Value Cloning For Architectures with Partitioned Register Banks* (Extended Abstract)

Darla Kuras† Steve Carr‡ Philip Sweany‡

Abstract

The ideal LIW (Long Instruction Word) architecture has one large register bank multiported to several functional units. Too many ports make this structure unrealizable in practice. As an alternative, we consider a LIW architecture whose register bank is partitioned into banks and whose functional units are distributed among the banks. Unfortunately, this design has a problem; operands that reside in different banks, but are required for the same operation, must be copied to a common bank. Some copies are necessary in order to take advantage of all functional units; however, too many can be costly. A non-trivial task of a LIW compiler is to find balance between inter-bank copies and functional unit utilization.

In this paper we present an experiment that shows how value cloning (keeping the same value in multiple register banks) can improve code generation on LIW architectures with partitioned register banks. Our results on an 8-wide LIW architecture show a 9% performance improvement with value cloning over code generated without cloning. Thus, we conclude that cloning is an important transformation for code generation in the presence of partitioned register banks.

1 Introduction

Computer architects are turning to higher and higher levels of instruction-level parallelism (ILP) to increase CPU performance. Unfortunately, large amounts of ILP hardware and aggressive instruction scheduling techniques put large demands on a machine's register resources. With these demands, it becomes difficult to maintain a single monolithic register bank. The number of ports required for such a register bank severely hampers access time [1, 3]. Partitioned register banks are one mechanism for providing high degrees of ILP with a high clock rate (Texas Instruments already produces several DSP chips that have partitioned register banks to support high ILP [7].) Unfortunately, partitioned register banks may inhibit achieved ILP. An instruction may be able to be scheduled in a particular cycle, but if its data resides in a register bank that is not accessible to the available functional unit, extra instructions must be inserted to move the data to the register bank of an available functional unit in order to allow execution. Therefore, a compiler must deal not only with achieving maximal parallelism via aggressive scheduling, but also with data placement among a set of register banks.

In this paper, we show how cloning read-only values and induction variables (*i.e.*, keeping copies of the values in multiple register banks) can improve the code generated for partitioned register banks. We begin with a discussion of previous work in generating code for partitioned register banks. We then describe our method, detail an experiment studying value cloning and finish with our conclusions.

2 Previous Work

Ellis described the first solution to the problem of generating code for partitioned register banks in his dissertation [2]. His method, called BUG (bottom-up greedy), is applied to a scheduling context at a time

*This research has been partially supported by Texas Instruments and NSF grant CCR-980781.

†Digital Equipment Corporation, 110 Spit Brook Road (ZK02-3/N30), Nashua NH 03062.

‡Department of Computer Science, Michigan Technological University, Houghton MI 49931-1295.

(*e.g.,* a trace). His method is intimately intertwined with instruction scheduling and utilizes machine-dependent details within the partitioning algorithm. Our method abstracts away machine-dependent details from partitioning, a feature that is extremely important in the context of a retargetable compiler.

Capitanio et al. present a code-generation technique for limited connectivity LIWs in [1]. They report results for two of the seven loops tested, that, for three functional units each with a dedicated register bank, show degradation in performance of 57% and 69% over code obtained with three functional units and a single register bank.

Janssen and Corporaal propose an architecture called a Transport Triggered Architecture (TTA) that has an interconnection network between functional units and register banks so that each functional unit can access each register bank [5]. Although they report results that show significantly less degradation than observed with partitioned register banks, we believe the interconnection network is too costly in both time and space.

3 Method

In this section, we give an overview of our partitioning scheme and a description of how value cloning fits into this scheme.

3.1 The Register Component Graph

Central to our partitioning scheme is the use of a register component graph (RCG). This graph is built for each function in the intermediate code. Each node of the graph represents a symbolic register and edges are drawn between registers used in the same operation. Note that there can be more than one edge between two nodes. Edges are drawn only between source and destination registers, based on a policy that the operation will be scheduled in the cluster that contains the destination register. Thus, unconnected registers in the RCG are good candidates to be placed in different register banks.

To aid in partitioning the RCG, we use another graph called the destination register interference graph (DRIG). The nodes in this graph represent destinations of operations in the program. There is an edge between two nodes in the DRIG if the two destinations registers appear as the destination of an operation in the same scheduled LIW instruction. The DRIG tells which operation should go in separate functional units.

Coloring decisions for a node in the RCG are done by calculating a benefit associated with each color (or register bank) and assigning the color with the highest benefit for that particular node. The benefit for a color, c, is increased for each adjacent node in the RCG that has been colored with c. The benefit for c is decreased for each adjacent node in the DRIG that has been colored with c. In computing benefits, we place more emphasis on parallelism. So, the reduction in benefit for each color from the DRIG is greater than the increase from the RCG.

3.2 Cloning

In a good partitioning method, interbank copies can be reduced by maintaining multiple copies of data (clones) across register banks. Not all values are good cloning candidates. Extra instructions may be required to clone and maintain a duplicate value. These instructions should not be more costly than the value's respective inter-bank copies. In this work two types of values are cloned: loop induction variables and read only variables.

Loop induction variables are used very frequently within a loop body and thus are good candidates for cloning. Since these variables are redefined in each iteration of the loop, their clones do require some maintenance; however, a simple increment may be cheaper than a copy. Thus, these clones generally have a smaller impact on the overall schedule length.

Unlike induction variables, read only clones require no additional instructions other than the actual cloning, which is no more expensive than an inter-bank copy and can be executed outside the loop body.

3.3 Cloning Example

The best way to see the potential benefit of value cloning is through an example. Consider the following example.

```
DO I = 1, 100
  A(I) = A(I) + 1
ENDDO
```

To expose some parallelism, we can unroll the loop one time and obtain the following:

```
DO I = 1, 100
  A(I) = A(I) + 1
  A(I+1) = A(I+1) + 1
ENDDO
```

Essentially, we've restructured the loop by packing two iterations of the old loop into one loop body. Note that with the exception of the induction variable, I, the two statements in this loop are completely independent. Intermediate code for this loop is listed in Figure 1 and its corresponding RCG in Figure 4. Read only and induction variables have been shaded in the RCG. From this graph we can obtain the partitioning: $P1 = r1, r2, r3, r4, r5, r6, P2 = r7, r8, r9, r10, r11$. The intermediate operations can then be scheduled for a LIW with 2 functional units and 2 register banks as in Figure 2. If each operation takes 1 cycle, total execution will take 2 + 50(10) + 1 = 503 cycles. If we remove the nodes that represent read only and induction variables (*r1* and *r2*) in this loop, we obtain the RCG in Figure 5. It can easily be bi-partitioned as follows: $P1 = r3, r4, r5, r6, P2 = r7, r8, r9, r10, r11$. If we create clones *r101* and *r102* of *r1* and *r2*, respectively, we can obtain the schedule in Figure 3 that takes only 4 + 50(8) + 1 = 405 cycles, almost a 20% improvement.

lda r1, A
load r2, I
L: mult r3, r2, 4
add r4, r1, r3
load r5, [r4]
add r6, r5, 1
store [r4], r6
add r7, r2, 1
mult r8, r7, r4
add r9, r1, r8
load r10, [r9]
add r11, r10, 1
store [r9], r11
add r2, r2, 2
cmp r2, 100
ble L
store I, r2

Figure 1: Code

lda r1, A	
load r2, I	
L: move r101, r1	
move r102, r2	
mult r3, r2, r4	add r7, r102, 1
add r4, r1, r3	mult r8, r7, 4
load r5, [r4]	add r9, r101, r8
add r6, r5, 1	load r10, [r9]
store [r4], r6	add r11, r10, 1
add r2, r2, 2	store [r9], r11
cmp r2, 100	
ble L	
store I, r2	

Figure 2: Original RCG Code

lda r1, A	
move r101, r1	
load r2, I	
move r102, r2	
L: mult r3, r2, r4	add r7, r102, 1
add r4, r1, r3	mult r8, r7, 4
load r5, [r4]	add r9, r101, r8
add r6, r5, 1	load r10, [r9]
store [r4], r6	add r11, r10, 1
add r2, r2, 2	store [r9], r11
cmp r2, 100	add r102, r102, 2
ble L	
store I, r2	

Figure 3: New RCG Code

3.4 Experimental Evaluation

We implemented value cloning in the Rocket [6] compiler and evaluated it on 49 software pipelined loops. These loops were extracted from Fortran benchmarks taken from the SPEC benchmark suite and from the numerical routines found in the book by Forsythe, Malcom and Moler [4]. They are the loops Rocket can software pipeline and partition using the greedy algorithm.

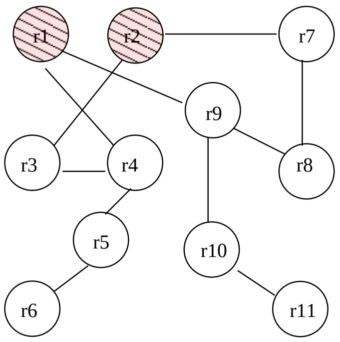

Figure 4: RCG for Loop

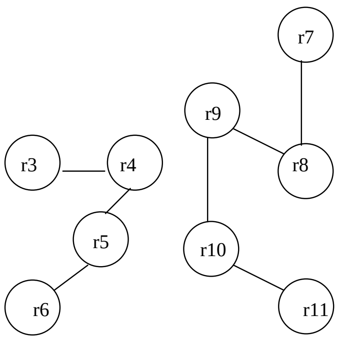

Figure 5: Modifies RCG

The machine model we chose included 8 general-purpose functional units, each connected to its own register bank. Each bank was assumed to have a sufficient number of registers. Integer operations required 2 cycles, and floating point operations required 4 cycles. Copy instructions for both integer and floating point values required 2 cycles. The code generated for this partitioned 8-wide LIW was compared with code generated for a similar, ideal, 8-wide LIW. The ideal LIW had the same characteristics as our target with the exception of a single register bank.

We evaluated the effect of value cloning by examining the initiation interval (II) obtained for the Fortran benchmarks with and without cloning. As shown in Table 1, on average, the partitioning algorithm without cloning observed a degradation of about 42% over the ideal II. When cloning is introduced, the degradation is reduced to around 31%. Similarly the greedy algorithm introduced on average 28 new copy instructions, and cloning reduced them to 23 (see Table 2). When only induction variables are cloned, the degradation in performance seen with the greedy algorithm is reduced to 39% and requires an average of 26 copy instructions. The cloning of read only variables reduced the degradation to about 35% and only 25 copies.

Benchmark Suite	Ideal II	Greedy II	Greedy-Clone II	Induction II	Read Only II
SPEC	100	141.08	133.86	136.92	140.31
FMM	100	113.54	113.54	114.92	113.54
Kernels	100	142.84	127.23	131.22	123.68
Other	100	154.39	136.65	165.04	143.12
All	100	141.73	131.11	139.41	134.83

Table 1: Normalized II

Benchmark Suite	Ideal Copies	Greedy Copies	Greedy-Clone Copies	Induction Copies	Read Only Copies
SPEC	0.00	46.92	37.77	41.54	43.23
FMM	0.00	14.83	13.83	14.17	14.33
Kernels	0.00	18.73	14.20	17.60	15.40
Other	0.00	27.53	22.80	25.87	24.33
All	0.00	28.43	23.04	26.06	25.39

Table 2: Copy Nodes in the DDG

As expected, value cloning read only and induction variables did reduce the number of copy instructions in almost all test cases. In general, it also reduced each loop's II. Value cloning may prove to be even more beneficial to resource constrained loops since they tend to be denser than recurrence constrained loops and copy instructions are difficult to insert in tight schedules.

While cloning often reduced II, there were some exceptions. The removal of values to be cloned from the RCG caused a new partitioning that resulted in worse code. Specificly, some read only values were cloned even though they were did not cause any copy instructions to be added. Additionally, some induction variables were cloned even when they were used in only one or two functional units.

In most cases, however, the cloning of read only variables or induction variables alone decreased the II achieved by the simple greedy partitioning. When cloned together, some loops reaped the improvements of both. Other loops were unable to achieve the full improvement of both. In a few cases, cloning both types of variables actually caused worse performance than cloning either one by itself. In these cases, the removal of read only and induction variables from the RCG and interference graph caused several other variables to be assigned to different register banks. Since the destination register of an operation determines its functional unit allocation, the new register assignment redistributed computation and unfortunately reduced parallelism.

4 Conclusions

In this paper, we have shown how cloning of both read-only values and loop induction variables can improve code generation for LIW architectures with partitioned register files. Our experiments showed a 9% performance improvement when cloning was applied.

In the future, we plan to restrict our aggressive cloning to cases where the schedule for the ideal LIW architecture indicates that it is likely to be needed. In addition, we plan to look at cloning array values and at the effects of cloning when loop unrolling is applied.

With increased demands for more speed, future processors will likely increase the levels of ILP far beyond that of modern architectures. With high levels of ILP, partitioned register banks become attractive and value cloning will be an important optimization in compilers for these architectures.

References

[1] Capitanio, A., Dutt, N., and Nicolau, A. Partitioned Register Files for VLIW's: A Preliminary Analysis of Tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)* (Portland, OR, December 1-4 1992), pp. 292–300.

[2] Ellis, J. *A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1984.

[3] Fisher, J., Faraboschi, P., and Desoli, G. Custom-fit processors: Letting applications define architecures. In *Twenty-Ninth Annual Symposium on Micorarchitecture (MICRO-29)* (Dec. 1996), pp. 324–335.

[4] Forsythe, G. E., Malcolm, M. A., and Moler, C. B. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[5] Janssen, J., and Corporaal, H. Partitioned register files for TTAs. In *Twenty-Eighth Annual Symposium on Micorarchitecture (MICRO-28)* (Dec. 1995), pp. 301–312.

[6] Sweany, P. H., and Beaty, S. J. Overview of the Rocket retargetable C compiler. Tech. Rep. CS-94-01, Department of Computer Science, Michigan Technological University, Houghton, January 1994.

[7] Texas Instruments. *Details on Signal Processing*, issue 47 ed., March 1997.