Operation Reuse on Handheld Devices

(Extended Abstract)

Yonghua Ding and Zhiyuan Li

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907
{ding,li}@cs.purdue.edu

Abstract. Compilers have long used redundancy removal to improve program execution speed. For handheld devices, redundancy removal is particularly attractive because it improves execution speed and energy efficiency at the same time. In a broad view, redundancy exists in many different forms, e.g., redundant computations and redundant branches. We briefly describe our recent efforts to expand the scope of redundancy removal. We attain *computation reuse* by replacing a code segment by a table look-up. We use *IF-merging* to merge conditional statements into a single conditional statement. We present part of our preliminary experimental results from an HP/Compaq iPAQ PDA.

1 Introduction

Compilers have long used redundancy removal to improve program execution speed. For handheld devices, which have limited energy resource, redundancy removal is particularly attractive because it improves execution speed and energy efficiency at the same time. In a broad sense, any reuse of a previous result can be viewed as a form of redundancy removal. Recently, our research group has investigated methods to expand the scope of redundancy removal. The investigation has resulted in two forms of operation reuse, namely *computation reuse* and *branch reuse*.

Computation reuse can be viewed as an extension of common subexpression elimination (CSE). CSE looks for redundancy among expressions in different places of the program. Each of such expressions computes a single value. In contrast, computation reuse looks for redundancy among different *instances* of a code segment or several code segments which perform the same sequence of operations. In this paper, we shall discuss computation reuse for a single code segment which exploits value locality [13, 14, 22, 26] via pure software means.

We exploit branch reuse through an *IF-merging* technique which reduces the number of conditional branches executed at run time. This technique does not require special hardware support and thus, unlike hardware techniques, it does not increase the power rate. The merger candidates include IF statements which have identical or similar IF conditions which nonetheless are separated by other statements. The idea of IF-merging can be implemented with various degrees of

aggressiveness : the basic scheme, a more aggressive scheme to allow nonidentical IF conditions, and lastly, a scheme based on path profiling information. In the next two sections, we discuss these techniques respectively and compare each technique with related work. We make a conclusion in the last section.

2 Computation Reuse

Recent research has shown that programs often exhibit value locality [13, 14, 22, 26], a phenomenon in which a small number of values appear repeatedly in the same register or the same memory location. A number of hardware techniques [7–9, 13, 14, 17, 21, 26] have been proposed to exploit value locality by recording the inputs and outputs of a code segment in a reuse table implemented in the hardware. The code segment can be as short as a single instruction. A subsequent instance of the code segment can be simplified to a table look-up if the input has appeared before.

The hardware techniques require a nontrivial change to the processor design, typically by adding a special buffer which may contain one to sixteen entries. Each entry records an input (which may consist of several different variables) and its matching output. Such a special buffer increases the hardware design complexity and the hardware cost, and it remains unclear whether the cost is justified for embedded systems and handheld computing devices. Using a software scheme, the table size can be much more flexible, although table look-up will take more time. The benefit and the overhead must be weighed carefully.

In our scheme, we use a series of filtering to identify *stateless* code segments which are good candidates for computation reuse. Figure 1 shows the main steps of our compiler scheme. For each selected code segment, the scheme creates a hashing table to continuously record the inputs and the matching outputs of the code segment. Based on factors such as value repetition rate, computation granularity estimation, and hashing complexity, we develop a formula to estimate whether the table look-up will cost less than repeating the execution. The hashing complexity depends on the hash function and the input/output size. The hashing table can be as large as the number of different input patterns. This offers opportunities to reuse computation whose inputs and outputs do not fit in a special hardware buffer.

2.1 How to Reuse

Computation reuse is applied to a *stateless* code segment whose output depends entirely on its *input variables*, i.e. variables and array elements which have *upwardly-exposed reads* in the segment. The output variables are identified by liveness analysis. A variable computed by the code segment is an output variable if it remains live at the exit of the code segment. If we create a look-up hash table for the code segment, the input variables will form the hash key. An invariant never needs to be included in the hash key. Therefore, for convenience, we exclude invariants from the set of input variables.

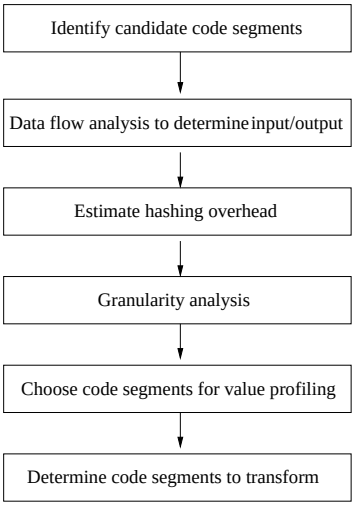

Fig. 1. Frame-work of the compiler scheme

The code segment shown in Figure 2(a) has an input variable *val* which is upwardly exposed to the entry of function *quan*. The array *power*2 is assumed to be invariant. The output variable is integer i which remains live at the exit of the function.

Our scheme collects information on three factors which determine the performance gain or loss from computation reuse, namely the *computation granularity*, the *hashing overhead*, and the input *reuse rate* of the given code segment. With the execution-frequency profiling information, it is relatively easy to estimate the computation granularity defined as the number of operations performed by the code segment. To get the reuse rate, we estimate the number (N_{ds}) of distinct sets of input values by value profiling and the number (N) of instances the code segment executed. We define the reuse rate ρ by the following equation:

$$\rho = 1 - \frac{N_{ds}}{N}$$

Based on the inputs and the outputs of the candidate code segment, we estimate the overhead of hashing table for computation reuse. The hashing overhead depends mainly on the complexity of the hash function and the size of each set of inputs and outputs.

To produce a hash key for each code segment, we first define an order among the input variables. The bit pattern of each input value forms a part of the key. In the case of multiple input values, the key is composed by concatenating multiple bit strings. In common cases, the hash key can be quite simple. For example, the input of the code segment in Figure 2(a) is an integer scalar, so the hash key is simply the value of the input. The hash index can simply be the hash key modularized by the hash size. Figure 2(b) shows the transformation result of the code segment in Figure 2(a).

```
int quan( int val ) {
   int i;

   for ( i = 0; i < 15; i++ ) {
      if ( val < power2[i] )
         break;
   }
   return (i);
}
```

(a)

```
int quan(int val ) {
   int i, key;

   if ( check_hash(val, hash_table, &key) == 0 ) {
      for (i = 0; i < 15; i++) {
         if (val < power2[i])
            break;
      }
      hash_table[key] = i;
   }
   else {
      i = hash_table[key];
   }

   return (i);
}
```

(b)

Fig. 2. An example code segment and its transformation by applying computation reuse

The hashing overhead depends on the size of the input and the output. The time to determine whether we have a hit is proportional to the size of the input. For a hit, the recorded output values should be copied to the corresponding output variables. For a miss, the computed output values must be recorded in the hashing table. In both cases, the cost of copying is proportional to the size of the output. In our scheme, we count the numbers of extra operations performed during a hit or a miss. (Note that a hit or a miss has the same number of extra operations.) A hashing collision can increase the hashing overhead. However, we assume there exist no hashing collisions.

2.2 Cost-Benefit Analysis

For a specific code segment, suppose we know the computation granularity C, the hashing overhead O, and the reuse rate ρ. The cost of computation before transformation equals C. The new cost of computation with computation reuse is specified by formula (1) below. Our scheme checks to see whether the gain by applying computation reuse, defined by formula (2), is positive or negative.

$$(C + O) \cdot (1 - \rho) + O \cdot \rho \tag{1}$$

$$C - [(C + O) \cdot (1 - \rho) + O \cdot \rho] \quad \equiv \quad \rho \cdot C - O \tag{2}$$

$$\rho \cdot C - O > 0 \quad or \quad \rho > \frac{O}{C} \tag{3}$$

In the above, computation reuse improve performance for the specific code segment if and only if the condition in formula (3) is satisfied. Obviously, reuse rate ρ can never be greater than 1. This gives us another criteria to filter out

code segments so as to reduce the complexity of value-set profiling. The compiler scheme removes code segments which do not satisfy $\frac{O}{C} < 1$ from further consideration. For the remaining code segments, value profiling is performed to get ρ.

After we obtain ρ, the compiler picks the code segments which satisfy formula (3) for computation reuse. Such code segments are transformed into codes that perform table look-up.

2.3 Value-set Profiling

Our scheme requires information on the reuse rate ρ which measures the repetitiveness of a set of input values for a code segment. This is in contrast to single-variable value profiling [5], where one can record the number of different values of the variable written by an instruction during the program execution. The ratio of this number over the total number of execution of the instruction defines the value locality at the instruction. (The lower the ratio, the higher the locality.) The locality of a set of values, unfortunately, cannot be directly derived based upon the locality of the member values. For example, suppose x and y each has two distinct values. The set of (x, y) may have two, three, or four distinct value combinations.

Therefore, our scheme first needs to define code segments for which we conduct value-set profiling. Given such a code segment, profiling code stubs can be inserted to record its distinct *sets* of input values. If we indiscriminately perform such value-set profiling for all possible code segments, the profiling cost will be prohibitive. To limit such cost, we confine the code segments of interest to those frequently executed routines, loops and IF branchs. Such frequency information is available by well-known tools such as *gprof* and *gcov*.

2.4 Experimental Results

We use Compaq's iPAQ 3650 for the experiments. The iPAQ 3650 has a 206MHZ Intel StrongArm SA1110 processor [1] and 32MB RAM, and it has 16KB instruction cache and 8KB data cache both 32 way set-associative. To test the energy consumption on the handheld device, we connect an HP 3458a high precision digital multi-meter to measure the actual current drawn on the handheld computer during the program execution.

We have experimented with six multimedia programs from Mediabench [16] and the GNU Go game. In our experiments, we use the default input parameters and input files as specified on the Mediabench web-site. The results from these programs are described below.

The two programs, G721_encode and G721_decode perform voice compression and decompression, respectively, based on the G.721 standard. They both call a function *quan* which have a computation reuse rate of over 99%.

The programs MPEG2_encode and MPEG2_decode encode and decode, respectively, MPEG data. Our scheme identifies the function $fdct$ for computation reuse in MPEG2_encode and the function $Reference\_IDCT$ in MPEG2_decode.

RASTA, which implements front-end algorithms of speech recognition, is a program for the rasta-plp processing. Its most time-consuming function *FR4TR* contains a code segment with one input variable and six output variables. The input repetition rate is 99.6%.

UNEPIC is an image decompression program. Its main function contains a loop to which our compiler scheme is applied. The loop body has a single input variable and a single output variable, both integers. The input has a repetition rate of 65.1%.

GNU Go is a go game. In our experiments, we use the input parameters "-b 6 -r 2", where "-b 6" means playing 6 steps in benchmark mode and "-r 2" means setting the random seeds as 2 (to make it easier to verify results). The function *accumulate_influence* contains eight code segments for computation reuse and the average repetition rate of inputs is 98.2%.

Table 1. Performance Improvement by Computation Reuse

Programs	Original (s)	Computation Reuse (s)	Speedup
G721_encode	2.01	1.53	1.31
G721_decode	3.69	2.76	1.34
MPEG2_encode	120.63	113.30	1.06
MPEG2_decode	83.02	46.06	1.80
RASTA	14.92	12.66	1.18
UNEPIC	1.73	0.76	2.28
GNUGO	788.05	654.51	1.20
Harmonic Mean			1.37

Tables 1 and 2 compare the performance and energy consumption, respectively, before and after the transformation. The machine codes (both before and after our transformations) are generated by GCC compiler (pocket Linux version) with the most aggressive optimizations (O3). The energy is measured in Joules (J).

Table 2. Energy Saving by Computation Reuse

Programs	Original (J)	Computation Reuse (J)	Energy Saving
G721_encode	4.59	3.56	22.4%
G721_decode	8.43	6.47	23.3%
MPEG2_encode	281.67	265.12	5.9%
MPEG2_decode	193.85	108.01	44.3%
RASTA	36.60	31.02	15.2%
UNEPIC	4.03	1.81	55.1%
GNUGO	1936.23	1613.69	16.7%

Since our computation reuse scheme is based on profiling, we test the effectiveness of the scheme with different input files. The program transformation is based on the profiling with default input files from the Mediabench web-site, and we run the transformed programs with other different input files. We show the results in Table 3. GNU Go has no input files, and we change the parameter from 6-step to 9-step. For each other program, we arbitrarily collect one input file from Internet or other benchmark suite such as MiBench [10]. We list the sources of input files in the second column of Table 3. For $G721$, we choose the input file *small.pcm* from the MiBench program $ADPCM$. We select the $tens\_015.m2v$, which plays table tennis, from Tektronix web-site, and extract the first 6 frames as the input of $MPEG2$ encode and decode. For $RASTA$, we choose the input file *phone.pcmbe.wav* in 1998's $RASTA$ test suite from ICSI. For $UNEPIC$, we get the input file $baboon.tif$ of $EPIC$, and we generate its $UNEPIC$ input file by running $EPIC$ with the $baboon.tif$ as input. The last column of Table 3 shows the effectiveness of our scheme. Based on the profiling information with the default input files, these programs applied the computation reuse scheme can achieve substantial performance improvement for other different input files.

Table 3. Performance Improvement for Different Input Files

Programs	Sources of Inputs	Original (s)	Computation Reuse (s)	Speedup
G721_encode	MiBench	9.12	6.77	1.35
G721_decode	MiBench	8.60	6.32	1.36
MPEG2_encode	Tektronix(table tennis)	175.36	147.47	1.19
MPEG2_decode	Tektronix(table tennis)	139.32	94.37	1.48
RASTA	ICSI(rasta_testsuite_1998)	37.87	31.98	1.18
UNEPIC	EPIC web-site(baboon.tif)	7.26	1.71	4.25
GNUGO	"-b 9 -r 2"	1485.28	1236.96	1.20
Harmonic Mean				1.43

2.5 Related Work

Since Michie introduced the concept of memoization [17], the idea of computation reuse had been used mainly in the context of declarative languages until the early 90's. In the past decade, many researchers have applied this concept to reuse the intermediate computation results of previously executed instructions [7–9, 13, 14, 21, 26]. Richardson applies computation reuse to two applications by recording the previous computation results in a *result cache* [21]. However, he does not specify how the technique was implemented, and the result cache in his paper is a special hardware cache. Sodani and Sohi [26] propose an instruction reuse method. The performance improvement of instruction level reuse is not significant, due to the small reuse granularity [27]. In the block and sub-block reuse schemes [13, 14], hardware mechanisms are proposed to exploit computation reuse in a basic block or sub-block. The reuse granularity on basic block

level seems still too small, and the hardware needs to handle a large number of basic blocks for computation reuse.

Connors and Hwu propose a hybrid technique [9] which combines software and hardware for reusing the intermediate computation results of code regions. The compiler identifies the candidate code segments with value profiling. During execution, the computation results of these reusable code regions are recorded into hardware buffers for potential reuse. Their compiler analysis can identify large reuse code regions and feed the analysis results to the hardware through an extended instruction set architecture. In the design of the hardware buffer, they limit the buffer size to 8 entries for each code segment.

3 IF-Merging

Modern microprocessors use deep instruction pipelining to increase the number of processed instructions per clock cycle. Branch instructions, however, tend to degrade the efficiency of deep pipelining. Further, conditional branches reduce the size of basic blocks, introduce control dependences between instructions, and hence may hamper the compiler's ability to perform code improvement techniques such as redundancy removal, software pipelining, and so on [4, 15, 30].

To reduce the penalty due to branch instructions, researchers have proposed many techniques, including static and dynamic branch prediction [2, 25], predicated execution [20, 23], branch reordering [30], branch alignment [6] and branch elimination [4, 15, 18], etc. Among these, branch prediction, especially dynamic branch prediction, has been extensively studied and widely used in modern high-performance microarchitectures. Branch prediction predicts the outcome of the branch in advance so that the instruction at the target address can be fetched without delay. However, if the prediction is incorrect, the instructions fetched after the branch have to be squashed. This situation results in a waste of CPU cycles and power consumption. Hence, a high prediction rate is critical to the performance of high-performance microprocessors. To achieve a high prediction rate, almost all high-performance microprocessors today employ some form of hardware support for dynamic branch prediction.

In contrast, processors designed for power-aware systems, such as mobile wireless computing and embedded systems, must take both the program speed and the power consumption into consideration. The concern for the latter may often be greater than for the former on many platforms. A branch predictor dissipates a non-trivial amount of power, which can be 10% or higher of the total processor's power dissipation. Such a predictor, therefore, may not be found on microprocessors have more stringent power constraints [19].

Hardware support for predicated execution [11] of instructions has been used on certain microprocessors, such as Intel XScale. Predicated execution removes forward conditional branches by attaching flags to instructions. The instructions are always fetched and decoded. But if the predicate evaluates to false, then a predicated instruction does not commit. Obviously, the effectiveness of predi-

cated execution highly depends on the rate at which the predicates evaluate to true. If the rate is low, then the waste in CPU cycles and power can be rather high.

It is also worth noting that branch prediction, as a run-time technique, generally does not help enhance the compiler's ability to improve codes. Recently proposed speculative load/store instructions expose the control of speculative execution to the software, which may increase the compiler's ability to pursue more aggressive code improvement techniques [29]. However, by today's technology, hardware support for speculative execution tends to increase power consumption considerably. Therefore, such support is not available on microprocessors which have more stringent power constraints.

In order to reduce the number of conditional branches executed at run time, we perform a source-level program transformation called *IF-merging*. This technique does not require special hardware support and it does not increase the power rate. Using this technique, the compiler identifies IF statements which can be merged to increase the size of the basic blocks, such that more instruction level parallelism (ILP) may be exposed to the compiler backend and, at run time, fewer branch instructions are executed. The merger candidates include IF statements which have identical or similar IF conditions which nonetheless are separated by other statements. Programmers usually leave them as separate IF statements to make the program more readable.

The idea of IF-merging can be implemented with various degrees of aggressiveness: the basic scheme, a more aggressive scheme to allow nonidentical IF conditions, and lastly, a scheme based on path profiling information.

```
if ( sign ) {
   diff = −diff;
}

......

if ( sign )
   valpred −= vpdiff;
else
   valpred += vpdiff;
```

(a)

```
if ( sign ) {
   diff = −diff;
   ......

   valpred −= vpdiff;
}
esle {
   ......

   valpred += vpdiff;
}
```

(b)

Fig. 3. An example code shows opportunity of basic IF-merging

3.1 A Basic IF-merging Scheme

In the basic scheme, we merge IF statements with identical IF conditions to reduce the number of branches and condition comparison. Figure 3(a) shows

an example extracted from the Mediabench suite. In the example code, two IF statements with identical condition are separated by other statements, which we call *intermediate statements*. Based on the data dependence information, we find that such intermediate statements have data dependences with the two merger candidates. Hence, we cannot move any of these intermediate statements before or after the new IF statement. We duplicate the intermediate statements and place one copy in the then-component of the merged IF statement, and another in the else-component. Figure 3(b) shows the transformed code by applying IF-merging on the code in Figure 3(a).

Throughout this section, we assume the source program is structured. Thus we can view the function body as a tree of code segments, such that each node may represent a loop, a compound IF statement, a then-component, an else-component, or simply a block of assignment statements and function calls. The function body is the root of the tree. If a node A is the parent of another node B, then the code segment represented by B is nested in the code segment represented by A. Unless stated otherwise, the merger candidates must always have the common parent in such a tree.

Obviously, in all of our IF-merging schemes, we need to be able to identify identical IF conditions, which requires symbolic analysis of IF conditions. To facilitate such analysis, we perform alias analysis [12], *global value numbering* [24], and transform the program into static single assignment (SSA) form [28], such that variables with identical values can be clearly identified. We then apply a set of normalization rules to the predicate trees of IF conditions, including the sub-trees that represent the arithmetic expressions in those conditions. Such normalization rules and the ensuing symbolic comparisons have been discussed extensively in the literature of software engineering and parallelizing compilers.

```
if (tmp1==−32768 && tmp2==−32768)
  tmp2 = 32767;
else
  tmp2 = 0x0FFFF&((tmp1 *
       tmp2 + 16384) >> 15) ;

if (tmp1==−32768 && sri==−32768)
  tmp1 = 32767;
else
  tmp1 = 0x0FFFF&((tmp1 *
       sri + 16384) >> 15) ;
```

Original Code

```
if (tmp1 == −32768) {
 if (tmp2==−32768)
  tmp2 = 32767;
 else
  tmp2 = 0x0FFFF&((tmp1*tmp2+16384)>>15);
 if (sri==−32768)
  tmp1 = 32767;
 else
  tmp1 = 0x0FFFF&((tmp1*sri+16384)>>15);
}
else {
 tmp2 = 0x0FFFF&((tmp1*tmp2+16384)>>15);
 tmp1 = 0x0FFFF&((tmp1*sri+16384)>>15);
}
```

Transformed Code

Fig. 4. Nonidentical conditions with common sub-predicates and its transformation

3.2 IF-Condition Factoring

The basic IF-merging scheme only identifies IF statements with identical IF conditions for IF-merging. Suppose the conditions are nonidentical but have common sub-predicates. By factoring the conditions we can also reduce the number of branches. The left-hand side of Figure 4 shows an example code extracted from Mediabench, and the right-hand side of Figure 4 shows the transformed code.

Our factoring scheme identifies IF statements with conditions containing common sub-predicates, and it factors the common sub-predicates from the conditions to construct a common IF statement, which encloses the original IF statements with the remaining sub-predicates as conditions.

3.3 IF-merging with Path Profiling

With path profiling information [3], we can make the IF-merging technique even more aggressive. For example, in the case of the code in the left-hand side of Figure 5, if the path profiling shows that majority of executions go to both $S1$ and $S2$, then we can transform the code into that showed in the right-hand side of Figure 5.

```
if ( a ) {
   S1;
}

if ( b ) {
   S2;
}
```

Original Code

```
if ( a&& b ) {
      S1;
      S2;
}
else if ( a )
      S1;
else if ( b )
      S2;
```

Transformed Code

Fig. 5. Example code shows IF-merging with profiling

We note that the probability of both *taken* in the two IF statements is p_{ab}. If p_{ab} is greater than 0.5, merging the two IF statements will reduce the number of branches. (The original code has two branches and the merged code has $1+2*(1-p_{ab}) < 2$ branches.) The number of comparison operations (denoted by λ) in the transformed code is defined by Formula (4) below, where p_a is the probability of *taken* in the first IF statement.

$$\lambda = (1 + p_a) + (1 - p_{ab})(1 + (1 - p_a)) \tag{4}$$

$\lambda = 3 - 2p_{ab} + p_a p_{ab}$
$\Rightarrow \quad \lambda \geq 3 - 2p_{ab} + p_{ab}^2 \quad \Rightarrow \quad \lambda \geq 2 + (1 - p_{ab})^2 \quad \Rightarrow \quad \lambda \geq 2$
$\lambda = 3 - 2p_{ab} + p_a p_{ab}$
$\Rightarrow \quad \lambda = 3 - p_{ab}(2 - p_a) \quad \Rightarrow \quad \lambda \leq 3 - 0.5(2 - p_a) = 2 + 0.5p_a \quad \Rightarrow \quad \lambda \leq 2.5$
Hence, the number of comparison operations in the transformed code ranges

from 2 to 2.5 when p_{ab} is greater than 0.5. The original code has two comparison operations. Although the number of condition comparisons is increased after merging, the performance has a net gain. Further, the then-component of the merged IF statement may present more opportunities for other optimizations.

```
if ( a ) {
   if ( b ) {
      S1;
   }
   else {
      S2;
   }
}
else {
   S3;
}
```

Original code

```
if ( b ) {
    if ( a ) {
        S1;
    }
    else {
        S3;
    }
}
else { /* !b => a */
    S2;
}
```

Transformed code

Fig. 6. Nested IF statements and the transformation of IF-exchanging

Another case for consideration is nested IF statements whose conditions are dependent. For example, the condition (or its negation) of the inner IF statement may derive the condition of the outer IF statement. (Obviously, the opposite is normally false. Otherwise we can remove the inner IF statement.) Given such nested IF statements, with profiling information on the *taken* probability, we can decide whether it benefits to exchange the nesting. Figure 6 shows an example code of nested IF statements in the left-hand side, and the code after the IF-exchange transformation in the right-hand side. In this example, we suppose the condition $!b$ (the negation of b) implies the condition a. (For example, suppose b is $X > 0$ and a is $X \leq 100$.) We further suppose that, based on profiling information, the *taken* probability of the outer IF statement (p_a) is greater than that of the inner IF statement (p_b). In the original code, both the number of branches and the number of comparison are $1 + p_a$, and in the transformed code, both of them are $1 + p_b$. Since p_a is greater than p_b, the IF-exchange will reduce both the number of branches and the number of comparison.

3.4 Experimental Results

We have experimented with eight multimedia programs from Mediabench [16]. Tables 4 and 5 show the performance and energy consumption, respectively, before and after IF-merging. The machine codes (both before and after our transformations) are generated by GCC (pocket Linux version) with the most aggressive optimizations (O3). Due to the space limit, detailed explanations are omitted.

Table 4. Performance improvement by IF-Merging

Programs	Original (s)	Optimized (s)	Speedup
ADPCM_coder	0.0670	0.0607	1.104
ADPCM_decoder	0.0639	0.0594	1.076
G721_encode	2.01	1.88	1.069
G721_decode	3.69	3.46	1.066
GSM_toast	1.11	1.04	1.067
GSM_untoast	0.51	0.47	1.085
PEGWIT_encrypt	0.424	0.412	1.029
PEGWIT_decrypt	0.240	0.236	1.017
Harmonic Mean			1.063

Table 5. Energy Saving by IF-Merging

Programs	Original (J)	Optimized (J)	Saving
ADPCM_coder	0.0324	0.0294	9.3%
ADPCM_decoder	0.0325	0.0299	8.0%
G721_encode	0.9403	0.8855	5.8%
G721_decode	1.7375	1.6311	6.1%
GSM_toast	0.5374	0.5049	6.0%
GSM_untoast	0.2426	0.2228	8.2%
PEGWIT_encrypt	0.2226	0.2171	2.5%
PEGWIT_decrypt	0.1260	0.1241	1.5%

3.5 Related Work

To reduce branch cost, many branch reduction techniques have been proposed, which include branch reordering [30], conditional branch elimination [4, 18], branch alignment [6], and predicated execution [20, 23], etc. As we finish writing this paper, we have discovered that part of our work in Section 3.3 is similar to a recent independent effort by Kreahling et al [15]. They present a profile-based condition merging technique to replace the execution of multiple branches, which have different conditions, with a single branch. Their technique, however, does not consider branches separated by intermediate statements. Neither do they consider nested IF statements, which we consider in Section 3.3. We have also given an analysis of the trade-off which is missing in [15]. Moreover they restrict the conditions in the candidate IF statements to be comparisons between variables and constants. We do not have such restrictions.

Calder and Grunwald propose an improved branch alignment based on the architectural cost model and the branch prediction architecture. Their branch alignment algorithm can improve a broad range of static and dynamic branch prediction architectures. In [18], Mueller and Whalley describe an optimization to avoid conditional branches by replicating code. They perform a program analysis to determine the conditional branches in a loop which can be avoided by code replication. They do not merge branches separated by intermediate statements. In [30], Yang et al describe reordering the sequences of conditional branches us-

ing profiling data. By branch reordering, the number of branches executed at run-time is reduced. These techniques seem orthogonal to our IF-merging.

4 Conclusion

In this extended abstract, we use computation reuse and IF-merging as two examples of expanding the scope of redundancy removal. We show that both program execution time and energy consumption can be reduced quite substantially via such operation reuse techniques. It is clear that profile information is important in both examples. We believe that a general model for redundancy detection can be highly useful for uncovering more opportunities of redundancy removal. As our next step, our research group is investigating alternative models for this purpose.

Acknowledgments

This work is sponsored by National Science Foundation through grants CCR-0208760, ACI/ITR-0082834, and CCR-9975309.

References

1. Intel StrongARM SA-1110 Microprocessor Developer's Manual. October 2001.
2. T. Ball and J. Larus. Branch prediction for free. *Proc. of the Conference on Programming Language Design and Implementation*, 1993.
3. T. Ball and J. Larus. Efficient path profiling. *Proc. of the 29th International Symposium on Microarchitecture*, December 1996.
4. R. Bodik, R. Gupta, and M. Soffa. Interprocedural conditional branch elimination. *Proc. of the Conference on Programming Language Design and Implementation*, 1997.
5. B. Calder, P. Feller, and A. Eustace. Value profiling. *Proc. of the 30th Int. Symp. on Microarchitecture*, pages 259–269, December 1997.
6. B. Calder and D. Grunwald. Reducing branch costs via branch alignment. *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.
7. D. Citron and D. Feitelson. Hardware memoization of mathematical and trigonometric functions. *Technical Report, Hebrew University of Jerusalem*, March 2000.
8. D. Connors, H. Hunter, B. Cheng, and W. Hwu. Hardware support for dynamic activation of compiler-directed computation reuse. *Proc. of the 9th Int. Conf. on Architecture Support for Programming Languages and Operating Systems*, November 2000.
9. D. Connors and W. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. *Proc. of 32nd Int. Symp. on Microarchitecture*, pages 158–169, November 1999.
10. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, December 2001.

11. J. Hennessy and D. Patterson. Computer architecture: A quantitative approach. *Second Edition, Morgan Kaufmann.*
12. M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Trans. on Programming Languages and Systems*, 21(4), 1999.
13. J. Huang and D. Lilja. Exploiting basic block value locality with block reuse. *In The 5th Int. Symp. on High-Performance Computer Architecture*, January 1999.
14. J. Huang and D. Lilja. Balancing reuse opportunities and performance gains with sub-block value reuse. *Technical Report, University of Minnesota*, February 2002.
15. W. Kreahling, D. Whalley, M. Bailey, X. Yuan, G. Uh, and R. Engelen. Branch elimination via multi-variable condition merging. *Proc. of the European Conference on Parallel and Distributed Computing*, August 2003.
16. C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *Proc. of the 30th Int. Symp. on Microarchitecture*, pages 330–335, December 1997.
17. D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.
18. F. Mueller and D. Whalley. Avoiding conditional branches by code replication. *Proc. of the Conference on Programming Language Design and Implementation*, 1995.
19. D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. *Proc. of the 8th International symposium on High-Performance Computer Architecture*, February 2002.
20. J. Park and M. Schlansker. On predicated execution. *Technical Report. HPL-91-58, Hewlett Packard Laboratories*, May 1991.
21. S. Richardson. Exploiting trivial and redundant computation. *Proc. of the 11th Symp. on Computer Arithmetic*, pages 220–227, July 1993.
22. S. Sastry, R. Bodik, and J. Smith. Characterizing coarse-grained reuse of computation. *3rd ACM Workshop on Feedback Directed and Dynamic Optimization*, December 2000.
23. J. Sias, D. August, , and W. Hwu. Accurate and efficient predicate analysis with binary decision diagrams. *Proc. of the 33rd International Symposium on Microarchitecture*, December 2000.
24. T. Simpson. Global value numbering. *Technical report, Rice University*, 1994.
25. J. Smith. A study of branch prediction strategies. *Proc. of the 4th International Symposium on Computer Architecture*, May 1981.
26. A. Sodani and G. Sohi. Dynamic instruction reuse. *Proc. of the 24th Int. Symp. on Computer Architecture*, pages 194–205, June 1997.
27. A. Sodani and G. Sohi. Understanding the differences between value prediction and instruction reuse. *Proc. of the 31th Int. Symp. on Computer Architecture*, pages 205–215, December 1998.
28. M. Wolfe. High performance compilers for parallel computing. *Addison-Wesley Publishing Company*, 1996.
29. Y. Wu and Y. Lee. Comprehensive redundant load elimination for the ia-64 architecture. *12th International Workshop, LCPC'99*, August 1999.
30. M. Yang, G. Uh, and D. Whalley. Efficient and effective branch reordering using profile data. *Trans. on Programming Languages and Systems*, 24(6), November 2002.