A Compiler Scheme for Reusing Intermediate Computation Results

Yonghua Ding
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907
ding@cs.purdue.edu

Zhiyuan Li
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907
li@cs.purdue.edu

Abstract

Recent research has shown that programs often exhibit value locality. Such locality occurs when a code segment, although executed repeatedly in the program, takes only a small number of different values as input and, naturally, generates a small number of different outputs. It is potentially beneficial to replace such a code segment by a table which records the computation results for the previous inputs. When the program execution re-enters the code segment with a repeated input, its computation can be simplified to a table look-up. In this paper, we discuss a compiler scheme to identify code segments which are good candidates for computation reuse. We discuss the conditions under which the table look-up costs less than repeating the execution, and we perform profiling to identify candidates which have many repeated inputs at run time. Compared to previous work, this scheme requires no special hardware support and is therefore particularly useful for resource constrained systems such as handheld computing devices. We implement our scheme and its supporting analyses in GCC. We experiment with several multimedia benchmarks and the GNU Go game by executing them on a handheld computing device. The results show the scheme to improve the performance and to reduce the energy consumption quite substantially for these programs.

1. Introduction

Recent research has shown that programs often exhibit value locality [9, 10, 16, 17], 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 [4, 5, 6, 9, 10, 13, 15, 17] 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 it will be adopted for embedded systems and handheld computing devices.

In this paper, we present a software scheme based on compiler analysis, profiling information and program transformation. The scheme identifies code segments which are good candidates for computation reuse. For each selected code segment, the scheme creates a hash table to continuously record the inputs and the matching outputs of the code segment. Based on factors such as the value repetition rate, the computation granularity, and the 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 size of the input and the output. Unlike the special hardware reuse buffer, the hash table is very flexible in its size. It 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. On the other hand, the overhead of accessing and updating the hash table is higher than the overhead for the hardware buffer.

We have implemented a prototype of our scheme, as well as the supporting compiler analyses, in the GCC compiler. We performed experiments by running several multimedia benchmarks [12] and the GNU Go game [1] on a Compaq iPAQ handheld device. The preliminary results show that, despite the hashing overhead, the compiler scheme reduces the execution time and the energy consumption quite substantially for these programs.

In summary, this paper makes the following main contributions:

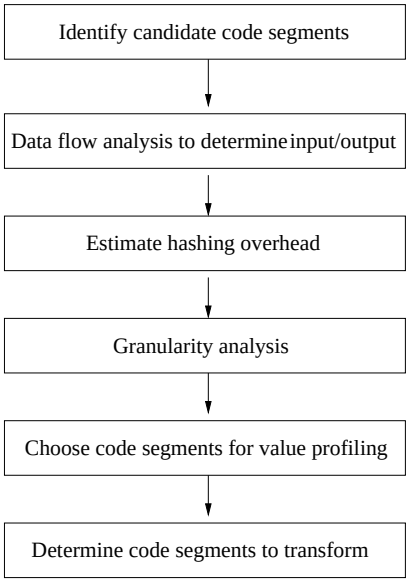

Figure 1. Frame-work of the compiler scheme

- We design and implement a compiler scheme to reuse the intermediate computation results. To the best of our knowledge, this is the first effort that uses a compiler approach for computation reuse without special hardware support.
- We present a cost-benefit analysis to identify code segments for computation reuse. For nested code segments, our cost-benefit analysis chooses the appropriate code segment to transform for computation reuse. Such analysis is absent in previous work.
- We give a detailed analysis of the experimental results. Previous work all use simulations in experiments. Our work is the first to provide experimental results from real measurement.

In the rest of the paper, we discuss our compiler scheme and its cost-benefit analysis in section 2, present experimental results in section 3, discuss related work in section 4 and make a conclusion in section 5.

2. A Compiler Scheme for Computation Reuse

Figure 1 shows the main steps in our compiler scheme. We will first briefly explain these steps and their ordering. We then address some of the key issues in the scheme.

2.1. The Main Steps

In our first step, we filter out code segments which are executed infrequently, in order to reduce value-profiling [3] cost. Our scheme requires information on the repetitiveness of a set of input values for a code segment. This is in contrast to single-variable value profiling, 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 and loops. Such frequency information can be collected using well-known tools such as *gprof* and *gcov*.

Next, we determine the input and output variables in the candidate code segments selected in our first step. We apply data flow analysis to the program. The inputs of a code segment are those variables or array elements that have *upward-exposed reads* in the code segment, excluding those recognized by the compiler as invariants at the entry of the code segment. (An invariant never needs to be included in the hash key.) 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. We have implemented such upward-exposure analysis and liveness analysis in a GCC compiler for C programs, including pointer and structure analysis necessary for identifying aliases. Algorithms for such analyses are well known.

Figure 2(a) shows an example code segment extracted from a Mediabench program which implements the G721 voice standard. The function $quan$ is invoked in several places within loops. The code segment has an input variable val which is upward exposed to the entry of function $quan$. The compiler recognizes $power2$ to be invariant after its initialization. The output variable is integer i which remains live at the exit of the function.

In the remaining steps, 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

```
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].output = i;
    }
    else {
        i = hash_table[key].output;
    }

    return (i);
}
```

(b)

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

number of distinct sets of input values (N_{ds}) by value profiling, and the number of execution instances of the code segment (N). We define the reuse rate R as

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

Based on the inputs and outputs of the candidate code segment, we estimate the hashing overhead 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.

The hashing overhead depends on the size of the input and the output. The time taken 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 hash table. In both cases, the cost of copying is proportional to the size of the output. In our scheme, we estimate the number of CPU cycles of the extra operations performed during a hit or a miss. (Note that a hit and a miss have the same number of extra operations.) A hash collision will increase the hashing overhead. To simplify the discussion, we assume there exist no hash collisions. During value-set profiling, we can count the hash collision rate for each value set and deduct the reuse rate accordingly. (In our experiments, only the program MPEG2 generates hash collisions.)

In the following subsections, we derive formulas to determine the performance gain or loss from computation reuse. Several other important issues are also discussed.

2.2. A Cost-Benefit Analysis

For a specific code segment, suppose we know the computation granularity C, the hashing overhead O, and the reuse rate R. The cost of computation before transformation equals C. The new cost of computation with computation reuse is specified by formula (1) below. The cost of computation now equals $C + O$ if hashing misses and it equals O if hashing hits. Our scheme checks to see whether the gain by applying computation reuse, defined by formula (2), is positive or negative. Hence, if and only if the condition in formula (3) is satisfied, the code segment has a performance gain from computation reuse.

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

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

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

Obviously, the reuse rate R 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 R.

After we obtain R, 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. Figure 2(b) shows the transformation result of the code segment in Figure 2(a).

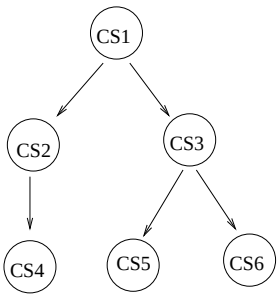

Figure 3. An example nesting graph shows nested code segments

```
static int quan( int val, short *table, int size) {
        int  i;

        for (i = 0; i < size; i++)
              if (val < *table++)
                    break;

        return (i);
}
```

Figure 4. Original code of $quan$ in G721

2.3. Nested Code Segments

In our current scheme, if one code segment is embedded in another and both segments can benefit from computation reuse according to the cost-benefit analysis, we choose only one of them for reuse. Examples of nested code segments include nested loops, loops in a routine, a routine call inside a loop and a routine calls another.

Based on the cost-benefit analysis in the previous subsection, we can determine the average performance gain, $R \cdot C - O$, for each code segment. Suppose such gain equals $g1$ for the outer code segment, and $g2$ for the inner code segment. Further suppose that, on average, n instances of the inner code segment are executed in each instance of the outer code segment. If the condition in formula (4) below is satisfied, then reusing the inner code segment will outperform that of the outer code segment. Otherwise, we reuse the outer code segment instead.

$$g1 - n \cdot g2 < 0 \tag{4}$$

If a code segment encloses two sequential code segments, the performance gain from the outer code segment will be compared with the sum of the gains from the two inner code segments.

To determine which code segments in those nested code segments to reuse, our compiler scheme constructs an interprocedural nesting graph to represent the nesting relationship among all code segments which can have performance gain from computation reuse. If the program contains recursive functions, then the nesting graph will contain cycles. For each non-singleton strongly connected component (SCC), we estimate the performance gain from computation reuse for each node in the SCC. The node with the best performance gain remains as a computation-reuse candidate according to the comparison described above. The rest of the nodes in the SCC are no longer candidates. We then condense the SCC into a single node. The estimated performance gain of the survivor in the SCC is assigned to the condensed node. After condensing all SCC's this way, the nesting graph becomes a DAG. We traverse the DAG bottom-up to determine which code segment in a nesting has the best performance gain according to the formula (4).

Figure 3 shows an example in which each node represents a code segment. An arc points from an outer code segment to an inner code segment. The compiler starts from the leaf nodes. It first compares the performance gain of $CS4$ with that of $CS2$ and marks the decision on the node $CS2$. It next compares the performance gain of $CS3$ with the sum of that of $CS5$ and $CS6$. The decision is marked on node $CS3$. Next, it compares the performance gain of $CS1$ with the sum of that recorded on $CS2$ and $CS3$. With that, the scheme can finally determine which code segments contribute the most performance gain.

2.4. Hashing Overhead Reduction

To reduce the hashing overhead, we apply code specialization to reduce the number of inputs and/or outputs of the candidate code segments. Specialization makes multiple versions of a code segment. In certain versions, some input variables become invariants. Specialization may increase the number of candidate code segments and hence the storage overhead for the hash table. However, the storage overhead is justified if the performance gain is substantial.

The code segment in Figure 2(a) is in fact a specialization of the function $quan$ in G721. Figure 4 shows the original code of $quan$, which has three input variables, an integer scalar $size$, an integer array $table$, and an integer scalar val. By an estimate, the hashing overhead outweighs the computation granularity. However, this function is called in several places in the program. For most of the call sites, the value of $size$ always equals 15 and the array $table$ is always a copy of another array $power2$ which is never changed after initialization. After specialization, these call sites will call a new version of $quan$ which has one input variable val. Computation reuse can now be applied to this version of $quan$ with a performance gain.

Entry	Input Var (IV) 1	...	IV n	Output Var (OV) 1	...	OV m
1						
2						
...						

Table 1. A hash table for a single code segment

To identify whether a variable is invariant in the execution of the code segment, our scheme performs a *code coverage analysis* to find all basic blocks which are in the execution paths from the first execution instance to the last execution instance of the code segment. If the variable remains unchanged in all these basic blocks, then it is invariant for the code segment.

2.5. Merging Hashing Tables to Reduce Storage Overhead

The hash tables consume extra memory. If we find multiple code segments with identical input variables, we can merge their hash tables to reduce the storage overhead. we include a bit vector in the merged hash table. Each bit represents one of the code segments and it records whether the output is available for a specific input. Table 1 shows the hash table for a single code segment. Table 2 shows the merged hash table for multiple code segments with identical input variables.

The GNU Go game is a good example to show the benefit of merged hash tables. The program has eight candidate code segments which have the identical set of input variables. Without merging, the transformed code runs out of memory on the Compaq iPAQ in our experiment. However, after merging the eight hash tables into one, the transformed code gets a speed up over 20% and an energy reduction over 16%. Further details are given in the next section.

3. Experimental Results

In this section, we first describe our implementation of computation reuse and then show the experimental results, including the performance and energy data.

3.1. Implementation

We have implemented the techniques described above, as well as a number of supporting analyses (see below), in the GCC compiler (v3.3). We implemented the new modules based on the abstract syntax tree in GCC. It is more difficult to collect information such as pointer deferences based on the lower level (i.e. the RTL level) intermediate representation in GCC. We implemented the following new modules in GCC:

- Call graph construction
- Clean-up
- Pointer analysis
- Control flow graph construction
- Def-use chains construction
- Code segment analysis
  - Granularity analysis
  - Hashing overhead analysis
  - Code coverage analysis
  - Array reference analysis for array input/output
- Code generation for computation reuse

In the call graph construction, we take into account function pointers and recursive functions. For recursive functions we compute their strongly-connected-component (SCC). The clean-up module is implemented to ease our subsequent analyses. For example, each function call in a complex expression is split from the expression in order to simplify the interprocedural analysis. We perform the pointer analysis [7] globally. For example, we can analyze a local pointer in one procedure which points to a local variable in another procedure. The construction of def-use chains [14] is also global because a definition in one procedure may be used in another procedure through pointers or global variables. In other words, there may exist a def-use chain whose definition and use are in different procedures. After we obtain the def-use chains for the entire program, we perform code segment analysis to identify candidate code segments for profiling. In code segment analysis, we estimate a lower bound on the granularity and an upper bound on the hashing overhead for each code segment. We confine the candidate code segment to a function body, a loop body, or an IF branch. To reduce the hashing overhead by reducing the number of inputs and outputs, we apply the code coverage analysis to identify invariant variables in the code segment. After profiling, we choose the code segments which satisfy formula (3) and transform them for computation reuse. Our implementation is a source-to-source transformation.

We implement a direct addressing hash table for each code segment. We generate the hash key by concatenating the values of input variables. If the hash key is not greater than 32 bits, we use the modularization to generate hash index. Otherwise, we perform a hash function [11] on the large hash key to generate a 32-bit hash key before the modularization. When hash collision happens, the previously recorded inputs and outputs in the entry is replaced

Entry	IV 1	...	IV n	Bit vector	Code Segment 1			...	Code Segment p		
					OV 1	...	OV m_1		OV 1	...	OV m_p
1											
2											
...											

Table 2. A Merged hash table

by the new inputs and outputs. The hash table size is determined based on the value profiling information, i.e., the number of distinct input patterns.

3.2. Experimentation Setup

We use a Compaq iPAQ 3650 PDA for the experiments. The iPAQ 3650 has a 206MHZ Intel StrongArm SA1110 processor [2] and 32MB RAM, and it has a 16KB instruction cache and an 8KB data cache, both being 32-way set-associative. By reducing the computation through reuse, we hope to reduce energy consumption on the PDA. To evaluate the impact of our transformation on 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. In order to get a reliable and accurate measurement, we disconnect the batteries from both the iPAQ and its extension pack, and we use a steady external 5V DC power supply instead. We use the built-in trigger mechanism in the multi-meter to start and stop measurement. After the test program starts running, the iPAQ triggers the multi-meter to read the current in a high frequency, and the trigger stops when the test program finishes. According to our measurement, the overhead associated with the triggering interrupts is less than 0.5% and the readings are consistent over repeated runs.

3.3. Test Programs

We have experimented with six multimedia programs from Mediabench [12] and the GNU Go game [1]. For other programs in Mediabench, our compiler scheme does not identify any significant code segments which benefit from computation reuse. 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 performs a linear search through a table. After a simple code specialization, the function $quan$ can be transformed to a function with one input variable and one output variable. The variation of inputs is small for both

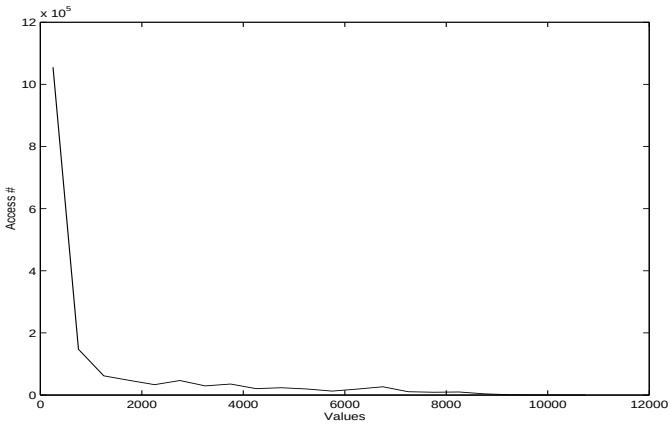

Figure 5. Histogram of input values in G721_encode

G721_encode and G721_decode. In our experiments, *quan* is invoked 1612942 times with 9155 different input patterns in program G721_encode, and it is invoked 2888970 times with 8884 different input patterns in program G721_decode. In both cases, the computation reuse rate is over 99%. The number of different input patterns, however, is much greater than the small hardware reuse buffer could store in the hardware approaches. Since both the input and the output are integers, the hash function is simple and the hash key is small. Figure 5 and 6 show the histogram of input values in function quan of G721_encode and G721_decode respectively. Figure 7 and 8 show the histogram of the table entries that were accessed.

As Figure 2(a) shows, the algorithm in the function quan is a linear search in a table whose elements increase by the power of 2. One might wonder whether computation reuse still benefits should the search be implemented differently. To answer this question, we revise the code by replacing the linear search with a binary search, and by replacing the table with shift operations respectively. We run the revised codes for both encode and decode. Figure 9 shows the revised function quan with a complete loop unrolling and a binary search. Figure 10 shows the function quan in which the table power2 is replaced by shift operations.

The programs MPEG2_encode and MPEG2_decode encode and decode an MPEG stream respectively. Our

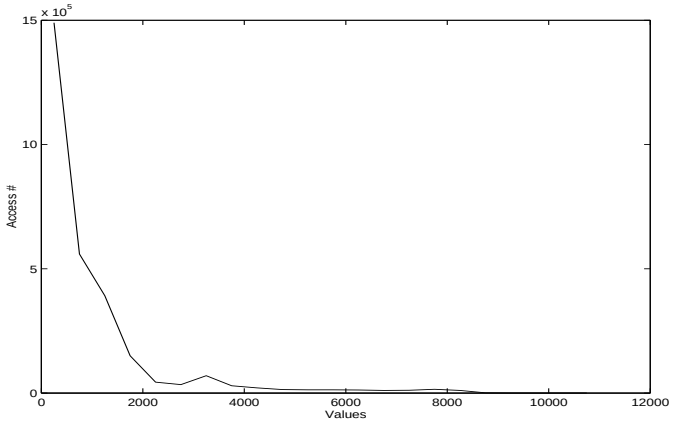

Figure 6. Histogram of input values in G721_decode

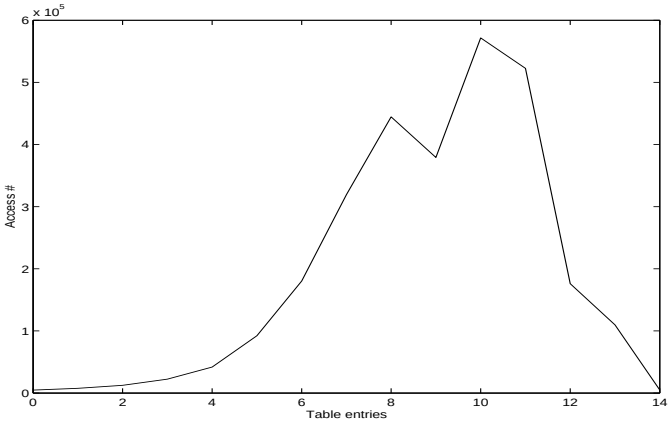

Figure 8. Histogram of accessed table entries in G721_decode

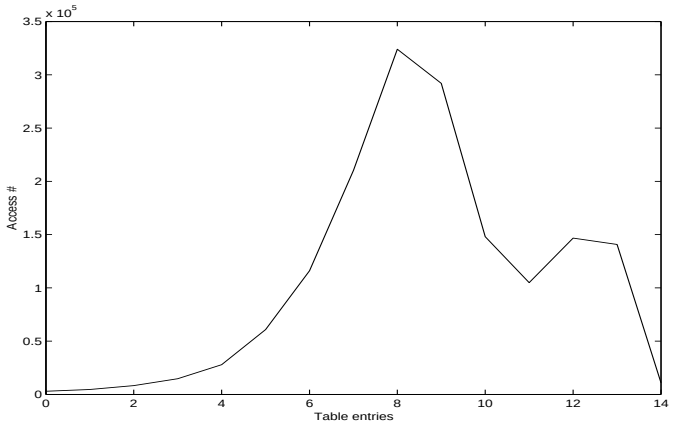

Figure 7. Histogram of accessed table entries in G721_encode

scheme identifies the function $fdct$ for computation reuse in MPEG2_encode and the function $Reference\_IDCT$ in MPEG2_decode. Both functions consume significant amount of computation time in their respective programs. They both have input and output of a 64-entry block. The hash key, being a 64-entry block, is much longer than the single integer used in the other programs mentioned above, and the hashing overhead is consequently higher. On the other hand, the computation granularity of the code segment is also considerably larger than those in the other programs. Hence, if the inputs have a high repetition rate, then it is still beneficial to reuse the computation. It turns out that the repetition rate is high in MPEG2_decode (48.6%), but not as high in MPEG2_encode.

$RASTA$, which implements front-end algorithms of speech recognition, is a program for the rasta-plp processing, and it supports the following front-end techniques: PLP, RASTA, and Jah-RASTA with fixed Jah-value. 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%. Figure 11 shows the histogram for the accesses of the distinct input patterns in the code segment of RASTA.

$UNEPIC$ 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%. Figure 12 shows the histogram of the input values in the code segment of UNEPIC.

GNU Go is a game which won the 8-th place in the 21st Century Cup in York, Pennsylvania and the second place in the European Go Congress. 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, each with four input variables and one output variable. Based on profiling, the input values fall in the range of [0,19]. For such small integer values, we use a single integer as the hash key which contains all four input values. The average repetition rate of inputs is 98.2%. Figure 13 shows the histogram of the input values in the code segments of GNU Go.

Table 3 lists statistics concerning several factors which affect the decision on whether to apply our computation reuse scheme. For each program, we show statistics only for the most significant code segment which has performance gain from computation reuse. The second column shows the computation granularity of one instance of the reusable

Programs	Computation	Overhead	DIP #	Reuse Rate	Hash Table Size
G721_encode	1.28	0.12	9155	99.4%	86KB
G721_decode	1.38	0.15	8884	99.7%	86KB
MPEG2_encode	13859	49.4	7617	9.8%	1.98MB
MPEG2_decode	12029	52.7	4068	48.6%	1.98MB
RASTA	333.7	59.5	31	99.6%	2KB
UNEPIC	29.45	0.61	22902	65.1%	512KB
GNUGO	26.3	2.14	46283	98.2%	4.47MB

Table 3. Factors which affect the optimization decision

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

   if ( val < power2[7] ) {
        if ( val < power2[3] ) {
             if ( val < power2[1] )
                  i = ( val < power2[0] ) ? 0 : 1;
             else
                  i = ( val < power2[2] ) ? 2 : 3;
        }
        else {
             if ( val < power2[5] )
                  i = ( val < power2[4] ) ? 4 : 5;
             else
                  i = ( val < power2[6] ) ? 6 : 7;
        }
   }
   else {
        if ( val < power2[11] ) {
             if ( val < power2[9] )
                  i = ( val < power2[8] ) ? 8 : 9;
             else
                  i = ( val < power2[10] ) ? 10 : 11;
        }
        else {
             if ( val < power2[13] )
                  i = ( val < power2[12] ) ? 12 : 13;
             else
                  i = ( val < power2[14] ) ? 14 : 15;
        }
   }
   return (i);
}
```

Figure 9. quan **with binary search**

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

      j = 1;
      for (i = 0; i < 15; i++) {
            if (val < j)
                  break;
            j = j << 1;
      }

      return (i);
}
```

Figure 10. quan **with table power2 replaced by shift operations**

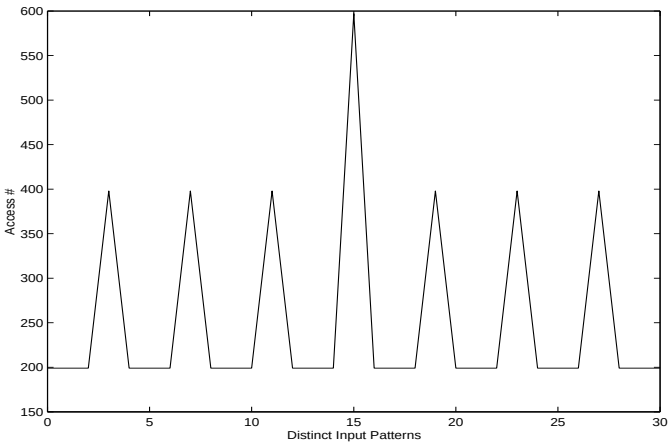

Figure 11. Histogram of distinct input patterns in RASTA

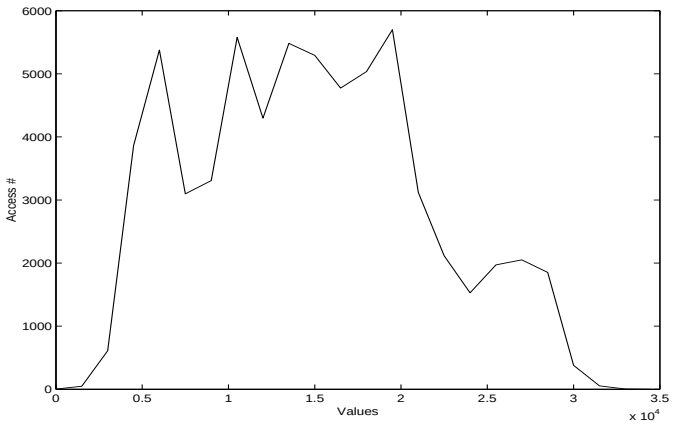

Figure 12. Histogram of input values in UNEPIC

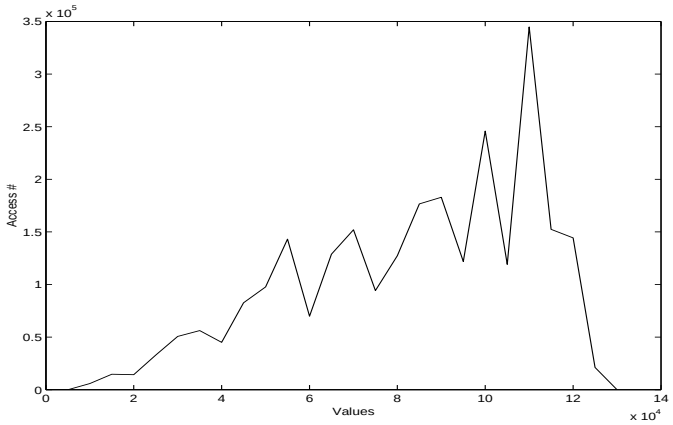

Figure 13. Histogram of input values in GNU Go

code segment in the unit of microsecond (μs). The third column shows the overhead (in μs) to access and manage the hash table in one instance of the execution of the code segment. The fourth column shows the number of distinct input patterns. The fifth column shows the reuse rate. The last column shows the extra memory consumption (in byte) of the hash table used in our experiment to get performance and energy data.

Table 4 lists the number of code segments analyzed, profiled, and transformed respectively in the specified functions. The last column shows the code size of the programs in the term of lines.

Table 5 shows the hit ratios when the hash table size is limited to 1-entry, 4-entry, 16-entry and 64-entry respectively. The LRU replacement policy is used. The results show that the hit ratio drops significantly for small hash tables. Note that in the current hardware proposals, the size of input/output is limited to no more than 8 entries for each code segment and the hash table size is limited.

Programs	Original (s)	Computation Reuse (s)	Speedup
G721_encode	4.40	2.82	1.56
G721_encode_s	4.12	2.78	1.48
G721_encode_b	2.84	2.55	1.11
G721_decode	8.01	5.00	1.60
G721_decode_s	7.49	5.00	1.50
G721_decode_b	5.15	4.56	1.13
MPEG2_encode	158.22	148.18	1.07
MPEG2_decode	96.65	53.22	1.82
RASTA	18.07	15.50	1.17
UNEPIC	2.14	0.93	2.30
GNUGO	1158.01	885.30	1.31
Harmonic Mean			1.46

Table 6. Performance Improvement with O0

Programs	Original (s)	Computation Reuse (s)	Speedup
G721_encode	2.01	1.53	1.31
G721_encode_s	1.83	1.51	1.21
G721_encode_b	1.60	1.48	1.08
G721_decode	3.69	2.76	1.34
G721_decode_s	3.42	2.73	1.25
G721_decode_b	2.99	2.73	1.10
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

Table 7. Performance Improvement with O3

3.4. Performance and Energy Data

Tables 6 and 7 show the performance improvement by our scheme applied to those seven programs mentioned above. G721_encode_s and G721_decode_s are modified codes with table power2 in function quan replaced by shift operations. In G721_encode_b and G721_decode_b, the linear search in function quan is replaced by a binary search. The harmonic mean excludes these modified codes. In Table 6, we first show results with both the original programs and the transformed programs compiled by GCC without any other optimization (O0). The second and third columns of the table show the execution time of original programs and transformed programs. By applying our scheme, 4 of the 7 programs have speedups greater than 1.5 over their

Programs	Functions	Analyzed CS	Profi led CS	Transformed CS	code size (lines)
G721_encode	quan, fmult, update	81	4	2	1.3K
G721_decode	quan, fmult, update	84	7	2	1.2K
MPEG2_encode	fdct	10	7	1	7.6K
MPEG2_decode	Reference_IDCT	11	5	1	8.2K
RASTA	FR4TR	27	3	1	6.1K
UNEPIC	main, collapse_pyr	69	1	1	0.9K
GNUGO	accumulate_influence	106	16	8	40K

Table 4. Number of code segments (CS)

Programs	1-entry	4-entry	16-entry	64-entry	64-entry Size (Byte)
G721_encode	0.1%	0.8%	3.1%	12.2%	512
G721_decode	0.04%	0.5%	2.3%	9.9%	512
MPEG2_encode	3.1%	5.1%	5.2%	5.4%	16384
MPEG2_decode	33.5%	44.7%	44.7%	44.7%	16384
RASTA	2.6%	17.9%	58.8%	99.6%	2048
UNEPIC	1.1%	1.1%	1.2%	1.4%	512
GNUGO	0%	0.01%	0.06%	0.3%	10240

Table 5. Hit Ratios with Limited Buffers

Programs	Original (J)	Comp. Reuse (J)	Energy Saving
G721_encode	10.25	6.60	35.6%
G721_decode	18.70	11.75	37.2%
MPEG2_encode	367.86	344.52	6.3%
MPEG2_decode	224.23	123.74	45.0%
RASTA	44.30	37.96	14.3%
UNEPIC	4.96	2.19	55.8%
GNUGO	2844.66	2185.35	23.2%

Table 8. Energy Saving with O0

Programs	Original (J)	Comp. 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%

Table 9. Energy Saving with O3

original programs. There is little performance improvement for MPEG2_encode because the reuse rate is only 9.8%.

Table 7 shows the performance results with both the original program and the transformed program compiled with the most aggressive optimizations (O3) in GCC. Our scheme is still shown to improve the performance of these programs considerably.

Tables 8 and 9 show the energy saving by our computation reuse scheme applied to these seven programs with O0 and O3 optimization levels respectively. Out of the seven programs, six achieve substantial energy saving by the software scheme. The second and the third columns of the tables show the energy consumption of the handheld computing device running the original programs and the transformed programs by applying the computation reuse, measured in the unit of Joule (J). We compute the energy consumption of the entire system of the handheld computer during the execution by the following equation:

$$energy = voltage * current\_drawn * elapsed\_time$$

In the above equation, $voltage$ is fixed to 5 volt since we use a steady external 5V DC power supply, and $current\_drawn$ is the average electrical current drawn during the execution of the program. The current fluctuates during the program execution, but the overall energy consumption is reduced in the transformed codes.

Figures 14 and 15 compare the speedups achieved by our scheme applied to these programs with different hash table sizes (in bytes). The optimal hash table size is determined based on the profiling information and is shown in the last column of Table 3. With the optimal size, the hash table holds all distinct inputs/outputs of its code segment with the default input file. Almost all these programs achieve good speedups by applying computation reuse with a hash table

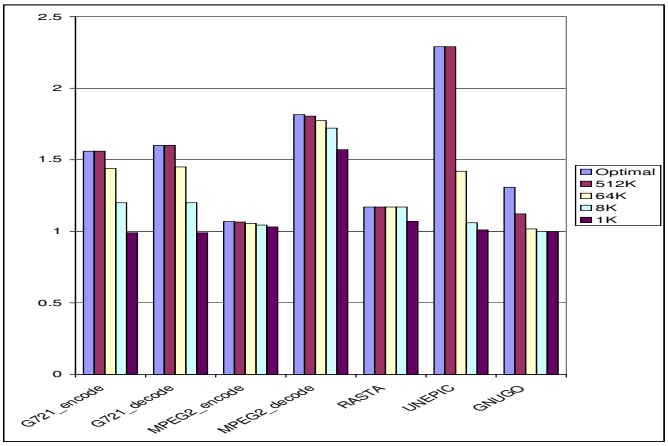

Figure 14. Under O0 optimization, speedups with different hash table sizes

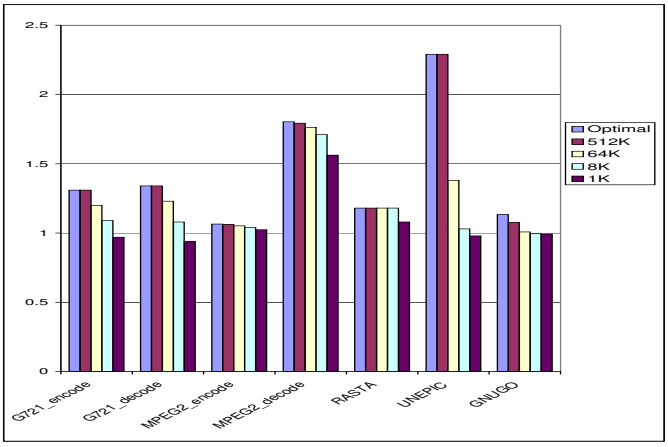

Figure 15. Under O3 optimization, speedups with different hash table sizes

of 512KB. This amount of storage overhead is affordable on handheld devices.

Since our computation reuse scheme is based on profiling, we evaluate the effectiveness of the scheme with different input files. The program transformation is based on the profiling with default input files from the Mediabench website, and we run the transformed programs with other different input files. We show the results in Table 10. GNU Go has no input files, and we change the parameter from 6-step to 9-step. For each of the other programs, we arbitrarily collect an input file either from the Internet or from another benchmark suite such as MiBench [8]. We list the sources of input files in the second column of Table 10. For $G721$, we choose the input file $small.pcm$ from the MiBench program $ADPCM$. We select the $tens\_015.m2v$ file, for playing table tennis, from Tektronix web-site, and we 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 10 shows the effectiveness of our scheme. After applying our scheme to these programs based on the profiling information with the default input files, substantial performance improvement is also achieved for the other input files.

4. Related Work

Since Michie introduced the concept of memoization [13], the idea of computation reuse had been used mainly in the context of declarative languages until the early 90's. In the past decade, researchers have designed various hardware mechanisms to reuse the intermediate computation results of previously executed instructions [4, 5, 6, 9, 10, 15, 17]. Richardson applies computation reuse to two applications by recording the previous computation results in a *result cache* [15]. However, he does not specify how to identify candidates for reuse, and the result cache in his paper is a special hardware cache. Sodani and Sohi [17] propose an instruction reuse method. The performance improvement of instruction level reuse is not shown to be significant, due to the small reuse granularity [18]. In the block and sub-block reuse schemes [9, 10], 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 [6] which combines software and hardware for reusing the intermediate computation results of code regions. Their scheme identifies the candidate code segments by value profiling for each instruction. In contrast, our scheme uses the compiler to select code segments based on *value-set* profiling. Under their scheme, the computation results of the reusable code regions are recorded into hardware buffers during the execution 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 no more than 16 entries for each code segment and limit the input/output register array to 8 entries. In our compiler scheme, there is no such limitation. Since accessing software hash tables are more time consuming than hardware buffers, we perform careful cost-benefit analysis to select reuse candidates. Such analysis is not done in previous work.

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

Table 10. Performance Improvement for Different Input Files (Under O3 optimization)

5. Conclusions

In this paper, we have presented a compiler scheme to reuse intermediate computation results. Our scheme uses profiling techniques to collect information on execution frequencies and value-set repetitions of important code segments. Based on a cost analysis, our scheme chooses code segments to transform for computation reuse. Our preliminary experimental results show that, for several Mediabench programs and the GNU Go game, the compiler scheme can result in significant performance improvement and energy-saving on a handheld computing device. The proposed transformations are best carried out by a compiler or a programming tool, because it is quite unnatural to write the program in the style of the transformed codes. Our scheme can be made more sophisticated in various ways. Most important of all, a candidate code segment can be a part of a loop body, a function body, or an IF branch, instead of the entire body. How to identify the most cost-effective part remains our future work.

6. Acknowledgments

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

References

[1] GNU Go. *http://www.gnu.org/software/gnugo/gnugo.html*.

[2] Intel StrongARM SA-1110 Microprocessor Developer's Manual. October 2001.

[3] B. Calder, P. Feller, and A. Eustace. Value profiling. *Proc. of the 30th Int. Symp. on Microarchitecture*, pages 259–269, December 1997.

[4] D. Citron and D. Feitelson. Hardware memoization of mathematical and trigonometric functions. *Technical Report, Hebrew University of Jerusalem*, March 2000.

[5] 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.

[6] 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.

[7] M. Das. Unification-based pointer analysis with directional assignments. *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 35–46, June 2000.

[8] 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.

[9] 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.

[10] J. Huang and D. Lilja. Balancing reuse opportunities and performance gains with sub-block value reuse. *Technical Report, University of Minnesota*, February 2002.

[11] B. Jenkins. A hash function for hash table lookup. *Dr. Dobb's Journal*, September 1997.

[12] 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.

[13] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.

[14] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5), May 1994.

[15] S. Richardson. Exploiting trivial and redundant computation. *Proc. of the 11th Symp. on Computer Arithmetic*, pages 220–227, July 1993.

[16] S. Sastry, R. Bodik, and J. Smith. Characterizing coarse-grained reuse of computation. *3rd ACM Workshop on Feedback Directed and Dynamic Optimization*, December 2000.

[17] A. Sodani and G. Sohi. Dynamic instruction reuse. *Proc. of the 24th Int. Symp. on Computer Architecture*, pages 194–205, June 1997.

[18] 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.