Node Prefetch Prediction in Dataflow Graphs

Newton G. Petersen
Martin R. Wojcik

The Department of Electrical and Computer Engineering – The University of Texas at Austin

newton.petersen@ni.com
mrw325@yahoo.com

EE382C: Embedded Software Systems – Final Report

May 8, 2002

Abstract

Dataflow languages provide a high-level description that can expose inherent parallelism in many applications. This high level description can be applied to automatically create efficient code and schedules based on patterns in the dataflow graphs and knowledge of the target architecture. When targeting a dataflow graph to custom hardware, it is sometimes advantageous to share nodes with similar functionality to save silicon. Any state information associated with the caller of the shared node must be stored and subsequently loaded upon firing. If prediction logic can predict which caller of a shared node is next, the associated state information can be prefetched while other nodes of the graph are executing. While some applications can be entirely scheduled at compile time, many multi-channel measurement and control applications require some degree of dynamic scheduling. Prefetching in statically scheduled dataflow graphs can be done at compile time, while prefetching in dynamically scheduled dataflow graphs requires dynamic prediction.

INTRODUCTION

Many measurement and control applications attempt to accomplish similar tasks on multiple channels. When targeting systems to custom hardware, sharing functionality between multiple channels can save silicon. Many times these functional units contain state information that must be loaded from memory before execution, and prefetching this state information can enhance system performance. The sequence of callers can be determined at compile time if the application is statically schedulable. If the sequence is dynamic in nature, a prediction mechanism is necessary to support prefetching.

Our main objective is to explore the feasibility of caller prediction as a method for predicting the next caller of a shared functional unit executing in a parallel processing system. Dataflow graphs are good at describing parallelism, and thus are a natural fit for modeling systems where the caller prediction mechanism may be beneficial. We examine related work in classifying dataflow graphs, and build on the concept of branch prediction in pipelined processors to implement our prefetch prediction unit for dynamically scheduled dataflow graphs.

RELATED WORK

Statically schedulable models of computation do not need prediction because all scheduling decisions are made at compile time; however, they can still take advantage of prefetching state data. Synchronous dataflow (SDF), computation graphs, and cyclo-static dataflow (CSDF) are all powerful models of computation for applications where the schedule is known at compile time [3]. For a valid schedule, it is possible to speed-up the process by simply pre-loading actors and their respective internal state data. Figure 1 shows the *prefetch* nodes explicitly in the diagram; however, the *prefetch* nodes could be added implicitly when targeting hardware capable of taking advantage of the parallelism exposed in the dataflow graph.

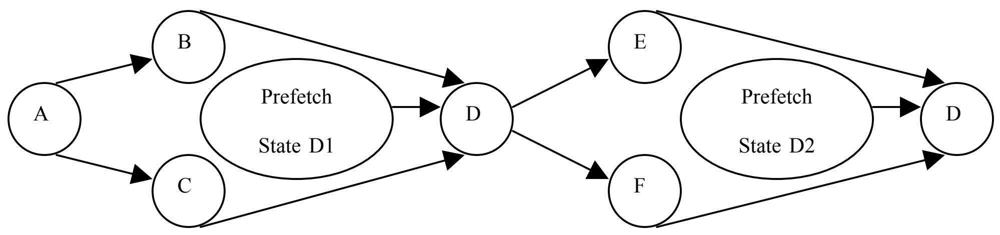

Figure 1: Homogeneous SDF graph with prefetch of state information for the shared node D

The idea of prefetching data is not new. Cahoon and McKinley have researched extracting dataflow dependencies from Java applications for the purpose of prefetching state data associated with nodes [2]. Wang *et al.*, when exploring how to best schedule loops expressed as dataflow graphs, also try to schedule the prefetching of data needed for loop iterations [5].

Dynamically Schedulable Models

While statically schedulable models of computation are common in signal and image processing applications and make efficient scheduling easier, the range of applications is restrictive because runtime scheduling is not allowed. Dynamically schedulable models of computation, such as Boolean dataflow, dynamic dataflow, and process networks, allow runtime decisions, but in the process make static prefetch difficult, if not impossible.

G, the LabVIEW™ graphical programming language, requires dynamic scheduling. According to Andrade *et al.*, the G model of computation is homogeneous, dynamically scheduled, and multidimensional [1]. Local variables, global variables, and other features of G make it possible to implement general process networks. The dynamic, yet structured nature of G makes certain subsets well suited to the exploration of prefetch prediction for shared G nodes. We used G as our prototyping environment for practical reasons.

Branch Prediction

Two-level branch prediction was pioneered by Patt and Yeh to help keep processor pipelines full for a greater percentage of the time [7]. This prediction model uses a lookup table of saturating two-bit counters that represent the likelihood that a branch will be taken given a certain history. As illustrated in Figure 2, the history register consists of data indicating if a branch was taken, or not taken, the past *n* times. A '1' represents a taken branch, and a '0' represents a branch not taken. The table therefore has 2^n entries.

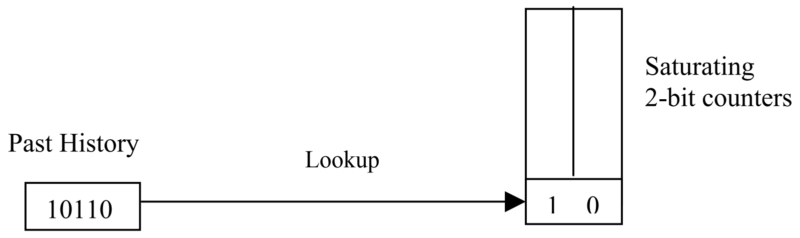

Figure 2: Two-level branch prediction

A '1' in the counter MSB predicts that a branch will be taken, while a '0' predicts that the branch will not be taken. This approach achieves a prediction success rate in the 90% range [4]. Patt and Yeh have tabulated the hardware costs to be significant for large history lengths [6].

IMPLEMENTATION

Our initial pass at a design of our prediction mechanism borrowed from the two-level branch prediction mechanism. Figure 3 shows the overall block diagram architecture. The past call history shift register keeps track of the last *n* calls to a shared node. The values held in the shift register are wired directly to a hash function that converts the history entry to a row lookup index into the call history table. The call history table has a column for each possible caller of the shared node.

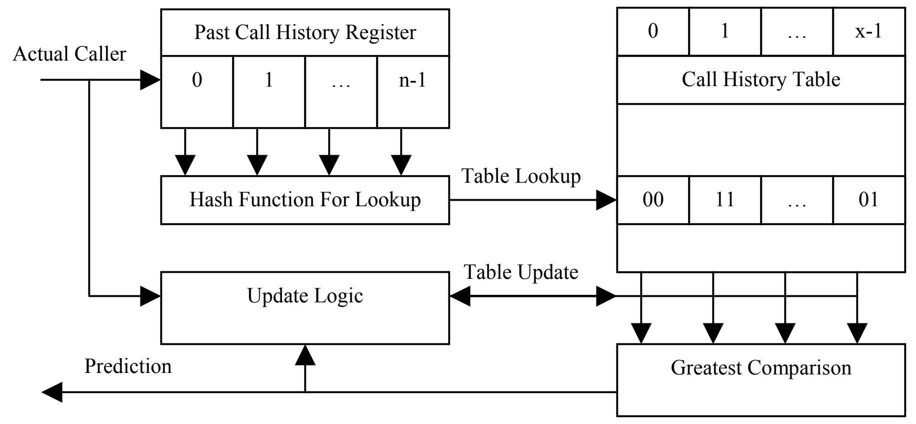

Figure 3: Block diagram of two-level caller prediction

Each cell in the call history table is managed by update logic to increment when a prediction is correct and decrement when a prediction is incorrect. When a prediction is desired, the column values across the history table are compared to find the greatest value.

Ideally, there would be a row in the history table for each possible value of the past history register. However, the number of rows grows exponentially (x^n) with the length of the history register. For three callers (*x)* and a history length of eight (*n)*, the direct map approach requires 6561 rows. A hashing function is necessary to eliminate this exponential dependence.

In the above predictor model, the maximum number of callers is analogous to a numeric base, and each position in the history shift register is analogous to a position in the numeric base. For example, for ten possible callers, a history register containing *9,6,2* maps to a lookup index of 962 using base 10. For *11* possible callers, a conversion is performed by evaluating *9*11^2 + 6*11^1 + 2*11^0*. A hardware modulo hash function can convert and multiply the running sum by a large prime number during each stage, but would only use the lower *k* bits of the direct mapped result. Hashing introduces collisions, which decrease the prediction accuracy.

However, if our application settles to a periodic schedule quickly, and the schedule changes infrequently, the need to keep track of past periodic behavior is eliminated.

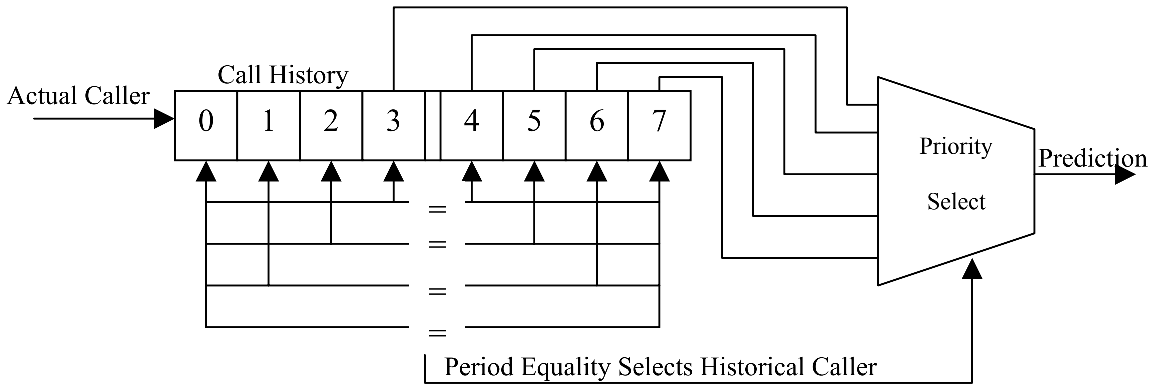

Figure 4: Predictor based on detecting a periodic calling sequence

A predictor model that determines the current periodic behavior in the current calling history is illustrated in Figure 4. The call history shift register in the period predictor is similar to the past history register of the two-level predictor. The period predictor first splits the history register in half, and then finds the part of the second half which best correlates with the beginning of the first half. For example, for a past call history of *12341234*, the first equality is true, and the period predictor selects *4* as the prediction. Similarly, for a history of *02345602*, the third equality is true, and period predictor selects *6* as the prediction. These comparisons assume that the period of the calling sequence is contained in the history register, and that call prediction latency is equal to the length of the history register if the period or the calling sequence changes.

RESULTS

We used a simple LabVIEW™ filtering application (Figure 5) to test our predictor models. The timer blocks provide the dynamic sample rate for the analog to digital converters (ADCs) and for the digital to analog converters (DACs) on the two independent channels. The sample rate also determines which set of coefficients will be loaded into the FIR block for filtering.

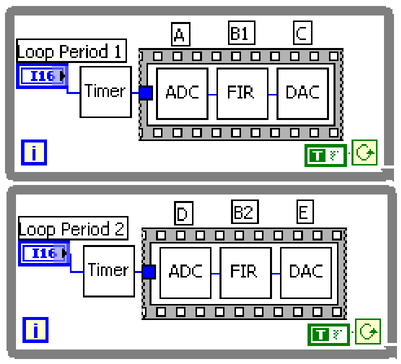

Figure 5: Dynamically scheduled LabVIEW™ filtering application using a shared FIR block

In the analysis that follows, the ADCs and DACs (blocks A, C, D, and E) take 20 cycles to execute, fetching the coefficients and current tap values (blocks F1 and F2) takes 31 cycles, and executing the shared FIR filter block (blocks B1 and B2) takes 6 cycles.

Figure 6 illustrates a schedule with three parallel threads of execution without prefetch prediction. For our analysis, we assume that loop 1 runs twice as fast as loop 2.

20	31	6	20	20	31	6	20	31	6	20	=211 cycles
A	F1		C	A	F1		C				Loop 1
		B1				B1			B2		FIR Exec
				D				F2		E	Loop 2

Figure 6: Schedule for loop 1 running twice as fast as loop 2 without prefetch prediction

The top row shows the execution time of each block, the second row shows the execution of blocks in loop 1, the third row shows when the shared FIR block executes, and the last row shows the execution of blocks in loop 2. Fetching occurs inline with the loop executions since we do not know which loop will call the FIR block next.

On the other hand, prefetch prediction tries to prefetch the data for the next caller while other blocks execute, as shown in Figure 7. We have noted a 33% improvement in system speed when our predictions are correct.

31	6	20	20	6	31	6	20	=140 cycles
A		C	A		C			Loop 1
F1	B1	F1		B1	F2	B2	F1	FIR Exec
			D				E	Loop 2

Figure 7: Schedule for loop 1 running twice as fast as loop 2 with prefetch prediction

For a period prediction unit with a history register long enough to contain the entire period, our predictions will always be correct. If our predictions are incorrect, our sampling schedule would shift, and our sampling rate would vary. A varying sample rate is acceptable for many feedback control applications, but not for most signal processing applications.

Before implementing prefetch prediction, we must examine if the performance enhancements outweigh the silicon cost. We are prototyping using Xilinx Virtex II™ FPGAs to provide the parallel processing for implementing dataflow graphs. The two-level lookup prediction approach uses over 500 slices, even without the block memory for the lookup table. As a point of reference, a symmetric 32-tap 32-bit FIR filter uses around 400 slices. The complexity and size of the two-level predictor makes its current practicality questionable.

The simplicity of the period predictor approach yields better implementation results. Figure 8 shows the size of the unit for varying history lengths and varying number of callers.

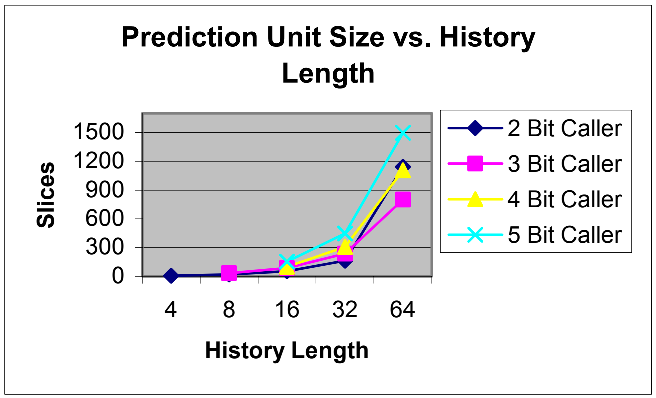

Figure 8: Periodic prediction unit size estimates

The size of the periodic predictor is reasonable for a history length of 32 or less, due to the special logic and routing resources in the Virtex II™ FPGA that implement the final mux in the period predictor [8]. The latency needed for a prediction is as low as one or two cycles, as only the parallel equality tests and the subsequent select are necessary.

CONCLUSION

The concepts of prefetching and branch prediction serve as the basis for our prefetch caller prediction of shared nodes on dataflow graphs. Shared nodes are common in multi-channel measurement and control systems. We have examined two mechanisms for implementing caller prediction, and have an implementation preference for detecting periodic behavior in the calling sequence. We showed a performance improvement of 33% for a simple multi-channel application.

REFERENCES

[1] H. A. Andrade and S. Kovner, "Software Synthesis from Dataflow Models for G and LabVIEW™," in *Proc. IEEE Asilomar Conference on Signals, Systems & Computers*, pp. 1705-1709, 1998.

[2] B. Cahoon and K. S. McKinley, "Data Flow Analysis for Software Prefetching Linked Data Structures in Java," in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 280-291, 2001.

[3] S. A. Edwards, *Languages for Digital Embedded Systems*, Kluwer Academic Publishers, 2000.

[4] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt, "An Analysis of Correlation and Predictability: What Makes Two Level Branch Predictors Work," in *Proc. IEEE/ACM Int. Sym. on Computer Architecture,* pp. 52-61, 1998.

[5] Z. Wang, T. W. O'Neil, and E. H-M. Sha, "Optimal Loop Scheduling for Hiding Memory Latency Based on Two-Level Partitioning and Prefetching," *IEEE Transactions on Signal Processing*, pp. 2853-2864, 2001.

[6] T.-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," in *IEEE/ACM Int. Sym. on Computer Architecture,* pp. 124-134, 1992.

[7] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *IEEE/ACM Int. Sym. on Microarchitecture*, pp. 51-61, 1991.

[8] Xilinx Inc., *The Virtex-II Handbook*, http://www.xilinx.com/products/virtex/handbook/index.htm.