploited due to complex data-dependency; second, we found that the code might include additional penalty caused by the more register-spills generated by calling conventions in the clustered code than in the non-clustered one, since the clustering decreases the occurrence of allocating the different TNs with non-overlapping live-ranges to the same register.

We next examined the effectiveness of the proposed PALF register allocation. Four register allocation schemes were evaluated: (1) the original register allocation in the ORC, (2) SA register allocation [14], (3) PALF (METIS), and (4) PALF (CHACO). Fig. 6 shows the benchmark performance gain relative to using original register allocation with the ORC as the baseline. The figure shows that our PALF approach provides an average speedup of 1.88 and 1.85 relative to the original approach when using the multilevel k-way partitioning and the randomized partitioning in Section 4.3, respectively, while the SA register allocation has an average speedup of 2.16, which can be considered as a lower bound since SA already has been approved to be an effective approach for such optimization problems [20]. However, a comparison of the compilation time between the SA approach and the PALF register allocation scheme reveals the better applicability of our proposed approach, as shown in Fig. 7. Our PALF approach implemented using the METIS and CHACO libraries achieves average compilation speedups of 31.62 and 29.16, respectively, compared with SA register allocation.

6. Related Work

There have been many prior studies in clustered VLIW architectures based on instruction scheduling. Ellis [21] proposed a popular method, BUG (Bottom-Up Greedy), to partition operations on a trace with scheduling in a two-phases sequence. Ozer et al. [22] proposed an algorithm, unified-assign-and-schedule (UAS), to combine cluster assignment and instruction scheduling together into a single phase. Nystrom and Eichenberger [23] presented an algorithm for modulo scheduling to perform partitioning with heuristics in a pre-modulo scheduling pass

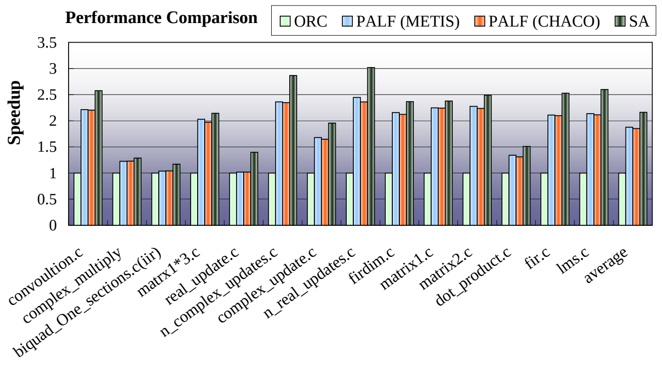


Figure 6. Benchmark performance speedups for three register allocation schemes

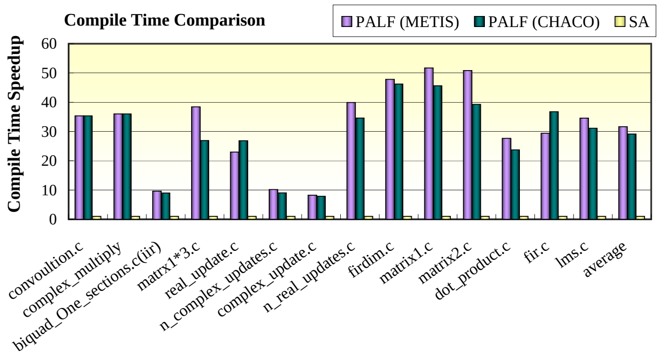


Figure 7. Benchmark compilation time comparison

to allow modulo scheduling to be effective. Codina et al. [24] used a similar strategy as UAS but focused on modulo scheduling. Using graph partitioners for clustering operations has also been investigated in several studies [5, 25]. Most of these studies were much different from our work, since our architecture has more features than clustering, and we mainly focus on register allocation targeted toward acyclic code to tie in with the phase ordering in ORC infrastructure.

7. Conclusion

Embedded DSPs are currently designed to exploit a high degree of instruction-level parallelism subject to the technological constraints of cycle time, power dissipation, and die area. The techniques used in their design commonly tend to include a clustered/partitioned architecture and low-power register file structures. In this work, we developed and implemented a novel heuristic approach for generating code for PAC VLIW DSPs that incorporates highly partitioned register files with a unique ping-pong structure. At the heart of this work is a proposed register file/bank assignment scheme that could be integrated with the existing unified register allocation methodologies to yield a feasible solution. The experimental evaluation using benchmark programs indicates that our register allocation scheme for PAC VLIW DSPs utilizes all register files efficiently and delivers comparable results to a SA approach but with a much lower compilation time.

REFERENCES

1. A. V. Aho, J. D. Ullman, and R. Sethi: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, 1986.
2. CEVA: CEVA-X1620 Datasheet. CEVA, 2004.
3. D. Chang and M. Baron: Taiwan's roadmap to leadership in design. Microprocessor Report, In-Stat/MDR, December 2004. http://www.mdronline.com/mpr/archive/mpr_2004.html
4. D. C.-W. Chang, C.-W. Jen, I-T. Liao, J.-K. Lee, W.-F. Chen, S.-Y. Tseng: PAC DSP Core and Application Processors. Procs. of the IEEE Int. Conf. on Multimedia & Expo, Toronto, July 9–12, 2006.
5. A. Capitanio, N. Dutt, and A. Nicolau: Partitioned register files for VLIW's: A preliminary analysis of tradeoffs. Procs. of the 25th Int. Symp. on Microarchitecture: Portland, OR, December 1–4, 1992; 292–300.
6. B. Hendrickson and R. Leland: The Chaco user's guide, version 2.0. Tech Report SAND95-2344. Sandia National Laboratories, Albuquerque, NM, October 1994.
7. R. Ju, S. Chan, and C. Wu: Open research compiler for the Itanium family. Tutorial at the 34th Int. Symp. on Microarchitecture, December 2001.
8. G. Karypis and V. Kumar: A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM Journal of Scientific Computing 1999; 20(1): 359–392.
9. T.-J. Lin, C.-C. Lee, C.-W. Liu, and C.-W. Jen: A Novel Register Organization for VLIW Digital Signal Processors. Procs. of 2005 IEEE Int. Symp. on VLSI Design, Automation, and Test, 2005; 335–338.
10. 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.
11. S Kirkpatrick, C. Gelatt, and P. Vecchi: Optimization by simulated annealing. Science, 220:671-679, 1983.
12. T.-J. Lin, C.-C. Chang. C.-C. Lee, and C.-W. Jen: An Efficient VLIW DSP Architecture for Baseband Processing. Procs. of the 21th Int. Conf. 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. Procs. of the 15th ACM Great Lakes symposium on VLSI, April 2005.
14. Y.-C. Lin, C.-L. Tang, C.-J. Wu, M.-Y. Hung, Y.-P. You, Y.-C. Moo, S.-Y. Chen, and J.-K. Lee: Compiler Supports and Optimizations for PAC VLIW DSP Processors. Procs. of the 18th Int. Workshop on Languages and Compilers for Parallel Computing, 2005.
15. R. A. Ravindran, R. M. Senger, E. D. Marsman, G. S. Dasika, M. R. Guthaus, S. A. Mahlke, and R. B. Brown: Increasing the number of effective registers in a low-power processor using a windowed register file. Procs. of the Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems, 2003; 125–136.
16. S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens: Register organization for media processing. Procs. of Int. Symp. on High Performance Computer Architecture, 2000; 375–386.
17. Texas Instruments: TMS320C64x Technical Overview. Texas Instruments, Feb 2000.
18. Chang, C. and Marculescu, D.: Design and Analysis of a Low Power VLIW DSP Core. Procs. of IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006.
19. V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr: DSPstone: A DSP-oriented benchmarking methodology. Procs. of Int. Conf. on Signal Processing and Technology, 1995; 715–720.
20. R. Leupers: Instruction scheduling for clustered VLIW DSPs. Procs. of Int. Conf. on Parallel Architecture and Compilation Techniques, Oct 2000; 291–300.
21. J. Ellis: Bulldog: A Compiler for VLIW Architectures. MIT Press, Cambridge, MA, 1985.
22. E. Ozer, S. Banerjia and T. Conte: Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. Procs. of the 31st Int. Symp. on Microarchitectures, Nov. 1998; 308–315.
23. E. Nystrom and A. E. Eichenberger: Effective cluster assignment for modulo scheduling. Procs. of the 31th Int. Symp. on Microarchitecture, Nov. 1998; 103–114.
24. J. Codina, J. Sanchez, and A. Gonzalez: URACAM: A unified register allocation, cluster assignment and modulo scheduling approach. Procs. of the 34th Int. Symp. on Microarchitecture, Dec. 2001.
25. M. Chu, K. Fan, and S. Mahlke: Region-based hierarchical operation partitioning for multicluster processors. Procs. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation, New York; 300–311.