Copy Elimination for Parallelizing Compilers *

David J. Kolson, Alexandru Nicolau, and Nikil Dutt

Dept. of Information and Computer Science, University of California, Irvine
Irvine, CA 92697, USA

Abstract. Techniques for aggressive optimization and parallelization of applications can have the side-effect of introducing copy instructions, register-to-register move instructions, into the generated code. This preserves program correctness while avoiding the need for global search-and-update of registers. However, copy instructions only transfer data between registers while requiring the use of system resources (ALUs) and are essentially overhead operations which can potentially limit performance. Conventional copy propagation and copy removal techniques are not powerful enough to remove these copies as, during loop parallelization, the lifetimes of the values copied may span over loop boundaries. In this paper, we present a technique for copy removal that incrementally unrolls a loop body and re-allocates registers to values so that no copy operations are required. We also present a heuristic version that limits the amount of unrolling and present experimentation that demonstrates the necessity of copy removal in gaining improved code performance.

1 Introduction

Optimizing compilers can generate many copy instructions, register-to-register move instructions, both due to the aggressive application of program transformations as well as the compiler's internal representation. "Classic" optimizations such as common sub-expression elimination [1] and induction variable elimination [1] as well as more sophisticated techniques such as redundant load elimination [4,20], variable or register renaming [11,22] and variable lifetime splitting [10], add copy instructions into the code in order to control compiler complexity as the global search-and-replace of registers within instructions each time that an optimization is performed is too costly.

Also, compilers which utilize a static single assignment (SSA) [12] internal form, for instance, must honor the requirement that each variable be assigned exactly once. This has the effect of breaking a variable's lifetime into several (shorter) lifetimes. A consequence of this is that, at join points in program flow, intermittent values may need to be transferred from one temporary to another (via the ϕ-function), thus potentially generating copy instructions in the final code.

* This work supported in part by ONR grant N000149311348 and ARPA grant MDA904-96-C-1472.

S. Chatterjee (Ed.): LCPC'98, LNCS 1656, pp. 275–289, 1999.

In the generation of sequential code, copy propagation is typically applied as a post-pass process to reduce/eliminate the number of copy instructions present and, in some cases, may be implemented within a graph coloring register allocator as the coloring of the source and target temporaries with the same color allows the copy instruction to be removed. Compilers which seek to expose and to exploit instruction-level parallelism (ILP) typically employ some of the same optimizations. Copy removal is then crucial to an ILP compiler as copy instructions are essentially overhead instructions which require system resources (ALUs) to execute[1]. Thus, the presence of copy instructions in the schedule represents a negative impact to the attainable performance of parallel code. However, the solutions available to sequential optimizing compilers: standard copy propagation and node coalescing during graph coloring, are unavailable to an ILP compiler.

During scheduling [14,15,17,21] and software pipelining [2,13,18,24], when iterations of a loop are overlapped, copy instructions potentially keep values live over loop boundaries and serve to 'queue' values for future use. Thus, these copy instructions are not amenable to removal via conventional copy propagation as, in parallel code, multiple values generated by the same instruction (but from different loop iterations) may be simultaneously live due to copy instructions and simple copy propagation would lead to incorrect results.

Also, in the context of ILP compilers, where an integrated approach to instruction scheduling and register allocation [23,5,3] is necessary as: 1) an instruction scheduler requires accurate information on the resources required by each instruction, and 2) resource re-allocation is necessary to reduce resource contention, continually applying a graph coloring algorithm to the code is too costly. Thus, approaches which rely on graph coloring to remove copy instructions by coalescing the source and target temporaries via coloring the respective nodes the same color are inappropriate.

In this paper, a generalized technique for copy removal is presented. This technique removes copy instructions that keep values live over loop iterations by unrolling the loop code and re-allocating registers to instructions. As a result, copies which preserve values within an iteration—"traditional" copies—are also removed. Thus, this technique subsumes conventional copy propagation techniques while providing a method for removing more advanced forms of copy instruction chains. In the context of ILP compilers, this is particularly useful as the realization of available ILP can be greatly reduced by copy instruction occupation of system resources (ALUs).

2 Introductory Example

As an introductory example, consider the code in Figure 1(a) which will serve to demonstrate how parallelization inserts copy instructions into code. In this example, the value written into $R0$ in node M is used by the instruction $R5 = R0 + 10$

[1] Practically, a copy $R1 = R2$ would be performed by executing some instruction as $R1 = R2 + 0$ or $R1 = R2 << 0$.

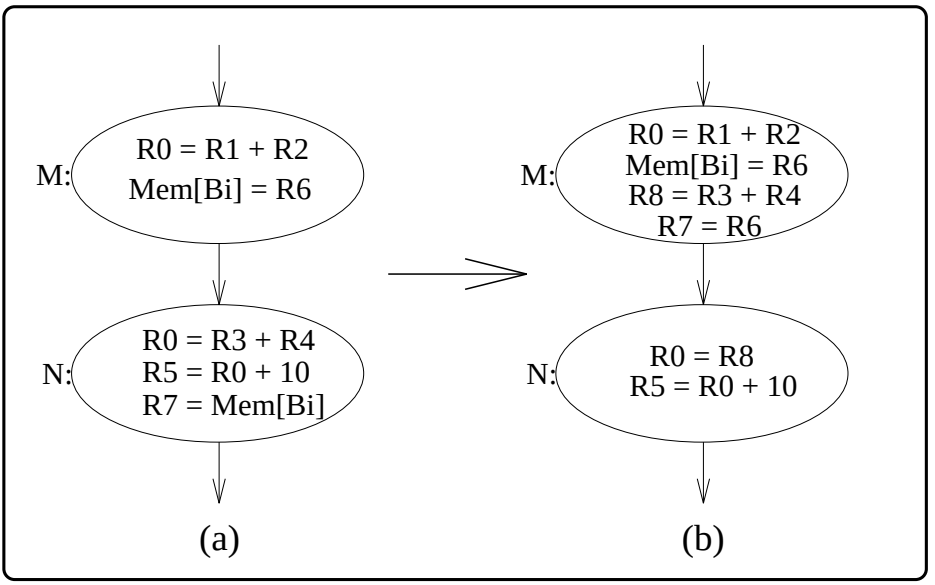

Fig. 1. Introducing copy instructions into the code.

in the next node[2]. During parallelization, the instruction $R0 = R3 + R4$ from node N may not be moved into (i.e., executed in parallel with the instructions of) node M as it redefines $R0$ and would cause incorrect values to be computed by the instruction $R5 = R0 + 10$ in node N. If a free register exists ($R8$ in this example), then the destination register of instruction $R0 = R3 + R4$ is renamed and the new instruction, $R8 = R3 + R4$, is moved into node M. To maintain correctness, a copy instruction $R0 = R8$ is necessary in node N to correctly move the value from $R8$ into $R0$, thereby compacting the code of (a) to that of (b).

Also, in node M, a value is stored[3] to the memory location B_i. In node N, this value is loaded from memory by the load instruction $R7 = Mem[B_i]$. Rather than re-loading the value from memory, the value stored to memory by instruction $Mem[Bi] = R6$ can be directly copied. Thus, the load is removed, resulting in the earlier availability of the value (i.e., the latency of the load is removed), and the copy instruction $R7 = R6$ is added to node M.

In the previous example, simple copy propagation may be used to eliminate the copies as the code is straight-line. However, during software pipelining when multiple iterations of a loop are overlapped and optimizations are performed, the

[2] Note that the value read by the instruction $R5 = R0 + 10$ is that produced by the instruction $R0 = R1 + R2$ in node M rather than the instruction $R0 = R3 + R4$ in node N as, due to the machine model, all operands are read before any results are written.

[3] For simplicity, loads and stores are shown here symbolically. Typically, the address is calculated into a register and analysis [4,20] is required to determine equivalency in memory references.

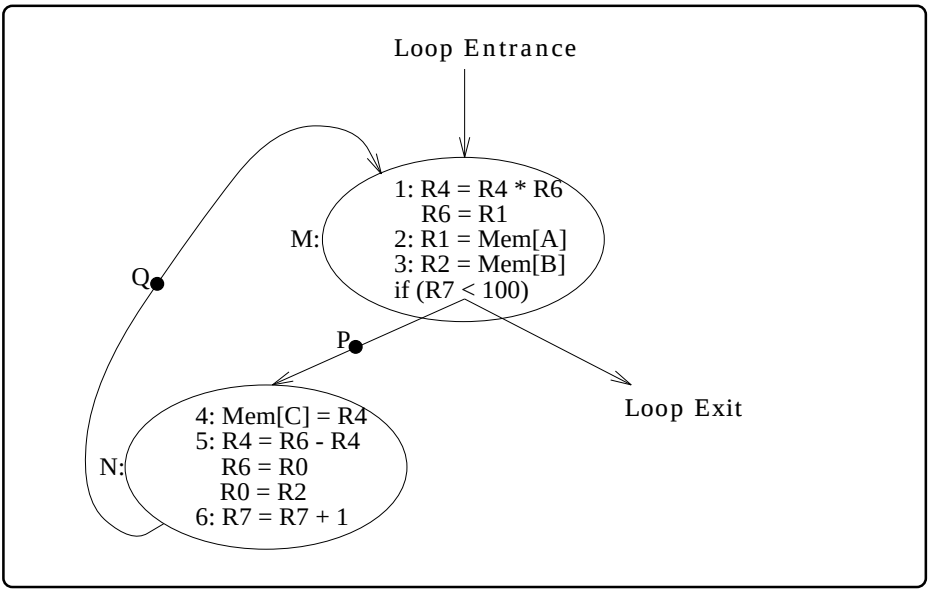

Fig. 2. Loop code with copy instructions.

uses of copied values can span across iteration boundaries. Thus, conventional copy propagation techniques [1] are not powerful enough to remove many of the copies introduced during loop parallelization.

As an example, in Figure 2, several optimizations were applied during the parallelization of a loop. As a result, several copy instructions are found in this code[4]. Conventional copy propagation cannot remove any of these copy instructions as those copies serve to keep values live over multiple iterations of the loop. For instance, the copy $R6 = R1$ in node M cannot be removed as it preserves a value loaded by $R1 = Mem[A]$ from the previous iteration (the previous execution of node M) and propagating $R1$ in place of $R6$ into the following node (into the instruction $R4 = R6 - R4$) would result in the use of an incorrect value (the newly loaded value from instruction #2 would be incorrectly used rather than the previously loaded value). Thus, conventional copy propagation is not adequate to remove these copies.

3 Related Work

Redundant memory instruction elimination [4,7,20] is a technique which minimizes the number of memory referencing instructions associated with array

[4] Note that four ALUs and two memory units are necessary to execute this parallel code, while, if copies were removed, only two ALUs and two memory units are necessary.

accessing. This technique will insert copy instructions into the schedule to eliminate a memory reference to a value which is already present in the register set.

Static renaming [11,22] is a method utilized during scheduling to allocate "on-the-fly" a currently unused memory location (register) when code optimization is prohibited due to false (anti- and output-) dependencies. A copy instruction is inserted to copy the generated value from the newly allocated register into the original register.

In the context of register allocation by graph coloring, several researchers [6,8,10,16] have addressed the problem of copy or move coalescing. Chaitin [8,9] proposes to combine the source and target nodes thereby producing a single node with the union of the interferences and removing the need for the move instruction. Since the degree of the new node is now higher, this can complicate the coloring process of the graph. Briggs *et al.* [6] have proposed a less aggressive, heuristic coalescing scheme which improves the colorability of the graph, but leaves copy instructions in the code. George and Appel [16] have extended Briggs' heuristic approach to improve coalescing in the SML/NJ compiler.

Chow and Hennessy [10] improve the quality of the spill code produced by a graph coloring allocator by splitting variable lifetimes at points in the code where register pressure is high. This allows a higher degree of freedom when coloring the graph, but requires move instructions when a variable's lifetime is not contained within the same register.

Another approach to register allocation for straight-line code is interval graph coloring. In [19], the interval graph coloring solution is extended to register allocation for loop graphs. In doing so, variable lifetimes are arbitrarily broken at loop boundaries and when the lifetime segments cannot be colored with the same color, copy instructions are necessary to transfer the value from one register to another.

4 Eliminating Copy Instructions

Copies generated during parallelization do not *produce* new values but, rather, *preserve* already computed values for future uses. In other words, multiple values produced in various iterations by an instruction are simultaneously live and transferred from definition to last use by chains of copy instructions. Copies related to a specific copy chain cannot be removed without affecting the 'queueing' of values for use. As depicted in the code of Figure 3, which has been compacted into one node, by unrolling the loop body sufficiently, the definition of a value and its last use become explicit thereby eliminating the need for the copy chain and, thus, enabling copy elimination[5]. In this example, the solution loop spans three iterations of the original loop.

[5] Once again, recall that, due to the machine model, all operands are read before any results are written. Therefore, in the first node, for instance, the value "X" used by A = X + B is that generated by the previous execution of that node.

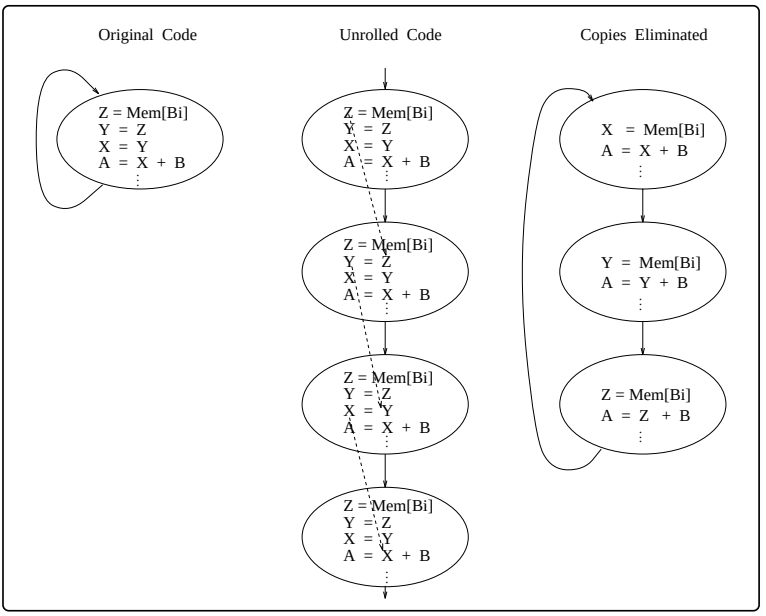

Fig. 3. Unrolling loop code to eliminate copies.

4.1 An Algorithm for Copy Elimination

Figure 4 contains an algorithm which performs copy elimination on a loop. As input, this algorithm takes a loop body or loop graph where each node contains instructions which are executed in parallel and produces a new loop without copy instructions which may span multiple iterations of the original (input) loop.

The first step of the algorithm is to compute the register mappings for each node in the graph. The register mapping for a node represents the contents of the registers *immediately preceding* the execution of that node and are derived similar to program data-flow analysis. (An algorithm for this is presented shortly.)

At this point, a loop template, or copy of the loop with its register mappings, is made and used for future reference. During copy elimination, the source registers of each instruction are looked up in this template to determine the tags for the values they use in the original loop. As values may be re-allocated to registers during copy elimination, it may become necessary to update or to change the source operands of some instructions. For instance, suppose the following is the register mapping for a node in the loop template:

$$(R0, 10.1)\ \ (R1, 9.0)\quad (R2, 7.0)\quad (R3, 10.0)$$

and the current instruction from that respective node in the current unrolling is $R2 = R0 - R3$ with a current register mapping of:

$$(R0, 9.0)\quad (R1, 10.0)\ \ (R2, 7.0)\quad (R3, 10.1)$$

In the original loop, this instruction uses values 10.1 for the first argument and 10.0 for the second. During copy elimination when considering the instruction

```
Procedure copy_elimination(L : loop)
Begin
    /* compute initial register maps for L */
    /* split nodes having multiple defs, recalc register maps */
    /* make loop template */
    /* add the header of L to the headers_list */
    /* add all backedges found in L to backedges_list */
    scan_and_reallocate (header_of(L));
    While (not empty backedges_list) {
        /* unroll along backedge b from backedges_list */
        /* add new iteration headers and backedges to respective lists */
        scan_and_reallocate (header);
        /* match backedges to header nodes */
    }
End copy_elimination
```

Fig. 4. An algorithm for copy elimination.

$R2 = R0 - R3$, these values (10.1 and 10.0) are looked up in the current register mapping to find the registers which contain them. If the registers currently containing those values are different from the source operands, the source operands are updated as appropriate, as is the case here where the first argument must be changed to $R2$ and the second argument becomes $R1$, thus, $R2 = R3 - R1$ is the instruction for the copy eliminated code.

Once the register mappings are calculated and the loop template is made, the header of the loop is added to a *headers_list* and all of the backedges of that loop are added to the *backedges_list*. The *headers_list* is used to keep track of all loop entry points to determine, after unrolling and copy elimination, if a backedge may be directed to any previous header that has an identical register mapping. The *backedges_list* is the list of loop iterative points along which an iteration of the loop is to be unrolled.

While there are backedges to unroll along, the algorithm iterates over the following steps: unroll the loop along that backedge; add the new header and backedges to the respective lists; *scan_and_reallocate*() (discussed further) and, finally, once copy elimination has been performed on the current unrolling, the backedges of this loop iteration are checked against all headers in the *headers_list*. Those backedges with register mappings that match the register mappings of an iteration header are directed to the respective matching header while those with no match are added to the *backedges_list*. The algorithm terminates once there are no more backedges left.

Computing Register Mappings

Figure 5 contains an algorithm patterned after dataflow analysis which computes the register mappings of a loop. To derive the mappings of a node, each instruction in the node is considered. If the instruction is a copy, then a new entry is made with the destination register and the value being copied.

```
Procedure compute_register_maps (L : loop)
    /* initialize all register maps to φ */
    changes = true
    while (changes) {
        changes = false
        add loop header to l
        while (not empty l) {
            remove node, N, from l
            Rmaps = reg_map_of(N)
            new_maps = copy(Rmaps)
            Foreach operation, op, in N {
                if (op is a copy)
                    (value.age) = lookup src1_reg_of(op) in Rmaps
                    Add (dest_reg_of(op), value.age) to new_maps
                else
                    Foreach map in new_maps with same id_of(op)
                        increment age
                    Delete all maps from new_maps with dest_reg_of(op)
                    Add (dest_reg_of(op), op_id_of(op).0) to new_maps
                endif
            }
            For all successors, S, of N {
                if (new_maps != reg_map_of(S))
                    changes = true
                    reg_map_of(S) = copy(new_maps)
                endif
                Add S to l
            }
        }
    }
End compute_register_maps
```

Fig. 5. An algorithm for computing register mappings.

If the instruction is not a copy, it defines a new value. An entry is added to the register mapping annotated with the destination register of the instruction and a tag of instruction identifier and 0 (zero signifies the birth of a value). Any annotation in the mapping with the same instruction identifier will have its *age* field incremented, as this value has become "older" by the generation of the new value. Also, any entry with the destination register is now killed and deleted from the mapping.

As an example of deriving the register mappings of a loop, the register mappings for the example of Figure 2 are derived in Table 1.

Scan-and-Reallocate

Figure 6 contains an algorithm for scanning a loop. This algorithm is similar to the algorithm for computing register mappings as it is necessary when reallocating registers to instructions to keep track of the values in the registers.

	Initially	Iteration of Algorithm 1^{st}	2^{nd}	3^{rd}	no changes
Maps At Point P	ϕ		($R0$, 3.1)	($R0$, 3.1)	($R0$, 3.1)
		($R1$, 2.0)	($R1$, 2.0)	($R1$, 2.0)	($R1$, 2.0)
		($R2$, 3.0)	($R2$, 3.0)	($R2$, 3.0)	($R2$, 3.0)
		($R4$, 1.0)	($R4$, 1.0)	($R4$, 1.0)	($R4$, 1.0)
				($R6$, 2.1)	($R6$, 2.1)
Maps At Point Q	ϕ	($R0$, 3.0)	($R0$, 3.0)	($R0$, 3.0)	($R0$, 3.0)
			($R1$, 2.0)	($R1$, 2.0)	($R1$, 2.0)
		($R4$, 5.0)	($R4$, 5.0)	($R4$, 5.0)	($R4$, 5.0)
			($R6$, 3.1)	($R6$, 3.1)	($R6$, 3.1)
		($R7$, 6.0)	($R7$, 6.0)	($R7$, 6.0)	($R7$, 6.0)

Table 1. Register mappings for the code of Figure ??.

```
Procedure Scan_and_Reallocate(L : loop)
    /* initialize all register maps to φ */
    reg_map_of(header) = /* output map of backedge */
    add loop header to l
    while (not empty l) {
        remove node, N, from l
        Rmaps = reg_map_of(N)
        new_maps = copy(Rmaps)
        Foreach operation, op, in N
            if (op is a copy)
                Remove op
            else
                Update_Args(op)
                if (dest_reg_of(op) ∈ Rmap and live)
                    dest_reg_of(op) = get free register
                Delete all maps from new_maps with dest_reg_of(op)
                Add (dest_reg_of(op), (op_id_of(op), 0)) to new_maps
            endif
        For all successors, S, of N {
            reg_map_of(S) = copy(new_maps)
            Add S to l
        }
    }
End scan_and_reallocate
```

Fig. 6. An algorithm for removing copies and updating register usages.

Initially, the register maps are initialized to ϕ and the entry register mapping to the loop is the register mapping found at the end of the previous iteration, or that found along the backedge unrolled upon. As the loop nodes are scanned, if an instruction is a copy, it is removed from the node. If not, the arguments to the instruction are updated as registers are re-allocated and the appropriate values may not still be in the used registers. Updating entails look-up of the sources in the loop template to determine the referenced values; those values are then found in the current mapping to obtain the register that currently contains the appropriate value(s). It might be necessary to re-allocate the destination of this instruction if that register contains a live value. Finally, values killed by this instruction are removed and an annotation is added for the processed instruction.

4.2 Heuristic Copy Elimination

Possibly the most noticeable feature of our copy elimination algorithm is that the final loop solution spans multiple iterations of the original loop in order to make value definitions and uses explicit. In some cases, it may not be desirable to unroll the loop for the necessary number of iterations. In this case, the algorithm may be parameterized with the maximal number of iterations to unroll. However, when this threshold value is reached, it is not guaranteed that the backedges for that unrolling depth will match any of the previous iteration headers. When the threshold is reached, a simple strategy may be employed to match backedges to headers so that a minimal number of copy instructions is introduced.

4.3 An Example

As an example, copy elimination is performed on the loop code of Figure 2 with the initial register mappings from Table 1.

The copy elimination algorithm applies the *scan_and_realloc*() procedure to the first iteration of the loop. As node M_0 is scanned, the first instruction considered is $R4 = R4 - R6$. The procedure *update_args*() looks up this instruction in the loop template and determines that argument one is the value (5.0) and argument two is the value (3.1). These values, (5.0) and (3.1), are looked up in the current register mapping (the register mapping found at point I in Figure 7) and are currently found in the registers $R4$ and $R6$, respectively. Since these values are already referenced in the appropriate registers, no changes to the instruction's operands are made. Since this is the last use of the value (5.0), contained in register $R4$, a new register is not needed for the destination register of the instruction. Lastly, an entry is made in the current register mapping of $(R4, (1.0))$.

The next instruction is the copy instruction $R6 = R1$ and is removed from the code. The next instruction, $R1 = Mem[A]$, is examined and is found to contain a live value (i.e., a value that is used beyond this node). Thus, a call to *get_free_register*() is necessary to re-allocate a register to the destination of this instruction. In this case, the function call returns the register $R2$ as the value

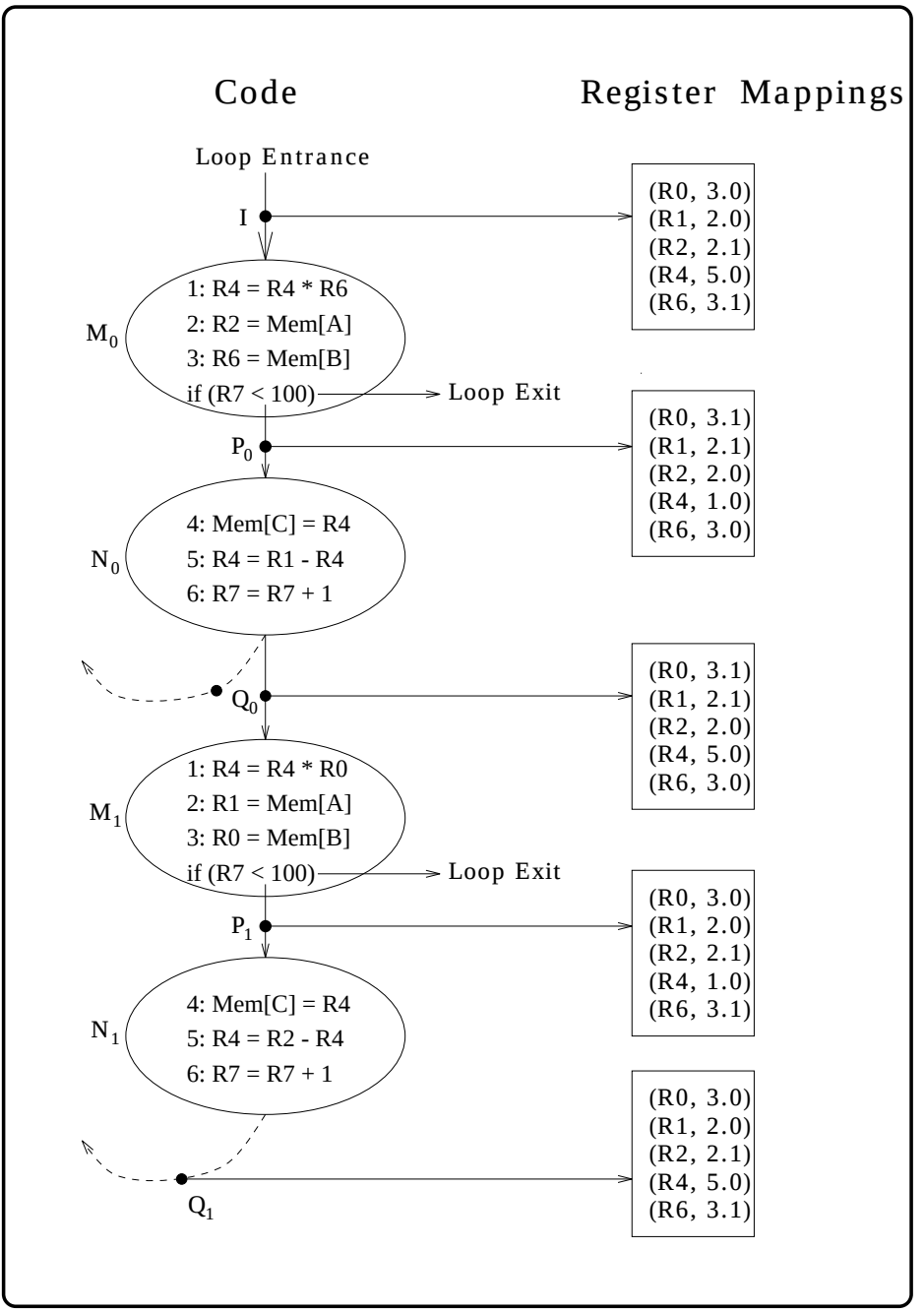

Fig. 7. Unrolling loop code to eliminate copies.

it contains, (2.1) becomes dead in this node[6]. Thus, $R1 = Mem[A]$ becomes $R2 = Mem[A]$.

When node N_0 is scanned, the argument to instruction $Mem[C] = R4$ is checked in the loop template. This instruction uses $R4$ which contains the value (1.0) at point P. In the current register mapping, $R4$ contains this value, so no updating takes place. The next instruction is $R4 = R6 - R4$ and uses the values (2.1) and (1.0), as source values, respectively. In the current mapping, those values are contained in $R1$ and $R4$, respectively. As the value (1.0) dies, the destination register of this instruction does not require updating. The next two instructions, $R6 = R0$ and $R0 = R2$, are copies and are removed. Finally, the last instruction requires no updating, leaving the code found in node N_0 of Figure 7.

This process continues and the derived mapping at Q_1 is found to match that at the loop top (point I), so the algorithm converges and the new loop spans two iterations of the original loop, containing no copies.

5 Experiments and Results

We conducted experiments on a suite of benchmarks which consisted of 11 numerical and scientific codes to study the effects of copy elimination on performance. Latencies given to our parameterized scheduler are one cycle for copy operations, two cycles for add/subtract operations, three cycles for multiply operations and three cycles for memory accessing operations. For functional unit constraints, two add/subtract units, one multiply and a single-ported memory which can handle only one request at a time, were used.

For each benchmark, three schedules were generated: the first schedule contained copy operations, the second schedule utilized our copy elimination algorithm and the third used a heuristic version with the maximum number of iterations, three. From these schedules performance measures were made. These performance improvement measures are with respect to sequential execution of the code and are measured as:

$$SU = \frac{cycles_{seq}}{cycles_{par}}$$

Thus, in our tables, columns labeled with "copies" refers to the speed-up of parallelized code which contains copy operations; "no copies" refers to the speed-up of parallelized code with copy elimination; and "heur" refers to the speed-up of parallelized code with the heuristic version of copy elimination. Percentage improvement (% Improvement) is the percentage improvement of the schedules

[6] When the instruction $R1 = Mem[A]$ generates a new value, it will cause all other values in the register mapping with the same identifier to become "older" (i.e., the age value increases). Thus, the value in $R1$, (2.0), will become (2.1) and the value in $R2$, (2.1), will become (2.2). As the value (2.2) is never used, the register containing that value is free for re-allocation in this node.

with copies eliminated versus the schedules that contain copies and is measured as:

$$Impr = \frac{SU_{nocopies} - SU_{copies}}{SU_{copies}} * 100$$

Our observed performance results on the benchmark suite are contained in Table 2 for codes with no copy elimination and codes where copies were removed. In all cases, copy elimination increased the performance of the parallelized code with percentage improvements ranging from 11% to 72%.

Table 2. Observed speed-up on benchmark suite.

Benchmark	copies	no copies	% Impr
2D-Hydro exerpt	1.27	1.99	57%
Cholesky Conj. Grad.	1.47	1.81	23%
Tri-diagonal Elim.	1.38	2.20	59%
GLR	1.61	1.79	11%
State Equations	1.11	1.91	72%
Partial Diff. Solver	1.48	1.99	34%
Integrator Pred.	1.05	1.56	49%
Difference Pred.	1.20	1.98	65%
Partial sum (scan)	1.80	2.25	25%
Difference sum	1.50	1.93	29%
2D Particle	1.70	1.90	12%

Table 3 contains the observed performance results on the benchmark suite for code with copy elimination and codes with heuristic copy elimination with an unrolling bound of three iterations. Also noted in the table is the number of iterations spanned by the optimal (i.e., no bounds on unrolling) copy elimination codes. In some cases, the heuristic version was able to derive the same solution as the optimal and in other cases derived solutions with results which are close to the optimal. It should be noted that, even though the optimal solutions span more iterations, the number of iterations spanned is not prohibitive.

6 Conclusion

Aggressive code motion and program optimization techniques, necessary for exploiting the parallelism inherent in application code, can have the side-effect of introducing many copy instructions—register-to-register move instructions, into the parallel code. These copies are necessary overhead for reducing compiler complexity, but their presence in the final code represents a hindrance to high performance as they consume functional resources, but perform no significant computation. In order to improve the attainable performance, copy elimination is necessary. This paper presents a technique which eliminates all copy instructions from parallel code by unrolling and remapping registers to values. As the

Table 3. Speed-up of copy elimination and heuristic.

Benchmark	no copies	heur	# Iters
2D-Hydro exerpt	1.99	1.89	4
Cholesky Conj. Grad.	1.81	1.81	3
Tri-diagonal Elim.	2.20	1.92	5
GLR	1.79	1.70	4
State Equations	1.91	1.91	3
Partial Diff. Solver	1.99	1.88	5
Integrator Pred.	1.56	1.56	3
Difference Pred.	1.98	1.98	3
Partial sum (scan)	2.25	1.99	4
Difference sum	1.93	1.80	4
2D Particle	1.90	1.83	4

increase in code size may be a consideration, a heuristic version is presented which bounds the amount of loop unrolling performed. Experimentation with a suite of benchmarks demonstrates that significant performance improvements are possible by eliminating copy instructions.

References

1. A. H. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley,Reading,Massachusetts, 1986. 275, 275, 278
2. A. Aiken and A. Nicolau. Perfect Pipelining: A New Loop Parallelization Technique. *Proc. of the 1988 European Symp. on Programming*, March 1988. 276
3. D. A. Berson, P.Chang, R. Gupta, and M. L. Soffa. Integrating Program Optimizations and Transformations with the Scheduling of Instruction Level Parallelism. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 207–221, August 1996. San Jose, California. 276
4. R. Bodik and R. Gupta. Array Data Flow Analysis for Load-Store Optimizations in Superscalar Architectures. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, August 1995. 275, 277, 278
5. D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. *Proceedings of SIGPLAN Architectural Support for Programming Languages and Operating Systems*, 26(4), April 1991. 276
6. P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994. 279, 279
7. D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Proceedings of the International Conference on Parallel Processing*, 1987. 278
8. G. Chaitin. Register Allocation and Spilling Via Graph Coloring. *Proc. of SIGPLAN Symp. on Comp. Const.*, 176, June 1982. 279, 279
9. G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation Via Coloring. *Computer Languages*, 6:47–57, January 1981. 279

10. F. Chow and J. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990. 275, 279, 279
11. R. Cytron and J. Ferrante. What's in a name? The value of renaming for parallelism detection and storage allocation. *Proceedings of the Internation al Conference on Parallel Processing*, August 1987. 275, 279
12. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. 275
13. K. Ebcioglu. Compilation Technique for Software Pipelining of Loops with Conditional Jumps. *20th Annual Workshop on Microprogramming*, 1987. 276
14. J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1985. 276
15. J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Comp.*, C-30(7), July 1981. 276
16. L. George and A. W. Appel. *Iterated Register Coalescing*. ACM Transactions on Programming Languages and Systems, 18(3):300–324, May 1996. 279, 279
17. J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. *International Conference on Supercomputing*, 1988. 276
18. T. Gross and M. S. Lam. Compilation for High-Performance Systolic Array. *Proceedings of the Symposium on Compiler Construction*, 1986. 276
19. L. J. Hendren, G. R. Gao, E. Altman, and C. Mukerji. A Register Allocation Framework Based on Heirarchical Cyclic Interval Graphs. *International Conference on Compiler Construction*, pages 176–191, April 1992. Paderborn, Germany. 279
20. D. J. Kolson, A. Nicolau, and N. Dutt. Elimination of Redundant Memory Traffic in High-Level Synthesis. *IEEE Transactions on the Computer Aided Design of Integrated Circuits and Systems*, pages 1354–1364, November 1996. 275, 277, 278
21. A. Nicolau. Uniform Parallelism Exploitation in Ordinary Programs. *Proceedings of ICPP*, August 1985. 276
22. A. Nicolau, R. Potasman, and H. Wang. Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism. *4th Int. Wksp on Lang. and Comp. for Par. Comp.*, 1991. 275, 279
23. S. S. Pinter. Register Allocation with Instruction Scheduling : A New Approach. *SIGPLAN PLDI*, 1993. 276
24. B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and An Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. *Micro-14*, June 1981. 276