A Feasibility Study for Methods of Effective Memoization Optimization

Daniel Mock

October 2018

Abstract

Traditionally, memoization is a compiler optimization that is applied to regions of code with few scalar inputs and outputs as well as no side effects. Each time a region of code is executed, the inputs and outputs are stored and if the inputs for a current invocation match a previous set of inputs, then the computation can be avoided be loading the saved output. In order to expand the opportunities for memoization, this project describes new methods to be able to cost effectively apply memoization to regions of code containing significant redundant computation. The regions of code we look for are the bodies of loops, as we need to eliminate enough redundant computation to offset the overhead associated with memoization. Composite structures, such as arrays, which are not easily memoizable due to their size and the time it takes to verify that their values have not changed, have been avoided when performing memoization in the past. We, however, propose a new method to quickly verify if composite structures such as arrays have been modified so that we can support memoizing regions of code that contain them. We present our findings on whether such a method for memoization is worthwhile.

1 Introduction

Memoization is a profiling technique which uses run-time behavior and a set of values to determine whether a set of operations are redundant and thus can be eliminated and be replaced by the old values previously computed. Thus, for a certain class of programs it can potentially provide speed-ups whose dependence is only on specific input values to a variety of code regions.

There are, of course, limitations. The largest limitation of memoization is overhead. When you enter a region of code that is a target of memoization, you have to check if any of the inputs differ from a previous time the code was executed. This is to guarantee that the outputs of the code region remain the same. The problem is that you have to store and check many values for this to work. This overhead grows prohibitively large as the number of input values increases. More input values must to be stored and then later checked. As the number of input values for a code region increases, the opposite applies for its potential to be memoized. More inputs are prone to having at least one value changing between different passes through the code region. Another crippling limitation is storing the data values and accessing the data values. If there are code regions that access large composite data structures, the entire composite structure must be stored to check that nothing changed. This alone makes memoizing most regions of code which reference such structures infeasible. The problem we try to address is how to memoize regions of code without increasing the run time of a program due to the amount of overhead involved in determining whether regions of code are memoizable during execution.

Traditional memoization techniques are quite limited in which code regions are supported for memoization. Historically, most memoization techniques just look for pure functions as memoization targets. These pure functions can't have many input values and they cannot access or change non-local data; i.e. a mathematical log function. This, however, is too limiting.

Most of the execution of a program is spent inside loop bodies, so only focusing on a subset of a program's functions (if at all) is not making use of the available potential for memoization. We attempt to discover this potential.

In this project, we look for potential memoization opportunities not just in functions, but in loop bodies. In order to avoid costly overhead from limiting potential memoizable code regions, we propose a new method for memoization which makes looking up changes to composite structures fast. We then collect simulation statistics to see if the cases we wish to optimize are even there to optimize. We present our findings of this exploration.

```
while (r != 0)
{
    t = s0 + r;
    if (t <= ix0)
    {
        s0   = t + r;
        ix0 -= t;
        q   += r;
    }
    ix0 += ix0 + ((ix1 & sign) >> 31);
    ix1 += ix1;
    r >>= 1;
}
```

```
if (check_inputs(r, s0, ix0, ix1, sign, loop_id
    )
{
    update_inputs(&s0, &ix0, &ix1, loop_id);
}
else
{
    while (r != 0)
    {
        t = s0 + r;
        if (t <= ix0)
        {
            s0   = t + r;
            ix0 -= t;
            q   += r;
        }
        ix0 += ix0 + ((ix1 & sign) >> 31);
        ix1 += ix1;
        r >>= 1;
    }
    save_inputs(s0, ix0, ix1, loop_id);
}
```

Figure 1: Loop before and after memoization transformation.

As an example, take a look at the loop in Figure 1. If we apply the suggested memoization technique that this paper presents to this loop, we can eliminate up to 5.7% of the total cycles (86,836,590 down to 81,835,019) executed of the **basicmath** benchmark.

2 Background

Memoization was first introduced as a concept in the late sixties. After which, people started to do research into this unexplored field of computation. One of those people, Richardson [2], presents a memoization technique that identifies code regions that have no side effects and have just one output. These code regions can be as simple as large functions, or even arbitrary sized chunks of instructions. When the output is calculated, he describes a way to get the value stored into a hardware cache with almost no overhead. Of course, determining which regions might have memoization potential is done with profiling.

There is a similar memoization technique to the one we present in this paper developed by Ding and Li of Purdue [1]. The main difference being that the memoization only occurs in the software and not the hardware. First, they look for code segments which are good candidates for memoization. They then perform data flow analysis to determine the inputs and outputs to the code region. The set values of the inputs to the candidate code region is then hashed into a hash table associated with the code region and the outputs are stored. The cost to perform the hash is estimated using an algorithm. This algorithm factors in how often the input values are the same, the computation granularity of the code region, and the hashing function complexity. If the hashing cost outweighs how often they think the code region can be memoized, then the code region is not memoized. Otherwise, the code segment is transformed to support their memoization technique.

3 Method for Detecting Updates to Composite Data Structures

Unlike the previous methods to employ memoization, which require lots of software overhead or special dedicated caches requiring lots of energy, our method relies only on a slight modification of the DTLB (and possibly page table) design, which can be queried and set by hardware as well as by instructions provided by the ISA. It all hinges on the existence of a hardware performance counter (HPC) that is incremented each time a store occurs. This will be used as a timestamp and is crucial to our proposed method. Data structures come in different sizes. Some can span many pages while some can be contained within a single page. Our solution for keeping track of regions that can be updated is to logically divide each page into sub-pages such that the number of sub-pages is a power of two, which represents an architectural design parameter whose best size will be experimentally selected. This ensures that we can track segments of large structures while also minimizing wasted space on tracking small ones. Many possible designs can accomplish this desired functionality. One possible design employs a unique timestamp for each sub-page and a unified timestamp for the entire data structure. We can obtain information about when any segment of a data structure has been last updated as well as when a particular sub-page of such a data structure has been updated. This is done by storing the data structure timestamps in a vector referred to as the *lastupdate_structures* array and coupling the sub-page timestamps with each DTLB entry.

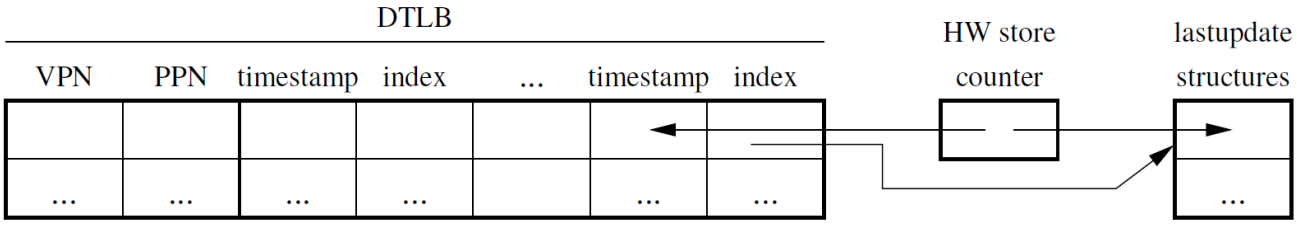

Figure 2: Detecting Updates to Structures

We propose to use the hardware structures in Figure 2 to detect when a data structure has last been modified. Each element of the *lastupdate_structures* array contains a HPC value which we use as a timestamp. The page table and the DTLB additionally include two fields for each sub-page: a timestamp and an index specifiying an element in the *lastupdate_structures* array. The high-order bits of the page offset are used to select the sub-page, thus the appropriate pair of timestamp and index values. Every store that occurs increments the HPC and the value of the HPC is assigned to both the timestamp field associated with the corresponding sub-page within the page and the element within the *lastupdate_structures* array by using the index field associated with the sub-page.

Figure 3 shows the set of proposed instructions used to check for updates to composite structures. *num_stores*, the

```
num_stores rd
```
(a) Get Current Store Count Value

```
get_index rd,rs
```
(b) Get Index Value

```
store_index rs,rt
```
(c) Store Index Value

```
last_update rd,rs,immed
```
(d) Get Last Update Count Value

```
chk_updates rd,rs,rt
```
(e) Checks for Updates to Structures

Figure 3: Proposed Instructions to Check for Updates to Composite Structures

instruction in Figure 3(a), is used to assign the value of the hardware store counter to the rd register. This is so that the total number of stores to a region of code after it is exited is recorded. The compiler compares this value to specified elements within the *lastupdate_structures* array that are associated with composite data structures accessed in the code region. In the case where the *lastupdate_structures* index value associated with a composite structure may not be known at compile time, then the instruction *get_index* in Figure 3(b) can be used to get the value of the index field. For dynamically allocated composite structures, the index field must be assigned at run time. The instruction *store_index* in Figure 3(c) performs this task. It's operands are an address and an index. The index of the sub-page in the DTLB entry associated with the address is updated. Figure 3(d) depicts the *last_update* instruction, which updates the timestamp of the sub-page or the *lastupdate_structures* array element with the provided value. The last instruction *chk_updates* in Figure 3(e) inspects the *lastupdate_structures* array to check if any composite structures were updated since the last time a region of code was executed. Once we identify target regions for memoization after the first phase of the compilation (after we have collected profiling data), we use these instructions to access the information needed to perform a check for redundancy. An example of what transforming a code region to support memoization looks like after the profiling phase is seen in Figure 4. We see that it is possible to exploit

```
sum = 0;
for (i = 0; i < N; i++)
   sum += a[i];
```
(a) Original Loop

```
static unsigned long long
   last_loop = 0;
static int last_sum;
...
if (last_loop >= last_update(A))
   sum = last_sum;
else {
   sum = 0;
   for (i = 0; i < N; i++)
      sum += a[i];
   last_sum = sum;
   last_loop = num_stores();
}
```
(b) Memoizing the Entire Loop

```
else if (last_update(A) > last_loop) {
   max = N/ELEMS_UNIT + N%ELEMS_UNIT != 0 ? 1 : 0;
   for (sum = i = j = 0; j < max; j++)
      if (last_loop >= last_update_unit(A, j)) {
         sum += sum_unit_a[j];
         i += ELEMS_UNIT;
      }
      else {
         for (t = 0; i < (j+1)*ELEMS_UNIT; i++)
            t += a[i];
         sum += sum_unit_a[j] = t;
      }
   last_sum = sum;
   last_loop = num_stores();
}
```
(c) Memoizing Portions of the Loop

Figure 4: Memoizing Iterations of a Loop

the potential for memoization without having to keep track of every data value. We just keep track of the last time used structures are updated.

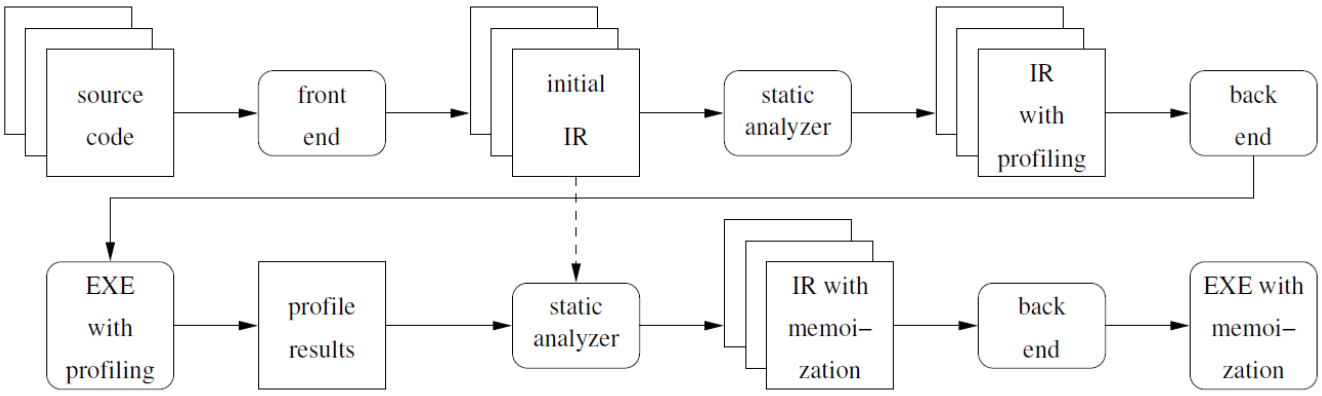

Figure 5: Proposed Memoization Process

4 Environment for Experimentation

Figure 5 shows the process we plan to use for memoization. During the first stage of the compilation process, we add the necessary profiling information to every loop and then generate an executable that will obtain the profiling results. With these results, we determine which loops are viable for memoization and transform those loops in the initial intermediate representation to produce code with the memoizable instructions added. This is input into the compiler back end to produce the executable that can eliminate redundant computation using memoization.

The compilation system we chose to use is **zephyr**. This is because the backend of the compiler uses **VPO** (Very Portable Optimizer), and we are familiar with how to use it. In order to identify and track potential memoizable code regions, we added a new instruction to the isa; *loop_entrance* which uniquely identifies when a given loop is entered. It's purpose is to track when we enter and exit a given loop. During the generation of the intermediate representation, after loops have been identified, we add a preheader to every loop (if not already generated) and insert the loop entrance instruction at the end of the preheader. Since a loop only has one preheader, and it must be entered before we enter the loop body, this instruction is guaranteed to execute before we begin execution of the loop. This signals to the simulator to keep track of all the loads and stores that occur during the loop's lifetime. Additionally, all exit blocks of a loop are marked with an instruction, *loop_exit*, that uniquely corresponds to the loop with which was entered. This enables us to know if any of the scalar values or composite structures a loop accesses are updated after the loop finishes execution.

The simulator we chose to use is **FAST**. This is mainly due to the fact that Dr. Whalley is partnered with Dr. Soner, who created FAST, and therefore we are very familiar with how to use and modify it. It simulates an in-order MIPS pipeline.

In order to track the necessary details to detect if memoization is possible for each loop nest, we modified the simulator in the following ways. When the simulator is run with a given benchmark, we choose to make it output the assembly file along with the memory contents. This file is parsed to determine the address ranges of the program as well as identify all global variables and their locations in addition to all functions and their address ranges. During simulation, when we encounter a *loop_entrance* instruction, we add the loop to the nest of currently executing loops. When a load or store occurs, the address of the memory location accessed is used to find the composite structure or scalar value being modified. The respective datum is modified and the cycle in which it was modified is saved with it. When we encounter the *loop_exit* instruction, we determine to which loop it belongs to and pop the loop off the loop nest. If we ever encounter the associated loop again, we check to see if any of the composite structures it accessed during a previous iteration were updated outside of the loop. If so, we choose to mark said iteration as not memoizable. If not, then if the scalar inputs are the same, we know that all the inputs

to the loop have not changed since some previous iteration, and that we mark said iteration as memoizable. If the set of scalar inputs are the same as some previous iteration, but only a subset of the composite structures are the same, then we mark the iteration as partially memoizable. It should also be noted that any loop iterations with side effects such as system calls are automatically not memoizable.

5 Results

We see from Figure 6 that the potential for memoization using the proposed method almost doesn't exist within the MiBench benchmark suite. Each benchmark has several bars of information, where each bar denotes the percentage of cycles of the overall program that are memoizable. Specifically, the percentage of cycles saved is based on how many previous entries to a loop are remembered that we can examine when determining if the scalars are the same as some previous entry. Note that regardless of how many previous loop entries we look at, if a page associated with a composite structure was updated (that some loop in question references), we get no benefit from looking at any of the entries. Only in the **basicmath** benchmark was there any memoization that was worth noting. However, all the memoization opportunities from **basicmath** came from the loop listed in Figure 1 or ones similar to it. Most of the loops did not reference any composite structures and each loop updated the values of a set of input scalars. Unfortunately, many of the loops in the other benchmarks either did not have these same properties, or they stored to arrays of data on the stack or heap thus invalidating them for memoization.

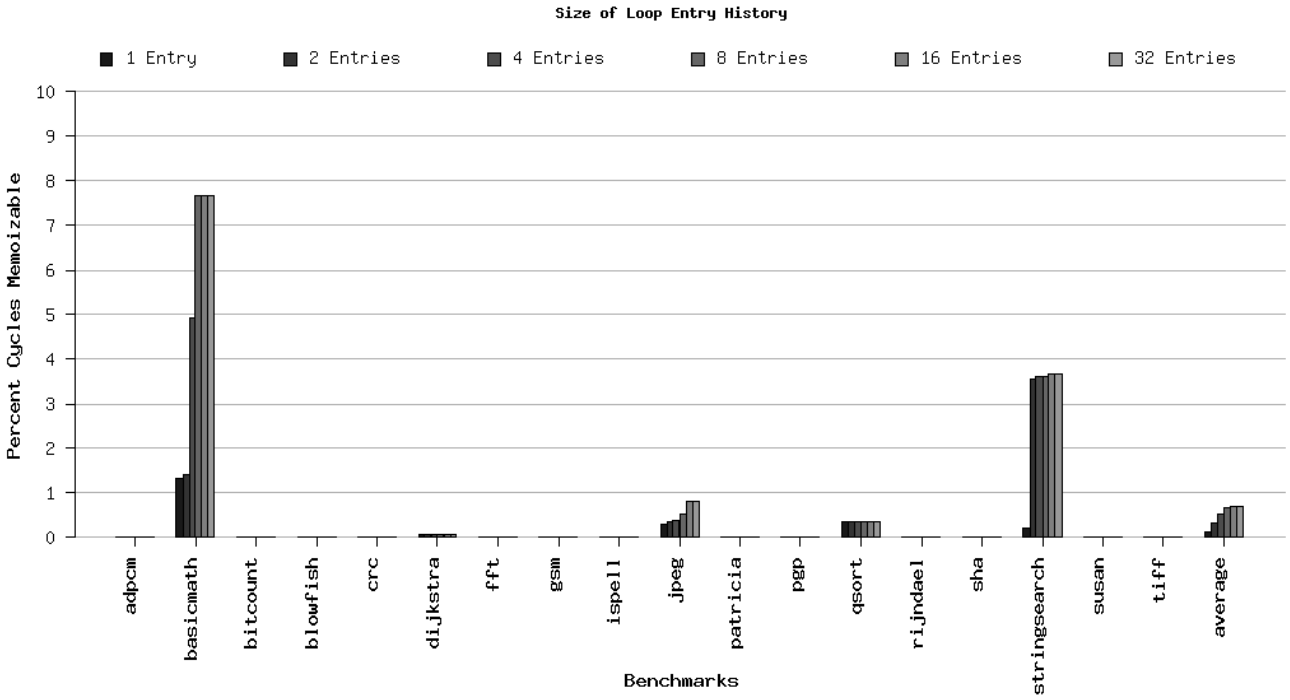

Figure 6: MEMO potential of MiBench benchmarks.

In **stringsearch**, we do see some success using our method. In this benchmark there is an array of strings that must be searched many times using a corresponding array of search strings. The length of a given string to be matched against must be calculated, and they do that by passing a pointer to the beginning of said string. By making the sub-page size small enough, the string arrays and the scalars used in the loop can be separated into distinct sub-pages. This allows the page containing the input scalars to be modified, but the page containing the string arrays that are being read from to remain

unmodified.

There were no loops that supported memoizing just a portion of their computations. It was either the entire loop is memoized or it's not.

6 Conclusions

There are almost no opportunities to exploit the proposed method for effective memoization optimization within the MiBench benchmark suite. However, there could be loops that are eligible for memoization, but that get disqualified because the composite structures they read from are too close to the input scalars whose values change over the course of the loop's execution. It might be worth seeing how much more potential memoization we can achieve if we align composite structures (regardless of if they are located in global memory or on the stack) on sub-page boundaries.

References

[1] Yonghua Ding, Zhiyuan Li *A Compiler Scheme for Reusing Intermediate Computation Results* Proceedings of the International Symposium on Code Generation and Optimization, 2004

[2] Stephen E. Richardson *Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation* 1992