

Optimizations and Oracle Parallelism with Dynamic Translation

Kemal Ebcioglu, Erik R. Altman, Sumedh Sathaye, and Michael Gschwind
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{kemal,erik,sathaye,mikeg}@watson.ibm.com

Abstract

We describe several optimizations which can be employed in a dynamic binary translation (DBT) system, where low compilation/translation overhead is essential. These optimizations achieve a high degree of ILP, sometimes even surpassing a static compiler employing more sophisticated, and more time-consuming algorithms [9]. We present results in which we employ these optimizations in a dynamic binary translation system capable of computing oracle parallelism.

1. Background and Motivation

Binary translation has attracted a great deal of attention lately [5, 10, 11, 18, 25, 26, 27]. Much (though not all) of this work has focused on functionally correct and efficient translation, as well as efficient translated code. There has been some work on optimizations uniquely suited to binary translation, such as determination of run-time constants for use in dynamic constant propagation, as well as other forms of value prediction. There has also been work on hardware structures which can expedite various forms of value prediction [17, 19].

However, we are not aware of any published work describing ILP extraction techniques *efficient enough* for use in a dynamic binary translation (DBT) framework. Although there is no hard and fast rule for what constitutes *efficient enough*, two criteria generally apply:

- An interactive user should observe no erratic performance due to time spent in translation, especially initial translation, e.g. an application initially taking a long time to respond to keyboard or mouse input.
- As a fraction of the overall runtime, the time spent in translation should be small. Alternatively, code/translation reuse should be high.

We have previously described an efficient technique for instruction scheduling for ILP [9, 10], including control and data speculation. In this paper, we describe several additional ILP extraction techniques suitable for use in such a binary translation system. These techniques cover **copy propagation**, **combining**, **load-store telescoping**, and **scheduling through indirect branches**. We have implemented all of these techniques in our **DAISY** system,

and have found overhead to be quite low — approximately 4000 *PowerPC* operations to translate one *PowerPC* operation [10]. Although we have not yet implemented them, we also describe algorithms for efficient **tree height reduction**, **software pipelining**, and **unification**.

The rest of the paper is organized as follows. Section 2 describes and illustrates each of the optimizations. Section 3 provides results showing the parallelism available when the optimizations described in Section 2 are applied. Section 4 discusses some related work, and Section 5 concludes.

2. Optimizations: Algorithms and Examples

For ease of understanding, we begin by providing a brief review of our scheduling algorithm, which is described in more detail in [9, 10]. Our algorithm maintains a **ready** time for each register and other resource in the system. This **ready** time reflects the earliest time at which the value in that register may be used. For example if the instruction `addi r3, r4, 1` has *latency* 1 and is scheduled at time 5, then the **ready** time for `r3` is $5 + 1 = 6$. When an operation such as `xor r5, r3, r9` is scheduled, our algorithm computes its earliest possible time $t_{earliest}$ as the *maximum* of the ready times for `r3` and `r9`. Our algorithm is greedy, and so searches forward in time from $t_{earliest}$ until a time slot with sufficient resources is available in which to place the instruction.

Resources fall into two broad types. First the appropriate type of functional unit (e.g. integer ALU) must be available on which to execute the instruction. Second, a register must be available in which to place the result of the operation. If the operation is scheduled in order (i.e. after predecessor operations in the original code have written their results to their original locations), then the result is just placed where it would have been in the original code. For example if `addi r3, r4, 1` is scheduled in order, the result is placed in `r3`. If the operation is executed speculatively then the result is placed in a register not present in the original architecture, and then copied into the original register in original program order. For example, if `addi r3, r4, 1` is executed speculatively, it might become `addi r63, r4, 1`, with `copy r3, r63` placed in the original location of the `addi`. Clearly the target architecture must have more registers than the original base architecture under this scheme.

2.1. Copy Propagation

There are a variety of instructions in the *PowerPC* architecture [14], and in most architectures for copying a value from one register to another. For example

```
addi  r3,r4,0
or    r3,r4,r4
oril  r3,r4,0
rlinm r3,r4,0,0,31
```

are all examples of operations which copy `r4` to `r3` in *PowerPC*. For high ILP, it is important to avoid including such instructions in a dependence chain. For example, consider the following fragment of *PowerPC* code,

```
addi  r3,r4,0
xoril r5,r3,0xFFFF
```

There is no real dependence between the `addi` and the `xoril`. These instructions can be performed in parallel as

```
addi  r3,r4,0      xoril  r5,r4,0xFFFF
```

In other words the earliest time $t_{earliest}$ for both the `addi` and the `xoril` is the same, since both only depend on the value in `r4`.

We implement **copy propagation** in order to recognize and deal with such cases. The heart of our **copy propagation** algorithm is a data structure:

```
typedef struct _rename {
    int         time;
    int         rename_reg;
    struct _rename *prev;
} RENAME;
```

This `RENAME` structure is kept for each integer register, e.g. `RENAME reg[32]` for an architecture such as *PowerPC* with 32 integer registers¹. All entries are initiated with the identity entry, for example:

```
reg[r4].time      = 0;
reg[r4].rename_reg = r4;
reg[r4].prev      = NULL;
```

Then as a *copy* instruction like `addi r3,r4,0` is scheduled at time slot `t`, the `r3` `RENAME` entry is updated:

```
reg[r3].time      = t;
reg[r3].rename_reg = r4;
reg[r3].prev      = &reg[r4];
```

(This assumes that register `r4` has not been modified in scheduling the current group of instructions.) In this way we know that from time 0 to `t-1`, the value currently in `r3` can be obtained from `r4`, and that after time `t`, the value can be obtained from `r3` itself. The update of this data structure can be done in constant time, while searching it in the worst case (where every instruction **copy propagates** the previous value) takes time proportional to the number of instructions. In practice, however, we have found **copy propagation** chains almost always contain only a few entries, typically 1.

¹The `RENAME` structure may also be kept for floating point or condition code registers.

2.2. Combining

Combining is in some ways a generalization of **copy propagation**. For example, consider the following *PowerPC* code fragment of a `forall` loop:

```
LOOP:
    lbz    r4,1(r3)
    stb    r4,101(r3)
    addi   r3,r3,1
    cmpi   cr0,r3,100
    bne    cr0,LOOP
```

Even though this is a `forall` loop (assuming the number of iterations is not greater than 100), the *apparent* loop carried dependence on `r3` prevents scheduling of the loop as a `forall` loop — unless **combining** is performed.

Figure 1(a) shows the loop with $2 \times$ unrolling, and indicates some of the dependences between operations, in particular those caused by apparent dependences on the loop induction variable in `r3`. Figure 1(b) shows how these operations might be scheduled on an infinite resource machine with unit latencies. Shaded/thickly outlined operations are from the second iteration of the loop. This schedule consumes 4 cycles for two iterations, and also consumes 5 issue slots. Figure 1(c) shows how these operations might be scheduled on the same machine after taking advantage of **combining**. Shaded operations directly benefit from **combining**. Thickly outlined boxes indicate operations from the second iteration. Only 2 cycles are required for the two iterations, but 9 issue slots are consumed.

Note that Figures 1(b) and (c) use *parallel semantics*. In other words for operations executed in the same cycle, all input values are read before any output values are computed. For example the `copy r3,r63` in the second cycle of Figures 1(c) does not change the input value of `r3` read by other operations in the second cycle. For this simple case, branches are evaluated left-to-right within the cycle, and if any branch takes, operations to the right of it in the cycle are not performed.

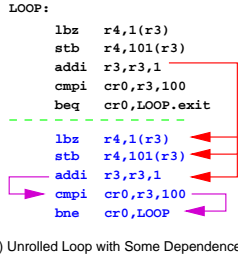
The **combining** in Figure 1(c) consists of computing the proper adjustments to immediate integer values used in `lbz`, `stb`, `addi`, and `cmpi`. For example, the address from which `lbz` loads in the second iteration is $2(r3)$, i.e. $r3_{orig} + 2$, since the `addi` increments `r3` by 1 each iteration.

Other forms of **combining** are possible. For example if the operation `add r3,r3,r4` occurs in a loop and `r4` is loop invariant, the value in `r3` can be computed for any given iteration as $r3_{orig} + Iter_num \times r4$. **Combining** is also not restricted to loops. For example, even if the following fragment is not in a loop

```
lbz    r4,1(r3)
addi   r3,r3,1
lbz    r5,1(r3)
```

the second `lbz` can have its address computed via **combining**:

```
lbz r4,1(r3)    addi r3,r3,1    lbz r5,2(r3)
```



(a) Unrolled Loop with Some Dependencies

VLOOP:

lbz r4,1(r3)	addi r63,r3,1			
stb r4,101(r3)	copy r3,r63	cmpi cr0,r63,100	lbz r4,1(r63)	addi r62,r63,1
beq cr0,LOOP.exit	stb r4,101(r3)	copy r3,r62	cmpi cr0,r62,100	
bne cr0,VLOOP				

(b) Parallel Schedule for Unrolled Loop with no combining

VLOOP:

lbz r4,1(r3)	addi r63,r3,1	cmpi cr15,r3,99	lbz r62,2(r3)	addi r61,r3,2	cmpi cr14,r3,98			
stb r4,101(r3)	copy r3,r63	copy cr0,cr15	beq cr15,LOOP.exit	copy r4,r62	stb r62,102(r3)	copy r3,r61	copy cr0,cr14	bne cr14,VLOOP

(c) Parallel Schedule for Unrolled Loop with combining

Figure 1. Effects of Unrolling and Combining

In order to efficiently implement the most common type of **combining** – for integer add and subtract operations, we use a data structure, similar to that used for **copy propagation**:

```

typedef struct _combine {
    int         time;
    int         combine_reg;
    int         offset;
    struct _combine *prev;
} COMBINE;

```

(Other types of **combining** are similarly treated.) As with **copy propagation**, a COMBINE structure is kept for each integer register. The only change from the RENAME structure is the new `offset` field, which is used to track how large a value must be added to `combine_reg` at time. Thus in the example just above, prior to scheduling the `addi r3, r3, 1` operation the COMBINE structure for `r3` is NULL. Afterwards it is

```

reg[r3].time         = 0;
reg[r3].combine_reg = r3;
reg[r3].offset       = 1;
reg[r3].prev         = NULL;

```

Thus if we schedule a subsequent operation, such as `lbz r5, 1(r3)`, performing an immediate addition or subtraction on the value in `r3`, we can immediately determine that if the operation is to be scheduled at time 0, 1 must be added to the value:

$$1+1(r3) = 2(r3)$$

As with **copy propagation**, the update of the COMBINE data structure can be done in constant time, while searching it in the worst case (where every instruction **combines**

the previous value) takes time proportional to the number of instructions. Again, in practice, however, we have found **combining** chains almost always contain only a few entries, corresponding to the degree to which we unroll loops (typically 4).

Note that it is important to maintain both **copy-propagation** and **combining** information, since **combining** can be used only on a small subset of operations. For example, if we encountered the operation `xor r3, r4, r5` and had a **copy-propagated** value for `r4` in `r8` at the time at which we wish to schedule the `xor`, we could merely replace `r4` with `r8`: `xor r3, r8, r5`. However, if we had only a *combineable* value, say `r4+7`, of `r4` at the desired scheduling time, nothing could be done, i.e. the `xor` would have to be scheduled later, possibly increasing the critical path.

2.3. Load-Store Telescoping

Load-Store Telescoping looks for cases when a load operation *must alias* with a previous store operation. When such cases are found, the dependence chain can be changed to avoid any access to memory. For example,

```

xor    r3,r5,r6
stw    r3,8(r1)
...
lwz    r4,8(r1)
xoril  r7,r4,0xFC

```

Assuming (1) that there are no changes to `r1` between the `stw` and the `lwz` and (2) that no other store operations occur between the `stw` and the `lwz` that may write

to $8(r1)^2$, such code can be transformed into the following parallel code:

```
xor r3,r5,r6
stw r3,8(r1) copy r4',r3 xoril r7',r3,0xFC
...
copy r4,r4' copy r7,r7'
```

Even if condition (2) does not hold and an ambiguous intervening store or load does occur, **load-store telescoping** may be employed if the hardware supports a **load-verify** operation [20] or some other means to detect an correct memory aliasing when it occurs. **Load-verify** loads a value and compares it to the value in its “*destination register*”. This destination register typically holds the value that was previously loaded, e.g. $r4'$ above. If the reloaded value and the value in the “*destination register*” are the same, then no change has occurred and execution can continue normally. Otherwise a trap occurs and execution must be fixed up and restarted. Thus the example above becomes:

```
xor r3,r5,r6
stw r3,8(r1) copy r4',r3 xoril r7',r3,0xFC
...
lwz.ver r4',8(r1) copy r4,r4' copy r7,r7'
```

Load-store telescoping holds special potential when translating unoptimized code in which most values are maintained in memory (typically the stack) and loaded and stored each time they are used. Such memory references can be eliminated from the critical path. Furthermore, since we propose to use this approach in a dynamic binary translation context, **load-store telescoping** can be done entirely transparently and unoptimized code can approach the performance of optimized code.

The importance of **load-store telescoping** is amplified by the use of **combining**, as **combining** allows us to determine that two different address expressions actually refer to the same location. For example,

```
stw r31,8(r1)
addi r1,r1,-64
lwz r3,72(r1)
xoril r4,r3,0xEE
```

After scheduling the `stw` and the `addi` at time 0, the **COMBINE** structure for $r1$ is

```
reg[r1].time = 0;
reg[r1].combine_reg = r1;
reg[r1].offset = -64;
reg[r1].prev = NULL;
```

Thus, when the `lwz` is scheduled at time 0, the address to which it refers is computed as $72-64(r1) = 8(r1)$, i.e. the same as the `stw` address. The resulting parallel code makes use of both **combining** and **load-store telescoping** to do everything in a single cycle:

²In a multiprocessor context, we may also require that no loads occur from the address at $8(r1)$, so as to maintain sequential consistency, i.e. so that a later load in the original code does not get an earlier value than an earlier load [16].

```
stw r31,8(r1) addi r1,r1,-64 ||
copy r3,r31 xoril r4,r3,0xEE
```

We also observe that the original register (e.g. $r31$) may have its value available even earlier, by **copy propagation**, **combining**, or further **load-store telescoping**. In other words, these optimizations can be repeatedly composed to find the earliest time at which a value is available. For example, consider the *PowerPC* code fragment above prefixed with additional operations:

```
stw r9,256(r18)
lwz r31,256(r18)
stw r31,8(r1)
addi r1,r1,-64
lwz r3,72(r1)
xoril r4,r3,0xEE
```

By two applications of **load-store telescoping** plus **combining**, the following single-cycle parallel code is obtained:

```
stw r9,256(r18) copy r31,r9 stw r9,8(r1) ||
addi r1,r1,-64 copy r3,r9 xoril r4,r9,0xEE
```

In order to perform **load-store telescoping**, we keep a list of all stores that we have seen on the current scheduling path. For each of these stores, we maintain several pieces of data in order:

- To determine if subsequent loads alias with the store. Some of the information necessary for this includes
 - The base address register.
 - The displacement from the base register.
 - **Combining** information about the base register.
 - The size of the store operation: 1/2/4/8 bytes.
- To determine the **copy propagation** and **combining** information associated with the source register of the store, e.g. $r9$ in `stw r9,256(r18)` above. This requires keeping a **RENAME** and a **COMBINE** struct for the source register of the store.

Given this list of stores and their associated information, each time a load is encountered, it is compared to all previous stores on the path to see if it *must alias*. If such a match is found, then the source register of the store can be used in place of the load as illustrated above. There are a few small additional details to make this work, and to make this work with indexed loads and stores like `stw r3,r4,r5`, but these do not change the general framework under which **load-store telescoping** operates. For example, even though a **load-store** pair may alias, resource constraints may not allow us to take advantage of **load-store telescoping** prior to the **store**, and after the **store**, the source register may be killed.

In the worst case, this algorithm is $O(N^2)$ in N , the number of operations, since operations could alternate between stores and loads with the result that $\frac{N}{2}$ loads would be compared to up to $\frac{N}{2}$ stores for a total of $\sum_{i=1}^{N/2} i = \frac{N^2}{8} + \frac{N}{4}$

```

V50:
  cmpl.indir cr15,PPC_LR,0x1000
  cmpl.indir cr14,PPC_LR,0x2000
  cmpl.indir cr13,PPC_LR,0x3000
  b         V51

```

```

V51:
  beq      cr15,V1000
  beq      cr14,V2000
  beq      cr13,V3000
  b        V52

```

```

V52:
  b      <Translation w/Start Corresponding to PPC_LR>
        OR
        <Binary Translator if no translation exists>

```

```

•••
V1000:
  # Translated Code from PowerPC 0x1000
•••
V2000:
  # Translated Code from PowerPC 0x2000
•••
V3000:
  # Translated Code from PowerPC 0x3000
•••

```

Figure 2. Translation of Indirect Branch

comparisons. However, in practice loads make up about 30% of most code and stores about 15%. This results in about $0.15N + 0.0225N^2$ comparisons on a typical path. A typical path has approximately 100 operations resulting in about 240 compares, or about 2.4 compares per operation. In any case, our scheduler uses a window to bound the maximum length of any path (to about $N = 250$ operations). For such a fixed upper bound N , the running time of this algorithm, is of course, constant.

2.4. Indirect Branches

Indirect (or register) branches can cause frequent serializations in our approach to dynamic binary translation (in which there is no operating system support for binary translation and all code from the original architecture is translated including OS code and low-level exception handlers). Such serializations can significantly curtail performance, and hence it is important to avoid them.

This can be accomplished for indirect branches by converting them into a series of conditional branches backstopped by an indirect branch. This is similar to the approach employed in Embra [30]. However, Embra checked only a single value for the indirect branch, whereas we check multiple values.

For example, consider a *PowerPC* indirect branch `blr`, (Branch to Link Register) which, the first 100 times it is encountered, goes to 3 locations in the original code, `0x1000`, `0x2000`, and `0x3000`. Then the binary translated code for this `blr` might as depicted in Figure 2.

We make several points about Figure 2:

- The `PPC_LR` value is kept in an integer register such as `r33` that is not architected in *PowerPC*.

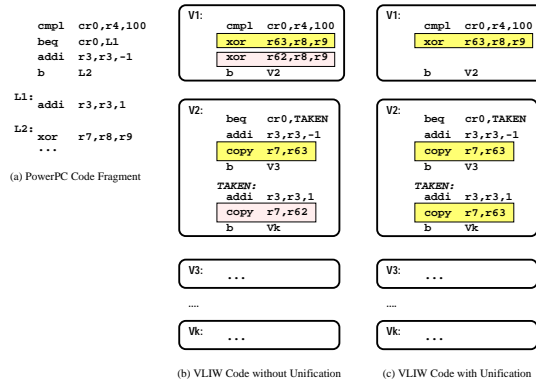


Figure 3. Effects of Unification

- Translated operations from `0x1000`, `0x2000`, and `0x3000` can be speculatively executed prior to `V1000`, `V2000`, and `V3000` respectively as resource constraints permit. Such speculative execution can reduce critical pathlengths and enable better performance.
- If additional return points such as `0x4000` are discovered in the future, the translated code can be updated to account for them — up to some reasonable limit on the number of immediate compares performed.
- A special form of compare, `cmpl.indir` is used because in *PowerPC* and most architectures, the register used for the indirect branch (e.g. `PPC_LR`) holds an *effective* (or *virtual*) address, whereas for reasons outlined in [9], it is important to reference translations by *real* address. The `cmpl.indir` operations in `V50` in Figure 2 translate the `PPC_LR` value to a real address before comparing it to the immediate (real address) value specified.
- It is also helpful if the `cmpl.indir` operation can specify a 32 or 64-bit constant, so as to avoid a sequence of instructions to assemble such an immediate value.

2.5. Unification

Our basic scheduling algorithm results in groups of instructions which are trees [10]. This can result in significant code explosion if the original code contains many *if-then-else diamonds*, as most integer code does. Figure 3(a) illustrates such a diamond in *PowerPC* code. If another such diamond occurred, the resulting tree for the group would have 4 total paths, while a 3rd diamond would result in 8 paths, etc. In other words code growth is exponential in the number of diamonds (and more generally in the number of *join points*). We use a variety of techniques to limit the size of groups [9, 11]. However, once groups are formed, we can still use **unification** to further reduce code explosion.

Returning to Figure 3, Figure 3(b) illustrates VLIW code that might be generated from the *PowerPC* code in Figure 3(a) without **unification**. All operations starting with the `xor` in Figure 3(a) are duplicated on both paths of the VLIW code as shown in Figure 3(b). (We have not speculated as much as is possible in this example, so as to simplify and highlight **unification** effects.) Clearly there is no need to compute the *same* `xor` result into both `r63` and `r62` in Figure 3(b). **Unification** notices this duplication and computes the result only once, as in Figure 3(c).

Unification is a form of *value numbering* [6], which is often employed in *common subexpression elimination*. With value numbering, as each expression is encountered, it is assigned a *value number*, typically an index into a hash table based on the input values/registers to the expression. Then as each new expression is encountered, its *value number* is compared to existing *value numbers*. If there is a match, then the earlier result can be reused and the value need not be recomputed.

In our context, expressions come from individual ALU operations. As each operation is scheduled into a VLIW instruction, a hash of the operation is stored in a unification field associated with the VLIW instruction. When subsequent operations are scheduled, they check the unification field of the VLIW instruction into which they are being scheduled. If the same expression is already computed in that VLIW, then there is no need to waste resources recomputing the value. Although the example in Figure 3 illustrated this for the case when operations beyond a join point are duplicated because of multipath scheduling, **unification** can be used for operations coming from arbitrary locations.

Actually the situation is slightly more complicated than just described. To check if this current operation (**CURR**) may be **unified** with a previously scheduled operation (**PREV**) in the unification field of a VLIW instruction, three conditions must be checked:

1. The input registers must be the same. Renaming and **copy propagation**, as described in Section 2.1, slightly complicate this check, as illustrated in Figure 4. The `and r7, r3, r8` after `L2:` in Figure 4(a) seems a likely candidate for unification as it occurs after a join point. However, as can be seen in Figure 4(b), `r3` maps to different registers in VLIW instruction `V2` where the `and` is scheduled. If the `beq` is taken, the value for `r3` is located in `r61` in `V2`. However, if the `beq` is not taken, the value for `r3` is located in `r63` in `V2`.

Thus before checking the unification field of VLIW instruction `Vx`, the scheduler must map all input registers `R` of operation `op` to the registers `R'` in which the value of `R` resides in `Vx`.

2. For **combineable** ops, the value of the **PREV** register and *offset* must be the same as those for **CURR** in the VLIW in which **PREV** occurs. This is similar to the previous case, and is illustrated in Figure 5. Figure 5(a) has a *PowerPC* code fragment in which the

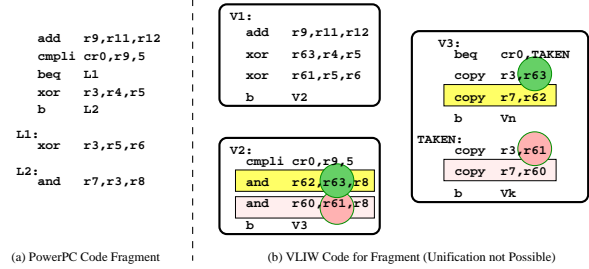


Figure 4. Unification not possible when input register mappings differ.

`final addi r7, r3, 8` operation occurs after a join point. However, in the VLIW code for this fragment in Figure 5(b), it can be seen that this `addi` operation needs different offsets: 7 and 9 because `r3` is modified differently on either side of the *diamond*. Hence this `addi` cannot be **unified**.

3. The destination register set by **PREV** must not be killed before **CURR** on the current path. This problem is illustrated in Figure 6. Figure 6(a) depicts a fragment of *PowerPC* code. The final `xor` statement is after a join, i.e. it occurs if both branches are taken *or* if the first branch falls through and the second branch is taken. However, depending on the order in which these paths are visited, **unification** of the `xor` may not be possible.

Figure 6(b) shows the skeleton of the group after scheduling the operations on the path where the first branch is taken (the path beginning at the top and ending at the lower right with `b Q`). (Note that the groups in Figure 6 are composed of multiple VLIW instructions, not just one VLIW.) The `xor` result is speculatively placed in `r63`. `r63` is subsequently live over the shaded area in Figure 6(b).

Figure 6(c) shows the skeleton of the group after scheduling the operations on the path when both branches fall-through (the path ending at the lower left with `b P`). The `and` result is for some reason unable to be speculated all the way to the start of the group, but can be speculated past the second branch. Since the `xor` value of `r63` is not live on this path, `r63` is used to hold the `and` result. This value of `r63` is live over the newly shaded area in Figure 6(c).

Figure 6(d) shows the skeleton of the group after scheduling the operations on the path when the first branch falls-through and the second is taken (the middle path ending at `b Q`). **Unification** of the `xor` is not possible because the result on path `T1` is in `r63` and `r63` is live with the `and` result along part of the path between `T3` and the start of the group, as is indicated by the shading.

A smart register allocator could clearly come up with an assignment that would allow **unification** here. However, our register allocator is simple, greedy, and

```

    cmpb    cr0,r4,100
    beq     cr0,L1
    addi    r3,r3,-1
    b      L2
L1:      addi    r3,r3,1
L2:      addi    r7,r3,8
    ...

```

(a) PowerPC Code Fragment

```

V1:      cmpb    cr0,r4,100
         addi    r63,r3,-1
         addi    r62,r3,1
         addi    r61,r3,7
         addi    r60,r3,9
         b      V2

```

```

V2:      beq     cr0,TAKEN
         copy    r3,r63
         copy    r7,r61
         b      V3
TAKEN:  copy    r3,r62
         copy    r7,r60
         b      V4

```

(b) VLIW Code Fragment (Unification Not Possible)

Figure 5. Unification not always possible with combining.

never backtracks, so as to work within the time constraints of dynamic binary translation [9, 10].

As with **load-store telescoping**, in the worst case, an operation must be checked against all previously scheduled operations to see if **unification** is possible, yielding $O(N^2)$ complexity, where N is the number of operations scheduled. However, in practice, we expect the unification field lookup to yield a very small number of operations to check as **unification** candidates, thus yielding a typical runtime that is $O(N)$.

2.6. Tree Height Reduction

Tree height reduction is illustrated in Figure 7. Figure 7(a) depicts a sequence of 7 add operations that form a dependence chain requiring 7 cycles to execute. Figure 7(b) shows how those add operations can be rearranged so that all results are computed in 3 cycles (at the cost of performing 13 add's instead of 7 and at the cost of doing up to 5 add's in one cycle instead of only 1).

The basic idea of **tree height reduction** is that operations in a linear dependence chain of length N can often be rearranged into a tree of operations of height $\lceil \log_2(N) \rceil$ by using the laws of *associativity* $((a + b) + c = a + (b + c))$, *commutativity* $(a + b = b + a)$, and *distribution* $(a(b + c) = ab + ac)$.

With our form of dynamic binary translation, which requires precise exceptions, all intermediate results must be computed as well, even if they eventually turn out to be dead. This is so that if an exception occurs, all registers will have their expected value from the original code. The shaded operations in Figure 7(b) are those *needed* to compute the *final* result in r16. Bold dependence arrows are also drawn between these *needed* operations. Dependences between other operations are drawn with smaller dependence arrows (or none at all in some cases to avoid clutter).

The requirement that intermediate results be computed prevents a straight-forward application of Brent's algorithm, which for any arithmetic expression with N operations, uses associativity, commutativity, and distribution to build in $O(N \log N)$ time, a tree of height at most $\lceil 4 \log_2(N - 1) \rceil - 1$ and using at most $3N$ function units [3,

4]. Brent's algorithm could be applied multiple times so as to compute each intermediate result not generated in creating the final result. However, in addition to increasing the time required for tree-height reduction, this approach may also require an excessive number of function units. Brent's algorithm could also be applied if it can be guaranteed (1) that no synchronous exceptions (such as page fault) can occur during execution of the expression in original program order and (2) that all intermediate results in the expression are dead. Alas, condition (1) requires that no loads occur anywhere in the expression, even if they are not part of the expression, and (2) is generally difficult to guarantee in our binary translation approach where full control flow graphs are not built.

Because of these difficulties, we propose a different, template-based approach. For clarity, we illustrate this approach on expressions using only one type of associative/commutative operation, e.g. add. However, templates could also be made for other commonly occurring cases, such as certain combinations of add and multiply. In this case, the distributive law can be applied as well. Our **tree height reduction** approach proceeds as follows:

- As each associative/commutative operation is scheduled, a check is made of its input registers:
 1. If neither of them is the result of an operation that is associative/commutative with the current operation, then there is no chance for **tree height reduction** of this operation.
 2. If (a) one of the input registers, **X** comes from an operation that is associative/commutative with this one, and (b) the other input has an *acceptable* availability time, then
 - Set the *depth* counter associated with this scheduled operation to one plus the value of the *depth* counter associated with the scheduled operation producing **X**.
 - Set the associative/commutative pointer associated with the current scheduled operation to the scheduled operation which produced **X**.

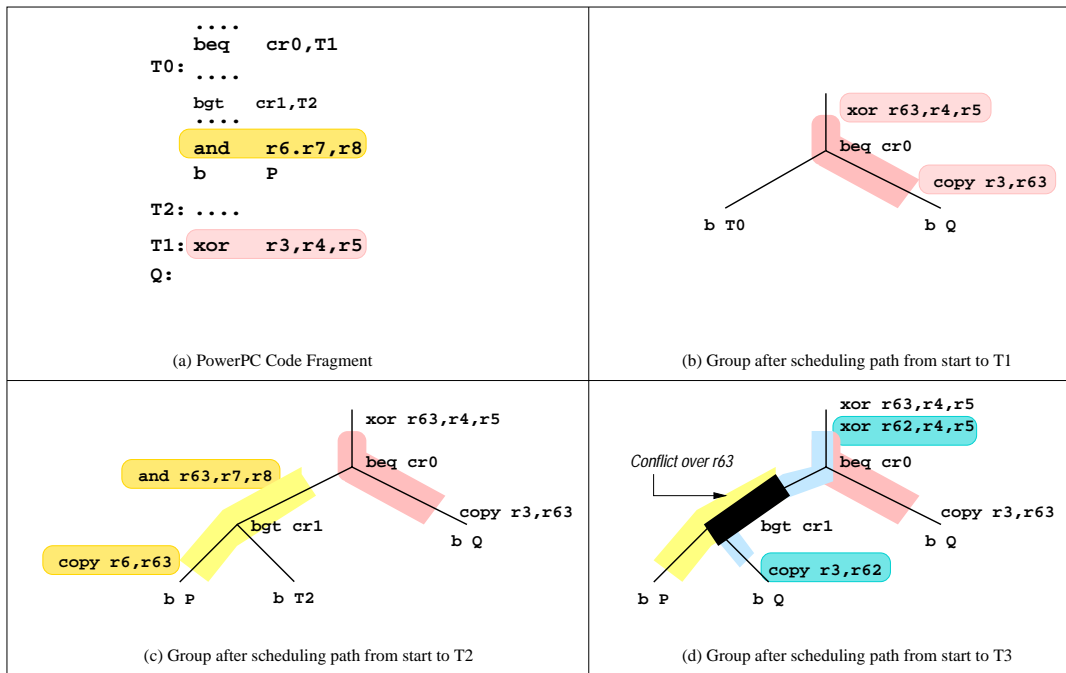
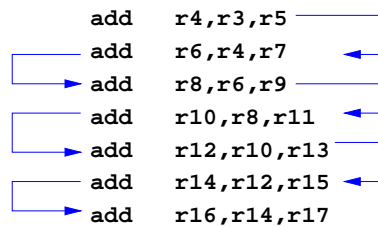
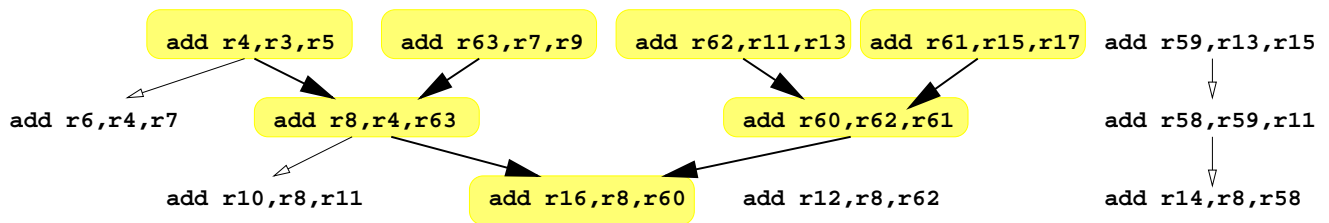


Figure 6. Unification sometimes fails because the destination register is live with another value.



(a) Associative operations in linear dependence chain.



(b) Operations after scheduling with tree height reduction.

Figure 7. Example of Tree Height Reduction

An *acceptable* availability time is any time that is less than the time of the operation starting this associativity dependence chain, i.e. the operation with $depth = 0$.

3. If both of the input registers come from operations associative/commutative with this one:
 - If input **X** would not have *acceptable* availability if input **Y** were chosen, but **Y** would have *acceptable* availability if input **X** were chosen, then choose **X**, and proceed as in (2).
 - If neither input **X** nor **Y** would have *acceptable* availability if the other were chosen, then give up on **tree-height reduction** for this chain.
 - If both inputs **X** and **Y** would have *acceptable* availability if the other were chosen, then choose the one that would yield the highest $depth$ and continue that dependence chain as in (2).

- If the $depth$ of an associative/commutative dependence chain reaches a “good” value, reschedule the operations in the chain using a **tree height reduction** template. The example in Figure 7 depicts a template for $depth = 7$.

Possible “good” values are 7 or 15: (1) The values 7 and 15 provide a full binary tree of computations. (2) They have good height reduction ratios, $7 : 3$ or $15 : 4$. In either case a marginally higher height would produce a worse ratio ($8 : 4$ or $16 : 5$). (3) The values 7 and 15 reflect plausible length sequences of dependent associative/commutative operations. Longer chains are somewhat unlikely and in any case can benefit from **tree height reduction** of shorter sequences within them. Shorter chains would receive relatively little benefit from **tree height reduction**.

This algorithm is $O(N)$ in N the number of operations:

- The data structures are updated in constant time as each operation is scheduled.
- Each operation is scheduled at most twice — originally and once more with **tree height reduction**.
- The number of operations required for **tree height reduction** is linear in the number of operations in the associative/commutative dependence chain.

2.7. Software Pipelining

A variation of *enhanced pipeline scheduling* [8] can be used efficiently with dynamic binary translation. Figure 8 gives an example illustrating the use of the algorithm.

- Figure 8(a) depicts a simple *PowerPC* loop which counts in $r6$ the number of entries in an array which are less than 9 and in $r5$ the number of entries which are greater than or equal to 9. The loop has 9 operations.

- This loop is initially scheduled as any other code as illustrated in Figure 8(b). More specifically three VLIW instructions encompassed by the bracket comprise the initial schedule. Figure 8 numbers the scheduling steps, and the text by the circled **1** also indicates the initial schedule. Note that the 9 *PowerPC* operations have already been reduced to 3 VLIW instructions.
- Given the initial schedule, Step 2 is to separate the first VLIW instruction $V1$ from the other two VLIW instructions $V2$ and $V3$ by a *fence* as denoted with **3** in Figure 8.
- This $V1$ becomes the loop prolog. A copy $V1'$ of $V1$ that can be thought of as the first VLIW instruction of the second iteration is appended after $V3$ as shown with Step 4.
- Step 5 illustrates the heart of the algorithm. The goal is to *schedule/software pipeline* all operations from this *2nd iteration* $V1'$ into VLIW's $V2$ and $V3$. For this goal to be met, two conditions must hold:
 - All operations from $V1'$ must be schedulable in the preceding VLIW's $V2$ and $V3$.
 - The set of operations in $V1'$ must produce the same outcome with parallel semantics as with sequential semantics. If not, $V1'$ must be modified so as to have correct sequential semantics. In this example, no changes are required. However, if $V1'$ wrote to $r3$ and later read from $r3$, the first write would need to be assigned a different register target, with that target being copied to $r3$ after the last (bottommost) read from $r3$ in $V1'$.
- The first 2 instructions ($V2, V3$) of Figure 8(c) show the result of scheduling the operations from $V1'$ into $V2$ and $V3$. The shaded areas indicate changes from Figure 8(b). Note that operations from $V1'$ are scheduled along both the *taken* and *fall-through* paths of $blt cr0, TAKEN1$ in $V3$. More precisely, $addi r63, r6, 1$ becomes $addi r59, r63, 1$ in $V2$ along the *taken* path, while becoming $addi r56, r6, 1$ along the *fall-through* path. Likewise $addi r61, r5, 1$ becomes $addi r57, r5, 1$ in $V2$ along the *taken* path, while becoming $addi r55, r61, 1$ along the *fall-through* path. **Unification** as described in Section 2.5 allows $lwz r3, 0(r62)$ and $addi r58, r62, 4$ to be common along both paths. After these steps, all results needed in $V1'$ are available at the time of $V3$. Thus $r3, r63, r62$, and $r61$ are appropriately updated along both non-exit paths through $V3$ as indicated by the highlighting in Figure 8(c). The now empty $V1'$ is discarded.
- Once this has been done, we mimic Steps 2, 3, 4, and 5 with Steps 6, 7, 8, and 9. More precisely, in Steps 6 and 7, $V2$ is fenced off from $V3$ and $V2$ is added to the loop prolog after $V1$ from Figure 8(b). Step 8 copies $V2$ to the end as $V2'$, and Step 9 software pipelines all of the operations of $V2'$ into $V3$.

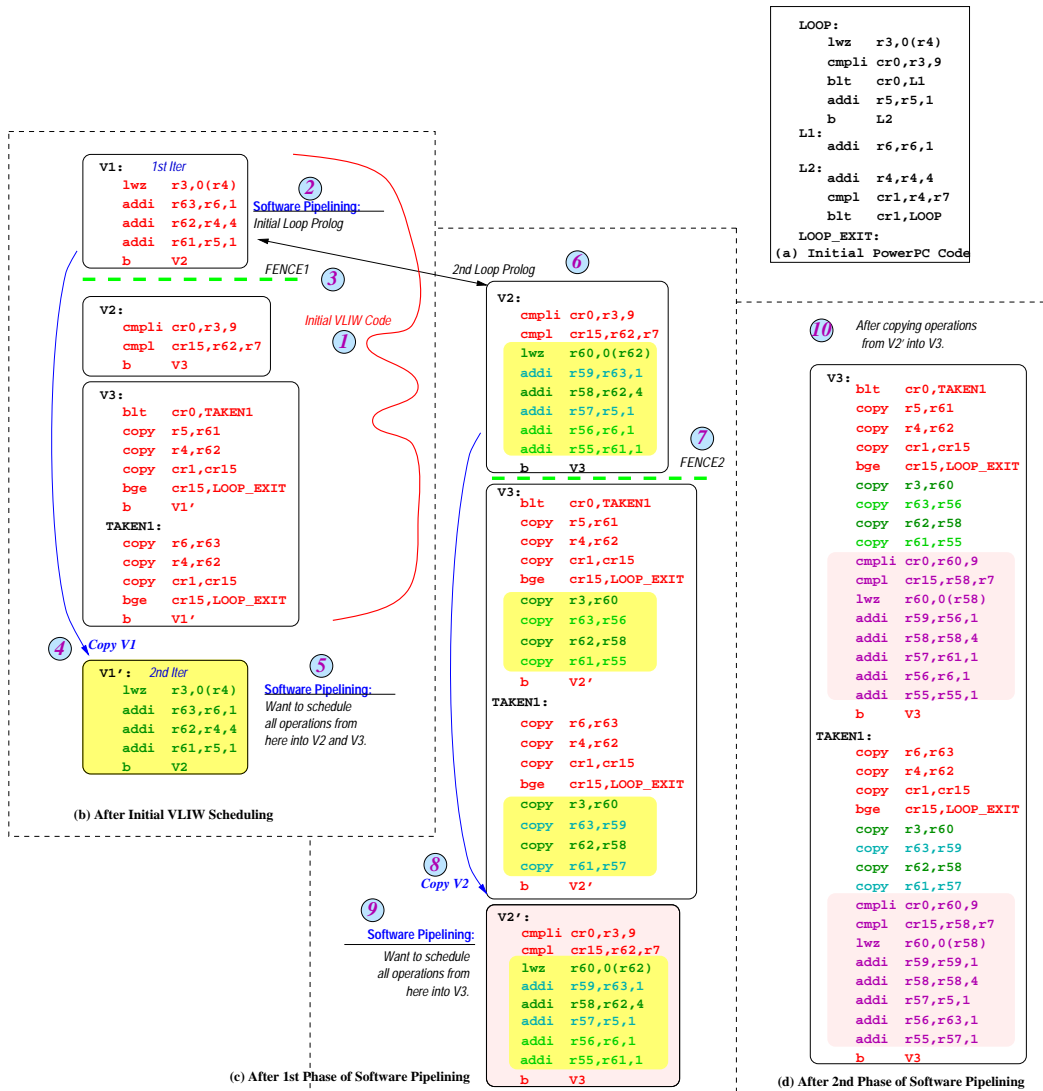


Figure 8. Example of Software Pipelining using variation of *Enhanced Pipeline Scheduling*

- Figure 8(d) shows the final result of this software pipelining — a loop which executes one iteration per cycle. Once again shaded areas show changes from Figure 8(c). Since all operations from iteration 1 have been scheduled, the algorithm stops. Notice that multiple instances of the same operation in the same VLIW can use a single resource in many implementations.

As this method of software pipelining is in some sense a variation on our basic scheduling algorithm, it has a worst case performance of $O(N^2)$, although in practice we expect it to be much closer to $O(N)$ [9].

3. Oracle parallelism results

In this Section we will provide some examples that show the combined effect of three optimizations discussed in this paper (copy propagation, combining, and load/store telescoping), on oracle ILP, using **SPECint95** and **TPC-C** traces for *PowerPC*.

In Table 1 are the results on oracle parallelism (infinite resource *PowerPC* IPC) on the first 300K instructions in each trace, turning on these three optimizations one by one,

then two at a time, and then all together. The IPC measure here is **Number of Original PowerPC instructions** / **Number of VLIW cycles** for an unboundedly wide VLIW. Note then that copy operations and speculative operations that do not contribute to the final result are not counted towards IPC.

As can be seen, there is a synergy between the optimizations such that the combined effect is sometimes greater than the individual effects. The combination of the optimizations described in the present paper can unleash a very high amount of oracle ILP on integer applications. The latencies used for this example were: 3 cycles for loads (4 for algebraic and partword loads); other latencies were similar to a *PowerPC 604*.

In Table 2 we provide the variation of oracle IPC with the length of the trace, for the case where all three optimizations are enabled. The oracle IPC can fluctuate as trace sections with lower or higher parallelism are navigated.

Of course, realistic machines will obtain much less parallelism, although these optimizations are likely to help there as well. In a separate paper [11], we describe a dynamic translation technique for obtaining a large window of in-

Prog	No Opt	Cpy	Cmb	Tel	Cpy+Cmb	Cpy+Tel	Cmb+Tel	All
compr	57.0	57.0	58.0	58.8	58.3	59.0	59.0	59.3
gcc	91.9	92.7	92.1	139.1	93.0	139.1	293.3	294.4
go	34.4	34.4	68.8	47.2	68.8	646.6	5263.2	5263.2
jpeg	18.0	18.0	18.0	72.9	18.0	73.0	373.1	374.5
li	21.7	22.5	22.2	27.2	23.1	27.2	1181.1	1960.8
m88k	98.7	98.8	1056.3	106.5	1119.4	106.7	2857.1	3000.0
perl	28.6	29.7	29.7	33.9	31.1	33.9	85.8	87.4
tpcc	25.8	26.7	27.0	50.0	28.1	51.3	58.1	60.9
vortex	33.2	33.2	193.3	38.4	194.3	38.4	495.9	499.2

Table 1. Oracle parallelism (IPC) with and without optimizations

Prog	100	1K	10K	100K	200K	250K	300K
compr	6	40	74	575	360	176	59
gcc	5	48	476	4762	9524	702	294
go	5	25	208	1754	3509	4386	5263
jpeg	10	53	13	125	250	312	375
li	6	42	238	2041	2326	2336	1961
m88k	7	56	556	1923	2667	2941	3000
perl	8	37	35	29	58	73	87
tpcc	7	14	44	67	55	58	61
vortex	8	40	370	1099	702	583	499

Table 2. Variation of oracle IPC with trace length

structions so as to enhance the effect of scheduling. That paper provides results for a 16-issue clustered machine, taking into account realistic CPI factors such as finite issue width and cache effects.

4 Related work

Most of the optimizations we have described are not new. Much of the contribution of this work is instead to show efficient implementations of these optimizations suitable for use in a dynamic binary translation system in which compile/translate time is very limited. More specifically:

- **Copy propagation** is a basic optimization described in [1].
- **Combining** is a form of *constant propagation* [1], whose application for extracting ILP was described in [21].
- **Load-store telescoping** of the sort we describe is unique to dynamic binary translation, since in order to be generally applicable, it must be possible to recover if **Load-store telescoping** turns out to be impermissible in a particular instance. Moshovos and Sohi described a hardware-based method for performing a similar function [19]. However, the window over which hardware works is much more limited. In addition, the power of **load-store telescoping** is multiplied in our context because of its interaction with **copy propagation** and **combining**. Also, when load-verify failures occur often, the code fragment is recompiled, this time not performing the telescoping on the offending load. This technique puts a bound on the overhead

of our approach, whereas hardware approaches may have to deal with incorrect guesses repeatedly.

- The problem of **scheduling through indirect branches** is related to memory aliasing [12, 13] and in particular the problem of determining all possible call sites for a function, and the problem of trying to determine all possible functions an indirect call may invoke. However, these all involve static analysis. Our approach takes advantage of the fact that it is done at runtime, and merely looks at the actual locations targeted by indirect branches and adds direct branch checks for those values. A related approach was described in [30] for use in the Embra simulation system.
- **Unification** has roots both in value numbering [6] and the scheduling work of Nicolau and Ebcioğlu [22, 7]. The approach described here is global in scope and quite efficient, but not as general as the Nicolau and Ebcioğlu approach.
- **Software pipelining** has a long history. Rau and Glaeser proposed modulo scheduling [24]. The approach here, however is much more closely related to enhanced pipeline scheduling [8], and again emphasizes an efficient implementation.
- A comprehensive **tree height reduction** algorithm was first proposed by proposed by Baer and Bovev [2]. This approach used only associativity and commutativity in performing its reduction. Brent [3] extended this approach by also making use of the distributive property. This early work applied only to operations in a single basic block. Nicolau and Potasman [23] proposed a **tree height reduction** technique capable of handling operations from multiple basic blocks. However, their algorithm was $O(N^2)$ in the number of operations N , whereas our algorithm is $O(N)$. In addition, our approach supports generation of the intermediate results needed for precise exceptions.

Considerable work has also been done on the limits of parallelism and *oracle parallelism* [29, 28, 15]. None of this work however examined the effect of performing optimizations such as **load store telescoping** and **combining** while scheduling for oracle parallelism.

5. Conclusion

We have described the implementation of several optimizations which can be employed in a *dynamic binary translation* system, where low compilation/translation overhead is essential. These optimizations have allowed us to achieve a high degree of parallelism in our **DAISY** experimental ILP compiler and toolset, and even to measure oracle parallelism.

References

- [1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers — Principles, Techniques, and Tools*, Addison-Wesley Publishers, Reading, MA, 1986.
- [2] J.L. Baer and D.P. Bovet, *Compilation of Arithmetic Expressions for Parallel Computations*, Proceedings of IFIP Congress, North-Holland, Amsterdam, pp. 340-346, 1968.
- [3] R. Brent, *The Parallel Evaluation of General Arithmetic Expressions*, Journal of the ACM, Vol. 21, No. 2, pp. 201-206, April 1974.
- [4] R. Brent and R. Towle, *On the Time Required to Parse an Arithmetic Expression for Parallel Processing*, International Conference on Parallel Processing, edited by P.H. Enslow, pp. 254, IEEE, August 1976.
- [5] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, J. Yates, *FX/32—A Profile-Directed Binary Translator*, IEEE Micro, Vol. 18, No. 2, pp. 56-64, March 1998.
- [6] J. Cocke and J.T. Schwartz, *Programming Languages and Their Compilers: Preliminary Notes*, Technical Report, Courant Institute of Mathematical Sciences, New York University, 1970.
- [7] K. Ebcioglu, *Some Design Ideas for a VLIW Architecture for Sequential-Natured Software*, In Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing), edited by M. Cosnard et al., pp. 3-21, North Holland, 1988.
- [8] K. Ebcioglu and T. Nakatani, *A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture*, In Languages and Compilers for Parallel Computing, D. Gelernter, A. Nicolau, and D. Padua (eds.), Research Monographs in Parallel and Distributed Computing, pp. 213-224, MIT Press, 1990.
- [9] K. Ebcioglu and E. Altman, **DAISY: Dynamic Compilation for 100% Architectural Compatibility**, Report No. RC 20538, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1996, <http://www.research.ibm.com/vliw/pubs.html>
- [10] K. Ebcioglu and E. Altman, **DAISY: Dynamic Compilation for 100% Architectural Compatibility**, Proceedings of ISCA-24, pp. 26-37, Denver, CO, June 1997.
- [11] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind *Execution-based Scheduling for VLIW Architectures*, To Appear in Proceedings of Europar-99, Toulouse, France, August/September 1999.
- [12] M. Emami, R. Ghiya, and L.J. Hendren. *Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*, Proceedings of SIGPLAN PLDI, pp. 242-256, Orlando, FL, June 1994.
- [13] L.J. Hendren, J. Hummel, and A. Nicolau, *Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs*, Proceedings of SIGPLAN PLDI, pp. 249-260, San Francisco, CA, June 1992.
- [14] IBM and Motorola, *The PowerPC Microprocessor Family: The Programming Environments Manual for 32-Bit Microprocessors*, www.mot.com/SPS/PowerPC/teksupport/teklibrary/manuals/pem32b.pdf.
- [15] M. S. Lam and R. P. Wilson, *Limits of Control Flow on Parallelism*, Proceedings of ISCA-19, pp. 46-57, Gold Coast, Australia, May 1992.
- [16] Leslie Lamport, *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, Vol. 28, No. 9, pp. 690-691, September 1979.
- [17] M.H. Lipasti and J.P. Shen, *Exceeding the Dataflow Limit via Value Prediction*, Proceedings of Micro-29, Paris, France, December 1996.
- [18] C. May, *MIMIC: A Fast System/370 Simulator*, Proceedings of SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, pp. 1-13, St. Paul, MN, June 1987.
- [19] A. Moshovos and G. Sohi, *Streamlining Inter-operation Memory Communication via Data Dependence Prediction*, Proceedings of Micro-30, Research Triangle Park, NC, December 1997.
- [20] M. Moudgill and J. Moreno, *Run-time Detection and Recovery from Incorrectly Ordered Memory Operations*, Report No. RC 20857, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1997, <http://www.research.ibm.com/vliw/pubs.html>
- [21] T. Nakatani and K. Ebcioglu, *Combining as a Compilation Technique for VLIW Architectures*, Proceedings of Micro-22, pp. 43-57, Dublin, Ireland, August 1989.
- [22] A. Nicolau, *Percolation Scheduling: A Parallel Compilation Technique*, TR 85-678, Department of Computer Science, Cornell University, 1985.
- [23] A. Nicolau and R. Potasman, *Incremental Tree Height Reduction for High Level Synthesis*, Proceedings of the 28th ACM/IEEE Design Automation Conference, pp. 770-774, San Francisco, CA, June 1991.
- [24] B.R. Rau and C.D. Glaeser, *Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing*, Proceedings of Micro-14, pp. 183-198, 1981.
- [25] G.M. Silberman and K. Ebcioglu, *An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures*, IEEE Computer, Vol. 26, No. 6, pp. 39-56, June 1993.
- [26] J.E. Smith, S. Sastry, T. Heil, T. Bezenek, M. Zhong, and V. Iyengar, *Achieving High Performance via Co-Designed Virtual Machines*, <http://www.ece.wisc.edu/jes/pitches/vms.ps>, November 5, 1998.
- [27] Sun Microsystems, *The Java Hotspot Performance Engine Architecture*, <http://java.sun.com/products/hotspot/whitepaper.html>, April 27, 1999.
- [28] K. B. Theobald, G. R. Gao and L. J. Hendren, *On the Limits of Program Parallelism and its Smoothability*, Proceedings of Micro-25, pp. 10-19, Portland, OR, December 1992.
- [29] D. W. Wall, *Limits of Instruction-Level Parallelism*, Proceedings of ASPLOS-IV, pp. 176-188, Santa Clara, CA, April 1991.
- [30] E. Witchel and M. Rosenblum, *Embra: Fast and Flexible Machine Simulation*, Proceedings of ACM SIGMETRICS'96, pp. 68-79, Philadelphia, PA, May 1996.