Pre-computing Function Results in Multi-Core and Many-Core Processors

Edward C. Herrmann, Prudhvi Janga, and Philip A. Wilsey
Experimental Computing Laboratory,
School of Electronic and Computing Systems,
PO Box 210030, Cincinnati, OH 45221–0030
herrmaec@mail.uc.edu, jangapi@mail.uc.edu, and philip.wilsey@uc.edu

***Abstract*—In recent years, the number of hardware supported threads in desktop processors has increased dramatically. All but the very lowest cost netbooks and embedded processors now have at least dual cores and soon systems supporting upwards of 8 to 16 hardware threads are likely to be commonplace. Unfortunately, it will be difficult to take full advantage of the parallelism emerging processors will be able to provide. To help address this issue, we are investigating mechanisms to pre-compute function results in separate threads running concurrently with the main program thread. The concurrent threads are forked automatically and without program modification. A critical component for the success of this idea is an ability to build a background thread that can pre-compute usable results in some effective manner. For some support functions (dynamic memory) exact arguments predictions for the function pre-computation are not necessary, for others (trigonometric functions) they are. In work with dynamic memory, we are able to pre-compute memory blocks and show modest speedup: saving approximately 25% of the dynamic memory costs. In studies with predicting argument values to trigonometric functions, we show that learning algorithms are able to successfully predict the next argument values approximately 44% of the time.**

I. Introduction

Recent trends have shown that parallel processing is emerging as the new frontier for the mass computing market [1], [2]. The point of diminishing returns has been reached in the field of instruction-level parallelism and power and heat concerns have further slowed additional processor performance gains [3], [4]. Hardware manufacturers have also joined the shift to more parallelism and a shift from multi-core to many-core is on the horizon. The road maps of all the major processor providers (Intel, AMD, Sun/Oracle, and IBM) clearly show this progression. The Intel i7 processor has hardware support for up to eight simultaneous threads, their new Xeon processor has 10 cores supporting up to 20 simultaneous threads. IBM's next generation Power7 product that supports up to 32 threads per chip [5]. Sun/Oracle already has single chip processors providing hardware support for up to 64 threads. Intel demonstrated a "single-chip cloud computer," which contains 48 fully functional x86 compatible cores [6]. Following these patterns, it is clear that desktop processors may soon contain hardware support providing capabilities for hundreds of simultaneously executing threads.

As the number of cores increases for desktop computers, it is important that software be able to take advantage of their additional parallel processing capabilities. The keys to successfully harnessing this power lies with new advances to better introduce parallelism into the practice of computer system and software development [2]. While approaches to improve our parallel programming capabilities are needed [1], we must also try to discover new techniques to harness this widely available parallelism. In addition, it is time to broaden our goals for parallelism away from strictly scalable computing; many-core processors provide untapped computational resources that we should strive to exploit to achieve any (even small) gains. While the goal for scalable parallelism is desirable, the possibility of gaining even modest speedup with the additional cores should be considered. Toward this end, we have initiated a series of studies to explore the development of techniques that easily and, ideally, transparently find additional speedup on many-core processors.

Our studies with many-core processors has focused on investigations to transparently fork separate background threads for pre-computing function results. For successful function pre-computation the background threads must be able to learn and then successfully react to the needs of the main program thread. For some functions

Support for this work was provided in part by the National Science Foundation under grant CNS–0915337 and by Sun Microsystems.

(*e.g.*, trigonometric functions), exact values for the function arguments must be predicted; in other cases (*e.g.*, dynamic memory) usable results can be achieved by indirectly monitoring the effects of the main program thread (in the case of dynamic memory, we observe the allocations of memory blocks of various sizes). Furthermore, it is also possible for the background threads to pre-compute and hold multiple return values with different expected arguments that can be matched and quickly returned to the main program thread when the actual argument is known.

In this paper, we present studies with function pre-computation. In particular, we examine function pre-computation for *dynamic memory* and then for *trigonometric functions*. Dynamic memory presents an opportunity where function results can be satisfied with imprecise argument predictions (as long as the value is equal or larger than the block size requested). In contrast, trigonometric functions require precise prediction of argument values. For dynamic memory, we use simple heuristics to monitor the typical sizes requested by the application program. For trigonometric functions, we study the use of a a generalized online learning algorithm to monitor and predict future argument values. With dynamic memory, we implement a full solution and show a reduction of 25% of the dynamic memory costs. Our studies with trigonometric functions are limited to studies of argument prediction. In particular, we captured argument histories from a couple of hours of general workstation operation and fed these histories into a learning algorithm. The learning algorithm then predicts the next value for the input argument. In the argument histories studied in this paper, the learning algorithms were able to achieve a prediction accuracy of 40-50%. Of course, the success of the prediction algorithms will be heavily dependent on a program's behavior; some programs will have regularity to (some or all of) their function arguments and benefit from threaded pre-computation others may not. Fortunately the approach we advocate is easily toggled on/off by the end user of an application program.

The remainder of this paper is organized as follows: Section II describes the constraints needed for a function to be candidate for pre-computation. Section III discusses some related work. Section IV presents our work with dynamic memory. Section V describes our experiments with argument prediction to trigonometric functions. Section VI describes the design of an generalized infrastructure to support the transparent pre-computation of function results. Finally, Section VII concludes our research and summarizes our results.

II. Functions Suitable for Pre-Computation

In order to be suitable for pre-computation in a background thread, a function must exhibit certain properties. In particular these properties must be met: (i) it must be side-effect free or any side-effects must not impact the correct processing of the main program thread, (ii) it must not rely on external program/machine state information to compute a correct result, and (iii) it must be functionally idempotent. For example, the `malloc` function will return a pointer to a memory container of acceptable size; *which container* may vary, but any suitably sized container is acceptable. The pre-computed `malloc` function can alter the free space lists of the operating systems, but this change does not impact the correct execution of the main processing thread. Any library function that can satisfy these requirements is a candidate for optimization through threaded pre-computation.

There are many factors that determine if threaded pre-computation will actually benefit an application. First the amount of speedup that can be gained will be limited by the amount of time a program spends inside the targeted function(s) (Amdhal's law). Application functions that do not contribute adequate computational costs to the total execution time will not see speedup as a result of the threaded pre-computation. Another factor is the amount of overhead involved in communication between the main program and the background thread. The main program argument values must be easily captured by the background thread (to facilitate learning and accurate prediction of future argument values). Similarly the background thread must be able to quickly transfer the pre-computed results back to the main program thread when the actual call occurs. The amount of time saved in pre-computing results must be greater than the communication overhead in order for speedup to occur.

Misprediction penalties will also affect performance. If the input to the next function call is mispredicted or (more precisely) if the set of pre-computed results do not cover the input argument, then the function output will have to be calculated in-line. Finally the timing of the calls in the program can also affect how effective function pre-computation will be. There must be sufficient time between successive function calls to allow the background thread to pre-compute the next result. If the function calls occur in more rapid succession, the pre-computed values may not be ready in time. This would cause the main program to wait for the output to be

calculated, reducing the amount of parallelism that can be extracted from the function.

III. Related Work

Although slightly related to speculative execution and value prediction in high performance out-of-order pipelined processors [4], threaded pre-computation is a parallelization technique that is more closely related to program futures software based value prediction [7]. Threaded pre-computation is also a technique that could be nicely matched with memoization for added performance enhancements. We review both memoization and futures below.

Memoization is a technique to capture previous results of a function call for reuse rather than recompute a later function invocation having the same input arguments [8]. Memoization requires that the function be referentially transparent and it works if the search for a previously saved result is lower than the runtime of the actual function invocation. Memoization could be an effective technique to combine with some uses of threaded pre-computation for even greater savings that either technique provides separately (specifically consider the empirical studies of Section V where several of the function argument traces are nearly always the same).

Another topic that is related to our work is the concept of computing futures. A future is defined as "a promise to deliver the value of a subexpression at some later time" [9], [10]. Certain parts of programs can be broken down into functional arguments, where futures can be used to evaluate argument expressions in parallel. Each future is assigned to an evaluator process to calculate its value. This allows each future to be calculated independently on separate processors. When the value of a future is needed, if the value is ready it is used immediately; otherwise, the process must block until the evaluation is complete. Futures use "eager evaluation" where once a value is anticipated to be needed, evaluation will start [9]. This approach introduces the possibility of wasteful calculation if features end up being unused. The initial implementation by Baker and Hewitt dealt with memory management by using futures in a garbage collection system [9].

Our threaded pre-computation approach is similar to futures in that we use eager evaluation by designating an extra processor to pre-compute function values before they are needed. However, futures are a programming feature that must be supported by the programming language and explicitly coded by the user into the application software. Our threaded pre-computation approach can be implemented by building a wrapper around library functions to thread and pre-compute function values without change to the application program (provided the program is dynamically linked and the dynamic link path can be modified to point first to the wrapper library).

The transparent threading of dynamic memory is explored and reported in [11], [12]. Both groups show improvement in application performance by transparent threading. In both projects, the idea is developed solely as a means to transparently deploy a parallel thread for dynamic memory management. In this paper we reposition these works as function pre-computation in order to generalize the concept for application with other functions. In particular we explore the online monitoring of the main program behaviors to support function pre-computation.

IV. Pre-Computing Dynamic Memory Operations

Our dynamic memory management library aims to offload, to a separate thread, many of the instructions required for the main program thread to allocate and deallocate memory blocks. We exploit the dynamic link step in contemporary Linux systems so that simply setting the Linux environment variable `LD_PRELOAD` points an application to the new threaded library before the standard system library. In this way, applications that are well served by our library can easily be configured to use it and others with insufficient dynamic memory remain pointed at the standard library (applications with no dynamic memory are not affected by either choice).

The background dynamic memory thread prepares for future memory requests by queuing up free memory blocks of predetermined sizes and making them available when requests for new allocations arrive. Free memory blocks are allocated in power of two sizes and stored into bins; new allocation requests use atomic moves to remove blocks from the bin with blocks large enough to service the input argument request size. Each bin maintains a counter to size the number of pre-allocated blocks in that bin for the background thread to maintain. This counter is managed by a heuristic algorithm that reacts to empty or near empty bins by increasing the pre-allocation block counter. Thus, for dynamic memory management, the pre-computation thread does not actually predict values of input arguments; instead it monitors the overall impact of the `malloc` argument values. The background thread is responsible for ensuring that

free blocks of memory are always available for future requests. The thread periodically refills any bins that are running low on free blocks by creating new free blocks and placing them into the bins. The background thread is also responsible for processing blocks recently freed by the program by either placing them back into the bins or returning them back to the operating system. In order to ensure no race conditions occur, the library uses a lock-free approach by utilizing atomic operations to ensure no two threads interfere with each other or the background thread.

As mentioned, each bin contains a counter variable that determines how many blocks can be stored in the bin queue. Since the frequency of memory allocation sizes is program dependent, each bin size starts off at zero and increases dynamically throughout the program based on the demand for blocks of that particular size. Whenever an allocation request arrives for a bin that is empty (called a bin miss), the bin size is increased by one. This prevents more allocation misses in the future and allows block sizes that are used more frequently to have larger buffers. The bin size grows until enough blocks are queued up so that bin misses no longer occur. This means that upon initial startup, bin misses will occur until the bins reach their optimal size. Since different programs exhibit different allocation patterns, this method of slowly increasing the bin sizes allows the bin queues to dynamically adapt to the request patterns of the program. Block sizes that are requested more frequently will have more blocks buffered in the bin queue, whereas block sizes that are used infrequently will have small queue sizes. In an effort to minimize the number of bin misses and simplify the complexity of the algorithm, bin sizes are not allowed to decrease. Each bin also contains a variable used to keep track of how many blocks are currently stored in the bin. This variable is used to quickly obtain the number of available blocks without having to traverse the entire linked list.

On bin refreshes, the manager thread cycles through all the bins and locates any that are under half full and refills them with new blocks. New blocks are allocated until the number of blocks stored equals half the bin size. The bins are only filled to half their capacity to allow room for blocks to be added from other sources (such as recently freed blocks). The manager thread refills bins in two cases. First, the manager thread is signaled by the program thread at regular intervals to refill the bins. This periodic refilling ensures that new blocks are constantly buffered so that future allocation requests can be satisfied. Secondly, in the case that a bin is empty when an allocation request arrives, the main thread signals the manager thread for an immediate refresh and then waits until a new block is available. After initial startup, this case should happen only rarely since the periodic refreshes will typically keep enough blocks buffered so that bins should never empty.

The `free` function works in a similar distributed fashion. When a free request arrives, the address is stored in a "free array" to be freed later by the manager thread. The manager thread periodically parses through the array, grabbing the addresses and freeing their corresponding blocks. This array acts as a circular queue, meaning addresses beyond the end of the array wrap around and are placed in the beginning slot. In the rare event that the free array is full during a call to `free`, the main thread signals the manager thread to clear the array and waits until a spot becomes available.

To obtain a quantitative analysis of the effectiveness our threaded library, we ran the SPEC CPU2006 benchmarks with both the standard library and our library [13]. The benchmark scores are calculated by timing the benchmark runs and comparing the times to the those of a fixed reference machine. The test system hardware consisted of a 2.66GHz Intel Core i7-920 processor with 3GB of RAM. The system `malloc` library used was the Linux `glibc` ptmalloc library. The standard SPEC benchmark scrips were used to perform the experiments and the reported results are the geometric mean of ten base benchmark runs of the library. The average variance of each test was below .003. The results from the tests are shown in Figure 1. For each benchmark, we show: the performance values output by the SPEC scripts using (i) the original system library (higher values are better), (ii) our threaded library, and (iii) the resulting speedup. The results show a maximum speedup of 6.42% (xlanchbmk). In all cases, this speedup is the result of a reduction of the dynamic memory costs by 25% (the total number of instruction for `malloc` and `free` were actually reduced by half, but the atomic instructions are approximately 20 times more expensive than the removed integer operations [14]).

V. Pre-computing arguments for Trigonometric Functions

Our experiences pre-computing dynamic memory operations encouraged us to consider expanding the approach to apply to more complex operations. The key question is: *can we develop algorithms that are effective at predicting future argument values for application programs*? The constraints on arguments to

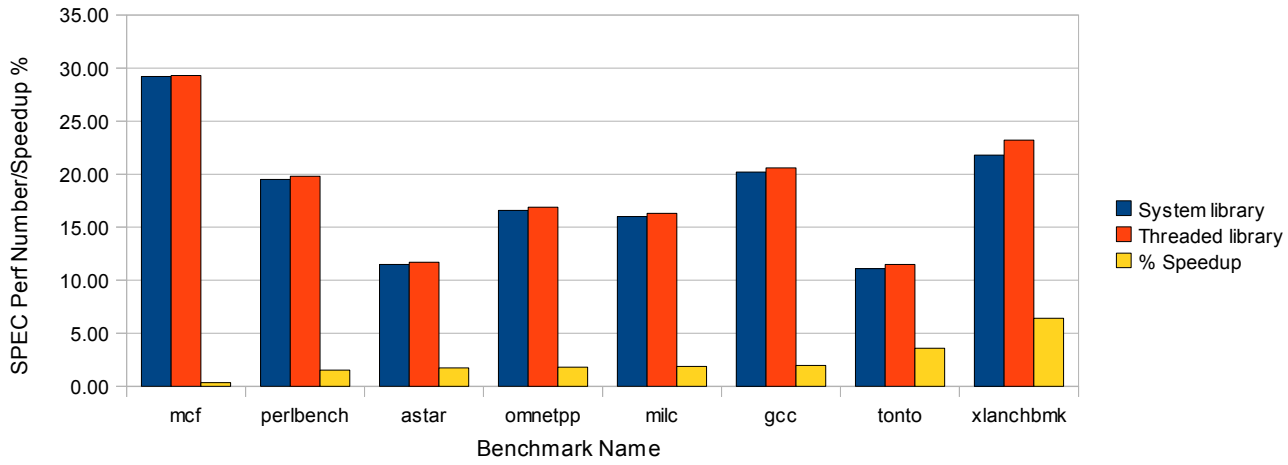

Fig. 1. SPEC Benchmark Result Summary on Intel i7

dynamic memory methods (specifically `malloc`) are somewhat alleviated because we can always use pre-allocated blocks that are slightly larger than the actual value requested. In order to generalize the work, the question becomes: are program arguments well behaved and occurring such that the future arguments to a value can be accurately predicted? To answer this question, we decided to examine the arguments passed into some of the standard trigonometric functions.

Rather than work with specific programs to capture argument values, we simply defined a library for `sin`, `cos`, and `tan` to record the process id (PID) and argument value to a file. Our library methods then used the regular system library functions to compute and return the correct values. We then setup a Linux workstation with the system path defined to use our library and captured the values for a couple of hours while the system was being used as a general purpose workstation. We captured a total of 155,362 uses of trig function values from 14 different PIDs. We discovered that all the arguments to the `tan` function were trivial, having only one or two distinct arguments and our predictor achieved a nearly 100% success rate on them. Therefore, we do not display the results (Figure 3) for the `tan` function.

The next step was to develop an online learning algorithm and use it to determine if we could reasonably predict the next argument that was used. The learning algorithm uses the current history to predict the next value; the predicted next value is compared against the actual next value. Only exact matches are considered a success. To keep things simple, we did not consider performance issues and implemented our algorithms in Mathlab. A generalized learning algorithm is evaluated so that it could potentially be used for an argument stream from any program/function that has a type compatible argument list. Lastly, we considered a match success/failure only on the immediately next argument used. This is a strict comparison and if combined with memoization, we could well find that the pre-computed result is actually used farther into the future. However, we have not yet performed any analysis on that possibility.

The algorithm that has been used to predict the future data values is based on context and hence is often referred to as *context predictor* [15]. Sazeides *et al* state that the context predictor has an accuracy from 50%-90% [16]. We have implemented their context prediction algorithm with slight modification. Context is considered as an interrelated set of data values (finite sequence of values) following certain pattern. In our context prediction algorithm we have considered a matrix to store different contexts as the learning process progresses. Based on the previous values a particular context is chosen and if the context contains a value after the previous value a prediction is made. However, if it does not contain any future value after the previous value a prediction based on the maximum frequency of occurrences in the particular context is made. When two or more values have the same maximum frequency in the same context then the algorithm makes a random selection. If a prediction previously made is wrong then the algorithm corrects the existing contexts or create new contexts so that it becomes more accurate as the predictions progress. A pseudo-code representation of the algorithm is shown in Figure 2. In this algorithm the matrix `M` is updated to store the different contexts in rows as the learning process progresses (`M` is initialized

```
if M == NULL then
  Initialize C, Pc, j, k to 0
else
  if Ac == Pc then
    if M[C][j+1]! = NULL then
      Pc = M[C][j+1]
      if k==15 then
        Set k to 0
      end if
      ACT[k] = Ac
      Increment k and j by 1
    else
      Set Pc to argument value with maximum
              frequency in the current context, C
    end if
  else
    Initialize i, l, Cfound to 0
    while ( Cfound == 0 ) and ( M[i][0] != NULL ) do
      find the current context using ACT[] and M[i][l] matrix
      if current context is found then
        set Cfound to 1
        C = i
        Pc = M[C][l] and exit loop
      end if
    end while
    if Cfound == 0 then
      Add new context at i+1
      set C=i+1 with M[C][0] = Ac
      Pc = Ac
    end if
  end if
end if
```

Fig. 2. Learning Algorithm to Predict The Next Input Argument

to a null matrix). In addition to the context matrix, `M`, we make use of four important parameters:

- `C`: The current context of argument values which points to a particular row in the matrix (`M[i]`).
- `Pc`: predicted argument value.
- `Ac`: Actual argument value.
- `ACT[]`an array to store the latest 15 actual argument values.

As it can be seen from the algorithm, if the actual value and the predicted value are the same (a success in the previous prediction) then the same context is continued and the next argument value to be predicted is the next value in the current or same context, `C`. If no future values exist under the current context, `C` then the argument value with the maximum frequency in the context is chosen to be the next predicted value. If the previous prediction is a failure then the same context is now longer valid. So, a search for a different context is initiated using the `ACT[]` array and the context matrix `M`. Once a different context is obtained, the next argument value is predicted as shown in the algorithm. However, if the search completes without finding a different context then a new context is added to the context matrix, `M` with only a single value in the context which is `Ac` (actual argument value) and the predicted value is also set to `Ac`.

Summarizing, based on the previous values a particular context is chosen and if the context contains

a value after the latest actual value a prediction is made. However, if it does not contain any future value after the latest actual value a prediction based on the maximum frequency of occurrences in the particular context is made. When two or more values have the same maximum frequency in the same context then we go for a random selection among them. If a prediction previously made is wrong then we correct the existing contexts or create new contexts so that it becomes more accurate as the predictions progress.

This algorithm was implemented in Matlab and used to predict the input arguments arguments to the sine, cosine, and tangent trig functions that were captured one afternoon from an operating Linux workstation. Figure 3 summarizes our match results. The results are organized into 2 groups (top row: `sin` and bottom row: `cos`). Each column shows a unique trig function/PID combination. The results for all uses of the `tan` function are above 99% as there were no uses that were non-trivial, therefore we do not show results for `tan`. Notice that even with `sin` and `cos` there are several bars that show nearly 100% prediction accuracy. This is because their input arguments consisted of only one or two distinct values that are easy predicted. However, what is interesting are results for other uses of `sin` and `cos`. In both cases, we show a prediction accuracy between 40% and 50%. Lastly, we have also evaluated the argument prediction process using other learning algorithms based on clustering the data values and using decision trees [15]. They have given approximately the same match results for the considered data and therefore we have not presented them.

While the computational costs of trigonometric functions may not be sufficiently high to merit pre-computation in a background thread, this result encourages us by showing that for some functions the prediction of future argument values can be reasonably accurate. The successful prediction and pre-computation of 40-50% of a computational expensive function call could drastically improve an applications performance. In the next section, we develop a design solution that can provide the infrastructure that will simplify the deployment of pre-computation in user library functions.

VI. Methods to Support Pre-Computing Function Values

Once a function is identified as a candidate for pre-computation the next task is deciding how to implement an effective and efficient prediction system. We are designing a software development environment to assist in the integration of pre-computation into existing function calls. A predictor class object could be coded to serve as an intermediary between the function calls and the actual function execution. Essentially the class would serve as a wrapper that resides between the program code and the function code. An instance of the predictor class would be instantiated for each function to be forked into a pre-computation thread. The normal calls to the function would be replaced by a call to a method of the predictor class. The same arguments would be passed in and the correct output would still be returned. Using this method all prediction, pre-computation, and threading details are abstracted. The programmer of the function pre-computation wrapper only has to create and setup the predictor object and write the function calls to go through it.

When creating the predictor, information about the function is needed to accurately replicate the structure of the function. The number and type of arguments to the function need to be known, as well as the type of the return value. Once this information is know, the method used to replace the function calls can be set to have the exact same interface structure as the real function. In addition, the predictor object must be passed a pointer to the original function so that it can use the existing implementation for the pre-computation activities. To illustrate how the predictor class would be used, some pseudo-code based loosely on C/C++ are shown. In this example, we show how the predictor class would be used to enable pre-calculation for a simple cosine function. For example, a sample constructor for the predictor class is:

```
//structure of the constructor
void Predictor(char* functionName,
               int numargs,
               char* returnType,
               char* argumentType,...);

//initializing the predictor
Predictor cosPredictor =
          new Predictor("cos", 1,
                        "double",
                        "double");
cosPredictor.setParams(
    <additional configuration params>);
cosPredictor.Initialize();
```

The next code block illustrates how the predictor class can be easily substituted into the code anywhere the original function would have been called.

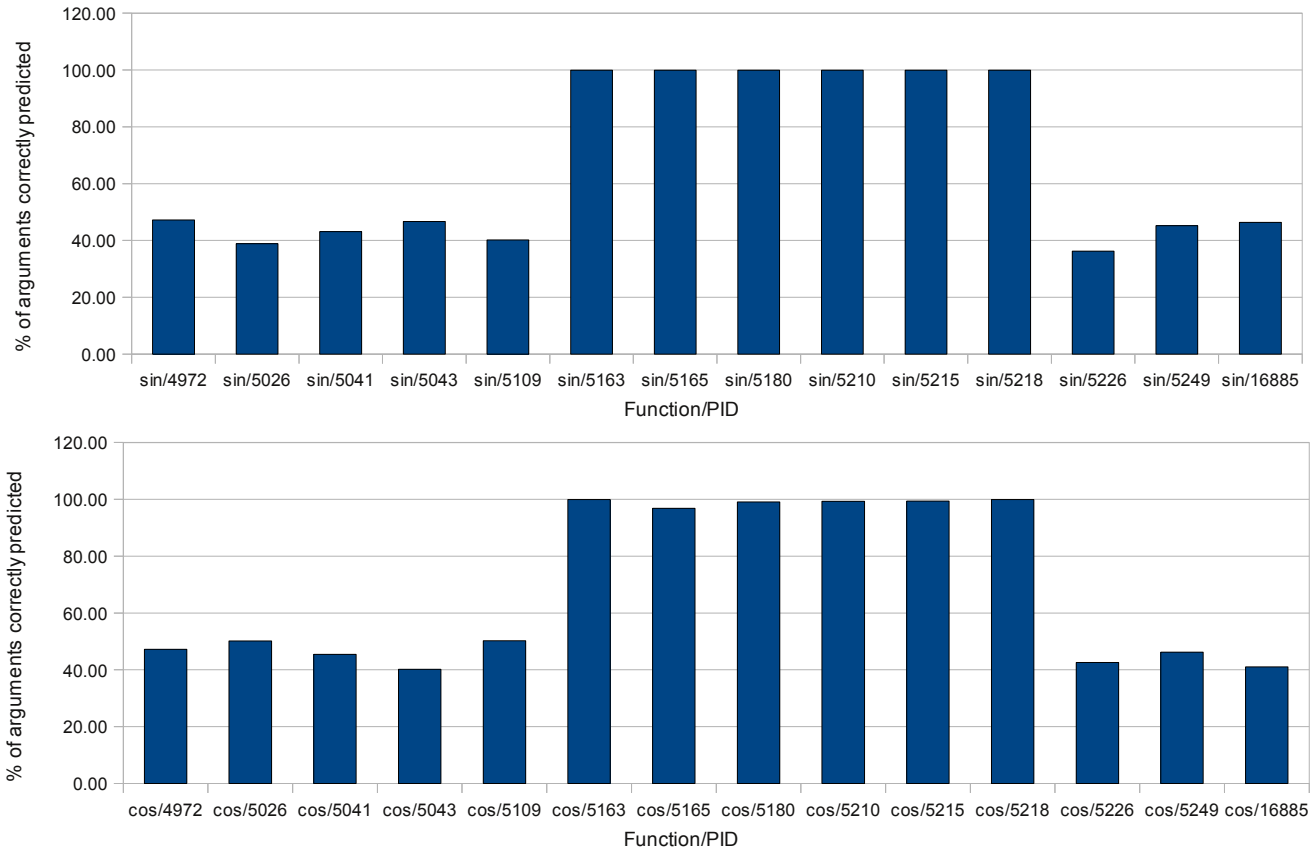

Fig. 3. Argument Match Results for Trigonometric Functions

```
//calling the function
xcoordinate =
    cosPredictor.callFunction(degrees);
```

During initialization, the predictor class uses the name of the library function to recurse down the dynamic link path to find a pointer to the function. In our dynamic memory library implementation, the `dlsym` function performed this task. The function pointer is used in conjunction with the function structure information to allow the predictor class to access the original library function. The function structure information is also used to set up the size and type of the argument arrays.

Internally the predictor class uses online learning algorithms to monitor arguments passed into the function and identifies the best prediction result. Once a certain configurable threshold is met, a separate thread is spawned to handle pre-calculation of future function calls. The main thread must communicate the argument patterns to the predictor thread periodically so that it can accurately predict the arguments to future function calls. Similarly, the predictor thread must provide the program thread with predicted future function outputs. These outputs would be stored with the predicted arguments that produced them so that the main thread can determine if the argument prediction was successful. It may even be beneficial to implement a buffer of future argument-output combinations to increase the chance of a correct prediction. Memoization could also be used to cache a certain number of previous function calls if it is found that calls are periodically duplicated. A happy medium must be reached; the more predicted argument pairs that need to be checked the longer the search overhead becomes. It is possible that the number of future predictions to be buffered and the frequency of thread communication could be adjusted, either statically through a configuration variable or dynamically during program execution (based on current performance). If an argument pair is incorrectly predicted, the function must be computed in-line by the main program thread.

The appropriateness of threaded pre-computation for a function will depend on how the function is used inside a particular application program. The predictor class could be coded to monitor overall effectiveness and adjust the amount of pre-computation based on the

prediction accuracy and processing power availability. It could even switch off the thread if the prediction and pre-computation fails to provide performance benefits. If no speedup is found using the predictor class, it does not have to be used.

One benefit of constructing and using a predictor class is that it opens the possibility of using multiple predictor threads in a single program. Each candidate function would spawn its own predictor thread, allowing a program to have as many predictor threads as it has candidate functions. If properly implemented, a generic predictor class could be used to easily adapt serial programs to many-core systems without placing the burden of parallel programming on the application developer.

VII. Conclusion

Emerging many-core processors provide unique opportunities and significant challenges for the parallel processing community. In particular, we need to rethink our conventional thoughts of parallelism where we pursue solutions whose success is measured by scalability with the number of processors/threads. We must also begin to think of these processors/threads as free/cheap resources that should be exploited whenever possible to gain any, and even modest, speedup opportunities.

In the work of this paper, we examine the prospects of transparently migrating user functions to concurrent, background threads that pre-compute function results for an application program. The background thread can use online learning algorithms to predict future arguments in order to facilitate the pre-computation of function results. While sharing many similarities with the concept of "program futures," threaded pre-computation is unique in that the threading and pre-computation is done transparently and without requiring parallel programming skills from the application programmer. Empirical studies with the argument values from trigonometric functions recorded from a running system show that an online learning algorithm is able to accurately predict the immediately next argument value 40-50% of the time.

As the computing market heads towards many-core parallel architectures, increasing system performance will ultimately depend on the ability of applications to more fully utilize parallelism. Finding ways to extract parallelism from user and system libraries with minimal programmer inputs provides one way to help accomplish this goal. Fortunately, threaded pre-computation can also be used in programs that are also manually parallelized to gain even more speedup. For example, our work with pre-computing dynamic memory management was successfully used with the multi-threaded firefox web browser.

References

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.

[2] A. Ghuloum, "Face the inevitable, embrace parallelism," *Communications of the ACM*, vol. 52, no. 9, pp. 36–38, Sep. 2009.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. Technical Report No. UCB/EECS–2006–183, Dec. 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2007.

[5] R. Kalla, "Power7: Ibm's next generation power microprocessor," in *Hot Chips 21*, Aug. 2009.

[6] Intel Press Release, Intel Corporation, "Futuristic intel chip could reshape how computers are built, consumers interact with their pcs and personal devices," Intel Press Release, Intel Corporation, Tech. Rep., Dec. 2009. [Online]. Available: http://www.intel.com/pressroom/archive/releases/20091202comp_sm.htm

[7] X.-F. Li, Z.-H. Du, Q.-Y. Zhao, and T.-F. Ngai, "Software value prediction for speculative parallel threaded computations," in *In First Value Prediction Workshop*, Jun. 2003, pp. 18–25.

[8] U. A. Acar, G. E. Blelloch, and R. Harper, "Selective memoization," *SIGPLAN Notices*, vol. 38, no. 1, pp. 14–25, Jan. 2003.

[9] H. Baker and C. Hewitt, "The incremental garbage collection of processes," *Proceedings of the Symposium on Artifical Intelligence and Programming Languages, SIGPLAN Notices*, vol. 12, no. 8, pp. 55–59, Aug. 1977.

[10] D. P. Friedman and D. S. Wise, "Cons should not evaluate its arguments," in *Third International Colloquium on Automata, Languages and Programming*, 1976, pp. 257–284.

[11] E. C. Herrmann and P. A. Wilsey, "Threaded dynamic memory management in many-core processors," in *Proceedings of the 2010 International Workshop on Multi-Core Computing Systems (MuCoCoS 2010)*. IEEE Computer Society, 2010.

[12] D. Tiwari, S. Lee, J. Tuck, and Y. Solihin, "Mmt: Exploiting fine-grained parallelism in dynamic memory management," in *IEEE International Parallel & Distributed Processing Symposium*, Apr. 2010.

[13] S. P. E. Corporation, "Spec cpu 2006," Aug. 2008 (last updated). [Online]. Available: http://www.spec.org/cpu2006/

[14] E. C. Herrmann, "Threaded dynamic memory management in many-core processors," Master's thesis, University of Cincinnati, Cincinnati, OH, 2009.

[15] P. Laird, "Identifying and using patterns in sequential data," in *4th International Workshop in Algorithmic Learning Theory*, 1993, pp. 1–18.

[16] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of the 30th International Symposium on Microarchitecture*, Dec. 1997, pp. 248–258.