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 – Literature Survey

March 25, 2002

Abstract

Dataflow languages can both provide a high level description and expose inherent parallelism of applications. This high level description can be applied to automatically create efficient code and schedules based on patterns in the dataflow diagrams and knowledge of the target architecture. Our project is focused on taking advantage of any inactivity in threads executing in parallel to predict and prefetch state information associated with each instance of a shared node. The prefetched state information is associated with the caller of each node. The prefetch prediction method is based on static knowledge of all the callers in an application's topology. This paper identifies classes of dynamically scheduled dataflow models of computation to which the prefetch prediction method might apply. We extend the concept of branch prediction to predict the caller of shared nodes. We identify applications where acceptable variance on quality of service provides possible compelling application spaces.

INTRODUCTION

Caller prediction is a method for predicting the next caller of a shared functional unit for applications executing in parallel and distributed processing systems. Since dataflow models can be used to describe these systems, we examine research on the different dataflow models of computation. Caller prediction gives us the ability to prefetch state information for a shared node. We plan to utilize the existing research on branch prediction for pipelined processors to implement our prediction logic. This functionality could allow dedicated hardware to increase the locality of state information dependent on the caller. Our research concentrates on determining the tradeoffs and potential performance benefits that can be expected from using our technique for certain classes of applications.

MODELS OF COMPUTATION

Statically Schedulable

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 [7]. 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.

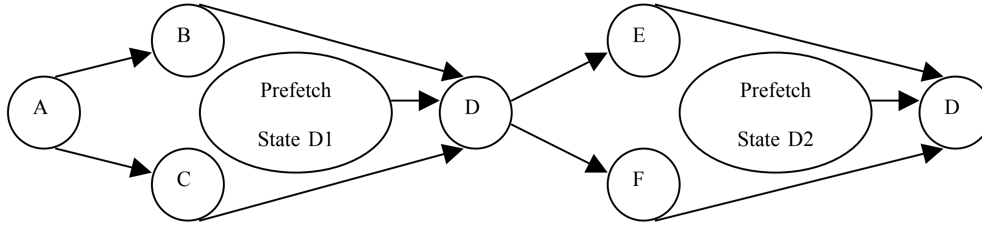

Figure 1: Homogeneous SDF graph with prefetch of state information for the shared node D

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 [5]. 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 [11].

Dynamically Schedulable

While statically schedulable models of computation are common in DSP applications and make efficient scheduling easy, the range of applications is very restrictive because no runtime scheduling is allowed. Dynamically schedulable models of computation allow runtime decisions, but in the process make static prefetch more difficult, if not impossible. In this situation we propose a caller prediction scheme that would allow prefetching state information associated with a shared node, even though the next caller is not known at compile time.

As J. T. Buck explains in his thesis, Boolean Dataflow (BDF) is a model of computation sometimes requiring dynamic scheduling [3]. The *switch* and *select* actors allow conditional dataflow statements with semantics for control flow as shown in Figure 2.

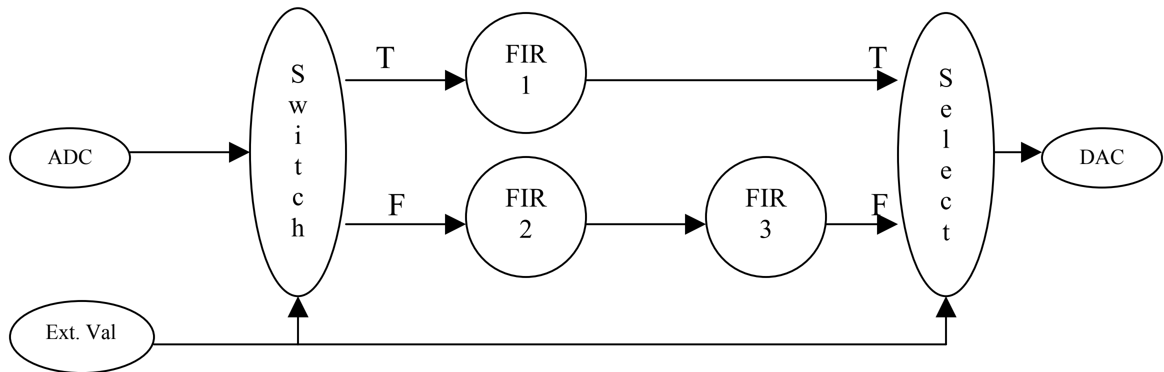

Figure 2: Homogeneous, quasi-static BDF graph

Scheduling in BDF can be either quasi-static or dynamic [3]. The *switch* and *select* actors can be combined to create *if-then-else* structures. More complex control structures can be implemented by combining multiple actors. In quasi-static BDF graphs, prefetching could be implemented by analyzing the program flow through the *switch* and *select* actors, and would not require prediction logic. However, for complex graphs requiring dynamic scheduling, caller prediction is a possible method for predicting the next caller of a shared node.

Cyclo-dynamic dataflow, or CDDF, as defined by Wauters et al. [12], extends cyclo-static dataflow to allow run-time, data-dependent decisions. Similar to a CSDF, each CDDF actor may execute as a sequence of phases *f1, f2, f3, f4, …, fn*. During each phase *fn*, a specific code segment may be executed. In other words, the number of tokens produced and consumed by an actor can vary between firings as long as the variations can be expressed in a periodic pattern [9]. CDDF allows run-time decisions for determining the code segment corresponding to a given phase, the token transfer for a given code segment, and the length of the phase firing sequence. The same arguments for using caller prediction for BDF graphs can be extended to include CDDF.

Dynamic dataflow, or DDF, extends the BDF model to allow any Boolean expression to serve as the firing rule. For example, the *switch* node in Figure 2 could allow a variable number of tokens on one of its inputs before firing [12]. DDF also allows recursion. Recursion is possibly too dynamic to support caller prediction, as state information would not only be associated with a caller, but also with a recursion depth varying at runtime. DDF requires a run-time scheduler to determine when an actor becomes executable [12], and is a possible model of computation that could benefit from our prefetch prediction method.

Lee and Parks describe dataflow process networks as a special case of Kahn process networks [10]. The dataflow process networks implement multiple concurrent processes by using unidirectional FIFO channels for inter-process communication, with non-blocking writes to each channel, and blocking reads from each channel. In the Kahn and MacQueen representation, run-time configuration of the network is allowed. While some scheduling can be done at compile time, for some applications most actor firings must be scheduled dynamically at run-time [10].

G, the LabVIEW™ graphical programming language, contains both dynamically and statically scheduled nodes that may be shared. According to Andrade et al., the G dataflow model of computation is homogeneous, dynamically scheduled, and multidimensional [2]. Locals, globals, and other features of G make it possible to implement general process networks. The current implementation of G does not allow recursion. The dynamic, yet structured nature of the G, makes certain applications well suited to the exploration of prefetch prediction for shared G nodes. For practical reasons, we plan to use the G language as a prototyping environment for our project.

BRANCH PREDICTION

Most modern general-purpose processors use branch prediction to improve processor performance. Branch prediction can be used to keep processor pipelines full for a greater percentage of the time by predicting execution flow after a branch. As a result, the processor does not wait for the branch to execute before starting to fetch and decode the next instruction. Most current branch prediction techniques use past branch history and associated data.

Two level branch prediction was pioneered by Yale Patt and Tse-Yu Yeh in 1991 [14]. This prediction model uses a lookup table of saturating two-bit counters, which represent the likelihood that a branch will be taken given a certain history. As illustrated in Figure 3, 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.

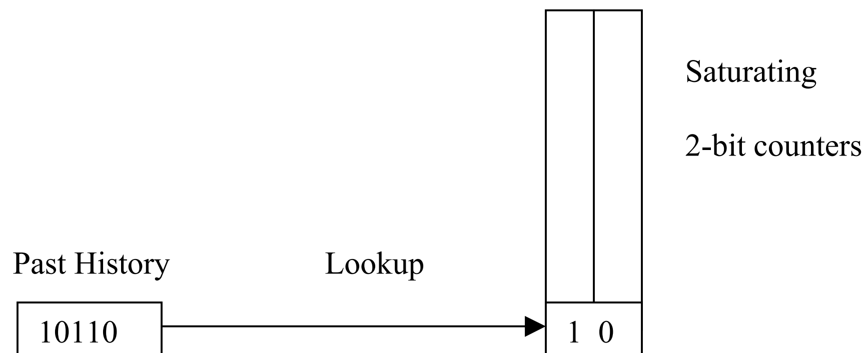

Figure 3: Two level branch prediction

A '1' in the MSB of the counter 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 on certain applications [8]. Patt and Yeh have also tabulated the hardware costs for this method to be significant especially for large history lengths [13]. Hardware costs of prediction

are important in our analysis of caller prediction, because if the hardware cost for prediction is larger per caller than the shared functional unit, the best solution would be to instantiate separate units for each caller.

Branch prediction has also been extended to handle indirect jumps, which in C language amounts to dereferencing a function pointer, and in C++ amounts to a virtual function call. Indirect jump prediction is closer to our caller prediction model because there is a possibility of multiple targets. By combining the branch history and the *jump from* address through a hash function, an index into a target cache of recently *jumped to* addresses can be created. The branch history could either consist of a global pattern history of conditional branches, or a record of the last *n jump to* addresses given the *jump from* address [6]. Our implementation of caller prediction will probably use the combination of conditional branch prediction and indirect jump prediction.

APPLICATIONS

There are a few characteristics we feel are important to any application taking advantage of caller prediction model for prefetching node instance data. Compelling applications cannot execute in hard real-time since the prediction logic may be wrong, resulting in jitter. Instead, we choose to concentrate on applications where the quality of service (QoS) can vary depending on how fast the application is running. Buttazzo et al. point out [4] that these applications are common due to the non-deterministic behavior of common low level processor architecture components, such as caching, prefetching, and direct memory access (DMA) transfers. Buttazzo's work suggests voice sampling, image acquisition, sound generation, data compression, video playback, and certain feedback control systems as application domains that can function at varying QoS depending on the execution rate of the application.

Feedback control systems are particularly interesting because they may contain several subsystems that share common components. Abdelzaher et al. explore a complex real-time automated flight control system that negotiates QoS depending on the load of the system [1]. The flight control system consists of a main function that controls the altitude, speed, and bearing of the plane, and also contains subsystems for flight control performance. Continuous analysis of all of the inputs to achieve optimal control is a large computation task. However, the task of flying a plane can be accomplished with relatively little computation power by carefully reserving resources and by tolerating less than optimal control functionality. PID (Proportional-Integral-Derivative) control is a possible compelling application domain because PID tuning parameters are relatively insensitive to rate changes. We hypothesize that many measurement and control applications may display similar properties of varying acceptable QoS.

CONCLUSION

The concepts of prefetching and branch prediction serve as the basis for our prefetch caller prediction of shared nodes on dataflow graphs. Prefetch caller prediction can improve performance of certain applications when described using dynamically scheduled dataflow models of computation. These applications should be targeted to custom hardware that is able to take advantage of the parallelism exposed in the dataflow graphs. One possible application domain includes mixed feedback control and measurement systems. Our research is concerned with exploring the feasibility of implementing caller prediction for prefetching and classifying applications where it makes sense.

REFERENCES

[1] T. Abdelzaher, E. M. Atkins, and K. G. Shin, “QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control,” in *IEEE Transactions on Computers,* pp. 1170-1183, 2000.

[2] H. A. Andrade and S. Kovner, “Software Synthesis from Dataflow Models for G and LabVIEW™,” in *Conference Record of the Thirty-Second Asilomar Conference on Signals, Systems & Computers*, pp. 1705-1709, 1998.

[3] J. T. Buck, *Scheduling dynamic dataflow graphs with Bounded Memory Using the Token Flow Model*. PhD Thesis, U. C. Berkeley, 1993. http://ptolemy.eecs.berkeley.edu/papers/jbuckThesis.

[4] G. Buttazzo and L. Abeni, “Adaptive Rate Control through Elastic Scheduling,” in *Proceedings of the 39th IEEE Conference on Decision and Control,* pp. 4883-4888, 2000.

[5] B. Cahoon and K. S. McKinley, “Data Flow Analysis for Software Prefetching Linked Data Structures in Java,” in *Proceedings of 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 280-291, 2001.

[6] P.-Y. Chang, E. Hao, and Y. N. Patt, “Target Prediction for Indirect Jumps,” in *Conference Proceedings of the 24th Annual International Symposium on Computer Architecture,* pp. 274-283, 1997.

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

[8] 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 *Proceedings of the 25th Annual International Symposium on Computer Architecture,* pp. 52-61, 1998.

[9] E. A. Lee, S. S. Bhattacharyya, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[10] E. A. Lee and T. M. Parks, ”Dataflow Process Networks”, in *Proceedings of the IEEE*, pp. 773-801, 1995.

[11] 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,” in *IEEE Transactions on Signal Processing*, pp. 2853-2864, 2001.

[12] P. Wauters, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-dynamic dataflow,” in *Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing*, pp. 319-326, 1996.

[l3] T.-Y. Yeh and Y. N. Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction,” in *Proceedings of the 19th Annual International Symposium on Computer Architecture,* pp. 124-134, 1992.

[14] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive branch prediction,” in *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 51-61, 1991.