

# Dynamic Concurrency Discovery for Very Large Windows of Execution

Jacob Nelson   Luis Ceze  
Computer Science and Engineering  
University of Washington  
{nelson, luisceze}@cs.washington.edu

## Abstract

*Dynamically finding parallelism in sequential applications with hardware mechanisms is typically limited to short windows of execution due to resource limitations. This is because precisely keeping track of memory dependences is too expensive. We propose trading off precision for efficiency. The key idea is to encode a superset of the dependences in a way that saves storage and makes traversal for concurrency discovery efficient. Our proposal includes two alternative hardware structures: the first is a FIFO of Bloom Filters; the second is an imprecise map of memory addresses to timestamps that summarizes dependences on-the-fly. Our evaluation with SPEC2006 applications shows that they lead to little imprecision in the dependence graph and find similar parallelization opportunities as an exact approach.*

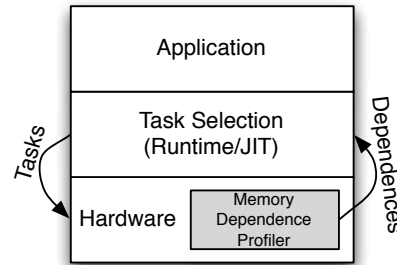
## 1. Introduction

Multicore microprocessors are ubiquitous today, and exploiting their potential requires parallel programs. But legacy sequential code still exists, and new sequential code is still being written. Parallelizing this code automatically (e.g., speculative multithreading) or manually benefits from information about where effort might be most fruitful. Data dependences define where concurrency, and thus opportunities for parallelization, exist.

Dependence detection to find parallelization opportunities has been done in many contexts before. Compilers can do it, but often must resort to conservative estimates without runtime information. Superscalar processors capture data dependences dynamically, but in windows of hundreds of instructions.

The goal of this work is to enable collection of memory dependences over windows of billions of instructions, with little resource usage and no performance overhead. We make three contributions. First, we introduce the concept of *imprecise dependence capture*, which trades accuracy in the captured dependence data for efficiency in the capture hardware. Second, we propose two hardware structures that use this concept to enable dependence capture over very large windows. Third, we demonstrate that these structures are able to find fruitful concurrency even with imprecision.

Our technique is applicable to a number of problems benefiting from dynamic memory dependence information, includ-



**Figure 1.** A high-level view of a system for dynamic speculative multithreading. This shows the context for this work, which enables efficient memory dependence collection for concurrency discovery.

ing dependence profiling, task recommendation for programmers [11], and speculative multithreading. Figure 1 shows one context in which our dependence collection structures could be applied: a system for dynamic speculative multithreading. The hardware would profile control and data dependences and communicate these to the runtime, which would use this information to partition the program into tasks for speculative execution. This paper focuses on capturing data dependences through memory from a sequential execution of a program; a complete system would combine this with other techniques to obtain a complete view of a program’s execution, select tasks, and ensure correct speculative execution.

## 2. Related work

There have been several pieces of work on dynamic collection of dependence information. We divide the space of prior work into three categories: (1) using profiler-collected information for program decomposition [6, 9]; (2) monitoring misspeculation events in speculative multithreading [7, 5]; and (3) hardware support for efficient on-line profiling [3, 12]. (2) and (3) are the most relevant to our work.

Prior work on speculative multithreading has explored using information on misspeculations to guide better task formation (e.g., [7, 4]). This indirectly produces information about dependences, since dependences are the cause of misspeculations.

However, the information that can be inferred is limited by the structure of tasks chosen beforehand. We approach this differently: we collect dependence information from the original sequential execution of a program, and therefore do not impose artificial restrictions on the dependence information collected.

Our work fits in the category of hardware support for efficient dependence profiling. To the best of our knowledge, the most relevant prior work was TEST [3], which proposed using timestamps per cache-line and a FIFO queue of memory references to keep track of recent accesses; dependences were computed based on that. The main limitation of that approach is the relatively short window size. Small windows limit the ability to find coarser-grain parallelism, which is what is desirable to tolerate the inter-core communication costs. We show instead that by allowing some amount of imprecision, we can design simple structures that are able to find coarse-grain parallelism in very large windows.

### 3. Model and Background

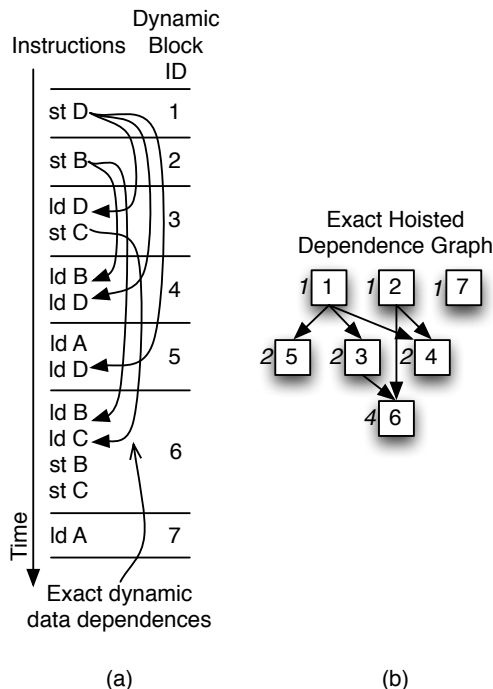
We model a program’s trace of execution as a directed graph. Nodes are dynamic instances of basic blocks; we refer to a particular dynamic basic block (DBB) with a *dynamic block identifier* (DBID), which is a tuple  $(BBID, timestamp)$  where BBID is the static basic block ID (i.e., the address of the first instruction) and the timestamp is a unique monotonically increasing value that identifies ordered instances of basic blocks. We assume all instructions execute in unit time, so the time to execute a DBB is given by its instruction count.

Edges are dynamic data dependences through memory. They impose an order on the sequence in which DBBs can be executed. By executing each DBB as early as possible in this order, we can find a lower bound on the execution time for a parallel version of the code. Some path through the graph will be the longest, defining the minimum execution time; this is the *critical path*. The ratio of the overall instruction count to the length of the critical path is the graph’s *available parallelism*.

Figure 2(a) shows a sequence of instructions divided into DBBs, along with the instructions’ exact dependences. We can reorder the sequence of execution so that each DBB runs as early as the dependence graph allows; Figure 2(b) shows this *maximally hoisted* graph with exact dependences. The critical path includes DBBs 1, 3, and 6 for a length of 7.

We chose to work at basic block granularity to ensure that no predefined task structure affects the data dependence information collected. Also, note that we only consider data dependences through memory in this work, since they are the hardest to detect and the biggest limiting factor in exploiting coarse-grain parallelism.

It would be ideal to identify these data dependences exactly. This is possible for small windows: simply store a history of writes along with DBIDs, and for each read, search for previous writes with matching addresses. But this fails for larger windows: some of the benchmarks we use for evaluation have millions of active variables. On a modern 64-bit machine, the history could require gigabits of on-chip storage. A more efficient solution is needed if we want to enable concurrency discovery in very large windows.



**Figure 2.** A sequence of executed instructions and their data dependences, split into dynamic basic blocks (a). The resulting maximally-hoisted dependence graph is shown with the blocks’ instruction counts (b).

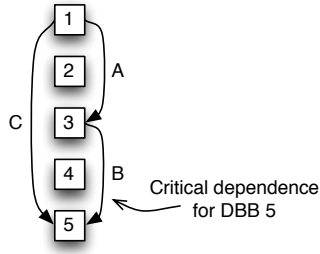
### 4. Imprecise dependence profiling

We will trade precision for resources and performance in computing the dependence graph. By capturing dynamic dependences that are imprecise in restricted ways, we may reduce storage requirements and search overhead while still capturing the important parts of the graph.

Our starting point will be the probabilistic set-inclusion data structure known as a Bloom Filter [1], whose clean mapping of set operations to hardware primitives has been exploited before [2, 10]. Bloom Filters may return false positives but will never return false negatives; this will be the source of our imprecision.

A Bloom Filter is a vector of  $m$  bits, along with  $k$  independent hash functions mapping a key to some subset of the bits of the vector. As in [2], we implement the  $k$  hash functions by partitioning a permuted key into  $k$  bit fields and applying an  $n$ -to- $2^n$  binary decoder to each  $n$ -bit field; the concatenation of all of these decoder results is the Bloom Filter’s bit vector. The size and position of each of the  $k$  bit fields, along with the choice of permutation, determines the Bloom Filter’s rate of false positives.

We present two structures for dependence detection. The first, called a *Bloom History*, is straightforward but resource-hungry; the second, called a *Bloom Map*, extends the first in



**Figure 3.** A dynamic data dependence graph. All blocks have the same instruction count. While DBB 5 has dependences from DBBs 1 and 3, the dependence from DBB 3 most limits concurrency.

a way that maps better to hardware but allows more imprecision. We are concerned only with the dependence detection hardware itself; we assume the processor provides information on the currently-executing DBB as well as a buffer to store discovered dependence edges until the system can process them.

To give useful results, our dependence collection structures must avoid too much imprecision about important dependences. Our structures aim for accuracy around the dependences that most limit concurrency near each DBB. One such *critical dependence* is shown in Figure 3. The Bloom History captures a superset of all dependences, ensuring critical dependences are also captured; the Bloom Map tries to capture only these critical dependences.

#### 4.1. Bloom History

A simple starting point is to encode the write set and read set of each DBB as Bloom Filters and store a history of encoded write sets; if the read set Bloom Filter for a DBB intersects with one of the write set Bloom Filters for a previously-executed basic block, a dependence exists between them. This is a *Bloom History*.

For each DBB, the Bloom History performs three steps. As the block executes, addresses read by the block are collected in a Bloom Filter, and addresses written by the block are collected in another Bloom Filter. Then, the read set Bloom Filter is intersected with each of the previous write set Bloom Filters; any intersections are recorded as dependence edges in an edge buffer. The contents of the read set Bloom Filter are not needed after this search is complete. Finally, the write set Bloom Filter is added to the write history, and the next DBB is executed.

Figure 4 shows a simple example with two hash functions (1-bit fields of the address) stored in 4-bit Bloom Filters. The code is the same as in Figure 2. Figure 4(c) shows the Bloom Filter encoding for addresses. Figure 4(e) shows the contents of the Bloom History after all DBBs have executed. Figure 4(b) shows the read set Bloom Filter values formed to search for intersections in the Bloom History for each DBB; these values are needed only during DBB execution and are not stored.

DBBs 1 and 2 perform only writes, updating the Bloom History. DBBs 3 and 4 demonstrate the Bloom History finding dependences that match the exact graph: DBB 3’s read set is compared with the Bloom History entries for DBBs 2 and 1, where a dependence is found; DBB 4’s read set is compared with the entries for DBBs 3, 2, and 1, finding dependences from DBBs 2 and 1. DBB 5 shows imprecision; even though address *A* is never written in the window of execution, the combination of addresses *A* and *D* read by DBB 5 (when ORed together as part of the Bloom Filter set union) produces a read set that intersects imprecisely with the write sets of DBBs 2, 3 and 1. This is *read aliasing*. The Bloom Filter for DBB 6’s writes to addresses *B* and *C* intersects with DBB 7’s read of address *A*, illustrating *write aliasing*.

Figure 4(d) shows the resulting maximally-hoisted dynamic dependence graph. It shares nodes 1, 3, and 6 with the exact graph but also includes 7, increasing the critical path length to 8.

Since we cannot create an arbitrarily large Bloom History, we must collect dependence information in windows. This limits the length of dependences we collect, and therefore may miss long dependences, such as the ones between iterations of large outer loops.

**Bloom History Hardware.** The structure of one potential hardware Bloom History is shown in Figure 5. It is organized like a shift register, storing a DBID and write set for each DBB. As execution progresses, the read set of the currently-executing DBB is intersected with all previous write sets in parallel; when the DBB finishes executing, edges that have been identified are latched in the edge buffer and the new write set and DBID are shifted in.

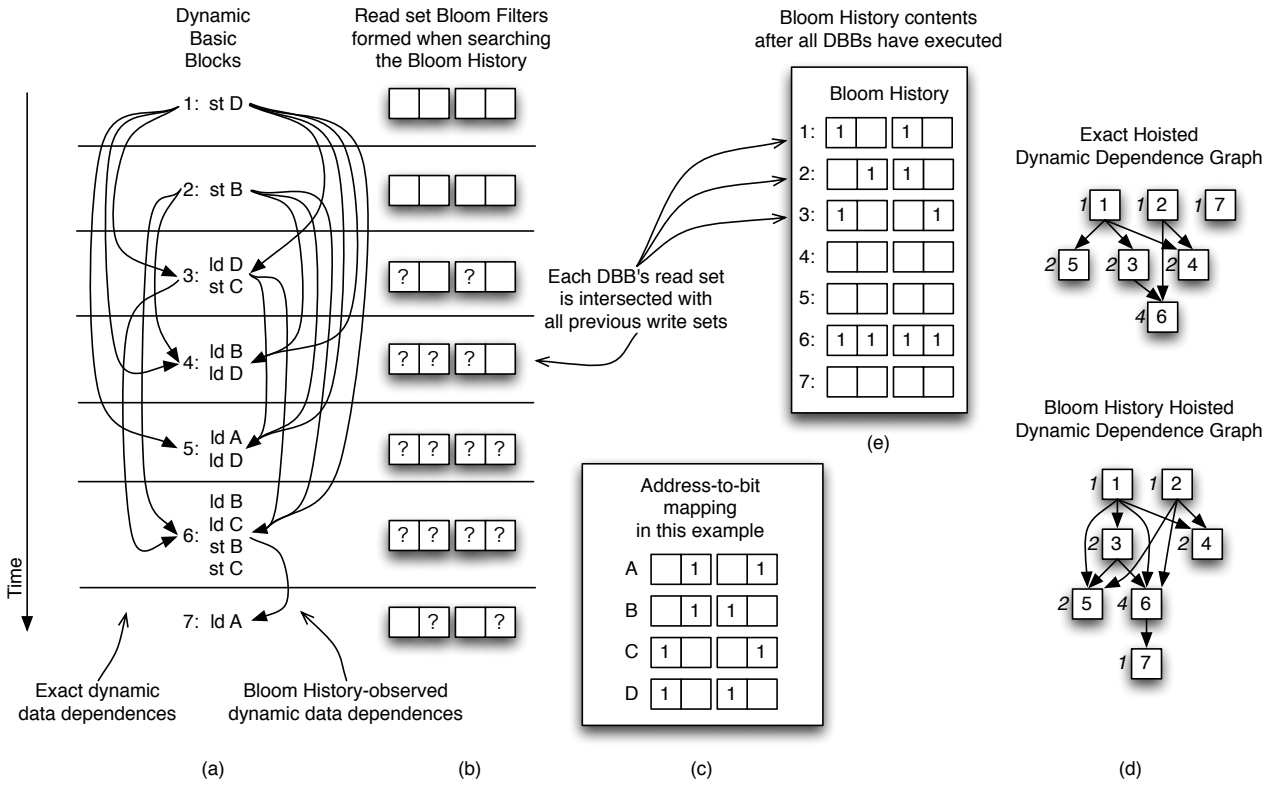
Read aliasing can be avoided by encoding and searching for each read address individually. We use this technique in our evaluation.

#### 4.2. Bloom Map

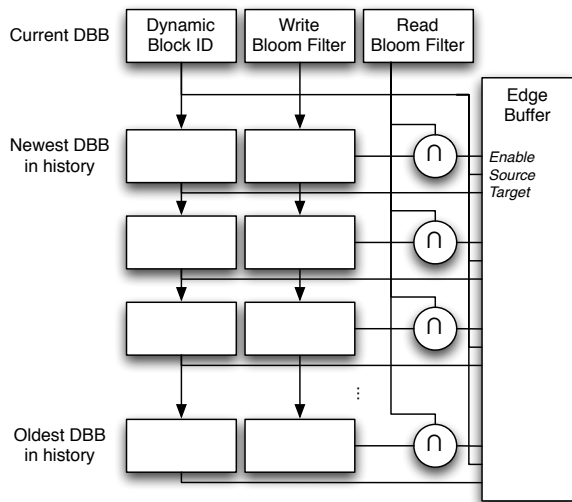
The Bloom History captures critical dependences, but it also captures many other dependences that do not limit concurrency. The Bloom Map attempts to focus its storage on the critical dependences of the graph. For a DBB on the critical path, one or more of its dependences forces it to be on the path. Often this is the DBB’s most recent (or shortest) dependence; the Bloom Map tries to store only that dependence.

The Bloom Map is a straightforward extension of a Bloom Filter that allows us to store a value at a key’s hash locations. Rather than setting bits, it stores the dynamic block identifier tuple (*BBID*, *timestamp*). When the Bloom Map is queried with an address, it returns the matching tuple with the most recent timestamp. This gives us essentially a “view” of the Bloom History: for each reading basic block, the Bloom Map tries to return the shortest dependence that the Bloom History would have found.

A Bloom Map consists of *k* arrays of (*BBID*, *timestamp*) tuples. Each array is indexed by a different hash function. Just as a Bloom Filter uses independent hash functions to reduce the rate of false positives, the Bloom Map uses multiple arrays to reduce imprecision in returned dependences. If all arrays



**Figure 4.** A sequence of executed instructions and their data dependencies, split into dynamic basic blocks (a) and observed by a Bloom History (b, e) with hash mapping shown in (c). The resulting precise and imprecise maximally-hoisted dependence graphs are shown with the blocks' instruction counts in (d).



**Figure 5.** Bloom History architecture.

agree that a dependence may exist, one of the stored dependence sources must be selected as the result. Careful design of this selection helps avoid imprecision in the returned dependences.

Figure 6 illustrates a Bloom Map with two arrays. During a write operation, we use the fields of the data address to index into each of the Bloom Map's arrays. Then we write the tuple into each selected array element, discarding any previous value. Reads are more involved: to process a single read, we use the fields of the read address to index into the Bloom Map's arrays. Then we must choose one of the selected tuples to return. Since a write to the address would have overwritten the tuples in all the arrays, by selecting the tuple with the oldest timestamp, we emulate returning the newest matching write from the Bloom History.

It is possible to process multiple reads, but an additional choice is required. First, since one hash function may point to multiple tuples in an array, we must choose one of them as the result for that array. The indexed tuples in an array may have been written by correct writes at different times; if we want to approximate returning the newest dependence to a reading DBB, we must choose the newest tuple in each array. Then we can again choose the oldest tuple between the arrays, since any correct dependence source must have updated all the arrays.

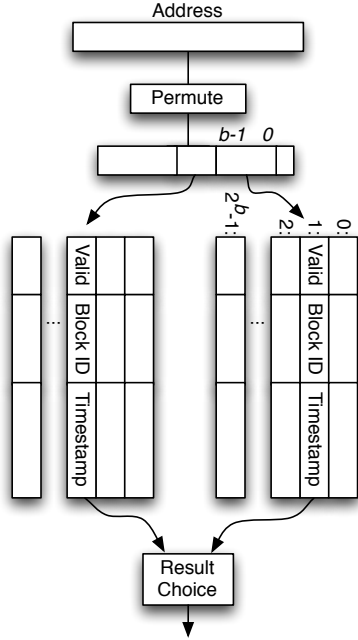


Figure 6. Structure of a Bloom Map.

Figure 7 shows a Bloom Map with two arrays observing a simple sequence of memory operations. The code, shown in Figure 7(a), is the same as Figure 2; Figure 7(c) shows the address mapping. Since a single Bloom Map is used for the whole window, in Figure 7(b) we show the contents of the Bloom Map whenever a DBB’s writes cause an update. We show only the timestamp of the DBIDs stored in the Bloom Map. Arrows indicate which cells of the current Bloom Map state are queried by a DBB’s read set and used in choosing a final result; these arrows are not stored.

The first two DBBs contain only writes, and thus only update the Bloom Map state. DBB 2 overwrites one of the tuples written by DBB 1. When DBB 3 queries the previous state of the Bloom Map, it must then choose between DBIDs 1 and 2; choosing 1, the older dependence source, yields a dependence that matches the exact graph. DBB 3 also writes, overwriting a relevant tuple for DBB 4’s reads, but again, choosing the older tuple between the arrays causes DBB 2 to be selected, matching the newest dependence in the exact graph. DBB 5’s query illustrates *shadowing*; the writes in DBBs 2 and 3 obscure the true dependence source. Read aliasing due to the inclusion of the (unmatched) load from *A* causes 3 to be returned, rather than 2 if the load from *D* was alone. Likewise, the writes in DBB 6 cause DBB 7’s query to find an edge from 6, even though the exact graph has no edge to 7.

Figure 7(d) shows the resulting maximally-hoisted dynamic dependence graph; again, it shares nodes 1, 3, and 6 with the exact graph but also includes 7, increasing the critical path length to 8.

**Result Choice Example.** Figure 8(a) demonstrates the three possible situations in choosing a result for a single read. DBBs 1 and 2 show the simplest case: if no intervening writes have overwritten the relevant tuples, any choice is acceptable.

If only some of the fields have been overwritten, the correct dependence source may still be found, as in DBBs 3 and 4. Since writes executed after the correct one will have newer timestamps, we may simply select the tuple with the oldest timestamp to find the correct dependence source.

If all the relevant tuples have been overwritten, we must return an imprecise dependence source. Returning any of the tuples will give us a dependence whose source DBB executed after the correct DBB, but the oldest tuple is closest to the correct tuple. In Figure 8(a), DBB 3 is a better choice than DBB 5 as a source for the dependence in DBB 6, since DBB 3 is closer to DBB 1, the correct source.

Figure 8(b) continues the example with multiple reads in each DBB. In DBB 7, it’s easy to see that we should choose DBB 5 as the dependence source, not DBB 3; choosing the newer dependence inside each array gives the right result. In DBB 9, by choosing the newest tuple in each array and the oldest tuple between arrays, we still get the correct source of DBB 5.

**Bloom Map Hardware.** Figure 9 shows a simple Bloom Map design that processes reads and writes individually. No associative structure or content-addressable memory is required for a Bloom Map; for both reads and writes, the arrays are indexed directly by fields of the address. Standard memory arrays can be used.

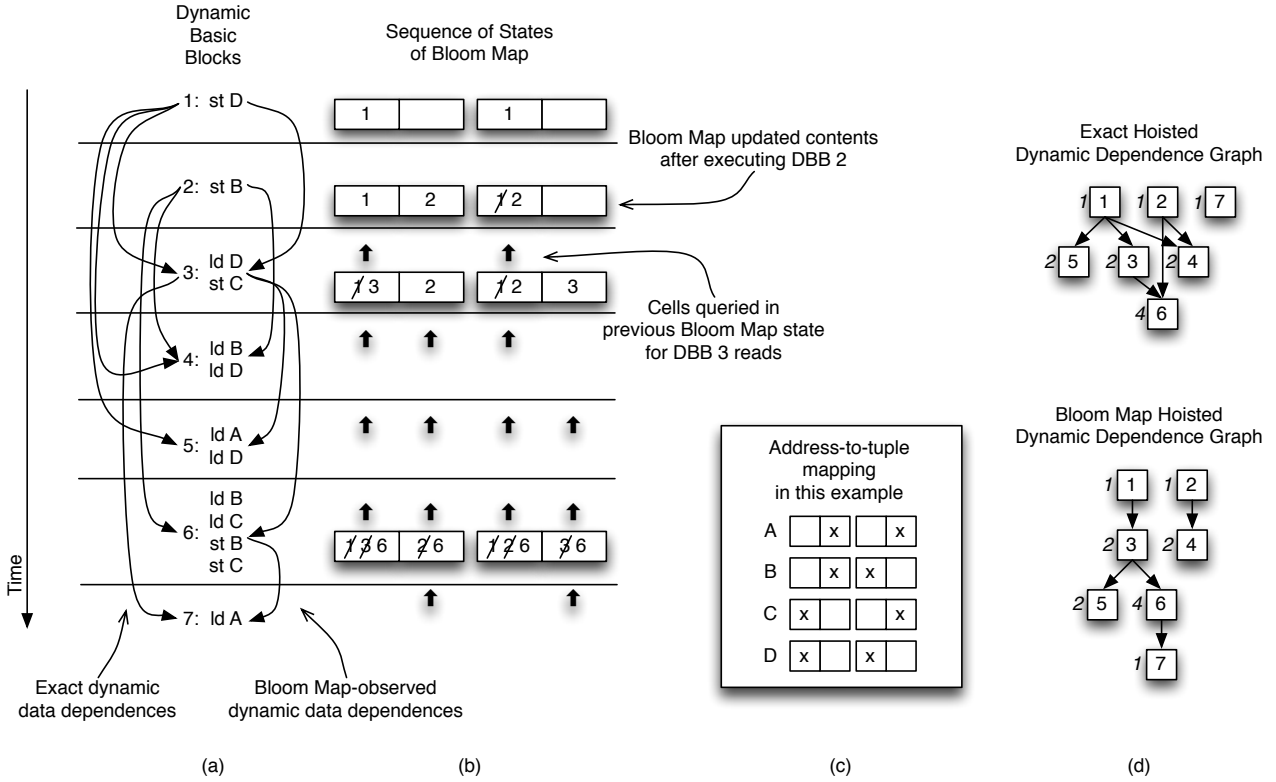
For each write address, we must store the DBID in the right location in each memory array. We do not want to compare the reads and writes of the same basic block, so we must insert a dynamic basic block’s writes after processing its reads. A conceptually simple way to implement this is to buffer the writes until the end of the dynamic basic block. Then, for each write, we use the fields of the write address to index into the arrays, storing the DBID tuple at each location.

Processing reads is similar. To find the correct tuple in each of the memory arrays, we use the appropriate field of the read address to index into the array. Once each array has chosen the relevant tuple, a final result must be chosen. We use a reduction tree made up of comparator-mux pairs. The comparator examines only the timestamp of the DBID. The output of the comparator drives the mux to pass along the correct tuple: the one with the oldest timestamp. If all arrays contained a valid tuple, the final result is valid.

Once the DBID tuple with the oldest timestamp ends up at the root of the tree, we have the dependence source: we store its DBID together with that of the currently-executing basic block (the destination) in the dependence edge buffer.

### 4.3. Discussion

The Bloom History and Bloom Map store data in different ways to capture different parts of the graph. The Bloom History stores write sets for each DBB; storage is allocated for each DBB whether or not it writes to an address. The Bloom Map stores potential dependence endpoints; storage is allocated for



**Figure 7.** A sequence of executed instructions and their data dependences, split into dynamic basic blocks (a) and observed by a Bloom Map (b) with hash mapping shown in (c). The resulting precise and imprecise maximally-hoisted dependence graphs are shown with the blocks' instruction counts in (d).

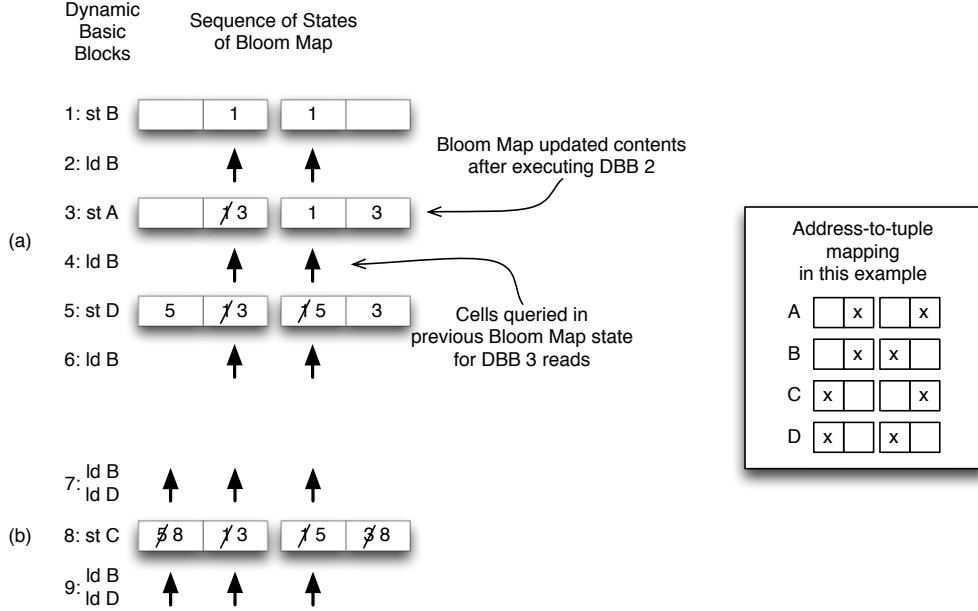
(and shared between) addresses that are written. The Bloom Map stores the most recent dependence source for an address, while the Bloom History stores all dependence sources for an address. For short critical dependences, the Bloom History wastes space storing dependences sources that don't matter. An analogy can be drawn to data encoding: the Bloom History stores a DBB's critical dependence in one-hot fashion, whereas the Bloom Map stores a DBB's critical dependence in a binary index encoding form. This suggests that a Bloom Map is likely to be more efficient in encoding dependences.

The Bloom History and Bloom Map each exhibit different types of imprecision in their captured dynamic dependence graphs. The Bloom History's imprecision within a window is limited to adding edges to the exact graph. Any edge in the exact dependence graph will be in the Bloom History's graph, but the latter may contain additional edges. Hoisting based on this graph will never lead to a critical path shorter than that in the exact graph: the Bloom History never over-estimates available parallelism. The Bloom Map emits only one dependence per DBB, and it will always emit a dependence no longer than the equivalent dependence found by an exact detector. Since we select the dependence with the oldest timestamp, and since timestamps always increase, any imprecision could only result

in an edge from a DBB executed after the correct one. But this imprecision may have consequences when computing available parallelism: shadowing may obscure a dependence that is part of the exact graph's critical path, leading the Bloom Map to find a shorter critical path and thus over-estimate concurrency. Imprecision may also lead the Bloom Map to find a longer critical path and under-estimate concurrency, just as with the Bloom History.

The Bloom History and Bloom Map both have limitations on the length of dependences they can collect. The Bloom History is limited by the window size, chosen at design time. The Bloom Map is limited by aliasing, and while the configuration of the Bloom Map's fields plays a role, this depends mainly on the series of addresses being stored.

Both the Bloom History and Bloom Map require comparison logic to identify dependences. The number of comparisons required for a Bloom History depends on the chosen window size. The number of comparisons required for a Bloom Map is independent of window size.



**Figure 8.** Examples for Bloom Map tuple selection. Only the timestamps of DBBs are shown. Cells show contents after the code for that DBB has executed. Arrows indicate which cells will be considered when computing the dependence for the reads in that DBB.

## 5. Evaluation

### 5.1. Setup

To explore the performance of our Bloom structures, we built a simulator using Intel’s dynamic binary instrumentation tool Pin [8]. We used SPECint 2006 as our benchmark set, running the test workload with base tuning parameters. We ran our experiments on a 64-bit Mac Pro running Ubuntu Linux, as well as on Large and Extra Large nodes at Amazon’s EC2 cloud computing facility. For each experiment, each benchmark was executed once, and the same trace of execution was used by each candidate structure.

The Bloom History structures have high simulation cost, so we implemented a sampling trace collector for experiments involving these structures. We chose a sample window size as large as practical; at each window boundary, we flipped a weighted coin to decide whether or not to sample in that interval. Dependences were tracked only within each sample interval, and the critical path was formed by concatenating the critical paths from each sampled interval. Available parallelism was calculated using only the instruction count from the sampled regions.

Addresses were compared at word granularity. Dependences were sampled in windows of 200,000 dynamic basic blocks to ensure that each sample window covered more than one million instructions. The sampled windows covered one percent of the program’s execution trace. An exact, hashtable-based software dependence detector was included in the runs to provide a base-

line for comparison. This detector observed the same trace as the Bloom structures and followed the same sampling rules.

We executed the SPECint 2006 benchmark set with a set of Bloom History and Bloom Map structures. We chose four Bloom Map configurations with bit budgets between 1Mbit (128KBytes) and 1Gbit along with Bloom History configurations for the larger three bit budgets, shown in Table 1. A small Bloom History could not be created, since even the 200,000 64-bit BBIDs required to cover our chosen simulation window would require 12Mbits of storage alone, exceeding the 1Mbit bit budget.

### 5.2. Results

We evaluate the performance of our mechanisms with two metrics. The first is edge error: our measure of how different the edges collected by our Bloom structures are from those in the exact dynamic dependence graph. Missing edges are included in this metric as well. This metric is defined as the ratio of edges not in both the precise and imprecise graphs (the symmetric difference) to total edges in both graphs, or in set-theoretic terms:

$$\text{edge error} = \frac{|\text{imprecise edge set} \Delta \text{precise edge set}|}{|\text{imprecise edge set} \cup \text{precise edge set}|}$$

We measured this edge error over the whole collected graph, as well as the graph of all edges from intermediate critical paths computed as the program executed. This latter metric is intended to show that the Bloom structure not only finds the crit-

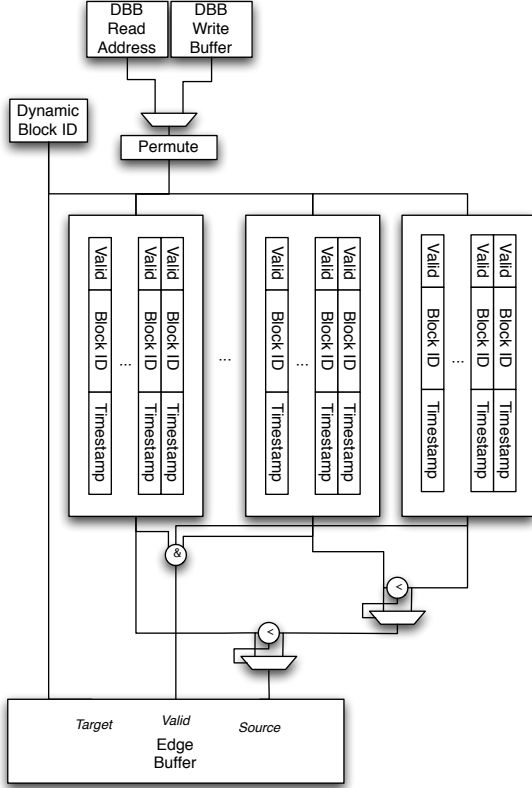


Figure 9. Bloom Map architecture.

| Name                  | Field configuration   | Size    |
|-----------------------|-----------------------|---------|
| Small Bloom Map       | 10 10 11 12           | 1Mbit   |
| Medium Bloom Map      | 15 15 15 15           | 16Mbit  |
| Medium Bloom History  | 2 2 2 3               | 16Mbit  |
| Large Bloom Map       | 18 18 19              | 129Mbit |
| Large Bloom History   | 2 2 2 3 4 6 9         | 129Mbit |
| X-Large Bloom Map     | 18 22 22              | 1Gbit   |
| X-Large Bloom History | 2 3 4 4 5 6 7 7 10 12 | 1Gbit   |

Table 1. Bloom History and Bloom Map configurations. Field configurations indicate width of bit fields taken from address; the rightmost number corresponds to the rightmost field, and fields are taken contiguously from the rightmost bit. Size indicates approximate number of one-bit storage elements required for the structure. All configurations used the identity permutation.

ical path after execution is finished, but that it also tracks the critical path during execution.

Figure 10 shows the overall edge error metric for the dependence graphs captured by the Bloom structures. We make two observations. First, the larger the structure, the smaller the edge error. This is untrue only for the X-Large Bloom History with

the benchmark `astar`, due to aliasing—a different configuration might eliminate this. Second, the Bloom Map yields lower edge error per bit than the Bloom History in all cases. The larger two Bloom Maps achieve zero edge error for a number of the benchmarks.

Figure 11 shows the critical path edge error metric for the dependence graphs captured by the Bloom structures. The edge error over intermediate critical paths is generally significantly less than the edge error over the entire dynamic dependence graph: our Bloom structures succeed at focusing their limited resources on the important dependences. In particular, the Small Bloom Map has a maximum critical path edge error of less than 25% even when the maximum overall edge error is nearly 90%.

The second metric we use to evaluate the Bloom structures is available parallelism: the ratio of the instruction count in the sampled trace to the length of the critical path through a maximally-hoisted version of the dynamic dependence graph captured by the Bloom structure. The goal here is to ensure that graph captured by Bloom structures matches that captured by an exact analyzer, and not to compute potential speedup.

Figure 12 compares the amount of parallelism discovered by the Bloom structures in the SPECint benchmarks with that found by an exact analyzer. Again, we make three observations. First, the structures are often able to find the same amount of parallelism as the exact analyzer, even with some edge error. This implies that imprecision in the dependence graph is not obscuring the true critical path, or that the imprecision leads us to identify a similar critical path. Second, the Bloom Map is more effective per bit at finding parallelism than the Bloom History. Even the smallest Bloom Map is able to find a significant fraction of the available parallelism found by the exact analyzer. Third, deviations from the exact available parallelism are correlated with critical path edge error. On some benchmark, most Bloom structures exhibiting this imprecision show a decrease in available parallelism.

The smaller Bloom Maps on the benchmark `h264ref` shows an over-estimate of available parallelism. This is a consequence of the imprecision allowed by the Bloom Map—if aliasing obscures a key dependence on the critical path, the imprecise dependence returned in its place may create a shorter critical path and thus lead to an over-estimate of concurrency. Other dependence information would need to be combined with this data to ensure the graph contains all important ordering constraints for execution.

## 6. Discussion

The Bloom Map is clearly a better choice based on our simulations. It is able to observe the entire trace of a program’s execution, potentially finding long, outer-loop concurrency. Since all writes are inserted into a single structure, aliasing is likely, but all the area can be allocated to this single structure, rather than spreading it across the window as the Bloom History does.

Could the Bloom History ever be a better choice? It has the advantage that for exact dependences that fit within a sample window, it will always find those dependences, avoiding the over-estimation of concurrency (rarely) allowed by the Bloom

Map. But given the Bloom History's significantly higher complexity and area and lower accuracy per bit, a Bloom Map of equivalent size will probably provide better performance.

## 7. Conclusion

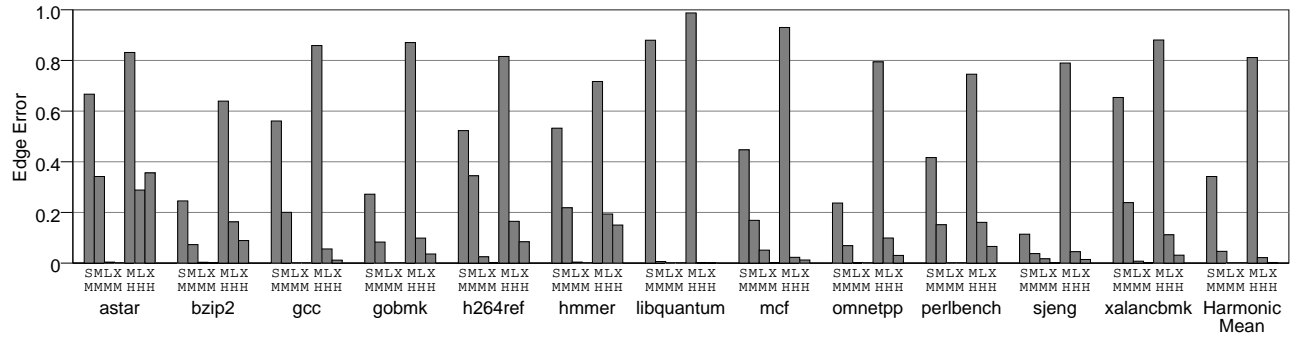
We have shown that allowing imprecision in dependence collection supports capturing important dependences with a small data structure. We described two efficient hardware structures for dependence collection that allow restricted classes of errors as a tradeoff for saving space and lowering collection overhead. Even with these errors, the structures are able to find similar opportunities for parallelization as a resource-hungry exact collector.

## Acknowledgments

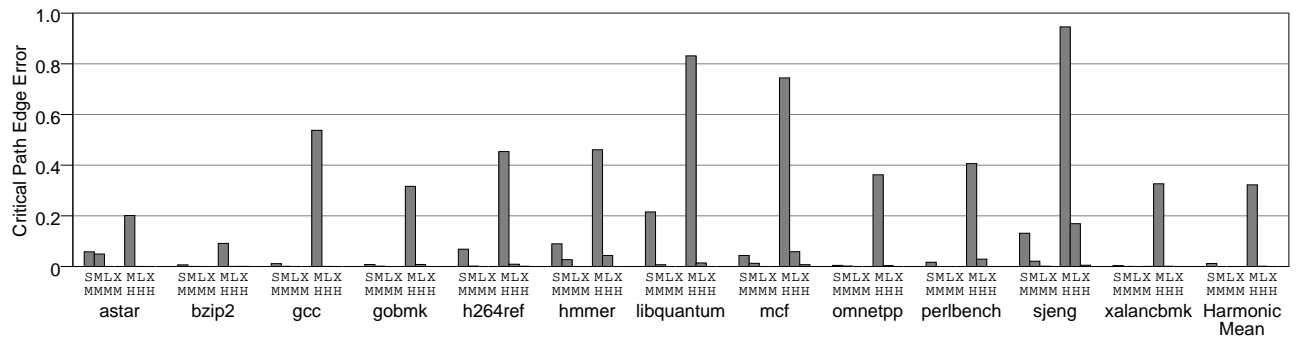
We thank Amazon.com for their donation of EC2 time. The anonymous reviewers and members of the WASP and Sampa groups at the University of Washington provided invaluable feedback in preparing this manuscript. Special thanks go to Doug Burger from Microsoft Research and Aaron Kimball from Cloudera for very helpful discussions in developing the ideas and the text. This work was supported in part by NSF under grant CCF-0702225 and a gift from Intel.

## References

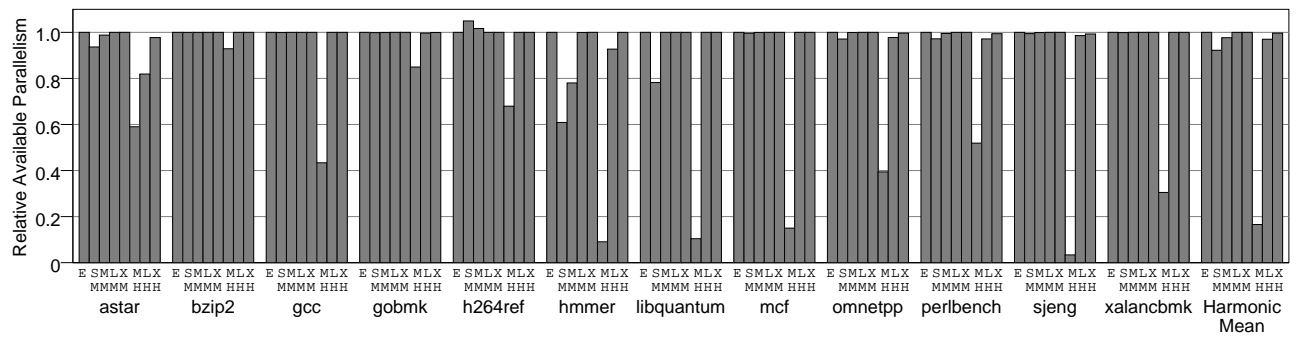
- [1] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, July 1970.
- [2] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *International Symposium on Computer Architecture*, 2006.
- [3] M. Chen and K. Olukotun. TEST: A Tracer for Extracting Speculative Threads. In *International Symposium on Code Generation and Optimization*, 2003.
- [4] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, 2000.
- [5] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software Behavior Oriented Parallelization. In *Conference on Programming Language Design and Implementation*, 2007.
- [6] T. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut Program Decomposition for Thread-Level Speculation. In *Conference on Programming Language Design and Implementation*, 2004.
- [7] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *Principles and Practice of Parallel Programming*, 2006.
- [8] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation*, 2005.
- [9] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. Tullsen. Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices. In *Conference on Programming Language Design and Implementation*, 2005.
- [10] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*, 2003.
- [11] C. von Praun, L. Ceze, and C. Cascaval. Implicit Parallelism with Ordered Transactions. In *Symposium on Principles and Practice of Parallel Programming*, March 2007.
- [12] C. Zilles and G. Sohi. A Programmable Co-processor for Profiling. In *International Symposium on High-Performance Computer Architecture*, 2001.



**Figure 10.** Overall edge error incurred by Small, Medium, Large, and X-Large Bloom Map configurations (SM, MM, LM, XM), and Medium, Large, and X-Large Bloom History configurations (MH, LH, XH) observing each of the SPECint benchmarks, as well as the harmonic mean of all the SPECint results. Good configurations have low edge error.



**Figure 11.** Edge error on intermediate critical paths incurred by Small, Medium, Large, and X-Large Bloom Map configurations (SM, MM, LM, XM), and Medium, Large, and X-Large Bloom History configurations (MH, LH, XH) observing each of the SPECint benchmarks, as well as the harmonic mean of all the SPECint results. Good configurations have low critical path edge error.



**Figure 12.** Available parallelism discovered in the SPECint benchmarks by Small, Medium, Large, and X-Large Bloom Map configurations (SM, MM, LM, XM), and Medium, Large, and X-Large Bloom History configurations (MH, LH, XH), as well as the harmonic mean of all the SPECint results, relative to available parallelism discovered by an exact analyzer (E). Good configurations match the exact analyzer.