A Study into Different Caching Strategies for Value Reuse Optimization

Alex Chen, Sahil Farishta, Ingab Kang, Joshua Segal

December 16, 2020

1 Abstract

Programs often redundantly execute the same functions with the same inputs in a deterministic fashion which wastes resources to recompute previously computed values. To solve this, programmers explicitly create caches for functions to avoid redundant computation. This technique is most popular in dynamic programming, but these problems are usually relatively small compared to the entire program. Caching becomes increasingly difficult to code and maintain as the code base scales in size. In this paper, we explore the performance effects of function value reuse. We automate the function caching functionality by writing a compiler optimization pass in LLVM combined with an external C++ caching library we created. Using LLVM, we perform a careful analysis of each function to make sure it is suitable for caching before running our optimization on it. We also investigate and implement two different caching algorithms: an LRU vector based cache and an LRU hashmap based cache. Both show significant improvement over the unoptimized versions, especially when computing recursive or repeated calls to math library functions. In our most significant benchmark, our optimizations show 10,000 times speed up over the unoptimized code.

2 Introduction

Most value reuse compiler optimizations today use compile time data flow analysis to do constant folding, global constant propagation, common sub expression elimination, etc. However, all of these optimizations are based on control flow graphs and not run time values. This means that there are still many redundant computations that get overlooked by the current compiler optimization passes. We explore different techniques to optimize code using run time optimizations by inserting small pieces of code throughout the program.

Additionally, most research has stayed away from trying to eliminate common function calls altogether because it is difficult to analyze the function's side effects. Side effects can include I/O functions, external device calls, reading or writing global variables, etc. In order to make a generalized compiler pass, we have to be conservative with what we cache. By only caching function values for functions that we know are cacheable, we can create a robust compiler optimization pass.

Caching function values is a common programming paradigm that many programmers use to significantly reduce the number of function calls. Caching recursive functions where function calls build off of each other is known as dynamic programming. This programming paradigm is used frequently to improve the performance of recursive functions. However, creating and maintaining a useful cache for an entire program becomes very difficult. This paper explores ways to automate this process to bring the massive performance benefits of caching to a program.

In this paper, we examine the significance of value reuse with functions using two methods. The first method uses a vector based LRU and the second method uses a hashmap based LRU method.

Having a compiler pass that can automate the creation and maintenance of a cache is usually more desirable than having programmers reprogram large parts of legacy code. Our research can significantly help improve this part of programming, so that legacy code can be optimized with little effort and the burden to create good program wide caches are lifted off of programmers.

3 Background

Here we discuss the various concepts that our solution uses. We look specifically at value reuse optimizations that can be performed using a cache based on the technique proposed by K. V. Seshu Kumar [1]. Additionally, we look at LLVM as a tool to create a compiler pass which will allow us to insert the code for our optimization at compile time.

3.1 Value Reuse Optimization

Value reuse refers to the elimination of redundant function calls [1]. A function call is redundant if it is a call to a pure function that has been called before with the same inputs. For a function to be pure, it must meet the following two requirements:

1. The function must return the same value each time it is called with the same input arguments.
2. The function must not have any effects outside of its stack space, including but not limited to using global variables and performing I/O operations.

If a function is pure, values that it returns can be stored in a cache that maps input arguments to return values as the program runs. On redundant calls to that function, the return value can be retrieved from the cache instead of computed again by executing the function.

Listing 1: Example of code that can be optimized by value reuse

```
1  int g = 5;
2
3  int addG(int a)
4  {
5      return a + g;
6  }
7
8  int main()
9  {
10     int a = 42;
11     ...
12     int w = sqrt(a);
13     int x = sqrt(a);
14
15     a = 52;
16     ...
17     int y = sqrt(a);
18     int z = addG(a);
19     printf("Can't cache this!");
20 }
```

Consider the C++ code in Listing 1. The call to sqrt on line 13 is redundant because sqrt is a pure function, so it is guaranteed to return the same value as the call to sqrt on line 12. If the value on line 12 was cached, then line 13 can simply retrieve that value instead of calling sqrt again. However, the call to sqrt on line 17 is not redundant because it has never been called with the same input before. Additionally, the call to addG on line 18 cannot be cached because addG uses the global variable g and the call to printf on line 19 cannot be cached because it performs an I/O operation.

3.2 LLVM

LLVM is an open source compiler for C and C++ programs [2]. It allows users to write passes that perform custom optimizations.

The key part of LLVM that is used in our implementation is the ModulePass, which gives access to all of the functions, basic blocks, and instructions in a program. By overloading this pass, we are able to determine which functions we can create caches for and insert instructions to instrument the caching logic as described later in Section 5.

4 Motivation

Value reuse optimization can have significant benefits for programs that have large numbers of calls to pure functions with the same input. For example, consider the code in Listing 2. Without value reuse optimization, this code would call the sqrt function 1000 times. However, with value reuse optimization, sqrt would only be called twice (once each for input arguments 10 and 20) and all subsequent calls would retrieve those values from cache. Since the sqrt function has $O(\sqrt{n})$ complexity and cache retrieval can have as low as $O(1)$ complexity, this would result in significant time savings. These savings only grow as the complexity of the function increases.

Listing 2: Example of code that benefits significantly from value reuse optimization

```
1  // Array with patterned data
2  int a[1000] = {10, 20, 20, 20, 10,
3      10, 20, 10, ...}
4
5  for(int i = 0; i < 1000; i++)
6  {
7      b[i] = sqrt(a[i]);
8  }
```

The same effect can be achieved with dynamic programming, but then it must be deliberately implemented by the programmer and only covers the functions that the programmer implements it for. However, a compiler could automatically insert code to perform value reuse on all pure functions in a program without any extra work done by the programmer. Thus, a compiler implementation of value reuse optimization is desirable.

5 Implementation

Our implementation of value reuse optimization is divided into two parts: an LLVM pass and a Function Cache Manager (FCM) written in C++. The LLVM pass runs through the target code and replaces any calls to a cacheable function with a function call to the FCM. The FCM manages the cache for each function, calling the function when there is no value cached for the arguments passed in, and evicting when the cache is full.

5.1 LLVM pass

The LLVM pass searches through the IR to find calls to cacheable functions and replaces them with a call to the FCM. To determine whether or not a function can be cached, the pass checks that each function meets two criteria: it must not use global variables, and it must not have any I/O operations such as printing or writing to a file.

The LLVM pass does this by iterating over each function in the module that is user defined. This is implemented by checking the namespace that the function belongs to, which prevents our code from optimizing calls to standard library functions. Optimizing these functions may lead to faster speed ups, but would require further checks to ensure that they are optimizable. The pass checks if each user defined function references global memory or performs an I/O operation in any of its instructions. If a function does either of these, then it is declared uncacheable. If a function calls another function, the pass checks to see if that function is cacheable or not, and if it isn't, then the caller function is declared uncacheable as well. If it hasn't been determined whether the called function is cacheable yet, the pass is run on that function before proceeding. Recursive function calls are ignored in this analysis, since if a function calls itself and is otherwise cacheable, then it is cacheable. If a function is found to be uncacheable, it is added to the set of uncacheable functions. Otherwise, it is added to the set of cacheable functions.

The pass then iterates through the calls to the set of cacheable functions. Each call is substituted with a call to the Function Cache Manager's (FCM) getCachedValue method. The FCM getCachedValue method takes in a function pointer to the function being cached, along with the original parameters. The return value of getCachedValue is then substituted everywhere the value from the original function call is used. The remainder of the work is completed by the FCM.

5.2 Function Cache Manager

The FCM code is written in C++ and is designed to manage the cache for each function. We began by implementing the method proposed by K.V. Seshu Kumar [1]. We then improved on the design by optimizing the cache search and eviction strategies. Each of our strategies works by creating a hashmap that maps function pointers to their respective caches. These caches contain a collection of structs which marks the arguments passed in and the corresponding return value. The FCM's main function is called getCachedValue, which is called directly from the program after the LLVM pass has been run. It takes the function pointer for the cached function and its parameters as arguments. The FCM then loads the corresponding cache collection for the function passed in, creating an empty cache for the function if this is the first time the function has been called. If the arguments passed in are in the cache, then the FCM simply returns the cached value. If the arguments passed in are not in the cache, then the FCM calls the cacheable function using the function pointer and arguments passed in. The FCM then adds the returned value to the cache along with the corresponding arguments, performing eviction if necessary, before returning the calculated value.

Each of our FCM implementations share this logic with different optimizations. We discuss them in greater detail below.

5.2.1 K. V. Seshu Kumar Baseline

The implementation described in Kumar's paper [1] was used as a baseline. It uses a vector to store the cached values and uses a round robin eviction strategy where each cache index is successively replaced. In his paper, Kumar simply performed the caching by hand; here we perform the caching dynamically using our LLVM pass. Kumar was able to decrease the number of function calls by up to 99% on various test cases with this method.

5.2.2 Least Recently Used Eviction

We use a Least Recently Used (LRU) eviction strategy to improve the performance of the implementation discussed in [1]. This approach adds an LRU bit to the cached element struct. The FCM keeps track of an LRU decrementer which starts off as the maximum value for an integer. Whenever an element is accessed or added to the cache, the LRU bit for that cache element is set to the value of the decrementer and then the decremeneter is decreased. When evicting, the cache element with the highest LRU value is replaced. We also perform a heuristic which begins searching the cache vector from the Most Recently Used element.

5.2.3 Least Recently Used with Hashing

We also implement an LRU eviction strategy that uses a hashmap to store the cached structs instead of a vector. This allows for faster lookup, insertion, and replacement times while adding memory overhead compared to our LRU implementation above. Additionally, there is added time overhead involved in creating the hashmaps and updating the size of each hashmap for each function.

6 Evaluation

6.1 Test Cases

We evaluate each of our implementations of value reuse optimization on the test cases shown in Table 1. The test cases can be divided into three different groups: Caching Advantageous, Caching Adversarial, and Not Cached. In all of our test cases, the function receives two integers as inputs and outputs an integer.

Group	Test Name
Caching Advantageous	Finonacci
	Iterative Fibonacci ST
	Iterative Factorial ST
	Pseudorandom Pow ST
	Towers of Hanoi
	LRU Biased
Caching Adversarial	Knapsack
	Matrix Chain Mult
Not Cached	Global Access
	Print Statement

Table 1: Test Cases

6.1.1 Caching Advantageous

The test cases belonging to this group are dynamic programs that benefit from caching function calls. The speedup that these tests show is best seen by analyzing the Fibonacci program. In a typical Fibonacci program, the n-th function calls on the (n-1)th and (n-2)th function, which then in turn calls on the (n-2)th, (n-3)th and (n-3)th, (n-4)th functions respectively. Therefore, if we cache the previous functions, we would be able to eliminate duplicate function calls.

Iterative Fibonacci Stress Test (ST) iteratively runs Fibonacci functions with differing n. Iterative Factorial ST computes the factorial of n, where n ranges from 0 to 1000. Pseudorandom Pow ST recursively computes the 400th power 40, 1600th power of 160, 6400th power of 640 on every 2nd, 3rd, and other iteration respectively. This test was created to demonstrate a test case where the same value would be reused, but in a random manner. Towers of Hanoi is a classic dynamic programming problem that we modified to accept the height of the tower being moved and the status of the three towers encoded as a single integer. Finally, LRU Biased is a test case that is designed to favor LRU over round-robin cache management by recursively calling the 0th function and the i-th function with i incremented on every iteration.

6.1.2 Caching Adversarial

These are cases where the program is less likely to call on previously cached inputs, but more likely to iterate through new inputs. In this case, FCM is not likely to offer any speedup as it is not likely to get a cache hit, so it would only incur the overhead of cache management. Knapsack computes the optimal selection of objects to store the most weight as possible within a specified limit. As a new input is used every iteration to explore a better selection, FCM is not likely to get cache hits. Matrix Chain Multiplication performs similarly as it computes the optimal sequence to compute a matrix multiplication and creates minimal reuse opportunities.

6.1.3 Not Cached

These tests were created to show that our LLVM pass successfully filters out uncacheable functions. The Global Access test accesses a global variable and Print Statement uses printf. Therefore, both cases should not be replaced with calls to FCM and should execute the same regardless of the optimization pass as they are not pure functions.

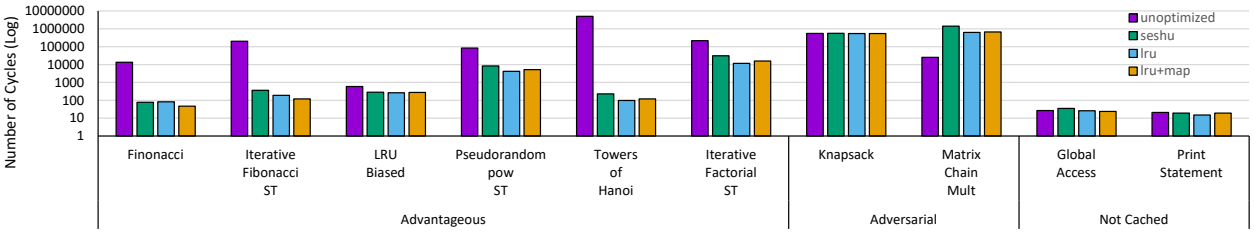

Figure 1: Experiment results showing the number of cycles at log scale needed to run each test case

6.2 System Configuration

Our tests were all conducted on the eecs583a server. As a limitation of using the course provided server, we were not able to monopolize the server for our test, which could introduce noise in our test results. Also, we were not given root access, so we were not able to disable turbo boost and fix the processor frequency, which could have been sources of noise in our results.

7 Analysis

Figure 1 shows the number of cycles needed to run each test case. As expected, the optimized code shows significant speedup on the Caching Advantageous tasks, slowdown on the Caching Adversarial tasks, and no speedup on the Not Cached tasks.

On the Advantageous tasks, LRU performed the best with an average of 8731 times speedup against unoptimized and LRU hashmap came in second with 7293 times speedup on average. The K. V. Seshu Kumar implementation was significantly slower than the others, with only 3771 times speedup on average. On the Adversarial tasks, all optimizations exhibited slowdowns, but Kumar's implementation performed the worst as LRU and LRU hashmap both employed searching optimizations, which reduced cache management overheads. For the Not Cached tasks, all configurations showed similar number of cycles (within ± 6 cycles of each other). We were also able to observe the same terminal outputs for Print Statement, confirming that the LLVM pass did not optimize for the Not Cached tasks.

Overall, if FCM was used, regardless of whether it was an Advantageous task or Adversarial task, our implementation of FCM outperformed Kumar's with an average of 2.3 and 1.9 times speedup for LRU and LRU hashmap respectively. This shows that our search algorithm and cache management improvements are good enough to manage the function cache efficiently.

8 Conclusion

In this paper, we demonstrate that it is possible to implement a caching strategy for value reuse optimizations at the compiler level. Using an LLVM pass, we can find functions that can be cached and implement various caching strategies to manage the caches for these functions to avoid redundant computations wherever possible. We have shown that a simple strategy which uses LRU as the eviction policy can improve performance by thousands of times on certain problems. These optimizations are akin to the optimizations seen through dynamic programming and require no extra work from an end user.

There is still work to be done in this area, however. Different caching algorithms may yield better performance on various problems, and not all functions benefit from this caching. Future work could involve developing a cost function which analyzes whether or not it is worth caching a certain function. Additionally, expanding this cost function using data profiling may allow us to determine the optimal cache eviction strategy for each individual function independently. Additionally, future work may perform optimizations more aggressively than we did, as we avoided functions that perform any sort of global or I/O accesses. However, these functions still may be cacheable with careful analysis. Future work may also extend these optimizations to run on any function, regardless of the number or type of inputs and outputs.

References

[1] K. Kumar, "Value reuse optimization: Reuse of evaluated math library function calls through compiler generated cache," *ACM SIGPLAN Notices*, vol. 38, 08 2003.

[2] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*. USA: IEEE Computer Society, 2004, p. 75.