Dynamic Reuse of Subroutine Results

Lior Shamir *

Department of Computer Science, Michigan Tech, 1400 Townsend Dr., Houghton, MI 49931, USA

Abstract

The paper discusses a concept of dynamic reuse of subroutine results. The described technique uses a hardware mechanism that caches the address of the called subroutine along with its arguments and returned value. When the same subroutine is called again using the same arguments, the returned value can be accurately predicted without an actual execution of the subroutine. Although this approach can be highly effective in some cases, it is limited to subroutines that do not use side effects, and use only *by-value* parameter passing. Since the proposed method requires that both the user and the compiler be aware of this mechanism, it might be more appropriate for specific computing-intensive applications, rather than standard all-purpose programming.

Key words: Subroutine reuse, Code optimization, Instruction reuse

1 Introduction

In computer systems, prediction of the future can be used in order to enhance performance [10, 13]. While at compile time only little can be learnt about the true behavior of the program execution, the future can be predicted at run-time based on previous events. This approach is commonly used to predict branches [1, 13] and values [10].

Many of the recent microprocessor architectures use value prediction [10, 3, 8] in order to increase parallelism. Since the variation of values being computed during the execution of a single program is relatively low [9], some of the values can be predicted at run-time before the actual execution of the computation. Value prediction techniques are used for reducing latencies caused by

* Corresponding author: Tel: (906) 487-2198 Fax: (906) 487-2933
Email address: lshamir@mtu.edu (Lior Shamir).

data dependencies [4], and branch prediction is used for reducing the latencies caused by control dependencies.

Sodani and Sohi [14] presented a technique for eliminating the execution of instructions by dynamically reusing previously executed instructions. However, information about previously executed instructions can be used not only for eliminating the execution of a single instruction, but also for eliminating the execution of some sequences of instructions. Kumar [7], presented a technique that uses compiler generated cache for reusing function calls, and showed that a substantial number of identical subroutine calls are executed during the execution of a computer program. In this paper we propose a hardware mechanism that eliminates subroutine calls by reusing the returned values of previous calls.

2 Dynamic Subroutine Reuse Overview

The notion of subroutines, in their various forms of functions, procedures or object methods, is widely used among software developers. A subroutine is a basic building block of a program that is often executed more than once during the program execution. A typical subroutine has a well-defined lean interface with its calling program, which is a list of arguments and a returned value. It is not seldom that subroutines are repeatedly executed with the exact same argument values, and their returned values are therefore identical. Since subroutines usually consume a relatively substantial amount of cycles, eliminating the execution of subroutines that have an already-known returned value can effectively increase performance. For instance, searching and analyzing genomic DNA sequences can be more efficient if values returned from previous computations can be used, without re-executing the computation [5]. Earth simulator [11] has also introduced the problem of intensive calls to the same subroutine [12]. Another example is massive execution of FFT transformations using computing-intensive machines such as Blue Gene [2, 6].

If the processor caches information regarding previous calls of a certain subroutine, it can compare the arguments of a given call to those of previous calls. If the arguments are identical, the processor can use the returned value computed by the previous identical call and continue without executing the subroutine. However, even a good and costless reuse of subroutine returned values cannot be applied to all subroutines. Subroutines that use side-effects must be executed even if the returned value is available from a previous call. A simple example is the subroutine *Sleep()*. Although the returned value of this subroutine is known, a programmer using this subroutine probably expects it to cause a delay, so its actual execution cannot be eliminated. However, many subroutines do not use side-effects, and the elimination of even a small

portion of these calls can lead to a considerable improvement in the overall performance. Consider the following simple code that transforms an RGB color image to an HSV color image.

```
for (y=0;y<image.height;y++)
  for (x=0;x<image.width;x++)
    {
      color=image[x][y].rgb;
      hsv=RGB2HSV(color.red,color.green,color.blue);
      image[x][y].hsv=hsv;
    }
```

In this case, most of the CPU cycles are consumed by the computationally expensive subroutine *RGB2HSV*. Since many of the pixels have the same *red*, *green* and *blue* values, a mechanism that reuses the returned value of the subroutine *RGB2HSV* can eliminate the execution of many of the subroutine calls and significantly increase the performance.

As mentioned earlier, returned values of subroutines cannot always be reused due to side-effects, by-reference parameter passing or I/O operations. Therefore, the proposed method requires that both the user and the compiler be aware of this mechanism. When the user is convinced that her subroutine has no side-effects and that its execution is safe to be eliminated at run-time, she can define the subroutine in a fashion that supports subroutine returned value reuse. For instance, a subroutine in C programming language might be as follows:

```
long _vpcall inc(long x)
{
  return(x++);
}
```

The *_vpcall* calling convention is an indication for the compiler that this subroutine is defined as safe for dynamic returned value reuse. The compiler should use this indication in order to generate a newly introduced *vpjsr* branch instruction (that will be discussed thoroughly in Section 3) rather than the regular *jsr* branch instruction in each call to that subroutine. The *vpjsr* instruction is an indication for the processor that the dynamic returned value reuse mechanism (that will also be described in Section 3) should be enabled. The requirement for compiler and user awareness of this mechanism is indeed a major downside of the proposed method. Since the proposed mechanism introduces some specific demands from the programmer, it might not be an optimal solution for standard all-purpose programming. However, for some specific computing-intensive tasks such as large-scale simulators, heavy multimedia processing and bioinformatics, using such an approach can substantially

improve the overall performance as explained in Section 4.

3 Dynamic Subroutine Reuse Implementation

3.1 Instruction Set

A traditional call to a subroutine is usually performed as follows: After the parameters are pushed into the stack, the *jsr* instruction is executed. The return address is also pushed into the stack, and the called subroutine address is copied to the PC. The proposed new branch instruction *vpjsr* is similar to the regular *jsr*, but performs several additional actions explained later in this paper. In order to exit from a subroutine call, the instruction *vpret* is introduced.
The *vpjsr* instruction is encoded using 3 fields:
1. Opcode
2. Subroutine address
3. Total size of subroutine arguments (in bytes)

While the purpose of the first two fields is trivial, the third field stores the total size (in bytes) of the subroutine arguments. For instance, a subroutine that has two arguments, each of the size of 4 bytes, will have a total size of arguments of 8 bytes. Similarly, the *vpret* instruction has an additional field that stores the size (also in bytes) of its returned value. Since the size of the arguments and returned values are known at compile time, compilers should find no difficulties in setting the right values to these fields.

3.2 Components

The implementation of the proposed method consists of two hardware components: the *Previous Subroutine Call Table* (PSCT), and the *Subroutine Argument Vector* (SAV). PSCT is a table that stores information regarding previous subroutine calls. Each entry in the table refers to one subroutine call and has 6 fields as described below:
1. *Address*: Stores the subroutine address. This field is used as an identifier of the subroutine.
2. *Arguments*: Stores the argument values used by the subroutine call.
3. *Psize*: The total size (in bytes) of the arguments.
4. *Rsize*: The total size (in bytes) of the returned value.
5. *Returned Value*: The value returned from the subroutine call.
6. *Valid*: Indicates whether the entry is used.

For instance, let S be a subroutine with two arguments: *P1*, which is of the type *byte*, and *P2*, which is of the type *unsigned short int*. Now assume that the subroutine S is called with the arguments *P1*=1 and *P2*=1025. The value of the field *Arguments* would be: 00000001000001000000001, where the leftmost 8 bits are the value of *P1* and the next 16 bits are the value of *P2*. The value of the field *Psize* is the total size of the arguments *P1* and *P2*, which is equal to 3. The value of *Rsize* is determined by the type of the returned value. For instance, if the returned value of S is *double*, then the value of *Rsize* should be 8, which is the size (in bytes) of a double precision variable.

The space allocated for the field *Arguments* is of a fixed size, while the actual size of the arguments can be different for each subroutine. Fortunately, most subroutines have a small number of arguments. For instance, most subroutines of the standard C *math* library have only one argument, and the average total size of arguments is just 9.73 bytes. Therefore, allocating a fixed number of 16 bytes as the size of the *Arguments* field should be enough for a wide range of subroutines. If a certain architecture requires a larger total size of arguments, the field *Arguments* can be set to a larger size with the sacrifice of increasing the space required for the implementation of the proposed mechanism. However, in any case, subroutines with a total size of arguments that exceeds the size of the field *Arguments* will not be able to use this mechanism and will be executed normally, without reusing their returned values. Since the total size of subroutine arguments is known at compile time, compilers can alert on any attempt of using the proposed method with an illegal total size of arguments.

The purpose of the SAV is to provide a fast access to the content of the top of the stack. The size of the SAV should be equal to the size of the *Arguments* field in the PSCT. The SAV is maintained by pushing and popping any value that is pushed or popped from the stack. I.e., when a word W is pushed into the stack, the SAV is left shifted and the value of W is copied to the right cells of the stack vector. When a word is popped from the stack, the SAV is right shifted.

3.3 Dynamic Subroutine Reuse Algorithm

The proposed *vpjsr* and *vpret* instructions are similar to the traditional *jsr* and *ret* instructions, but with several exceptions. The next paragraphs describe the simple algorithms performed when a subroutine is called (using *vpjsr*) and exits (using *vpret*).

1. Subroutine call: A subroutine call using the *vpjsr* instruction is basically similar to the execution of the regular *jsr* instruction. However, when *vpjsr* is executed, an associative lookup in the PSCT is performed in order to find a

previous call to the same subroutine with the same argument values. Such a previous call is found if there is a valid entry in the PSCT with an *address* field equal to the address of the called subroutine, and an *arguments* field equal to the SAV. Since the size of the *arguments* field is fixed, it might be greater than the total size of arguments of the subroutine. Therefore, the field *Psize* is used in order to ignore the irrelevant bits of the *arguments* field. For instance, if the value of *Psize* is equal to 4, only the first 4 bytes of the SAV are compared to the first 4 bytes of the *arguments* field of the PSCT entry.

Since the SAV is a reflection of the top of the stack, comparing the SAV to the field *arguments* of a PSCT entry actually compares the values at the top of the stack to the values that were at the top of the stack when the PSCT entry was created. If the values are identical, the arguments values that were used in that previous call are identical to the argument values that are about to be used. In this case, the subroutine is not executed and the returned value is simply taken from the *returned value* field of the PSCT entry.

If no such entry is found in the PSCT, a new entry is created. The value of the *address* field in this entry is the address of the subroutine, and the value of the SAV is copied to the *arguments* field. At this time, the value of the field *valid* is 0. Since the number of entries in the PSCT is finite, a certain replacement policy is required. Replacement policies such as FIFO and LRU can be applied for this task.

2. Return from subroutine: Upon return from the subroutine (using the *vpret* instruction), the returned value of the subroutine call is copied to the *returned value* field of the PSCT entry, and the *valid* field of that entry is set.

Since associative lookup is performed each time *vpjsr* is executed, it may be that the time required for completing the lookup is greater than the actual time needed for executing the subroutine. Therefore, programmers should avoid using the *__vpcall* calling convention for computationally inexpensive subroutine. The execution time of a subroutine can be changed at runtime, and is dependent on the values of the arguments, global variables, etc. Practically, in most cases programmers can estimate the computational cost of the average execution of a given subroutine. For instance, a short subroutine that does not include loops and does not call other subroutines is expected to be computationally inexpensive. Otherwise, execution time is expected to be shortened if the proposed mechanism is used.

4 Performance Evaluation

The proposed method was experimented by modifying the benchmark programs such that calls to subroutines with a total size of arguments less than or equal to 16 bytes were replaced with other subroutines doing the same thing. For instance, calls to the function *RGB2HSV(x)* were replaced by calls to the subroutine *my_RGB2HSV(x)* described below:

```
HSV my_RGB2HSV(RGB x)
{
  unsigned char parameters[SIZE_OF_PARAMETERS];
  unsigned char ret_val_buffer[SIZE_OF_RETURNED_VALUE];
  HSV returned_value;
  memcpy(parameters, &x, sizeof(x));
 if (psct.sub_call((void *)my_RGB2HSV,parameters, sizeof(x),sizeof(HSV),ret_val_buffer))
     return((HSV)ret_val_buffer);

  returned_value=RGB2HSV(x);

  memcpy(ret_val_buffer, &returned_value,sizeof(returned_value));
 psct.set_ret_value((void *)my_RGB2HSV,parameters, sizeof(HSV),ret_val_buffer);
  return(returned_value);
}
```

The *psct* object (mentioned in lines 7 and 11) is a software simulation of the PSCT component described in Section 3, so that the modification provides a software simulation of the mechanism and allows counting the number of subroutines that were executed or dynamically eliminated. The results of using a PSCT with 256 entries are described in Fig. 1. In this experiment, each entry in the PSCT has 16 bytes allocated for subroutine arguments and 8 bytes for the returned value, and the FIFO replacement policy is applied.

In order to maintain the correctness of programs, the experiment included only *math* library subroutines, which are known to be side-effect free. The modification performed a simulation of the PSCT, by comparing the argument values to previous calls.

The overall speedup when using the proposed method is provided in Fig. 2.

Although the results provided in Fig. 2 do not introduce a tremendous improvement, the proposed mechanism can be highly effective in some common tasks of processing many repetitive data, and in some cases the performance improvement can be dramatic. In order to demonstrate the efficacy of the proposed method, we examined the behavior of the trivial code described in

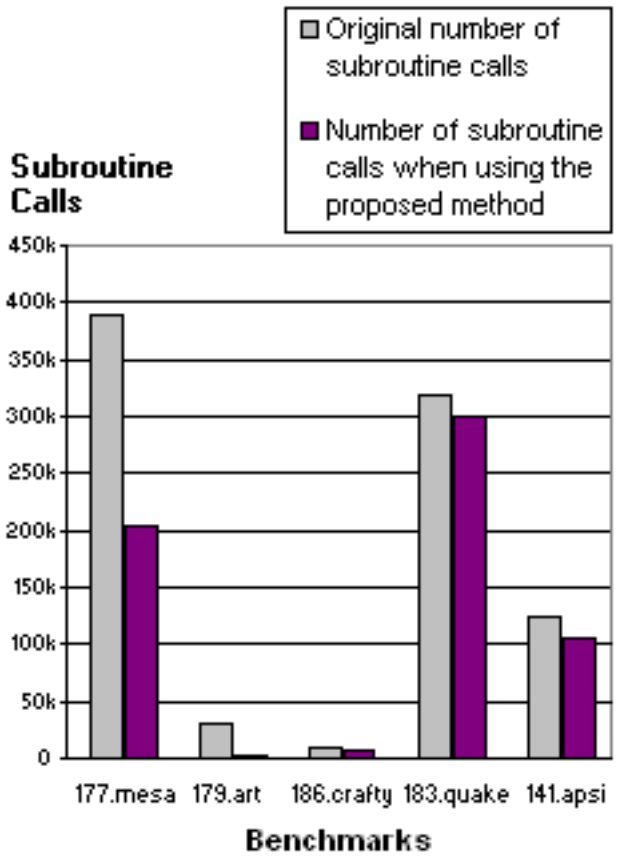

Fig. 1. Number of actual calls to math library subroutines with and without using the proposed method

Section 2 that transforms an image from RGB color space to HSV color space. Table 1 presents the results of running the code described above on a standard 525×320 true color image.

When using the same approach with a 256 color image, the increase in performance becomes extreme as described in Table 2. Since there are only 256 possible RGB vectors in the given image, there is no need to call *RGB2HSV* more than 256 times. Therefore, the proposed method allows this toy program to eliminate all but 256 calls to *RGB2HSV*, and the total number of calls is constant (at 256) and independent of the number of pixels. This approach can be used to enhance the performance of specific computing-intensive applications such as large-scale simulators, bioinformatics, image processing and multimedia, where repetitive calls to subroutines are not rare, and in many cases consume a substantial portion of the total CPU resources.

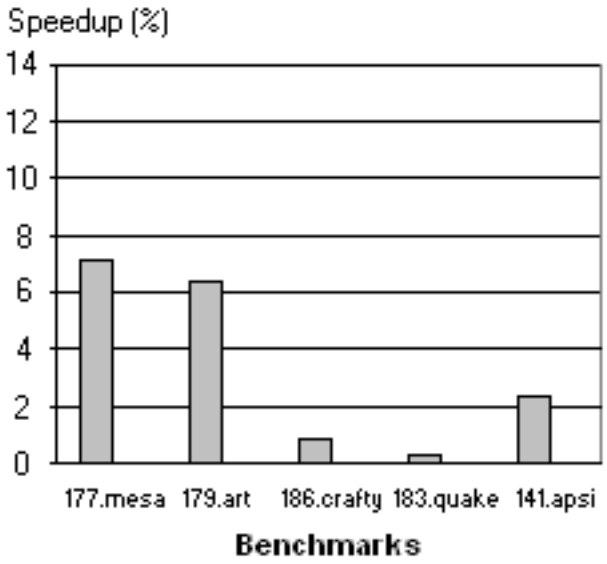

Fig. 2. Overall speedup

Table 1
Percentage of eliminated subroutine calls when converting a 525 × 320 true color image from RGB to HSV

PSCT entries	number of eliminated calls	number of executed calls	percentage of eliminated calls
2	69800	98200	41.5
3	76645	91355	45.6
4	80833	87167	48.1
10	91862	76138	54.6
20	98174	69286	58.4
30	103990	64010	61.8
50	109499	58501	65.1
100	115825	52175	69
256	131262	36738	78.1

5 Conclusion

Dynamic elimination of repetitive identical subroutine calls can improve performance. The improvement is dependent on the behavior of program at runtime, but also on the size of the PSCT and the entries replacement policy. More entries in the PSCT allow storing more previous subroutine calls, and therefore increase the efficacy of this mechanism. The downside of this technique is that compilers and users should be aware of this mechanism and support

Table 2
Percentage of eliminated subroutine calls when converting a 525 × 320 256-color image from RGB to HSV

PSCT entries	number of eliminated calls	number of executed calls	percentage of eliminated calls
2	87733	85267	49.2
3	95466	72534	56.8
4	102212	65788	60.8
10	120602	47398	71.7
20	132309	35691	78.7
30	138376	29264	82.3
50	145707	22293	86.7
100	158643	9357	94.4
256	167744	256	99.85

it. The presented method can be applied only to subroutines that do not use by-reference parameter passing, side-effects, or perform I/O operations. Improper use of the technique might undesirably eliminate subroutine calls and can lead to an unpredicted behavior of the computer program. Therefore, the proposed approach might be more appropriate for specific computing-intensive applications, rather than standard all-purpose programming.

References

[1] A.N. Eden and T. Mudge. The YAGS branch prediction scheme, Proc. MICRO–31 (Dallas, TX, 1998) 69–77.

[2] M. Eleftheriou, B. Fitch, A. Rayshubskiy, T.J. Christopher, R. Germain, Performance Measurements of the 3D FFT on the Blue Gene/L Supercomputer, Proc. Intl. EuroPar Conf. (Lisbon, 2005) 795–803.

[3] F. Gabbay. Speculative execution based on value prediction, *EE Department Technical Report 1080*, Technion – Israel Institute of Technology, Haifa, Israel, 1996.

[4] J.L. Hennessy and D.A. Patterson. Computer Architecture: A Quantitative Approach, Second Edition, (Morgan Kaufmann Publishers, 1996).

[5] X. Huang, L. Ye, H. Chou1, I. Yang and K. Chao. Efficient combination of multiple word models for improved sequence comparison, Bioinformatics 20(16) (2004) 2529–2533.

[6] IBM Journal of Research and Development 49(2/3) (2005) Special Issue on Blue Gene.

[7] K.V. Seshu Kumar. Value reuse optimization: reuse of evaluated math library function calls through compiler generated cache, ACM SIGPLAN Notices 38(8) (2003) 60–66.

[8] M.H. Lipasti and J.P. Shen, Exceeding the dataflow limit via value prediction, Proc. MICRO-29 (IEEE, Paris, 1996) 226237.

[9] M.H. Lipasti and J.P. Shen. The performance potential of value and dependence prediction, Proc. EUROPAR-97 (Springer, Passau, 1997) 1043–1052.

[10] M.H. Lipasti, C.B. Wikerson and J.P. Shen. Value locality and load value prediction, Proc. ASPLOS-7 (Cambridge, MA, 1996) 138–147.

[11] D. Normile. Earth Simulator' Puts Japan on the Cutting Edge, Science 295 (2002) 1631–1633.

[12] A. Sinya et al., Study of the Standard Model of Elementary Particles on the Lattice with the Earth Simulator, Annual Report of the Earth Simulator Center (2003) 175–179.

[13] J.E. Smith. A study of branch prediction strategies, Proc. 8th Annual Symp. on Computer Architecture (Minneapolis, MN, 1981) 135–148.

[14] 8. A. Sodani and G. S. Sohi. Dynamic instruction reuse, Proc. ISCA-25 (Barcelona, 1998) 194–205.