New Methods for Exploiting Program Structure and Behavior in Computer Architecture

Amir Roth and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St. Madison, WI 53706
{amir, sohi}@cs.wisc.edu

Abstract

Micro-architectural techniques of the next decade will have to be more efficient and scalable in order to handle growing workloads and longer communication and memory latencies. We believe that information about program structure, the data and control relationships between instructions, can be used as a powerful framework for new techniques. We argue that program structure information has several inherent advantages over frameworks that associate information either with instructions in isolation or with data. We present summaries of four novel methods that apply program structure information to memory system problems from disambiguation and data cache bandwdith to prefetching and coherence optimization.

1 Introduction

Processor performance has improved at a dramatic rate in the past decades, powered by increasingly faster circuits as well as architectural and micro-architectural techniques that allow for both a high clock frequency and increased parallelism. Our continuing challenge is to sustain this rate into the next decade and beyond.

Current designs stand on a foundation of accumulated knowledge that itself assumes a certain set of processing demands and technology parameters. In the future, however, data working sets will be much larger and communication relatively more expensive. If tried and true techniques do not scale up to these tremendous workloads, we will need innovative new designs that will be more efficient and effective.

We believe that one principle underlying these novel methods will be an increased reliance on *program structure and behavior information*. Compared with techniques that exploit statistical data properties like spatial and temporal locality, *program based methods* have inherent advantages that allow them to leverage more compact representations into more significant gains. In addition, these novel techniques will also be capable of new and fundamentally different operations that will complement conventional methods.

Our research focuses on novel applications of program structure information in all aspects of processing. However, in this paper we summarize our efforts towards improving memory system performance. Current trends suggest that this will be a dominating issue for architectures of the near future. Memory latency is continuously increasing in relation to processing speeds. With the addition of wider, more aggressive pipelines, the latency cost of a single memory request is up to hundreds of instruction execution opportunities. And while growing transistor budgets allow more memory to be integrated on a chip and reduce the probability of off-chip traffic, new workloads are more than capable of exceeding these allowances. Future architectures will need greater data cache and off-chip bandwidth, lower latency cache access, better automatic management of large structured data sets, and faster inter-processor memory communication.

The rest of the paper presents synopses of four recently proposed program based techniques that attack these important aspects of memory system performance. We show how program structure information can be used to prefetch classes of data structures whose access patterns defy conventional address prediction and prefetching techniques. A second kind of information can be used to overcome the load issue delays introduced by address disambiguation and cache access. A third kind helps to expedite data sharing pattern detection in shared memory multiprocessors.

2 Rationale for Using Program Structure

The micro-architectural memory techniques of today, like cache hierarchies and prefetch engines, achieve their performance objectives by exploiting traditional notions of *program behavior*, like spatial locality, temporal locality, and address stream predictability. The future innovations

we speak of require that new aspects and parameters of program behavior be defined, studied, and exploited.

Although seemingly unrelated, the problem of branch prediction provides a good illustration of the kind of fundamental changes in approach we suggest. For almost a decade, branch predictors were based upon analyzing the history of a particular branch in isolation. A major leap in performance occurred when it was realized that the outcome of a branch was correlated not only with its own outcome history, but also with the previous outcomes of *other* branches. The injection of some notion of *program structure* into the process was the catalyst for a wave of powerful innovation in branch prediction. While branch prediction has benefitted from the incorporation of program structure notions, memory hierarchy design has, for the large part, ignored such information. If program structure, in this case the relationships between instructions that access memory, is taken into consideration, better designs may result.

We define program structure in general as the control and data dependence relationships among instructions or instruction groups such as basic blocks, loop bodies or entire functions. There are two fundamental reasons that underlie the predictive and computational power of program structure information:

- *Causality.* We refer to program structure as *primary* information. *Secondary* information refers to all observable program behavior, from branch outcomes to memory references and data. Program structure (primary) information produces all observable (secondary) program behavior, from branch outcomes to memory references. Current methods collect, analyze, and exploit different kinds of secondary information in ad hoc ways. Program structure information can be collected in a more unified way and then used to recreate pieces of secondary information on demand.
- *Stability.* At least on modern computers, program structure is invariant. Captured and analyzed once, it can be used repeatedly with confidence, regardless of a particular data context.

Causality and stability allow program structure to form the information basis for a great variety of micro-architectural methods. They also endow these methods with several inter-related properties that increase robustness, scalability and efficiency:

- *Early availability.* Associating information with data implies that in order to access the information, the same data must be available. In current pipelines, and depending on the particular kind of data involved, this may not occur until a forwarded value has been received or the instruction has generated an address. Late availability precludes optimizations that must occur early in the pipeline. Technically, program structure information, which can be accessed using instruction identity, can be retrieved before the instruction itself is fetched. Early availability enables optimizations early in the pipeline and increases the reach and effectiveness of other optimizations.
- *Compactness*. All programs are essentially data parallel. While the size of the program remains fixed (or at least relatively so), the amount of data processed may grow arbitrarily large. Consequently, any information associated with the program can be managed in a fixed (and relatively small) amount of space. On the other hand, storage required to manage data-associated information can grow to sizes proportional to the data set.
- *Learning amortization*. Associating information with data requires that the information be *learned* for every data item. As a result, data must be accessed at least once before the given technique can be applied to it. For data that is accessed once (or once before the information is lost), this presents a problem. Program structure information is learned once per instruction and can subsequently be applied even to never before seen data.

With these properties and *a priori* advantages in mind, we revisit several aspects of memory system function and performance with an emphasis on the application of program structure information. Not surprisingly, we focus on the relationships among memory operations. In particular, we exploit *memory dependences* or *address dependences* which relate static sets of loads and stores which dynamically reference the same set of addresses. We also exploit *load value dependences* which track the use of loaded values.

3 Prefetching Linked Data Structures

As data sets continue to grow and relative memory latencies increase, proactive memory hierarchy placement via prefetching becomes increasingly important. Hardware data prefetching is usually driven by address prediction which in turn relies on patterns extracted from the address stream. While address prediction is effective for some classes of structured data (e.g., arrays), others continue to pose problems. Linked data structures (LDS) are especially troublesome. Although currently less common than arrays, LDS may become more prevalent with the widespread adoption of object oriented programming languages like C++ and Java. LDS traversal involves long chains of dependent memory accesses. When they miss in the cache, these LDS accesses typically form a critical

chain and serialize the program. Without parallelism, prefetching's effect at reducing operation latency becomes critical. However, LDS reference streams have little expressed arithmetic regularity and render conventional address predictors and prefetch engines useless.

With address based schemes largely ineffective, we use program structure information to construct a prefetching scheme that avoids explicit address prediction. One such technique is *dependence based prefetching* (DBP) [5] which, at a high level, isolates the program thread responsible for traversing the LDS and pre-executes it. Internally, DBP represents the thread as a collection of explicit data dependence relationships. By ignoring control flow, this representation allows the thread to be replayed off the critical processing path and without the overheads and uncertainties of sequencing. To prefetch, we obtain an LDS root address and allow the representation to unroll the rest of the structure. As it executes, data it touches are prefetched into the cache. Dependence based prefetching is fundamentally different than conventional address-based prefetching. Rather than generating addresses by guessing statistically (which is what we would do without program structure information), we identify the program components and dependences that perform the desired operation and mimic them. DBP is quite effective for prefetching LDS with results comparable to, or better than, the best known software schemes.

The internal representation that drives DBP is nothing more than the data dependence relationships between the instructions in the program. Specifically, since LDS traversal is characterized by chains of loads that feed other loads, we capture the relationships between static load pairs. Once these dependences are established, they can be queried using producer load identity to obtain the identities and descriptions of all consuming LDS loads. This representation allows the processor to speculatively *instantiate* instructions in a dataflow manner. The whole process can be viewed as a form of super-aggressive scheduling that is not restricted to selecting instructions from within a sequential window.

Dependence based prefetching is an example of a technique that is *enabled* by program structure information. Although address based counterparts of this technique may be constructed, their effectiveness at predicting LDS addresses are likely to be low.

4 Streamlining Memory Communication

In addition to storing large amounts of structured data, memory is also used as an inter-operation communication device. In load-store architectures, stores take values produced by creator operations (def) and write them in memory so that subsequent loads may read them and pass them on to other operations (use). We use program structure information to streamline this communication mechanism.

Conventional memory communication is a laborious and somewhat inefficient process. A store computes an address and deposits the value into the memory hierarchy. To pick up that value, a subsequent load must compute an address, wait for all intervening store addresses to become available, and finally access the memory hierarchy if there were no conflicts. Add to this the fact that the store and load are just proxies and what should be a simple act of passing a value from one instruction to another is unnecessarily but inevitably delayed because of the indirect way of naming the communication channel.

Although inefficient, memory communication is a highly structured activity. Loads do not communicate with arbitrary stores or vice versa. Rather, static loads and stores

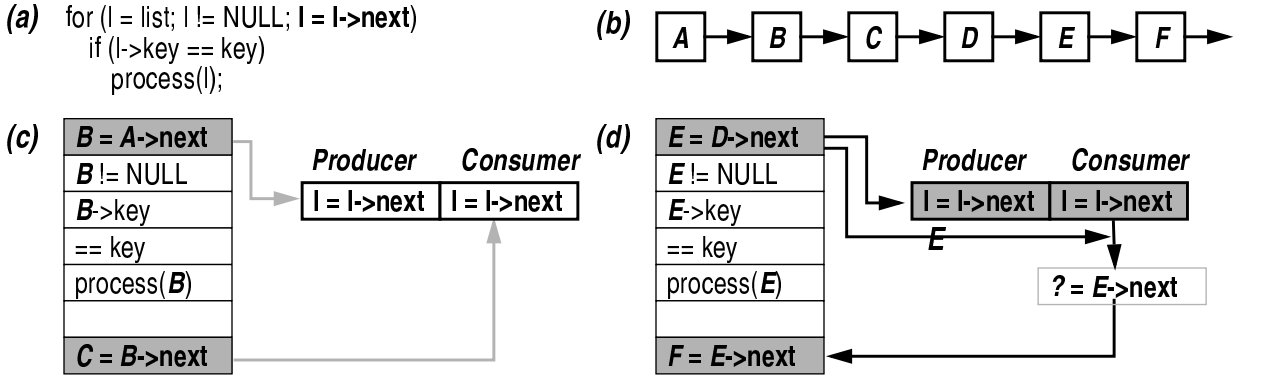

Figure 1. Dependence Based Prefetching. *(a) A list traversal code with the induction load in bold. (b) The traversed list with letters as node addresses. (c) During the first several loop iterations, the dependence predictor learns the identities of the LDS loads and constructs a representation of their dependence relationships. (d) During subsequent iterations, the dependence representation is consulted to spawn instances of the appropriate loads as prefetches.*

are partitioned into small *communicating groups* with a low frequency of inter-group communication. Information about this structure, which we call *memory dependence information*, can be used to circumvent the delays inherent in each component step: disambiguation, memory access, and *def-use* value transfer. As we explain, this structure can be exploited to alleviate some of these inefficiencies.

We begin by attacking the delays associated with address based disambiguation. With communication groups small and stable, and the frequency of inter-group communication low, the utility of a full address-based disambiguation procedure diminishes greatly. By dynamically identifying communicating groups, group membership information can be used to assess load issue status without the need for calculating the addresses of all previous stores. *Memory dependence speculation* was introduced by Moshovos et. al. [3] in the Multiscalar context and later by Chrysos and Emer [1] in a superscalar environment. In addition to avoiding disambiguation, memory dependence information can be used to *synchronize* waiting loads with the stores they depend on, avoiding unnecessarily long issue delays for loads that cannot issue immediately.

With disambiguation out of the way, memory dependence information can be used to eliminate memory access. To do this, we use the memory dependence tag (the communication group tag) to name the communication channel. We then pass values directly from store to load using that name (as if it were a register location). Since a communication group may contain multiple static stores, the actual store instance used in any single communication is determined dynamically in a process that is reminiscent of (not surprisingly) register renaming. Specifically, a store places its value in a location tagged by the communication group identifier. All subsequent group loads that occur before the next group store pick up the value. The next group store overwrites the value, and the cycle repeats. As a surgical extension of the basic mechanism, it is possible to allow group loads (in addition to group stores) to deposit values in the communication channel. This provides low latency access to memory values in situations where multiple loads access the same memory location.

These mechanisms, simultaneously introduced by Moshovos and Sohi as *speculative memory cloaking* [4] and by Tyson and Austin as *memory renaming* [6], eliminate cache access and address calculation allowing stores and loads to communicate directly. These forms of streamlined, low-latency communication are enabled by associating communication channels with program entities like loads, stores, and communication groups, rather than with addresses, which in and of themselves contain no explicit communication information.

Recall, in memory communication a store-load pair is simply a intermediate channel used to pass a value from some creating instruction (*def*) to another instruction (*use*) for use. The final reduction in communication latency is achieved when we speculatively convert these *def*-store-load-*use* dependence chains into more direct and efficient *def-use* ones, completely removing memory and its associated delays from the communication path. *Speculative memory bypassing*, due to Moshovos and Sohi [4], can be performed when both the *def* and *use* are simultaneously in flight. By endowing renaming logic with program structure information, *use* can be renamed using *def*'s

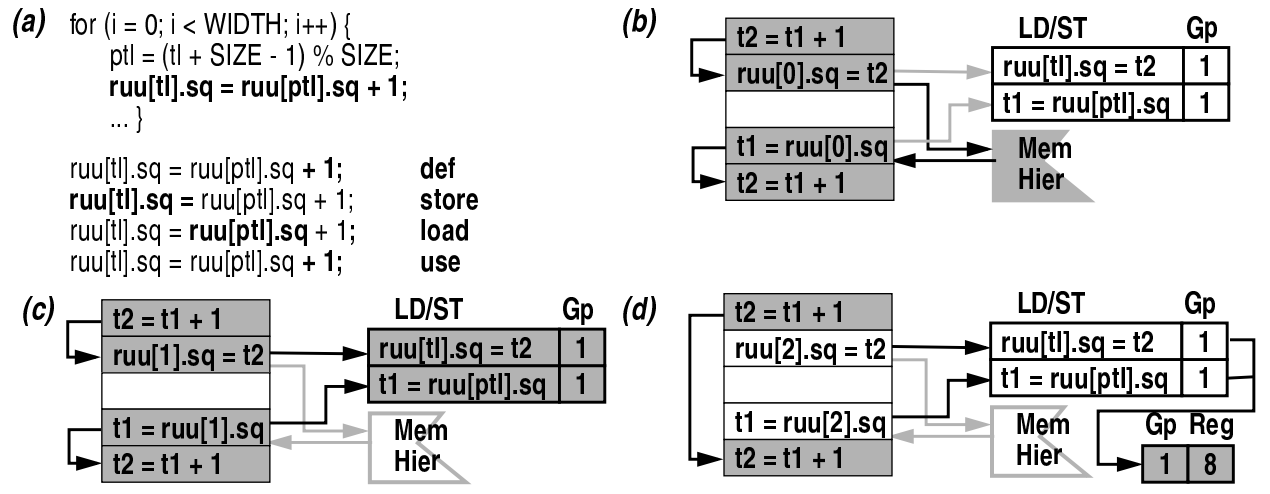

Figure 2. Speculative Memory Cloaking and Bypassing. *(a) loop code with DEF-store-load-USE dependence detailed. (b) First and second iterations: memory communication proceeds via cache, store/load dependence learned and marked with group tag. (c) Cloaking: store and load communicate via group tag. DEF-store and load-USE communication occur as usual. (d) Bypassing: If DEF-USE are both in-flight, load-store dependence is used to rename USE using DEF's (rather than load's) output physical register. Value then flows directly via the register.*

rather than load's physical register mapping, allowing the value to flow from *def* to use via a shared physical register.

Synchronization, cloaking and bypassing are examples of optimizations that are *enabled* by the early availability property of program structure information. Address based counterparts of these optimizations do not exist. The entire focus of these methods is to avoid address-based memory communication, relegating it to the status of a verification step and moving it off the processor's critical path.

5 Amplifying Cache Bandwidth

Increasingly parallel processing increases the demand for data cache bandwidth. Conventional methods for supplying additional bandwidth are *replication*, which is costly to implement, and *interleaving* which requires additional multiplexing logic and may still incur conflicts. We use program structure information to reduce bandwidth demands by shunting *transient* (short lived) memory values to a different structure and relieving the cache from having to service them.

We define transient memory values as ones that are either read or overwritten shortly after being written initially. Value transience is associated with a store instruction and is structurally determined by the temporal proximity of future loads and stores to the same memory location. Transient values are sent to the *transient value cache* (TVC). In order to be of any service, subsequent stores and loads to the same address must be directed to the TVC as well. This is done speculatively using the concept of a *transient value group*, an extension of the communication group concept used in cloaking implementations.

As in cloaking, transient value communication must be verified using the traditional memory path. Loads that are directed towards the TVC and do not find their value there must be sent to the cache, and similarly for stores that are not actually overwritten. However, this verification can occur after TVC access or writeback and only for mispredictions. In fact, Moshovos and Sohi [4] suggest that the TVC can be used to reduce the cache bandwidth demands of cloaking/bypassing verification.

Unlike dependence based prefetching, synchronization and cloaking, the TVC is address-based and as such does not require the use of program structure. However, the predictor used to direct instructions into the TVC is program based, an address based implementation would be both more costly and less accurate.

Cloaking, bypassing and the TVC are examples of how program structure information can be used to compactly represent of value attributes (here inter-reference times and lifetimes) and to manage the portion of the memory hierarchy close to the processor core. By capturing these and other value attributes, program structure can be used to reduce traffic, short-circuit long transactions, and supply added bandwidth in other memory hierarchy parts. In fact, program based designs may partition the memory system horizontally rather than vertically, with each partition specializing in handling values with certain attributes.

6 Improving Shared Memory Performance

The power of program structure information can also be brought to bear in the shared memory multiprocessing domain. Shared memory multiprocessors feature interprocessor, in addition to intra-process(or), memory communication. Writes by one processor must be propagated to other processors to provide communication and a consistent view of memory. For bandwidth reasons, cache coherence is typically implemented on a demand basis. However, specialized protocols that reduce both latency and traffic can be used once characteristic communication (data sharing) patterns are recognized. For instance, in migratory sharing, data items are first brought into local memory in read-only mode and incur a second coherence event and added latency when the block is subsequently written and the mode must be changed to modified. If a migratory sharing pattern can be predicted at the time of the read, the read request can be changed to a write request, saving the second long latency transaction. Producer-consumer sharing can be similarly optimized.

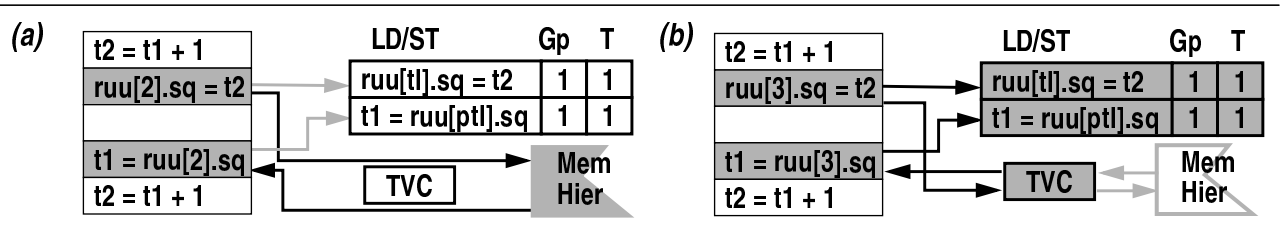

Figure 3. Transient Value Cache. *Using the cloaking/bypassing code example (a) First iteration: memory communication/verification consumes expensive data cache resources, temporal proximity of communicating store-load pairs is recorded. (b) Subsequent iterations: store-load pairs diverted to TVC, conserving data cache bandwidth.*

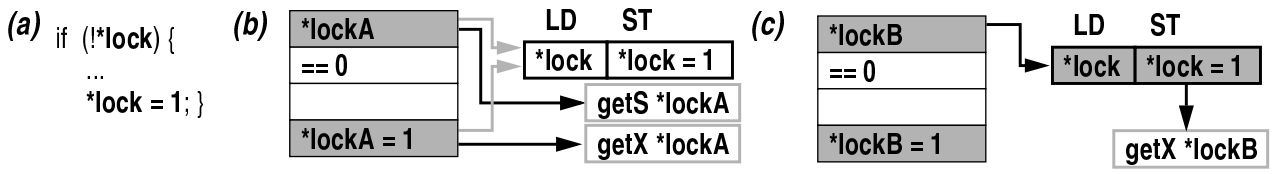

Figure 4. Migratory Sharing Pattern Prediction. *(a) Synchronization code in which a lock is first read initiating read miss protocol handling (getS) and subsequently written launching a write fault coherence action (getX). (b) First time through the code both coherence actions execute, dependence between load and store is marked. (c) Subsequently, using this information the load can initiate the corresponding store event, reducing store latency and network traffic.*

Instead of waiting for a consuming processor to ask for shared data, a producing processors can speculatively presend it, reducing latency at the consumer.

With these obvious benefits, much work has been devoted to efficient automatic detection and exploitation of these sharing patterns. Conventional detection schemes are almost exclusively address based, associating sharing pattern predictions with cache blocks. Although effective, these techniques are space inefficient and possess little predictive power since sharing information is maintained, and must be learned, on a cache block basis.

Our intuition tells us that program based predictors are better suited to this task. Processor communication is performed by the program, not the data. More fundamentally, communication is specified by program structure, the dependence relationships between reads and writes. With such static underpinnings, it is unlikely that the same code can engender multiple communication patterns at run time. Kaxiras and Goodman [2] have shown this to be the case, with compactness and learning amortization paying large dividends. Program based detectors associate sharing information with instruction groups and apply coherence-protocol optimizations based on participant instruction identity rather than cache block address. They require less storage than conventional predictors and experience the unoptimized learning phase only once per code group rather than once per communicated cache block.

One potential obstacle to the implementation of program based coherence optimization is that, although transparent to the user, it requires program structure information to flow through well established system interfaces that don't currently provide such capabilities. The coherence bus, for instance, observes only data addresses, not instruction identities. However, the near future promises to alleviate these problems. Processors will likely have integrated coherence-controllers and such integration will surely soften that interface boundary. Highly visible interface changes will not take place unless motivated by compelling performance gains. At the same time, the perception of high cost discourages research avenues with interface requirements, reducing the likelihood that performance justification will be found. This initial work certainly provides evidence that program structure information has the potential for significant performance improvements that will justify its inclusion in a system interface.

7 Conclusions and Future Directions

We believe that focusing our attention on *program structure information* will result in new techniques that are powerful, compact, and can scale to meet performance demands. The techniques described here, though effective for their intended purpose, are still in "first-cut" stages. We have also investigated applications of program structure information in the areas of branch prediction, instruction prefetching, scheduling and automatic restructuring of data for increased performance. We continue to study these techniques and to search for new enhancements.

References

[1] G.Z. Chrysos and J.S. Emer. Memory Dependence Prediction using Store Sets. In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153, Jun. 1998.

[2] S. Kaxiras and J. Goodman. Improving CC-NUMA Performance Using Instruction Based Prediction. In *Proc. 5th International Symposium on High Performance Computer Architecture*, pages 161–17, Jan. 1999.

[3] A. Moshovos, S.E. Breach, T.N. Vijaykumar, and G.S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proc. 24th International Symposium on Computer Architecture*, pages 181–193, Jun. 1997.

[4] A. Moshovos and G.S. Sohi. Streamlining Inter-Operation Communication via Data Dependence Prediction. In *Proc. 30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[5] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[6] G. Tyson and T. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proc. 30th International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.