Copy Propagation Optimizations for VLIW DSP Processors with Distributed Register Files *

Chung-Ju Wu Sheng-Yuan Chen Jenq-Kuen Lee

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

Abstract. High-performance and low-power VLIW DSP processors are increasingly deployed on embedded devices to process video and multimedia applications. For reducing power and cost in designs of VLIW DSP processors, distributed register files and multi-bank register architectures are being adopted to eliminate the amount of read/write ports in register files. This presents new challenges for devising compiler optimization schemes for such architectures. In our research work, we address the compiler optimization issues for PAC architecture, which is a 5-way issue DSP processor with distributed register files. We show how to support an important class of compiler optimization problems, known as copy propagations, for such architecture. We illustrate that a naive deployment of copy propagations in embedded VLIW DSP processors with distributed register files might result in performance anomaly. In our proposed scheme, we derive a communication cost model by cluster distance, register port pressures, and the movement type of register sets. This cost model is used to guide the data flow analysis for supporting copy propagations over PAC architecture. Experimental results show that our schemes are effective to prevent performance anomaly with copy propagations over embedded VLIW DSP processors with distributed files.

1 Introduction

Digital signal processors (DSPs) have been found widely used in an increasing number of computationally intensive applications in the fields such as mobile systems. As the communication applications are moving towards the conflicting requirements of high-performance and low-power consumption, DSPs have evolved into a style of large computation resources combined with restricted and/or specialized data paths and register storages. In modern VLIW DSPs, computation resources are divided into clusters with its own local register files to reduce the hardware complexity.

* This paper is submitted to *LCPC 2006*. The correspondence author is Jenq Kuen Lee. His e-mail is jklee@cs.nthu.edu.tw, phone number is 886-3-5715131 EXT. 33519, FAX number is 886-3-5723694. His postal address is Prof. Jenq- Kuen Lee, Department of Computer Science, National Tsing-Hua Univ., Hsinchu, Taiwan.

In cluster-based architectures, the compiler plays an important role to generate proper codes over multiple clusters to work around the restrictions of the hardware. Data flow analysis is an important compiler optimization technique. Available expressions, live variables, copy propagations, reaching definitions, or other useful sets of properties can be computed for all points in a program using a generic algorithmic framework. Current research results in compiler optimizations for cluster-based architectures have focused on partitioning register files to work with instruction scheduling [13] [16]. However, it remains open how the conventional data flow analysis scheme can be incorporated into optimizations over embedded VLIW DSP processors with distributed files by taking communication costs into account.

In this paper, we present a case study to illustrate how to address this register communication issue for an important class of compiler optimization problems, known as copy propagations, for PAC architectures. Parallel Architecture Core (PAC) is a 5-way VLIW DSP processors with distributed register cluster files and multi-bank register architectures (known as ping-pong architectures) [1] [8] [9]. Copy propagation is in the family of data flow equations and traditionally known as an effective method used as a compiler phase to combine with common available expression elimination and dead code elimination schemes. We illustrate that a naive deployment of copy propagations in embedded VLIW DSP processors with distributed files might result in performance anomaly, a reversal effect of performance optimizations. In our proposed scheme, we derive a communication cost model by the cluster distance, register port pressures, and the distance among different type of register banks. The profits of copy propagations are also modeled among program graphs. We then use this cost model to guide the data flow analysis for supporting copy propagations for PAC architectures. The algorithm is modeled with a flavor of shortest path problem with the considerations of shared edges in program graphs. Our model will avoid performance anomaly produced by conventional copy propagations over distributed register file architectures. Our compiler infrastructure is based on ORC/Open-64 compiler infrastructure and with our efforts to retarget them in a VLIW DSP environments with multi-cluster and distributed register architectures. We also present experimental results with DSPstone benchmark to show our schemes are effective to support copy propagations over embedded VLIW DSP processors with distributed register files.

The remainders of this paper are organized as follows. In Section 2, we will introduce the processor architecture and register file organizations of PAC VLIW DSP processors. Section 3 presents motivating examples to point out performance anomaly phenomenon with copy propagations over embedded VLIW DSP processors with irregular register files. Section 4 then presents our algorithm and solution to this problem. Next, Section 5 gives experimental results. Finally, Section 6 presents the related work and discussions, and Section 7 concludes this paper.

2 PAC DSP Architecture

The Parallel Architecture Core (PAC) is a 32bit, fixed-point, clustered digital signal processor with five way VLIW pipeline. PAC DSP has two Arithmetic Logic Units (ALU), two Load/Store Units (LSU), and one single Scalar unit. The ALU and LSU are organized into two clusters, each containing a pair of both functional unit (FU) types and one distinct partitioned register file set. The Scalar unit can deal with branch operations, and is also capable of load/store and address arithmetic operations. The architecture is illustrated in Figure 1.

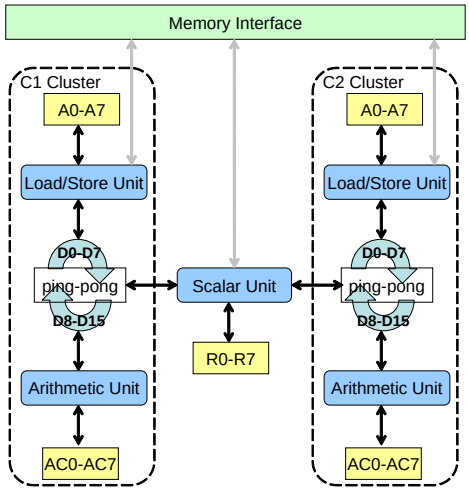

Fig. 1. The PAC DSP architecture illustration

As in Figure 1, the register file structure in each cluster is highly partitioned and distributed. PAC DSP contains four distinct register files. The A, AC, and R register files are private registers, directly attached to and only accessible by each LSU, ALU, and Scalar unit, respectively. The D register files are shared within one cluster and can be used to communicate across clusters. Each of the D-register files have only 3 read ports and 2 write ports (3R/2W). Among them, 1R/1W are dedicated to the Scalar Unit, leaving only 2R/1W for the cluster FUs to use. The remaining set of 2R/1W ports are not enough to connect to both cluster FUs simultaneously. Instead, they are switched between the LSU/ALU: during each cycle, the access ports of each of the two D-register files (in a single cluster) may be connected to the LSU or ALU, but not both. This means that access of the two D-register files are mutually-exclusive for each FU, and each LSU/ALU can access only one of them during each cycle. For one individual public register sub-block, we can't perform reading and writing on it in two different FUs at the same time. Due to this back-and-forth style of register file access, we call this a 'ping-pong' register file structure. We believe this

special register file design can help us achieve low-power consumption because it retains an effective way of data communication with less wire connections between FUs and registers. Note that the public register files are shared register but can only be accessible by either LSU or ALU at one time. PAC DSP processor [1] is currently developed at ITRI STC, and our laboratory is currently collaborating with ITRI STC under MOEA projects for the challenging work to develop high-performance and low-power toolkits for embedded systems under PAC platforms [12], [16], [18], [19], and [20].

3 Motivating Examples

This section gives examples to motivate the needs of our optimization schemes. Consider the code fragment below:

```
Code Fragment 1
    (1)  x := t3;
    (2)  a[t2] := t5;
    (3)  a[t4] := x + t6;
    (4)  a[t7] := x + t8;
```

The traditional technique for compilers to optimize the above code is to use `t3` for `x`, wherever possible after the copy statement `x := t3`. The related work in optimizing this code sequence by the copy propagation technique can be found in Aho's book [4]. Following the common data flow analysis and copy propagation applied to Code Fragment 1, we have the optimized code below:

```
Code Fragment 2
    (1)  x := t3;
    (2)  a[t2] := t5;
    (3)  a[t4] := t3 + t6;
    (4)  a[t7] := t3 + t8;
```

This propagation can remove all data dependency produced by `x := t3`, providing the compiler with possibility to eliminate the assignment `x := t3`. However, the scheme above is not appropriate for the design of PAC DSP architecture. Due to this specific-architecture design with clustering and distributed register files , extra intercluster-communication code needs to be inserted if there occurs the data flow across clusters. Suppose `t3` is allocated to a different cluster from `t6`,`t8`, and `x`, the insertion of intercluster-communication code will then need to be done if applying conventional copy propagation. Such overhead of communication code increases the total cycles of the optimized code compared with non-optimized one. Figure 2 is an example of VLIW code fragment. Code bundle at the left-hand side represents one propagation path exists from Cluster 2 to Cluster 1, i.e. TN2 (Temporary Name, which is referred as a virtual register representation) can be propagated from Cluster 2 to Cluster 1. Code bundle at the right-hand side shows extra inter-communication costs needed after propagation.

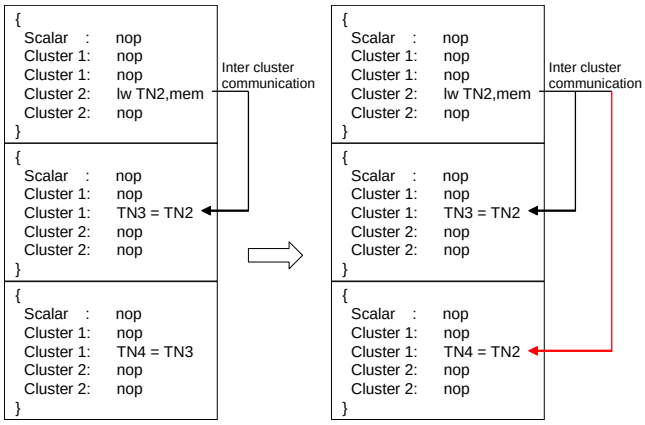

Fig. 2. A VLIW Code Example for Inter Cluster Communication

Not only does the clustered design make data flow across clusters an additional issue, but also compiler needs to take the distributed register file structure into consideration. The private access nature of A and AC registers makes data flows more difficult. 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`, `ac2` are allocated to the variables `x`, `t3`, `t6` and `t8`, 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 line 3 and line 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 LSU and AC register files are also only attached to ALU. If `d2` is replaced with `a1`, compiler must insert extra copy instructions for private register access properties. 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.

In addition, the reduced wire connection is another important issue. Referring to the short Code Fragment 4 and Code Fragment 5, the left part of Figure 3 illustrates how Code Fragment 4 being scheduled into bundles and also shows read/write ports attached to D register files, and the right part of Figure 3 shows Code Fragment 5. Note that we arrange all the instructions into cluster 1 to avoid the cross-interference between port pressure and clustered design because we want to focus on the port pressure issue.

```
Code Fragment 4
    (1)  LW   d2,  a0,  16
    (2)  COPY  ac2, d3
    (3)  SW   d4,  a0,  40
    (4)  ADD  d5,  ac2, d2
```

After propagating `d3` to `ac2`, the resulted code is as follows:

```
Code Fragment 5
    (1)  LW   d2,  a0,  16
    (2)  COPY  ac2, d3
    (3)  SW   d4,  a0,  40
    (4)  ADD  d5,  d3, d2
```

We observe that there are 3 read ports needed in the second bundle, but our architecture only has 2 available read ports and 1 available write port. Due to the port constraint, the bundle must be separated. Figure 4 illustrates the final bundles of Code Fragment 5.

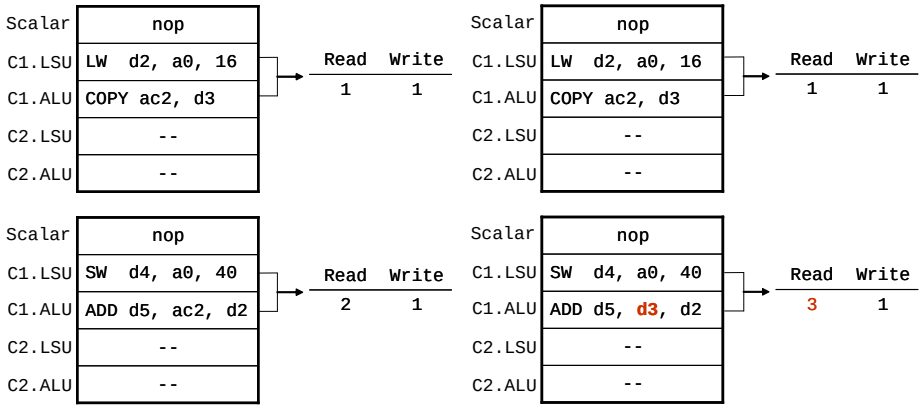

Fig. 3. The bundles of Code Fragment 4

In summary, Figure 2 illustrates a scenario that there might be data flows from one cluster to another cluster. In Code Fragment 3, due to private registers can only be accessible by the corresponding function units, compiler has to allocate a new temporary register first and then move data from one register to the temporary register. Propagation makes access between two different private register file types increases register pressure. In Code Fragment 4 and 5, compiler does not need to spend extra registers or communications through memory. However, due to the reduced wire connections with global register files, the instruction scheduler can only schedule them into two different bundles and fill the empty slots with nops. We name the above three behaviors as 'performance anomaly'. In the following section, this problem is solved by deriving cost

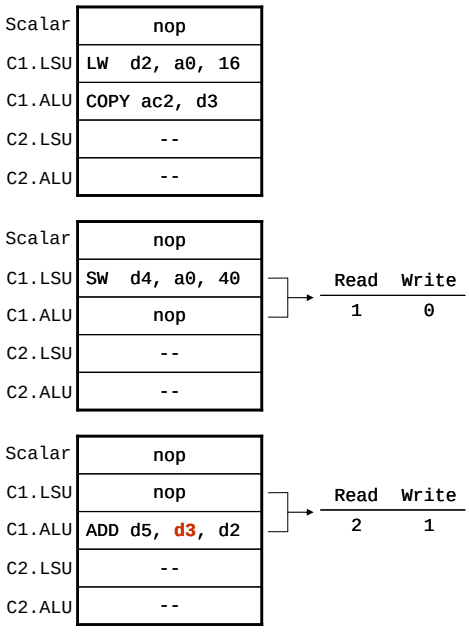

Fig. 4. Schedule of Code Fragment 5 according to the register ports constraint

models and using the cost models to guide the copy propagation process for performance benefits.

4 Enhanced Data Flow Analysis on PAC Architecture

4.1 Cost Model and Algorithm

As mentioned in section 3, a naive application of data flow analysis scheme to programs on PAC DSP actually increases execution cycles because of memory interface access, register pressure, and separated bundles. In the following discussions, we will first introduce our cost models, and we will then develop an algorithm based on our cost models to guide the analysis process to avoid performance anomaly.

Our cost models for data flow analysis are to model the total weights we spend and the total gains we get. We have defined several attributes for evaluating the costs and gains of data propagation. The total weights of data flow path are the costs of propagation from the TN n of instruction p to the TN m of instruction q. Note that one TN (Temporary Name) of register type is referred as a virtual register required to be allocated to a physical register in the machine level IR used in compilers.

We also build equations to evaluate the extra communication costs of data propagations from variable n to variable m, i.e. the three performance anomaly

effects mentioned in section 3. We define our cost equation as follows:

$$Cost(n, m) = PP(n, m) + RP(n, m) + CBC(n, m), \tag{1}$$

where $PP(n, m)$ shows the port pressure caused by data flows from variable n to variable m. And $PP(n, m)$ is the extra cycles caused by the separation of bundles. We rewrite $PP(n, m)$ as

$$PP(n, m) = \lceil \frac{k_r - p_r}{p_r} \rceil + \lceil \frac{k_w - p_w}{p_w} \rceil, \tag{2}$$

where k_r/k_w is the number of read/write ports needed after data flows from variable n to variable m, and p_r/p_w is the number of read/write port constraint we have mentioned in section 2. k_r, k_w, p_r, and p_w need to be calculated according to the instructions in n and m, respectively.

$RP(n, m)$ represents the register pressure caused by data access between two different private register file types. Due to the distributed register file constraint, one extra copy instruction must be inserted to move data from one private register to a temporary register. $RP(n, m)$ returns the number of extra copy instructions. $CBC(n, m)$ returns the cost of memory access cycles when propagating across clusters. PAC DSP provides a special instruction pair (BDT and BDR) to broadcast data from one cluster to another.

Table 1 shows the corresponding cost functions used in each kind of data flow path. Note that we have local register A for data movement units, local register AC for ALU unit, register D as a ping-pong register to be interleaved between ALU and load/store units.

Data Flow	Cluster1.D	Cluster1.A	Cluster1.AC	Cluster2.D	Cluster2.A	Cluster2.AC
Cluster1.D	–	PP	PP	CBC	CBC	CBC
Cluster1.A	–	–	RP	CBC	CBC	CBC
Cluster1.AC	–	RP	–	CBC	CBC	CBC
Cluster2.D	CBC	CBC	CBC	–	PP	PP
Cluster2.A	CBC	CBC	CBC	–	–	RP
Cluster2.AC	CBC	CBC	CBC	–	RP	–

Table 1. Costs in each data flow path

The total gains are the reduced communication codes and the reduced copy assignments from propagations between TN n of instruction p to TN m of instruction q. We define the total gains as

$$Gain(n, m) = RCC(n, m) + \sum_{j \in path(n,m)} ACA(c[j]), \tag{3}$$

where $RCC(n, m)$ represents the original communication cost on this n-m path, and the communication cost can possibly be reduced if the assignment is done

directly instead of going through a sequence of copy propagations. $ACA(c[j])$ is to calculate the number of all available copy assignments which can be reduced along this $n - m$ data flow path. $c[j]$ is the intermediate copy assignment on $n - m$ path, and $path(n, m)$ represents the set of intermediate nodes in the flow path from n to m.

We view each variable as a node, and the data flows between those nodes form an acyclic DFG. Our analysis algorithm mainly comprise 2 procedures. In the first procedure, we perform the ordinary copy propagation algorithm illustrated in Figure 5. Let U be the 'universal' set of all copy statements in the program. Define $c\_gen[B]$ to be the set of all copies generated in block B and $c\_kill[b]$ to be the set of copies in U that are killed in B [4]. The conventional copy propagation algorithm can be stated with the following equation.

$$out[B] = c\_gen[B] \cup (in[B] - c\_kill[B]) \tag{4}$$

$$in[B] = \bigcap_{P \textit{ is a predecessor of } B} out[P] \quad \text{for } B \text{ not initial} \tag{5}$$

$$in[B_1] = \emptyset \quad \text{where } B_1 \text{ is the initial block} \tag{6}$$

Note that we don't perform the step 3 in Figure 5 at this time. After performing the first two steps of the copy propagation algorithm in Figure 5, we keep every traversed nodes in the same data flow path into a list L. While finding out all possible nodes, we import these nodes in L into the other equations (equation (1), and equation (3)) to find a data propagation path with the best profits. Finally, we propagate data according to the best data flow path. Note that we can choose not to take the data flow path if no path makes a profit. The algorithm in Figure 6 shows the whole processes of both the weight evaluation and the data flow selection.

The first step of enhanced data flow algorithm does the initial work to find out the concerned nodes of a propagation path from $node_n$ to $node_m$. The nodes form an acyclic data flow tree. Step 2 evaluates the initial weight of each edge (i, j). By step 2, we can calculate the initial weight of this $n-m$ path. The initial weight can be estimated by $Gain(n, m)$ since they tell the same cost but from different views. In step 3, we perform both the equation (1) and the equation (3) to check if there are some short cuts to go. Note that the *gains* represent both the communication cost and the available copy assignments we can save by going through the short cut, and the *costs* show the extra inter/intra cluster costs on the short cut. We iterate several times over this tree graph, using k as an index. On the kth iteration, we get the best profit solution to the propagation path finding problem, where the paths only use vertices numbered n to k. Note that if this results in a better profit path, we remember it. Due to the comparison with initial weight, the outcome path must be no more than the weight of no-propagation method and naive propagation method. After iterations, all possible short cuts have been examined, and we output the proper propagation path by step 4. The algorithm produces a matrix p, which, for each pair of nodes u and v, contains an intermediate node on the least cost path from u to v. So the best

Algorithm 1: Copy Propagation Algorithm

Input: A flow graph G, with ud-chains.
c_in[B] represents the solution to Equation (4), (5), (6). And du-chains.
Output: A revised flow graph.
Method: For each copy s: **x:=y** do the following.

1. Determine those uses of **x** that are reached by this definition of **x**, namely, s: **x:=y**.
2. Determine whether for every use of **x** found in (1), s is in c_in[B], where B is the block of this particular use, and moreover, no definitions of **x** or y occur prior to this use of **x** within B.
3. If s meets the conditions of (2), then remove s and replace all uses of **x** found in (1) by **y**.

Fig. 5. Copy Propagation Algorithm.

profit path from u to v is the best profit path from u to $p[u,v]$, followed by the best profit path from $p[u,v]$ to v. Step 3 of the algorithm is done with a flavor of the shortest path problem, but only now that we model the problem for copy propagation and register communications.

4.2 Advanced Estimation Algorithm

The goal of the *Enhanced Data-Flow Analysis Algorithm* is to collect the information of the weights and gains of propagation at each point in a program. If multiple nodes have the same ancestors, they should share the weights and gains from their ancestors. The Figure 7 shows a new evaluating method to solve this sharing problem on a propagation tree.

In the first step, we deal with the issue for shared edges for determining which path is doing copy propagation and which path does not. In that case, the intermediate assignment will not be eliminated by dead code eliminations. This can still be done, but we need to reflect this in our cost model for GAINS calculated in equation (3). Three small steps are performed. In step 1.a, we first find the set of all propagation paths. Note that we only need to find out those paths are not sub-path of other paths, as dealing with the long path (non-proper sub-path) in copy propagation will cover all cases. Next in Step 1.b, we try to mark the intermediate stops in all propagation paths according to output of Path routine in Figure 6 for each path. Next in Step 1.c, we re-adjust the cost model for GAINS if there are intermediate nodes which will not be eliminated eventually in dead code elimination phase due to the share edge decides to keep the intermediate stops.

Algorithm 2: Enhanced Data Flow Analysis

Input: Inputs in Copy Propagation Algorithm. (Figure 5).
Output: A proper propagation path.

1. Perform the first and the second steps in Copy Propagation Algorithm in Figure 5 to traverse all possible propagation nodes on $n - m$ path .
2. **for** $i = n$ to m **do**
   for $j = n$ to m **do**
   /* Evaluate the initial weight, $w[i, j]$. */
   /* This weight includes the communication*/
   /* costs and all the copy assignments */
   /* along path before propagation. */
   Estimate the initial weight $w[i, j]$;
   end
   end
3. **for** k = n to m **do**
   for i = n to m **do**
   Compute $Gain(i, k)$ and $Cost(i, k)$.
   for j = n to m **do**
   Compute $Gain(k, j)$ and $Cost(k, j)$.
   $profit = Gain(i, k) - Cost(i, k) +$
   $Gain(k, j) - Cost(k, j)$;
   if $(w[i, j] - profit) < w[i, j]$ **do**
   $w[i, j] = w[i, j] - profit$;
   $p[i, j] = k$;
   end
   end
   end
   end
4. /* Output a proper propagation path from */
   /* u to v */
   Path (u, v, p) {
   $k = p[u, v]$;
   if $(k == Null)$ **return**;
   Path(u, k);
   output the node k;
   Path(k, v);
   }

Fig. 6. The Enhanced Data Flow Analysis Algorithm.

Algorithm 3: Available Copy Assignment Estimation Algorithm

Input: A propagation tree
Output: Proper weights of all propagation paths

Step 1.a:
Find the set of all the propagation paths (all the non-proper propagation paths), PP.
Step 1.b:
For each path $p \in PP$ do {
Mark each element in the output of Path routine in Figure 6 for p as intermediate stop.
}
Step 1.c:
For each path $p \in PP$ do {
Compare the elements of intermediate stops in p with the elements from the output of Path routine in Figure 6 for p. If there are additional elements in the path of p marked as intermediate stops, revise cost for the GAINS of p.
}
Step 2.a:
For each path $p \in PP$ do {
Use reference counting to count the reference count for each node in p.
}
Step 2.b:
For each path $p \in PP$ do {
Revise GAINS for p by using the reference counting information acquired in the previous step.
}

Fig. 7. Available Copy Assignment Estimation Algorithm.

In step 2, we also deal with shared edges, but for fine-tuning the cost model. As if there are shared edges, the gains of copy propagations should be counted only once (or the benefit needs to be distributed among shared paths). A reference counting scheme can be used to see the amount of sharing. This is done in Step 2.a. This information can then be used to re-adjust the cost model for GAINS in equation (3).

5 Infrastructure Designs and Experiments

We now first describe our compiler testbed for our proposed copy propagations over cluster-based architecture and distributed register files. Our compiler platform is based on ORC and we retarget the compiler infrastructure for PAC architecture. ORC is an open-source compiler infrastructure released by Intel. It is originally designed for IA-64. ORC is made up of different phases. The ORC compilation 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, is a part of the Pro64 compiler released by SGI [11]. PAC architecture introduces additional issues with register allocation under comparison between different platforms. In our compiler infrastructure, we first implemented a partitioning scheme to partition the register file among clusters. This is known Ping-pong Aware Local Favorable (PALF) register allocation [12] [16] to obtain a preferable register allocation scheme that well partitions register usage into the irregular register file architectures in PAC DSP processor. The algorithm involves the proper consideration of various characteristics in accessing different register files, and attempts to minimize the penalty caused by the interference of register allocation and instruction scheduling, with retaining desirable parallelism over ping-pong register constraints and inter-cluster overheads. After the phase of register allocation and instruction selections, we then move into the phase of EBO (basic block optimizations). EBO was a phase originally in ORC for the basic block optimizations and carrying out optimization such as copy propagations, constant folding, dead code eleminations. Our enhanced copy propgation algorithm is implemented in this phase.

We use the PAC DSP architecture described in section 2 as the target architecture for our experiments. The proposed enhanced data flow analysis framework is incorporated into the compiler tool with PAC ORC [5], and evaluated by the ISS simulator designed by ITRI DSP team. We also implement the METIS graph partitioning library [6] for the register allocation scheme. The benchmarks used in our experiment are from the floating-point version of DSP-stone benchmark suite [7]. Notice that benchmarks are indexed with numbers to identify the specified basic block we used in this experiment. We focused on the major basic blocks as copy propagation was implemented in peephole optimizations for basic block optimizations.

Three versions are compared in our research work. The base version is one without copy propagation mechanism. The original version is one from a work that only performs the naive copy propagation algorithm in Figure 5. The En-

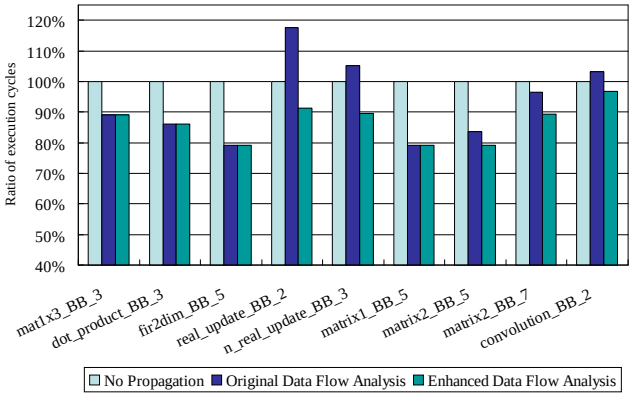

Fig. 8. Ratio of execution cycles in basic block codes.

hanced Data-Flow Analysis scheme proposed in our work is to perform all phases in Figure 6. Both the original version and the Enhanced Data-Flow Analysis scheme are incorporated with dead code elimination.

Figure 8 shows that our scheme can achieve an average of 15.0% reduction comparing to the base method. Note that from our experiment, the original copy propagation version (Figure 5) suffers a performance loss in benchmarks *real_update_BB_2*, *n_real_update_BB_3*, and *convolution_BB_2*. That's because the naive copy propagation produces lots of inter communication codes and register pressure in *real_update_BB_2* and *n_real_update_BB_3*. The test program *convolution_BB_2* suffers redundant inter communication codes. Although the naive propagation version can reduce some of the unnecessary copy assignments, it is still out-performed by our proposed scheme. The test programs *mat1x3_BB_3*, *dot_prod-uct_BB_3*, *fir2dim_BB_5*, and *matrix1_BB_5* show that our methods can keep the good nature of the naive propagation version. And the other benchmarks prove that our proposed methods can also reduce the performance anomaly over by distributed register files.

6 Related Work

High-performance and low-power VLIW DSP processors are of interests lately for embedded systems to handle multimedia applications. To achieve this goal, clustered architecture is one well-known strategy. Examples are given in this work [8] [9] [10]. The presence of distributed register file architecture presents a challenge for compiler code generations. Earlier work focused on the partitioning of register file to combine with instruction scheduler [12] [13] [16]. While the partitioning scheme for distributed register file is important, there are more challenging problems ahead as evidenced in this work that we need to handle copy propagations over such architectures. [17] provides techniques to support

copy propagation during register allocation which is known as node coalescing in the interference graph. Our work presents an approximation to deal with these issues in post register phase that we give cost models to guide the process for copy propagations on embedded VLIW DSP processors with distributed register files and multi-bank register structures.

Performance anomaly was earlier also found in the problem of array operation synthesis. The work for Fortran 90 and HPF programs [14] [15] was done in the context of array operations and source languages for distributed memory parallel machines. With the distributed memory hierarchies moving from memory layers into register levels, the performance anomaly was also observed in the register layers. Previous work was done in loop levels and source levels, while this work needs to carefully model register communication and architecture constraints in the instruction levels.

7 Conclusion

In this paper, we presented an enhanced framework for copy propagations over VLIW architectures with distributed register files. This presented a case study to address the issues for how to address compiler optimizations for conventional optimizations schemes over distributed register file architectures. Experimental results show that our scheme can maintain the benefits of copy propagation optimizations while prevent performance anomaly. Future work will include the integration of cost models to cover more cases of compiler optimization schemes such as common available expression eliminations.

References

1. David Chang and Max Baron: Taiwan's Roadmap to Leadership in Design. Microprocessor Report, In-Stat/MDR, December 2004. http://www.mdronline.com/mpr/archive/mpr_2004.html.
2. C. M. Overstreet, R. Cherinka, M. Tohki, and R. Sparks. Support of software maintenance using data flow analysis. Technical Report TR-94-07, Old Dominion University, Computer Science Department, June 1994.
3. C. M. Overstreet, R. Cherinka, and R. Sparks. Using bidirectional data flow analysis to support software reuse. Technical Report TR-94-09, Old Dominion University, Computer Science Department, June 1994.
4. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, November 1985.
5. Cheng-Wei Chen, Yung-Chia Lin, Chung-Ling Tang, Jenq-Kuen Lee. ORC2DSP: Compiler Infrastructure Supports for VLIW DSP Processors. *IEEE VLSI TSA*, April 27-29, 2005.
6. George Karypis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1): 359-392, 1999.
7. V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology*, pp.715-720, October 1994.

8. 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.
9. Tay-Jyi Lin, Chie-Min Chao, Chia-Hsien Liu, Pi-Chen Hsiao, Shin-Kai Chen, Li-Chun Lin, Chih-Wei Liu, Chein-Wei Jen. Computer architecture: A unified processor architecture for RISC & VLIW DSP. In *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, April 2005.
10. S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. *International Symposium on High Performance Computer Architecture*, pp.375-386, 2000,
11. SGI - Developer Central Open Source - Pro64 http://oss.sgi.com/projects/Pro64/.
12. Yung-Chia Lin, Yi-Ping You, Jenq-Kuen Lee. Register Allocation for VLIW DSP Processors with Irregular Register Files. *International Workshop on Languages and Compilers for Parallel Computing*, January 2006.
13. R. Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proceedings of International Conference on Parallel Architecture and Compilation Techniques*, pp.291-300, October 2000.
14. Gwan-Hwan Hwang, Jenq-Kuen Lee and Roy Dz-Ching Ju. A Function-Composition Approach to Synthesize Fortran 90 Array Operations. *Journal of Parallel and Distributed Computing*, 54, 1-47, 1998.
15. Gwan-Hwan Hwang, Jenq-Kuen Lee, Array Operation Synthesis to Optimize HPF Programs on Distributed Memory Machines. *Journal of Parallel and Distributed Computing*, 61, 467-500, 2001.
16. Yung-Chia Lin, Chung-Lin Tang, Chung-Ju Wu, Jenq-Kuen Lee. Compiler Supports and Optimizations for PAC VLIW DSP Processors. *Languages and Compilers for Parallel Computing*, 2005.
17. Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *Conference on Programming Language Design and Implementation*, 1992.
18. Yi-Ping You, Ching-Ren Lee, Jenq-Kuen Lee. Compilers for Leakage Power Reductions. *ACM Transactions on Design Automation of Electronic Systems*, Volume 11, Issue 1, pp.147-166, January 2006.
19. Yi-Ping You, Chung-Wen Huang, Jenq-Kuen Lee. A Sink-N-Hoist Framework for Leakage Power Reduction. *ACM EMSOFT*, September 2005.
20. Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, Jenq-Kuen Lee. Interprocedural Probabilistic Pointer Analysis. *IEEE Transactions on Parallel and Distributed Systems*, Volume 15, Issue 10, pp.893-907, October 2004.