Tolerating Delinquent Loads with Speculative Execution

Chuck (Chengyan) Zhao, J. Gregory Steffan, Cristiana Amza, and Allan Kielstra†

Dept. of Electrical and Computer Engineering
University of Toronto
{czhao,steffan,amza}@eecg.toronto.edu
†IBM Toronto Laboratory
IBM Corporation
kielstra@ca.ibm.com

ABSTRACT

With processor vendors pursuing multicore products, often at the expense of the complexity and aggressiveness of individual processors, we are motivated to explore ways that compilers can instead support more aggressive execution. In this paper we propose support for fine-grain compiler-based checkpointing that operates at the level of individual variables, potentially providing low-overhead software-only support for speculative execution. By exploiting this checkpointing support to improve the performance of sequential programs, we investigate the potential for using speculative execution to tolerate the latency of *delinquent loads* that frequently miss in the second-level (last level on-chip) cache. We propose both data and control speculation methods for hiding delinquent load latency. We develop a theoretical timing model for speculative execution that can yield up to 50% relative speedup. Our initial testing using synthetic benchmarks strongly supports this model.

1. INTRODUCTION

While today's computer hardware is characterized by the abundance of processor cores in multicore chips, the individual processors themselves are generally not much more aggressively speculative or out-of-order than previous designs. Instead the primary technique to cope with mounting latency to off-chip memory is multithreading, such as Intel's Hyper-threading and SUN's multithreaded Niagara processor: in these designs the long latency of an off-chip load miss can be tolerated by executing another thread for the duration of the miss. However, there is a dearth of threaded software—especially for desktop computing—which will limit the impact of solutions that depend on multithreading alone.

Prefetching is also a well-studied technique for addressing memory latency, via both hardware and compiler techniques. However, prefetching for irregular data accesses can be difficult, since irregular data accesses are difficult to predict and since there is a close trade-off between tolerating latency and increasing overhead and traffic. This environment underlines the importance of selective compiler techniques for tolerating memory latency.

One way to be more selective is to focus on *delinquent loads* (DLs) [8, 36]. A DL is a particular memory load in a program that frequently misses in a cache—typically the last-level cache on-chip. In other words, for many applications a small number of DLs contribute a large fraction of all last-level cache load misses. Hence DLs, should they be reasonably persistent across target architectures, may be a good focal point for compiler optimization.

1.1 Tolerating DLs with Compiler-Based Checkpointing

We propose a software-only method for checkpointing program execution that is implemented in a compiler. In particular, our transformations implement checkpointing at the level of *individual variables*, as opposed to previous work that checkpoints entire ranges of memory or entire objects. The intuition is that such fine-grain checkpointing can (i) provide many opportunities for optimizations that reduce redundancy and increase efficiency, and (ii) facilitate uses of checkpointing that demand minimal overhead, such as tolerating DL latency. We propose two methods of tolerating DL latency that exploit compiler-based fine-grain checkpointing to implement software-only control and data speculation. We evaluate the performance potential through both theoretical analysis and synthetic benchmark testing on

real machines.

1.2 Contributions

We make the following contributions in this paper:

- we implement a software-only checkpointing framework that leverages on compiler analysis and targets aggressive overhead reduction;
- we propose control and data speculative compiler transformations that will overlap with DLs;
- we propose a theoretical performance model of relative speedups and implement synthetic benchmarks whose evaluation results strongly support the modeling.

2. RELATED WORK

Our techniques are based on a wide spectrum of existing work in related areas, including prefetching [8, 24], multithreading [11, 38], checkpointing [12, 19], speculation [9, 13] and identifying DLs.

Panait at el. [36] investigated techniques to identify DLs statically. They examine code at the assembler level, categorize memory load instructions into various groups, and calculate a final weight based on profiling information obtained through training. They single out 10% of data loads that generate 90% of all cache misses. However, their approach is based on short-distance predictable memory behaviors. Thus their scheme is applicable only in isolating level-1 DLs. In addition, the identified DLs are memory locations in assembly format, which is nontrivial to map to source locations. Zhao at el. [55] introduced a lightweight and online runtime methodology to identify DLs. They observe that bursty online profiling and mini simulation of short memory traces can largely represent the underlying memory behaviors. Their simulation provides 61% overall accuracy with only 14% extra runtime overhead. However,they also introduces a 57% false positive ratio, a prohibitive number for any speculative compiler adopting their technique. We identify DLs through an efficient software cache simulator based on PIN [4, 23, 35]. It can be configured to deal with artificially many levels of cache and is capable of identifying DLs at any designated cache level. It provides service to map loadPCs back to source program locations, which is particularly useful to enable compiler optimizations.

3. COMPILER-BASED FINE-GRAIN CHECKPOINTING

Checkpointing [12, 19, 21, 40, 48, 49] is the process of taking a snapshot of program execution so that execution can rewind to that snapshot later if desired. Checkpointing has a wide range of uses and includes both hardware and software implementations. While proposed hardware-based solutions [2, 30] can perform well, they have yet to be adopted broadly in commercial systems. Software-only checkpointing solutions [19, 21, 37, 49] are therefore more immediately practical, although their inherent overheads can be prohibitive. In contrast with past work on coarse-granularity checkpointing based on copying large memory regions or cloning objects, in this section we propose a relatively lightweight compiler-based approach to checkpointing that operates at the level of individual variables.

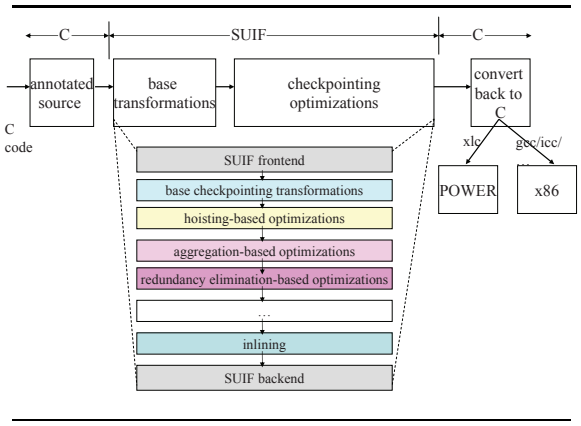

Figure 1: Checkpointing system overview

Overview Figure 1 presents a high-level overview of our checkpointing system. The system takes as input a `C`-based program, with annotations that indicate where a checkpoint region begins and ends, as well as code that decides whether the checkpoint should be committed or rewound. Our checkpointing transformations and optimizations are implemented as passes in the SUIF [3, 14] compiler, which outputs transformed `C` code that can then be compiled to target a number of platforms (currently x86 via `gcc` and POWER via IBM's `xlc` compilers). This source-to-source approach allows us to capitalize on all of the optimizations of the back-end compilers.

Undo-Log vs Write-Buffer The most important design decision in a checkpointing scheme is the approach to buffering: whether it will be based on *write-buffer* [16, 26] or alternatively an *undo-log* [18, 31]. A write-buffer approach buffers all writes from main memory, and therefore requires that the write-buffer be searched on every read. Should the check-

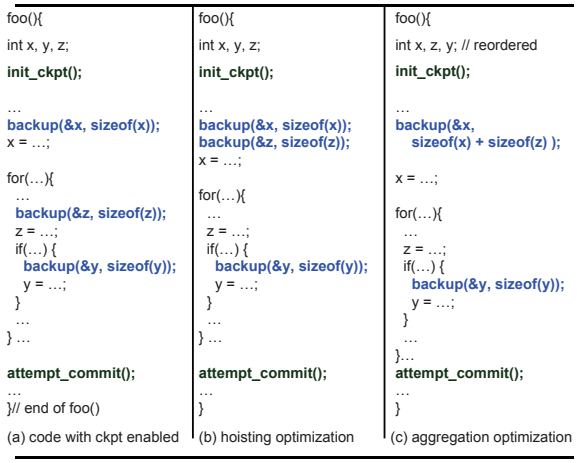

Figure 2: Fine-grain Checkpointing Optimizations

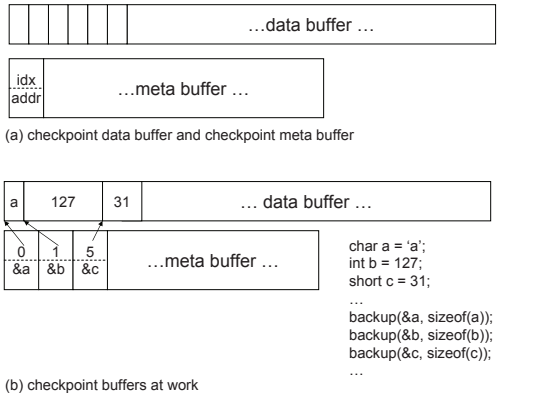

Figure 3: Undo-log buffering mechanism.

point commit, the write-buffer must be committed to main memory; should the checkpoint fail, the write-buffer can simply be discarded. Hence for a write-buffer approach the checkpointed code proceeds more slowly, but with the benefit that parallel threads of execution can be effectively checkpointed and isolated (e.g., for some forms of optimistic transactional memory [16, 28]). An undo-log approach maintains a buffer of previous values of modified memory locations, and allows the checkpointed code to otherwise read or write main memory directly. Should the checkpoint commit, the undo-log is simply discarded; should the checkpoint fail, the undo-log must be used to rewind main memory. Hence for an undo-log approach the checkpointed code can proceed much more quickly than a write-buffer approach. For this work, since we are considering only a single thread of execution with focus on performance, we proceed with an undo-log approach.

Base Transformation Given that we implement an undo-log based approach, the base pass of the checkpointing framework is to precede all writes with code to backup the write location into the undo-log. As illustrated in Figure 2(a), within the specified checkpoint region the variables `x`, `y`, and `z` are all modified and preceded with a `backup()` call. The `backup()` call takes as arguments a pointer to the variable to be backed up and its size in bytes. Figure 3 illustrates our initial design of an undo-log, where we have divided the undo-log into two structures: (i) a data buffer which is essentially a concatenation of all backed-up data values of arbitrary sizes; and (ii) a meta-data buffer which stores the length and starting address of each element. As an example, Figure 3(b) shows the contents of an undo-log after three `backup()` calls. When a checkpoint commits, we simply move the data and meta buffer pointers back to the start of each buffer; when a checkpoint must be rewound, we use the meta buffer to walk through the data buffer, writing each data element back to main memory. In future work we will more thoroughly investigate possibilities and trade-offs in the implementation of the undo-log.

Optimizations Our base transformation for fine-grain checkpointing provides significant opportunities for optimizations. Given the initial code shown in Figure 2(a), we can perform several optimizations. For example, as illustrated in Figure 2(b) a *hoisting* pass which will hoist the backup of any variable written unconditionally within a loop to the outside of that loop (variable `z` in the example); note that such hoisting would not be performed by a normal hoisting pass since the write to the variable is not necessarily loop invariant. Note also that we do not hoist variable `y` in the example since it is only conditionally modified—whether to hoist such cases is a trade-off that will be studied in future work. A second optimization is to *aggregate* `backup()` calls for variables which are adjacent in memory, potentially rearranging the layout of the variables to ensure that they are adjacent.[1] Aggregation reduces the overhead of managing adjacent variables individually (variables `x` and `z` in the example). We have implemented an in-

[1]Note that for a source-to-source transformation this isn't necessarily a safe optimization as the back-end compiler may further rearrange the variable layout—an implementation in a single unified compiler would not have this problem.

lining pass so that a `backup()` is not actually implemented as a procedure call but instead consists only of the bare instructions for performing the backup. In future work, we will also investigate redundancy optimizations to remove redundant and unnecessary `backup()` calls.

4. DELINQUENT LOADS IDENTIFICATION AND PERSISTENCE

DL Identification We identify DLs by profiling second-level (L2) cache misses using a cache simulator based on PIN [23] that we developed for this work. The PIN framework identifies each memory-access instruction from the application and directs them to a software cache model. Within each memory access, the software model captures necessary access signatures (read vs. write, effective memory address, length of data, etc.) and performs efficient cache simulation. The software cache model is easily configurable when dealing with various cache configurations, including the total levels of cache, cache size, cache-line size, degree of associativity, replacement policy, etc.

One compelling feature of this infrastructure is that, when a benchmark is compiled with debug information, it allows us to directly associate load and store instructions with their corresponding source code location. Hence the simulator can reliably map each load instruction that is responsible for a large fraction of L2 cache misses back to the offending source code location. In this paper, we will consider a particular load instruction to be a delinquent load if it is responsible for greater than 10% of all L2 cache misses for a program. We will also refer to the actual percentage of L2 cache misses as the *significance* of that delinquent load (i.e., a load that is responsible for all of a program's L2 cache misses would have a significance of 100%).

DL Persistence We use SPEC2000INT [10] benchmarks, compiled with compilers of various vendors (*gcc* and *icc*), versions (*gcc* 3.4.4, 4.0.4, 4.1.2, 4.2.4, 4.3.2, and *icc* 9.1), and optimization levels (*O0, O2* and *O3*) to study DL locations and properties. We configure the cache simulator with 2-level cache that covers a large variations of cache size, cache line size and degree of associativity.

Our initial investigation of all SPEC2000INT `C` benchmarks found that only a subset of the applications contain DLs. Within that subset, the DLs have the following persistent properties:

- the DLs are persistent across various L2 cache configurations (size, line size, ways of associativity), as long as the working set doesn't entirely fit into the L2 cache;
- the DLs are persistent across different compilers, including vendors, versions and optimization levels;
- the DLs are persistent across inputs (training or reference).

Interested readers should refer to [54] for further detail.

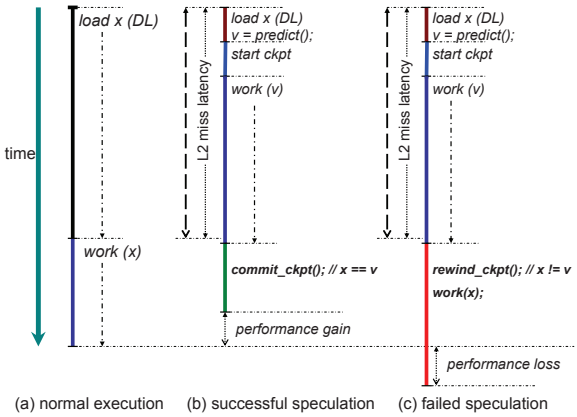

Figure 4: Overview of tolerating a DL with speculative execution.

5. TOLERATING DELINQUENT LOADS WITH SPECULATIVE EXECUTION

In this section we propose two techniques that leverage compiler-based fine-grain checkpointing to tolerate DLs, namely data and control speculation. For a single-threaded speculation, we must make a prediction about the resulting value of a DL and execute code that uses that prediction to make progress rather than waiting for the DL result value from off-chip. This approach exploits the parallelism provided by a wide-issue superscalar processor that can execute instructions with memory access in parallel. Ideally the latency of the DL is hidden when the prediction is correct, but execution can rewind and re-execute using the correct DL value should the prediction be incorrect.

5.1 Overview

Figure 4(a) illustrates the challenge presented by a DL: the L2 miss latency for a DL can be lengthy, and the computation that follows the DL (`work()`)

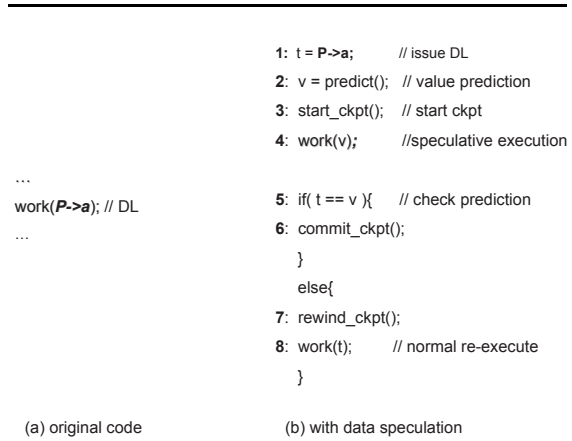

Figure 5: Tolerating a DL via data speculation.

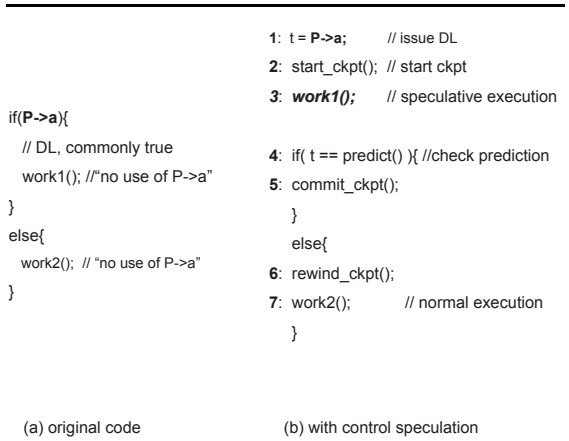

Figure 6: Tolerating a DL via control speculation.

likely depends on the DL's result value (`x`). Figure 4(b) provides an overview of how to tolerate a DL by overlapping the DL miss latency with speculative execution of the subsequent code using a predicted value (`v`). The DL is scheduled as early as possible, followed by the generation of a predicted value (`v`).

The computation proceeds using the predicted value (`work(v)`), with that computation being checkpointed to support execution rewind. When the computation is complete, we compare the predicted value with the actual value, and if they are equal then we can commit the checkpoint (as shown in Figure 4(b)). Ideally such a successful prediction and speculation will result in a performance gain relative to the non-speculative original code. Should the value be mispredicted, as illustrated in Figure 4(c), then we must rewind the checkpoint and re-perform the computation with the correct result value of the DL (`work(x)`). The combined overheads of checkpointing as well as rewinding and retrying the computation can result in a performance loss relative to the original code.

5.2 Data Speculation

The first method of tolerating DL latency that we evaluate is *data speculation* (*DS*) where we predict the result value of the DL and use it to continue execution speculatively, as illustrated in Figure 5. After issuing the DL as early as possible (*1*), predicting the DL's data value (*2*), starting the checkpoint (*3*), and speculatively executing based on that predicted value (*4*), we then attempt to commit the speculation. The commit process first checks whether the prediction was correct (*5*): if so then the checkpoint is committed (*6*), otherwise the checkpoint is rewound (*7*) and the computation is re-executed using the correct DL result value (*8*).

5.3 Control Speculation

Whenever the result value of a DL is used *solely* within a conditional control statement, as shown in Figure 6, we have an interesting opportunity: rather than predicting the exact result value of the DL we can instead merely predict the boolean result of the conditional—which ideally will more easily be an accurate prediction than predicting the exact result value. We call this form of speculation *control speculation (CS)*, which is essentially a special-case of data speculation. The speculative compiler transformations of tolerating control speculation are given in Figure 6.

Modern processors perform branch prediction and speculatively execute instructions beyond the branch—however this speculation is limited to the size and aggressiveness of the processor's issue window. With compiler-based control speculation we can ideally speculate more deeply, allowing greater opportunity for tolerating all of the latency of a DL.

6. PERFORMANCE

In this section, we give both a theoretical performance model and a practical evaluation of the proposed speculative techniques on real machines. We first present a mathematical analysis of the implicit DL memory overlapping model and give theoretical predictions of potential performance benefits. We show that the theoretical model predicts approximately 50% relative speedup. We then ap-

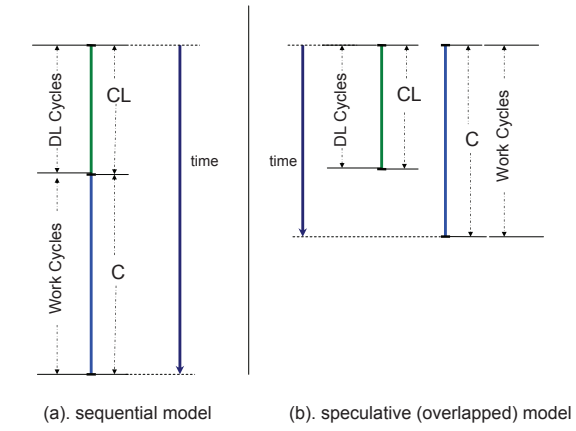

Figure 7: Ideal timing model

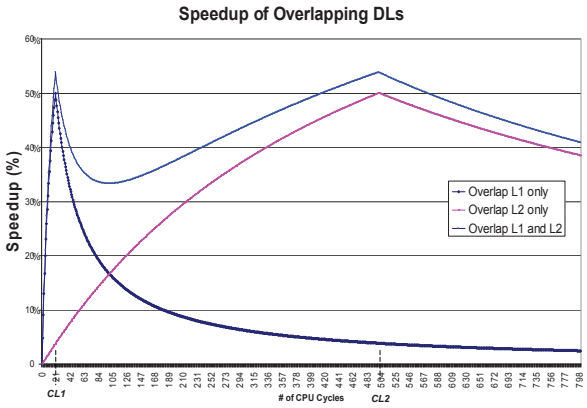

Figure 8: Relative speedup of ideally overlapped execution with DLs on various levels of cache

ply this model on synthetic benchmarks running on real machines and demonstrate that the relative performance gain of the synthetic benchmarks closely matches the theoretical prediction.

6.1 Theoretical Performance Modeling

Figure 7 illustrates the ideal timing model for overlapping execution with DLs. Figure 7(a) is the normal sequential model where the total execution time is the sum of both DL cycles and the work cycles whose continuation relies on the DL. This represents the conditions where the DL's value is immediately needed to allow execution to proceed, thus the program stalls until the DL value returns. Under the overlapped model (Figure 7(b)), the program continues with the predicted value while the memory system is serving the DL. This resembles a level of memory-level parallelism though there is no explicit parallel thread needed to fetch the DL. Thus the total execution time is the *maximum* of the two. This models the cases when either the DL's value not being immediately needed or the DL being used to make a predictable control-flow decision and therefore its precise value is less important.

Let CL denote the cycles of a cache miss (DL) and let C denote the cycles of work that overlaps with the DL, we have

$$T_{sequential} = CL + C$$

$$T_{speculate} = \max(CL, C)$$

Let S denote the relative speedup of overlapping execution with DL, we give the definition of S

$$S = \frac{T_{sequential} - T_{speculate}}{T_{sequential}} = \frac{CL + C - \max(CL, C)}{CL + C} \tag{1}$$

Thus the ideal theoretical relative speedup for only overlapping with only a L1 cache is

$$S = \frac{CL_1 + C - \max(CL_1, C)}{CL_1 + C} = \begin{cases} \frac{C}{CL_1+C}, & \text{if } C < CL_1 \\ \frac{CL_1}{CL_1+C}, & \text{if } C \geq CL_1 \end{cases}$$

Similarly the ideal theoretical relative speedup for only overlapping with only a L2 cache is

$$S = \frac{CL_2 + C - \max(CL_2, C)}{CL_2 + C} = \begin{cases} \frac{C}{CL_2+C}, & \text{if } C < CL_2 \\ \frac{CL_2}{CL_2+C}, & \text{if } C \geq CL_2 \end{cases}$$

In addition, we obtain the theoretical relative speedup for overlapping with combined L1 and L2 cache by aggregating individual speedups:

$$S = \begin{cases} \frac{C}{CL_1+C} + \frac{C}{CL_2+C}, & \text{if } 0 \le C < CL_1 \\ \frac{CL_1}{CL_1+C} + \frac{C}{CL_2+C}, & \text{if } CL_1 \le C < CL_2 \\ \frac{CL_1}{CL_1+C} + \frac{CL_2}{CL_2+C}, & \text{if } C \ge CL_2 \end{cases}$$

Figure 8 presents three theoretical relative speedup curves for overlapping with L1 cache only, with L2 cache only, and overlapping with combined L1-and-L2 cache respectively. It shows both the overall similarity and individual differences. For ease of comparison, we fix the L1 cache miss latency to 20 cycles (CL_1) and L2 cache miss latency to 500 cycles (CL_2).

The curve that overlaps with L1-only workload goes sharply to its peak from 0 to CL_1 (20) cycles in the beginning. Since the L1-miss-and-L2-hit cycles are relatively short, the curve has only limited room to stretch before reaching its theoretical peak, which is predicted to be 50% when the overlapped cycles (C) equal to L1-miss-and-L2-hit cycles (CL_1). The curve that overlaps with L2-only work can be treated as horizontally scaling the L1 curve to match with L2-miss-and-memory-hit cycles (CL_2) and its theoretical performance upper-bound is also 50%. Given ideal workloads, the two theoretical speedups can further combine and generate an aggregated effect that can cross the 50% threshold, presented as the CL_2-centered triangle-like area in Figure 8.

6.2 Micro Benchmarks

We developped a set of synthetic benchmarks for real-machine evaluation. This includes linked list (linklist), binary search tree, B-tree, red-black tree, AVL tree, and hashtable, etc. They behave similarly in that accesses to dynamically allocated data structures result in frequent cache misses (DLs). We use linklist as the representative for this initial study. We make each node in the linklist larger than the cache line size on the machine it evaluates. To exacerbate the situation, we randomize the starting address of each node, which helps undermine the hardware prefetcher. By adjusting the number of nodes in the linklist, we achieve the effect of either polluting only the L1 cache (L1-DL), or polluting both L1-and-L2 caches (L1L2-DL) through a single linklist traversal. The empirical list size we use is 4K nodes for L1-DL and 2M nodes for L1L2-DL, respectively. We use *RDTSC* [1, 50] for fine-grain time measurement.

The machine used for evaluating the benchmarks has a single-core 3.0GHz Pentium-IV CPU, with a 16KB 4-way set-associative L1 data cache, a 12KB 8-way set-associative L1 instruction cache, and a 512KB 8-way set-associative shared L2 cache. The cache-line size is consistent at 64B. Each measurement data point is the arithmetic average of at least 5 independent runs.

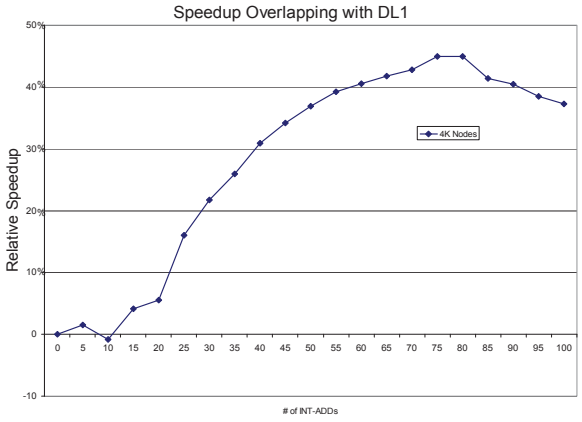

Figure 9: Relative speedup overlapping with L1-only DL on real machine

6.3 Micro Benchmark Performance

Figure 9 shows the relative speedup of overlapping L1 DL using linklist. The workload to overlap with DL is a loop performing accumulation of integer adds (INTADDs, shown on the x-axis), while the y-axis gives the relative speedup. Figure 9 is very similar to the theoretical prediction of L1 speedup curve given in Figure 8. It reaches its maximum of 45% while overlapping roughly 70 INTADDs.

When performing testing on real machines, a workload that pollutes the L2 cache must already have the L1 cache polluted. It is difficult to obtain the performance figure with a workload that overlaps with only the L2 cache (L2 DL). We thus focus on workloads that overlaps with L1-and-L2 (L1-L2) DL.

Figure 10 shows the relative performance result when overlapping with L1-L2 DLs. In stage 1, the curve reaches around 35% speedup at roughly 70 INTADDs. This agrees with our own measurement given in Figure 9 and it is the effect of mostly overlapping L1 DL. In stage 2, the curve maintains stableness over 35% with maximum reaching very close to the 50% theoretical peak. This closely matches the L1-and-L2 prediction given in Figure 8 where a wide range of 35%+ relative performance is expected after stage 1.

6.4 Challenge with Real-World Applications

We give theoretical predictions on performance analysis which overlaps with various levels of cache. We verify this claim with micro benchmarks that can reach very close to the theoretical peak. These results are obtained under ideal conditions that (i)

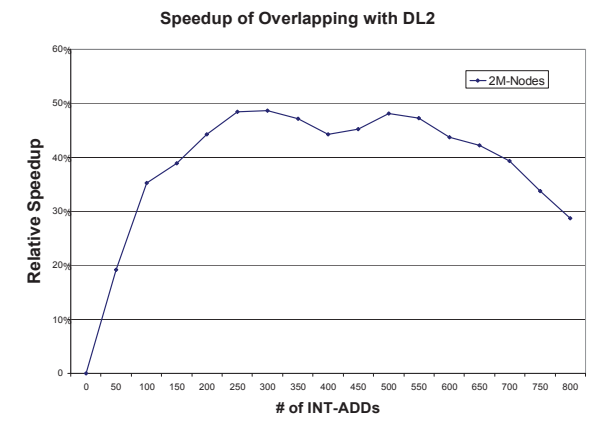

Figure 10: Relative speedup overlapping with L1-and-L2 DL on real machine

there is no need to do checkpointing because the workload has no global side effect (similar to a *pure* function), and (ii) there is no failed speculation because the involved predictor can yield 100% prediction accuracy. However, such ideal situations may not hold under non-synthetic benchmarks on real machines.

In the future, we plan to investigate the feasibility of applying the control and data speculation transformations we introduced in this paper to real-world applications (e.g., the DL-intensive applications in the SPEC2000INT suite). We expect some major challenges. First, even with all checkpointing optimizations enabled, checkpointing overhead is non trivial and can't be ignored. Second, the branch or value prediction's success rate plays an important role because failed predictions will directly translate into failed speculation and triggers the recovery and retry overhead. Third, the compiler needs to find enough work that can potentially overlap with the identified DL. Finally, the compiler needs to recognize an ideal sweet spot to terminate speculative overlapping.

6.5 Summary

We present theoretical performance analysis that models speculative execution overlapping with various levels of DLs. We predict the relative speedup will be around 50% through the model and verify it with real synthetic benchmarks that can reach very close to the theoretical peak. The results are obtained on real machines under ideal speculative conditions. The encouraging results motivate us to do further exploration using real-world applications.

7. CONCLUSIONS AND FUTURE WORK

In this paper we present our discovery that level-2 DLs from cache-miss intensive applications are persistent across a wide variety of cache architectures and input data sets. Motivated by this persistence, we present compiler transformations dealing with both control speculation and data speculation. We conduct theoretical performance modeling that predicts around 50% relative speedup. Our study using synthetic benchmarks strongly supports this claim.

We plan to investigate speculative execution that overlaps with DLs on real-world benchmark applications (e.g., the SPEC2000INT suite). The DL-persistent nature that exists in these applications provides an ideal granularity to further explore speculative execution that can be enabled through compiler transformations.

The emergence of hardware transactional memory [27] provides ideal hardware acceleration for fine-grain checkpointing. We plan to capitalize on the reduced overhead to use it to implement fine-grain speculative optimizations such as tolerating DL latency. We also plan to pursue alternative client optimizations for compiler-based fine-grain checkpointing such as debugging support, and possibly as part of an optimized software transactional memory (*STM*) [17, 41].

8. ACKNOWLEDGEMENTS

This work is funded by support from both IBM and NSERC. Chuck is supported by an IBM CAS Ph.D. fellowship since 2007.09. The authors would like to thank the anonymous reviewers for their feedback and insightful comments. The authors would also like to thank Mihai Burcea for the resourceful discussions during development.

9. REFERENCES

[1] Using the rdtsc instruction for performance monitoring. In *Pentium II Processor Application Notes, Intel Corporation*, 1997.

[2] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: An efficient, scalable alternative to reorder buffers. In *IEEE Computer Society*, 2003.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The suif compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[4] P. P. Bungale and C.-K. Luk. Pinos: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd ACM/USENIX International Conference on Virtual Execution Environments (VEE 2007)*, 2007.

[5] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *International Symposium on Computer Architecture archive*, 1999.

[6] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th annual international symposium on Computer architecture (ISCA '97)*, May 1997.

[7] I. cheng K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[8] J. Collins, H. Wang, D. Tullsen, C. Huges, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *ACM SIGARCH Computer Architecture News*, May 2001.

[9] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Tolerating dependences between large speculative threads via sub-threads. In *International Symposium on Computer Architecture (ISCA)*, June 2006.

[10] S. P. E. Corporation. Spec2000 integer benchmark suites. 2000.

[11] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: A platform for next-generation processors. In *IEEE/ACM International Symposium on Microarchitecture*, 1997.

[12] W. Elnozahy, D. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems, pp. 39-47*, October 1992.

[13] S. Fung and J. G. Steffan. Improving cache locality for thread-level speculation. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[14] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. In *IEEE Computer*, December 1996.

[15] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ACM SIGOPS Operating Systems*, December 1998.

[16] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *CM SIGARCH Computer Architecture News*, March 2004.

[17] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *The Twenty-Second Annual Symposium On Principles Of Distributed Computing*, 2003.

[18] H. V. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Procs. of the International Conf. on Very Large Databases (VLDB)*, 1993.

[19] G. Kingsley, M. Beck, and J. Plank. Compiler-assisted checkpoint optimization using suif. In *First SUIF Compiler Workshop*, 1995.

[20] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Checkpointed early load retirement. In *High-Performance Computer Architecture (HPCA)*, 2005.

[21] C. Li, E. Stewart, and W. Fuchs. Compiler-assisted full checkpointing. In *Software-practice and Experience, Vol 24(10), 871-886*, October 1994.

[22] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ACM SIGOPS Operating Systems Review*, December 1996.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI 05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[24] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pages 222-233*, October 1996.

[25] C.-K. Luk and T. C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. In *In IEEE Transactions on Computers, Vol. 48, No. 2*, Feburary 1999.

[26] A. Mcdonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ACM SIGARCH Computer Architecture News*, 2006.

[27] S. Microsystems. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc(r) processor. Feburary 2008.

[28] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: Log-based transactional memory. In *High-Performance Computer Architecture (HPCA)*, 2006.

[29] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *International Symposium on Computer Architecture (ISCA)*, 1997.

[30] A. Moshovos and A. Kostopoulos. Cost-effective, high-performance giga-scale checkpoint/restore. In *Computer Engineering Group Technical Report*, November 2004.

[31] J. E. B. Moss. Log-based recovery for nested transactions. In *Proceedings of the 13th International Conference on Very Large Data Bases*, 1987.

[32] T. C. Mowry. Tolerating latency through software-controlled data prefetching. March 1994.

[33] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Architectural Support for*

Programming Languages and Operating Systems, 1992.

[34] W. Ng and P. Chen. The symmetric improvement of fault tolerance in the rio file cache. In *Proceedings of 1999 Fault Tolerance Computing (FTC)*, 1999.

[35] H. Pan, K. Asanovic, R.Cohn, and C.Luk. Controlling program execution through binary instrumentation. In *SIGARCH Computer Architecture News 33, 5*, 2005.

[36] V. Panait, A. Sasturkar, and W.-F. Wong. Static identification of delinquent loads. In *International Symposium on Code Generation and Optimization*, March 2004.

[37] J. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. In *IEEE Technical Committee on Operating System and Application Environments, Special Issue on Fault-Tolerance*, 1995.

[38] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Seventh International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[39] B. Rychlik, J. Faistl, B. Krug, and J. Shen. Efficacy and performance impact of value prediction. In *Parallel Architectures and Compilation Techniques (PACT)*, 1998.

[40] C. S. An evaluation of recovery related properties of software faults. In *Ph.D. thesis*, 2004.

[41] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, and C. C. M. and. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Principles and Practice of Parallel Programming(PPOPP)*, 2006.

[42] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, 1997.

[43] J. E. Smith. A study of branch prediction strategies. In *SIGARCH: ACM Special Interest Group on Computer Architecture, 25 years of the international symposia on Computer architecture (selected papers)*, 1998.

[44] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *International Symposium on Computer Architecture (ISCA)*, June 2000.

[45] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th Annual IEEE/ACM International Symposium on Microarchitecture*, 2001.

[46] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *International Symposium on Computer Architecture*, 1995.

[47] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[48] Y. Wang, Y. Huang, K. Vo, P. Chung, and C. Kintala. Checkpointing and its applications. In *25th Int. Symp. On Fault-Tol. Comp., pp. 22-31*, June 1995.

[49] J. Whaley. System checkpointing using reflection and program analysis.

[50] P. Work and K. Nguyen. Measure code sections using the enhanced timer. In *Intel(R) Software Network*, 2008.

[51] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *the 24th annual international symposium on Microarchitecture (MICRO)*, 1991.

[52] P. yung Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO)*, 1995.

[53] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[54] C. C. Zhao, G. Steffan, and C. Amza. Compiler-based checkpointing and the potential for tolerating delinquent loads. In *Technical Report, Department of Electrical and Computer Engineering, University of Toronto*, 2009.

[55] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W. fai Wong. Ubiquitous memory introspection. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2007.