Removing Anti Dependences by Repairing

M. Anton Ertl Andreas Krall
{anton,andi}@mips.complang.tuwien.ac.at

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
anton,andi@mips.complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/home.html
Tel.: (+43-1) 58801 4474
Fax.: (+43-1) 505 78 38

Abstract. Anti dependences (write-after-read dependences) constrain the reordering of instructions and limit the effectiveness of instruction scheduling and software pipelining techniques for superscalar and VLIW processors. Repairing solves this problem: If the definition of a variable is moved before a previous use of that variable, compiler-generated repair code reconstructs the value that the definition destroyed. Repairing features several potential advantages over register renaming, another technique for removing anti dependences: less register pressure, less loop unrolling and fewer move instructions.

Key Words: anti dependence, repairing, register renaming, instruction-level parallelism, speculative execution

1 Introduction

Computer designers and computer architects have been striving to improve uniprocessor performance since the invention of computers. The next step in this quest for higher performance is the exploitation of significant amounts of instruction-level parallelism. Therefore, superscalar and VLIW (very large instruction word) machines have been designed, which can execute several instructions in parallel. In order to use these resources the instructions are reordered by the hardware [Tho64, Tom67, PHS85, Soh90] or by compiler techniques like basic block instruction scheduling [LDSM80, HG83, GM86, EK92], trace scheduling [Fis81, Ell85] and software pipelining [RG81, Lam88, Rau94]. To ensure correctness, the order between dependent instructions must be maintained, which restricts reordering and parallelism.

Dependences exist between writes and reads (data flow dependences), reads and writes (anti dependences) and between writes (output dependences) to the same register or memory location. In this paper, we will discuss only dependences through registers. We will also concentrate on anti dependences. Although the techniques discussed here can be used to eliminate output dependences, (partial) dead code elimination [KRS94, BC94] is more appropriate for this purpose.

Another problem for exploiting significant amounts of instruction-level parallelism is the limited amount of registers (e.g., $\leq$ 32 integer registers on all

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: Compiler Construction (CC'96), Springer LNCS 1060, pages 33-43

popular architectures). By contrast, functional units tend to become abundant; compilers will have a hard time utilizing all of them all the time.

We discuss anti dependences and existing techniques for dealing with them in Section 2. In Section 3 we introduce a new technique, that often reduces register pressure, but usually pays for this with more instructions: repairing. In Section 4 we demonstrate the advantages of repairing with a small example. Finally, we show the potential of repairing with empirical data derived from instruction traces of real-life applications (Section 5).

2 Anti Dependences

Anti dependences (and output dependences) are, in some sense, false dependences. They are not caused by the data flow between instructions, but by reusing registers. Several methods for dealing with anti dependences have been proposed:

2.1 Register Renaming

Anti dependences can be removed (or at least moved) by register renaming [PW86]. This technique can be implemented in hardware [Tom67, PHS85, Soh90] and as compiler optimization [PW86, Lam88]. Note that only compiler-based renaming techniques can increase the reordering freedom for the compiler.

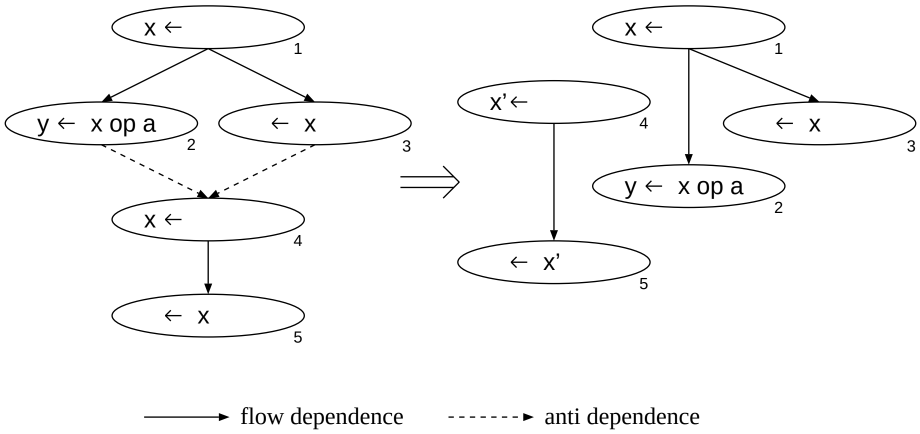

Fig. 1. Register renaming

Figure 1 shows how register renaming works: Originally, register x is used in two live ranges, resulting in two anti dependences, one from each use (read) of the first live range to the definition (write) of the second live range. Register renaming transforms the second live range such that it uses the register x'.

Renaming has two restrictions:

- Renaming the whole second live range may be impossible, because one of the uses requires a specific register (e.g., to satisfy the calling convention).
- Renaming does not work, if the two dynamic live ranges involved are statically the same live range (e.g., a live range in a loop).

Both problems can be solved by moving x' to x as soon as the first live range no longer needs x (at the cost of an additional move instruction). The second problem can also be solved by separating the live ranges statically by code replication (e.g., loop unrolling).[1]

2.2 Rematerialization

Instead of renaming one of the live ranges such that the definition of the second does not destroy the value used in the first live range, the compiler can reconstruct the value of the first live range just before the value is used. Rematerialization reconstructs the value by simply recomputing it. Rematerialization of constants has been proposed [CAC$^+$81] and successfully used [BCT92] as an alternative to spilling in register allocation.

Figure 2 shows, how the scheduler can rematerialize a constant (in instruction $\bar{1}$). In this example, rematerialization moved instruction 3 down across 4 and 5, which originally (anti-) depend on 3. The resulting code still contains antidependences, but they are different and may hinder scheduling less (if this arrangement were not profitable, the compiler would use rematerialization differently or not at all).

Rematerialization reduces the lifetime of the result of a computation, but it may increase the lifetime of the source operands. This may cause higher register pressure and more loop unrolling. A simple way to avoid this problem is to rematerialize only constants, because they have no input operands. This approach is used by [BCT92].

3 Repairing

Like rematerialization, repairing reconstructs the value that was in the register before it was overwritten by the definition of the second live range. In contrast to rematerialization, repairing reconstructs the value from downstream values using the inverse operation.

In Figure 3, the value of the first live range is used to compute y in instruction 2. Later, y is used to reconstruct that value in register x' using the inverse operation $\overline{op}$ (instruction $\overline{2}$).

To apply this transformation, the following conditions must be satisfied:

[1] The combination of register renaming and loop unrolling is known as modulo variable expansion [Lam88].

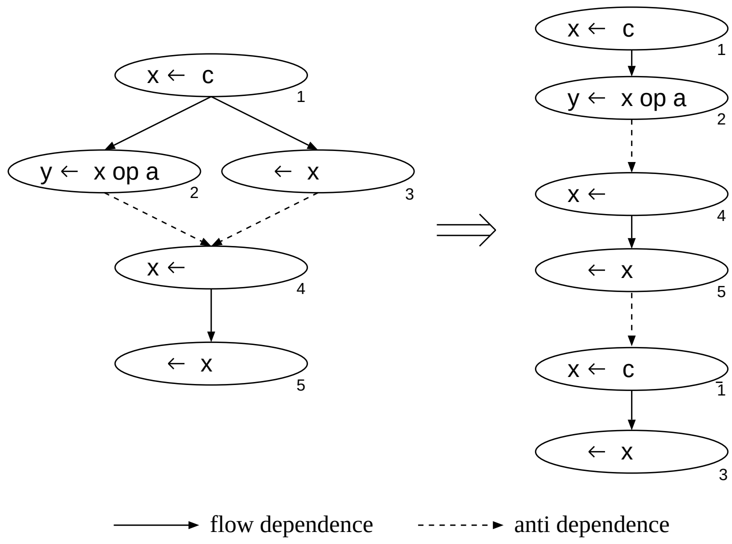

Fig. 2. Rematerialization of a constant

– Another value has been computed from the value destroyed by the second definition.
– This computation is invertible. This includes arithmetic and logic operations like add (with modulo arithmetic), subtract, rotate, exclusive or, negation and bitwise complement, but not multiply or floating point operations, which can lose information.

At first sight, repair code seems to make the program worse, especially when compared to register renaming, which (apparently) costs nothing but a few loop unrollings. But in many cases the repair code can use an otherwise unused execution unit, can be combined [NE89] with other operations or optimized in some other way.

Repairing also introduces new data flow dependences ($2 \rightarrow \overline{2} \rightarrow 3$ in Figure 3). These dependences pose no problem to the scheduler. It can chose between repairing and other methods depending on the way in which it wants to arrange the instructions 2, 3 and 4. The data dependences introduced by repairing just mean that repairing cannot be used for certain arrangements. Fortunately, for those arrangements where repairing offers the greatest benefits (i.e., the scheduler wants to move instruction 3 far down), it can be applied.

The potential advantages of repairing over register renaming are:

less register pressure Repairing often uses one register less between the time when the value is destroyed and the time when it is repaired.

less loop unrolling If the lifetime of a variable is l cycles, and a loop iteration is initiated every s cycles, then at least $\lceil l/s \rceil$ values for the variable must

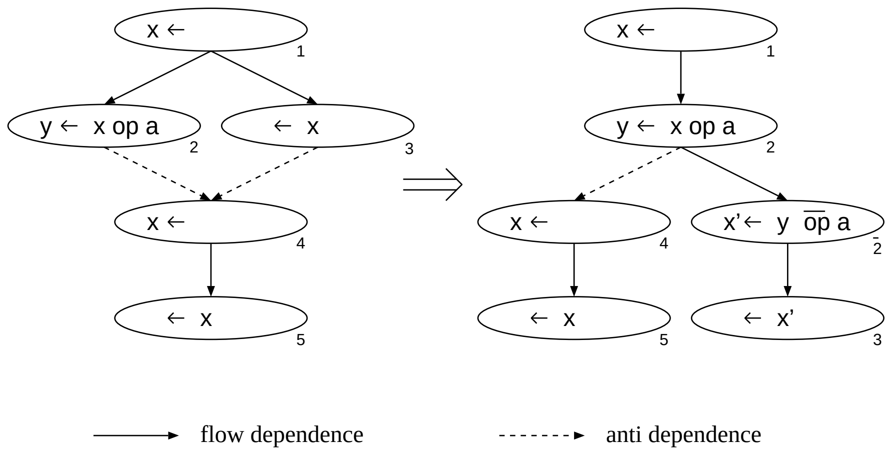

Fig. 3. Repairing using the inverse operation ($\overline{op}$)

be kept alive concurrently. The loop must be unrolled that many times in order to to address the values in different registers. Repairing shortens the lifetime of registers, which in turn lowers $\lceil l/s \rceil$ and the unrolling factor.

fewer move instructions Unless the compiler performs an unhealthy amount of code replication, register renaming introduces move instructions at control flow joins. These moves can often be avoided with repairing.

However, repairing also has a potential for making a program worse. Apart from adding an operation, it can also lengthen the lifetime of the values that are needed for the reconstruction. The result of the operation to be inverted (y in Figure 3) is used elsewhere anyway, and keeping it alive for repairing is certainly better (with respect to register pressure) than keeping the original value alive; but lengthening the lifetime of the other operand (a in Figure 3) can cause higher register pressure than register renaming. Of course, if the operation needs only one operand in a register (i.e., the operation is unary or it has an inline (small constant) operand), repairing is guaranteed to be profitable with respect to register pressure.

In some cases, the repairing operation and the operation using this repaired value can be combined [NE89], providing the benefits of repairing without any cost. For example: additions or subtractions with immediate operands can be combined with additions, subtractions and comparison instructions with an immediate operand or with memory instructions; negations can be combined with additions or subtractions. Figure 4 shows an example, where the scheduler moves an `sw` instruction down.

In comparison with rematerialization, repairing results in less register pressure in the worst case: Both extend the live ranges of the values necessary for the reconstruction down to the instruction performing the reconstruction. But

`addu $5, $4, 8`
`sw $3, 4($4)`
`addu $4, ...`
	$\Rightarrow$
	repairing
		`addu $5, $4, 8`
		`addu $4, ...`
		`...`
		`subu $6, $5, 8`
		`sw $3, 4($6)`
			$\Rightarrow$
			combining
				`addu $5, $4, 8`
				`addu $4, ...`
				`...`
				`sw $3, -4($5)`

Fig. 4. Repairing used with combining (MIPS assembly)

rematerialization can extend them down all the way from the instruction that computed the value to be rematerialized originally, whereas repairing can extend one (a in Figure 3) down from the invertible instruction (which uses the value to be repaired and is therefore later than the instruction that computed that value) and the other down from the last instruction that uses the value computed by the invertible instruction (y in Figure 3), which is even later. In particular, repairing is guaranteed to be profitable (with respect to register pressure), if the repairing instruction needs only one register operand, whereas rematerialization is not always profitable for the analogous case.

In the preceding discussion we always wrote about "extending live ranges". Of course, repairing and rematerialization can be applied to these live ranges, too, where appropriate; still, on average, a longer live range will cause higher cost, be it register pressure, reconstruction or move instructions, or loop unrolling.

The most important application of repairing will be compiler-based speculative execution. Global code reordering techniques like trace scheduling and software pipelining move instructions up before branches. This is only legal, if the destination register of the moved instruction is dead on the other path. However, by inserting repair code in the other path the compiler can lift this restriction (see Figure 5). Note that instruction 2 need not reside in front of the branch from the beginning—it may have moved up, too.

The actual algorithm for repairing depends on the scheduling framework. E.g, a trace scheduling [Ell85] compiler would first schedule a trace without restrictions from anti dependences, then (in the bookkeeping phase) it would determine the applicability and profitability of renaming, repairing and rematerialization for each anti dependence, and apply the least costly applicable transformation, and finally it would allocate the registers.

4 Example

We demonstrate the advantages of repairing with a small example. Figure 6 shows the C function strlen, which computes the length of a zero-terminated string. Figure 7 shows the assembly language output of a compiler for the MIPS R3000. We have changed the register names to make the program more readable.

Figure 8 shows a version of the loop that is software-pipelined using register

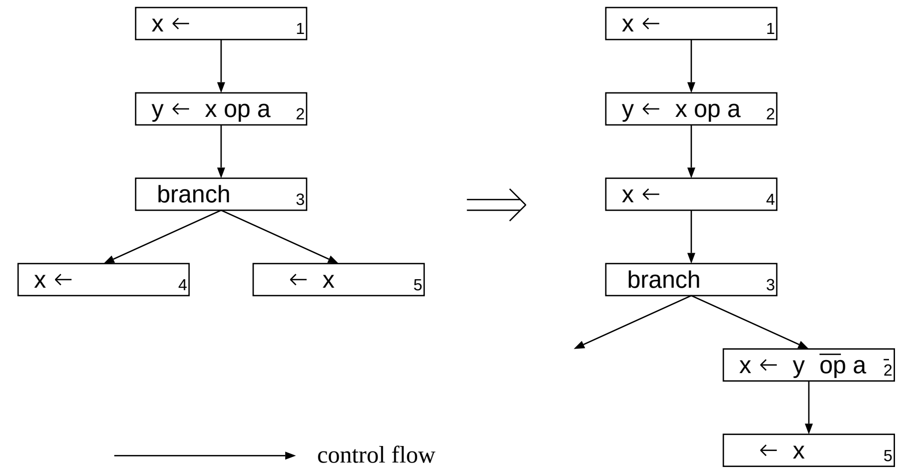

Fig. 5. Repairing applied to speculative execution (control flow graph)

```
int strlen(char *s) {
  char *t = s;
  while (*s != '\0')
    s++;
  return s-t;
}
```

Fig. 6. The C function strlen

```
 #   1    int strlen(char *s) {
strlen:
 #   2    char *t = s;
        move    t,s           # t=s
 #   3    while (*s != '\0')
        lb      t0,0(s)       # t0=*s
        beqz    t0,end        # while (t0 != '\0')
loop:
 #  4     s++;
        addu    s,s,1         # s++
        lb      t0,0(s)       # t0=*s
        bnez    t0,loop       # while (t0 != '\0')
end:
 #   5    return s-t;
        subu    v0,s,t        # return_value = s-t
        j       ra            # return
```

Fig. 7. MIPS R3000 assembly language source of strlen

```
                  move t,s
      lb_0 t0,0(s)    addu_1 s1,s,1
      lb_1 t1,0(s1)   addu_2 s2,s1,1
loop: lb_n  t2,0(s2)   addu_n+1 s3,s2,1 beqz_n-2 t0,end
      lb_n+1 t3,0(s3) addu_n+2 s,s3,1  beqz_n-1 t1,end1
      lb_n+2 t0,0(s)  addu_n+3 s1,s,1  beqz_n t2,end2
      lb_n+3 t1,0(s1) addu_n+4 s2,s1,1 bnez_n+1 t3,loop
                  move s,s3
end:              subu v0,s,t     j ra
end1:             move s,s1       b end
end2:             move s,s2       b end
```

Fig. 8. Software pipelined version of strlen with register renaming

renaming.[2] We assume a load latency of 2, a branch latency of 1, and a processor that has enough resources to execute one line of Figure 8 per cycle. The indices of the instructions indicate the iteration the instruction belongs to. This example nicely demonstrates the disadvantages of register renaming. The `addu`s are executed speculatively three iterations in advance and therefore their results live four cycles (they are used in the off-loop arms). Therefore the number of different registers necessary for `s` and the loop unrolling factor is $\lceil l/s \rceil = 4$. The result of the `lb` lives for only three cycles, but since the unrolling factor is four, we must give four registers to it, too[3]. At the exit of the loop `move` instructions have been generated to reunite the `s` values into one register.

```
                  move t,s
      lb_0 t0,0(s)   addu_1 s,s,1
      lb_1 t1,0(s)   addu_2 s,s,1
loop: lb_n  t2,0(s)   addu_n+1 s,s,1 beqz_n-2 t0,end
      lb_n+1 t0,0(s) addu_n+2 s,s,1 beqz_n-1 t1,end
      lb_n+2 t1,0(s) addu_n+3 s,s,1 bnez_n t2,loop
end:                 subu s,s,3
                     subu v0,s,t   j ra
```

Fig. 9. Software pipelined version of strlen with repair code

Figure 9 shows another version of the loop, this time software-pipelined with repairing and register renaming. `s` satisfies the conditions for repairing with the inverse operation and can safely be destroyed by incrementing it. Therefore `s` needs only one register. It does not pay off to destroy and repair the results of the `lb`s, so we have to use register renaming in this case. Since these results live for three cycles, the loop is unrolled three times. At the off-loop path, `s` has to

[2] For simplicity, we assume that the loads cannot have exceptions. Speculative execution of trapping instructions is discussed in, e.g., [EK94].

[3] We could have saved the one register by unrolling $\mathrm{lcm}(4,3) = 12$ times [Lam88].

be repaired to its proper value. `s` has been destroyed by incrementing thrice. Therefore the repair code consists of three decrements that have been combined into one decrement by three. In summary, repairing saves four registers (44%), one loop iteration (25%) and some other code as well.

5 Potential

This section shows how important repairing is for real-world programs. We produced traces (up to 100,000,000 instructions) of various applications and counted the antidependences in them and how many of them can be removed with various forms of repairing.

This trace-based method has some disadvantages: it does not see all antidependences that the compiler has to consider (in particular, it does not see antidependences to off-trace instructions), and it treats all antidependences equal, no matter how important or unimportant they are for the compiler. The advantage of this method is that it is independent from the compiler; if, in contrast, we implemented repairing in a compiler and presented empirical data based on experiments with this compiler, the results would strongly depend on the scheduler and on the register allocator of that compiler. Note that the results we present do not depend much on the compiler; although the compilers we used performed register allocation, this has little influence, because almost every use of a value causes an antidependence, independent of the register allocator, and the uses themselves are also quite independent of the register allocator (as long as moves and spilling are minor factors). Our empirical data supports this view: you cannot tell from the data which compiler produced the code.

The applications used are: abalone, a board game; agrep, an approximate pattern matcher; dvips, a filter used in typesetting; gcc-cc1, a part of the GNU C compiler; gzip, a compression program; and sicstus, a Prolog interpreter. All programs were compiled for the Alpha architecture under OSF/1, either with gcc-2.7.0 (abalone, gcc-cc1, sicstus) or with cc-3.1.1 (the other programs).

Figure 10 shows the results. The column *instructions* displays the trace length, *anti dep/inst.* the number of anti dependences per instruction, and the next three columns display what portion of these anti dependences can be eliminated with various forms of repairing: *repairing* comprises all forms of repairing, *one reg. operand* are those forms of repairing that are guaranteed to be profitable with respect to register pressure, and *combinable* are those cases where the repairing code can be combined with the instruction that uses the repaired value (and therefore repairing is for free, in addition to being profitable).

18%–34% of all antidependences can be removed with repairing. Only about half of them (8.6%–16.6%) are guaranteed to be profitable with respect to register pressure according to our simple one-register-operand criterion, so it is probably a good idea to invest a little more in profitability analysis. 4.1%–12.6% of the anti dependences can be repaired for free, providing the benefits of repairing without any cost.

program	instructions	anti dep./ inst.	repairable	one reg. operand	combinable
abalone	100,000,000	1.25	30.1%	15.8%	8.1%
agrep	29,251,288	1.35	34.0%	8.6%	4.2%
dvips	51,155,896	1.22	18.2%	10.9%	4.1%
gcc-cc1	100,000,000	1.17	25.7%	16.6%	12.6%
gzip	100,000,000	1.37	27.2%	14.1%	4.8%
sicstus	91,433,314	1.16	31.8%	15.8%	10.7%

Fig. 10. Portion of anti dependences that can be removed with various forms of repairing

Compilers for register-starved architectures (in particular, the 386 architecture) can employ repairing of combinable instructions now to reduce the register pressure. For other architectures, there are probably still a few years left until register pressure becomes a significant problem. The large amount of parallel units available by then will make any form of repairing attractive that reduces register pressure.

6 Conclusions

We have introduced repairing, a compiler technique that can remove anti dependences and reduce register pressure: If an instruction writing to a register is moved up across an instruction reading from that register, repairing reconstructs the destroyed value from derived values using the inverse operation.

Repairing often has advantages over other techniques for removing anti dependences: Repairing produces less register pressure and it produces shorter live ranges, requiring less loop unrolling or fewer move instructions.

18%–34% of all anti dependences can be removed by repairing, about half of them are guaranteed to reduce register pressure (others may be profitable, too), and 4.1%–12.6% of antidependences can be removed in a way that reduces register pressure without increasing the number of executed instructions.

References

[BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, 1994.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 311–321, 1992.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):45–57, 1981. Reprinted in [Sta90].

[EK92] M. Anton Ertl and Andreas Krall. Instruction scheduling for complex pipelines. In *Compiler Construction (CC'92)*, pages 207–218, Paderborn, 1992. Springer LNCS 641.

[EK94] M. Anton Ertl and Andreas Krall. Delayed exceptions — speculative execution of trapping instructions. In *Compiler Construction (CC '94)*, pages 158–171, Edinburgh, April 1994. Springer LNCS 786.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[GM86] Phillip B. Gibbons and Steve S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, 1986.

[HG83] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 147–158, 1994.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[LDSM80] David Landskov, Scott Davidson, Bruce Shriver, and Pattrick W. Mallet. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[NE89] Toshio Nakatani and Kemal Ebcioğlu. "Combining" as a compilation technique for VLIW architectures. In 22^{nd} *Annual International Workshop on Microprogramming and Microarchitecture (MICRO-22)*, pages 43–55, 1989.

[PHS85] Yale N. Patt, Wen-mei Hwu, and Michael Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *The* 18^{th} *Annual Workshop on Microprogramming (MICRO-18)*, pages 103–108, 1985.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining. In *International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, 1994.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techgniques and an easily schedulable horizontal architecture for high performance scientific computing. In *14th Annual Microprogramming Workshop (MICRO-14)*, pages 183–198, 1981.

[Soh90] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined processors. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[Sta90] William Stallings, editor. *Reduced Instruction Set Computers.* IEEE Computer Society Press, second edition, 1990.

[Tho64] J. E. Thornton. Parallel operation in Control Data 6600. In *AFIPS Fall Joint Computer Conference*, pages 33–40, 1964.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.