Compiler-Based Checkpointing and the Potential for Tolerating Delinquent Loads

Chuck (Chengyan) Zhao, Greg Steffan, and Cristiana Amza
{czhao,steffan,amza}@eecg.toronto.edu

Dept of Computer and Electrical Engineering, University of Toronto
Toronto, Ontario, Canada, M5S 3G

UT EECG Technical Report
Number UT-EECG-TR-2009-0017
April 2009

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
URL: http://www.ece.toronto.edu

Compiler-Based Checkpointing and the Potential for Tolerating Delinquent Loads

Chuck (Chengyan) Zhao, Greg Steffan, and Cristiana Amza
{czhao,steffan,amza}@eecg.toronto.edu

Dept of Computer and Electrical Engineering, University of Toronto
Toronto, Ontario, Canada, M5S 3G

University of Toronto, Dept. of ECE, UT EECG TR-2009-0017
April 2009

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-grained compiler-based checkpointing that operates at the level of individual variables, potentially providing low-overhead software-only support for speculative execution. As an initial attempt to exploit 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. After demonstrating that delinquent loads are persistent across different cache architectures and program inputs for several SpecINT2000 benchmarks, we propose and evaluate both data and control speculation methods for hiding delinquent load latency. Through an initial study of the delinquent loads in the MCF benchmark we find that our ability to improve performance via such speculation is limited by (i) the unpredictability of delinquent load result values, and (ii) the limited amount of computation on which to speculate.

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 Hyperthreading 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) [9, 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

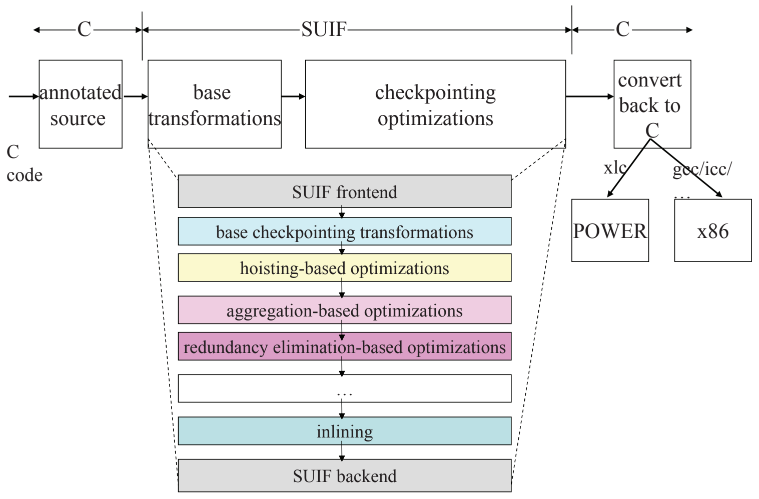

C
SUIF
C
annotated source
base transformations
checkpointing optimizations
convert back to C
C code
xlc
gcc/icc/...
POWER
x86
SUIF frontend
base checkpointing transformations
hoisting-based optimizations
aggregation-based optimizations
redundancy elimination-based optimizations
...
inlining
SUIF backend

Figure 1: Checkpointing system overview

work that checkpoints entire ranges of memory or entire objects. The intuition is that such fine-grained 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. Subsequently we demonstrate that DLs in several SpecINT2000 benchmarks are indeed persistent across a wide range of second-level (L2) cache sizes and architectures, and that modern compilers and processors do not alleviate them. Finally, we propose and evaluate two methods of tolerating DL latency that exploit compiler-based fine-grained checkpointing to implement software-only control and data speculation.

2 Support for Compiler-based Fine-grained Checkpointing

Checkpointing [13, 20, 21, 40, 48, 49] is the process of taking a snapshot of program execution so that we 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 [3, 30] can perform well, they have yet to be adopted broadly in commercial systems. Software-only checkpointing solutions [20, 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.

Overview Figure 1 presents a high-level overview of our checkpointing system. We take 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 [15, 4] compiler, which outputs transformed `C` code that we can then compile to target a number of platforms (currently x86 via `gcc` and POWER via IBM `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* [17, 26] or alternatively an *undo-log* [19, 31]. A write-buffer approach buffers all writes from main memory, and therefore also requires that the write-buffer be searched on every read. Should the checkpoint 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 [17, 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

```
foo(){
int x, y, z;
init_ckpt();

…
backup(&x, sizeof(x));
x = …;

for(…){
  …
  backup(&z, sizeof(z));
  z = …;
  if(…) {
    backup(&y, sizeof(y));
    y = …;
  }
  …
} …

attempt_commit();
…
}// end of foo()
```

(a) code with ckpt enabled

```
foo(){
int x, y, z;
init_ckpt();

…
backup(&x, sizeof(x));
backup(&z, sizeof(z));
x = …;

for(…){
  …
  z = …;
  if(…) {
    backup(&y, sizeof(y));
    y = …;
  }
  …
} …

attempt_commit();
…
}
```

(b) hoisting optimization

```
foo(){
int x, z, y; // reordered
init_ckpt();

…
backup(&x,
   sizeof(x) + sizeof(z) );

x = …;

for(…){
  …
  z = …;
  if(…) {
    backup(&y, sizeof(y));
    y = …;
  }
  …
}…
attempt_commit();
…
}
```

(c) aggregation optimization

Figure 2: Fine-grain Checkpointing Optimizations

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 we focus on 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 back-up 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 hence 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, which can be of arbitrary size; 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 opportunity for optimization. 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 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 host variable `y` in the example since it is only conditionally modified—whether to hoist such cases is actually a trade-off that will be studied. 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 are also investigating redundancy optimizations to remove redundant and unnecessary `backup()` calls, and have implemented an inlining pass so that `backup()` is not actually implemented as a procedure call but instead consists only of the bare instructions for performing the back-up.

[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.

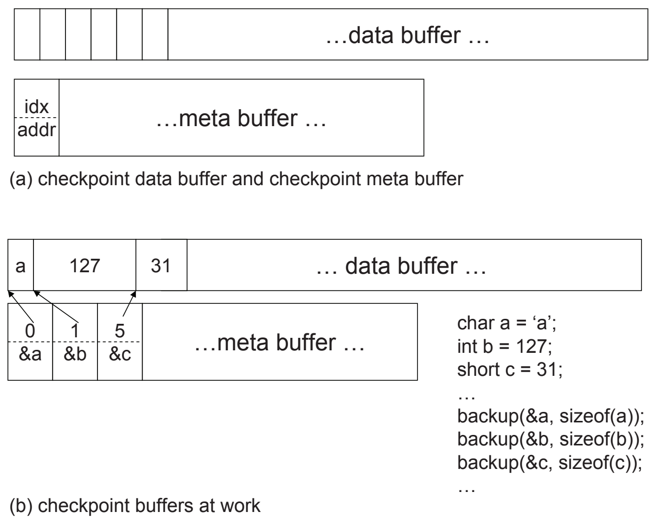


Figure 3: Undo-log buffering mechanism.

3 Identifying and Measuring Delinquent Loads

In this section we describe our methodology for identifying and measuring DLs, including our benchmark applications and profiling infrastructure.

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. 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 we 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%).

Benchmark Applications In this study we focus on the Spec2000Int [11] benchmarks, compiled using gcc-4.1.2 with -O3 optimization. Our initial investigation of all `C` benchmarks found that only a subset of the applications contain DLs, as listed in Table 1. This table also lists the particular `ref` input that we use for each benchmark, as well as the significance for each of the DLs in each benchmark (assuming a 256KB L2 cache with 32B cache lines and 2-way set associativity). As is evident from the table, these few DLs are responsible for a very large fraction of all L2 cache misses for these applications, ranging from 13.6% for VPR to 89.6% for MCF.

Name	Input Data	% L2 misses
mcf	inp.in (ref)	DL0: 14.4% DL1: 31.1% DL2: 23.7% DL3: 9.7% DL4: 5.4% DL5: 5.3% **total: 89.6%**
bzip2	input.program (ref)	DL0: 16.8% DL1: 12.2% DL2: 18.3% DL3: 14.9% **total: 62.2%**
vortex	ref	DL0: 15.7% DL1: 12.6% DL2: 11.5% **total: 39.8%**
parser	ref.in	DL0: 10.4% DL1: 18.6% **total: 29.0%**
vpr	ref	DL0: 13.6% **total: 13.6%**

Table 1: The most significant DLs

4 Delinquent Load Persistence

To consider optimizing DLs in a compiler, we first want to be confident that the DLs for a program are not sensitive to a particular size or configuration of the L2 cache. In this section we measure the *persistence* of L2 load misses and DLs in our benchmark applications across a broad range of L2 cache architectures. We also measure persistence across program inputs and compiler vendors.

4.1 Persistence across L2 Architectures

We measure a wide range of L2 cache architectures, with sizes varying from 256KB to 4MB, cache-line size varying from 32B to 128B, and associativity varying from 2 ways to 16 ways. Table 2 summarizes the combinations that we study for each cache size—the index is only to indicate the relative order of the combinations, and is the implied x-axis for each cache size for the remaining result graphs in this paper.

To start, in Figure 5 we present the average number of L2 cache load misses per 1000 instructions across our L2 cache configuration space. Note that even a single L2 cache miss per 1000 instructions is fairly significant, since an L2 miss can result in a 300-500 cycle miss penalty depending on the processor. It is evident that there are a significant number of L2 load misses regardless of configuration—especially for MCF which suffers more than 20 L2 load misses per 1k instructions for the smaller L2 cache sizes. Once the L2 cache size is 2MB or larger, the incidence of L2 cache misses is greatly reduced. This is partly because the SpecINT2000 benchmark suite was not designed to properly exercise processors with greater than 1MB on-chip L2 caches, so for designs with 2MB L2 caches or larger the working set for most applications is resident. MCF is an exception, and continues to have a relatively frequent occurrence of L2 cache misses even for large L2 caches.

Figure 4 demonstrates the persistence of each of the DLs in our benchmarks across L2 cache architectures. Focusing on the experiments using the `ref` inputs, we see that DLs are generally persistent across architectures, with the significance of the DLs for some applications dropping off for L2 caches that are 2MB or larger. Since MCF has such a large incidence of DLs we also include DLs that comprise more than 5% of all L2 cache misses. For MCF, three of the DLs remain persistent even for a 4MB L2 cache. For VPR, the main DL becomes insignificant for 2MB or larger L2 caches. For both BZIP2 and PARSER, while some DLs become less significant as L2 cache size increases, one of

Index	Line-size	Assoc.
0	32B	2
1	32B	4
2	32B	8
3	32B	16
4	64B	2
5	64B	4
6	64B	8
7	64B	16
8	128B	2
9	128B	4
10	128B	8
11	128B	16

Table 2: Cache configuration space explored (across a range of L2 cache sizes).

the DLs becomes more significant.

4.2 Persistence Across Program Inputs

Figure 4 also demonstrates DL persistence across program inputs, by showing results for both the `ref` and `train` inputs for each benchmark. Most DLs remain persistent across architectures (up to 1MB L2 caches), although for VPR its single DL becomes insignificant, as does one of PARSER's DLs. This is likely due to the fact that `train` inputs are generally not as large as `ref` inputs, reducing the L2 cache capacity required to fit the working set for these benchmarks.

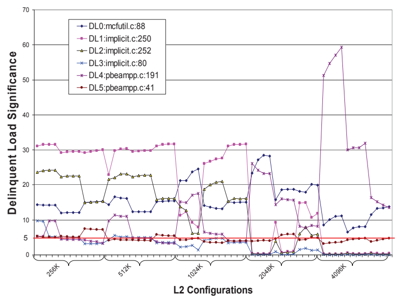
4.3 MCF: A Deeper Look

In this section and for the remainder of this paper we focus on MCF, since it has the most significant DLs of any benchmark. Our first question is whether DLs are consistent across different compilers. In Figure 4(k) we compile MCF with `gcc` version $4.0.4$ and in Figure 4(l) we compile with Intel's `icc` version 10.1 [2], both running on the `ref` input. Four of the DLs remain consistent across the two compilers, while two of them change between the two compilers (DL3 and DL5).

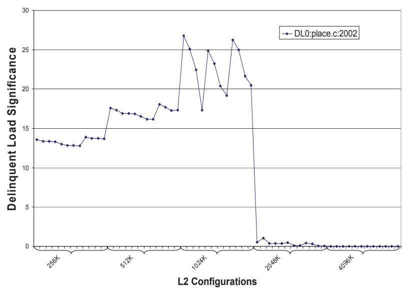
Figure 6 shows the corresponding code for the top six DLs in MCF, which provides insight into DL characteristics in integer applications. First, all DLs reside within a pointer access, fetching a field from a structure. Examining source code shows that all DLs are part of a linked-list traversal. All link-list nodes are dynamically allocated which presents little inter-node spatial locality—implying that conventional prefetching techniques will not be effective for these DLs. Second, the majority of DLs are within 1-level pointer access (DL0 to DL4), while only DL5 is through multiple levels of pointer indirection. This matches the style of single-level link list where most actions happen within the single node that is currently being accessed. Third, DLs are more likely to happen within a frequently-accessed field of a big structure whose size is larger than the cache-line size. Knowing the size of arc is 32B (DL1 to DL5) and the size of node is 60B (DL0), they are either equal-to or larger-than the smallest cache-line that we simulate (32B)—loading a different linked-list node of either type is more likely to cause cache misses on such architectures. Finally, when there are multiple levels of pointer access (DL5) this is more likely to be a DL, because such accesses are very unlikely to remain within a single cache line.

4.4 Summary

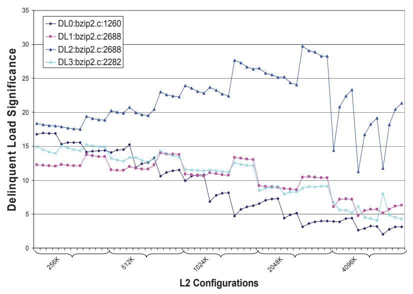
For code containing large numbers of L2 cache load misses and exhibiting DL behaviors, we observe that the DL source locations and cache behaviors are persistent across various cache architectures provided the working set originated from these DLs won't entirely fit into the cache. This persistence motivates us to investigate compiler techniques to tolerate the latency of DLs.



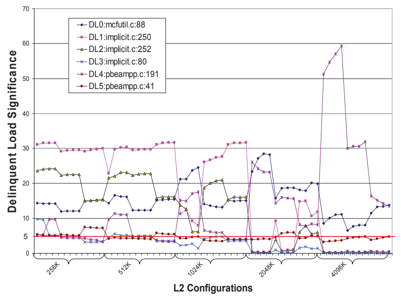
(a) MCF:ref



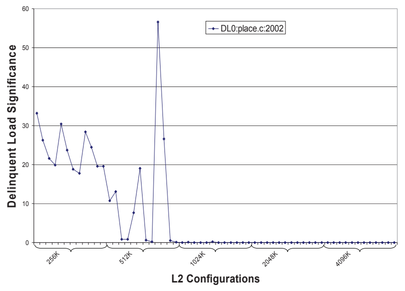
(c) VPR:ref



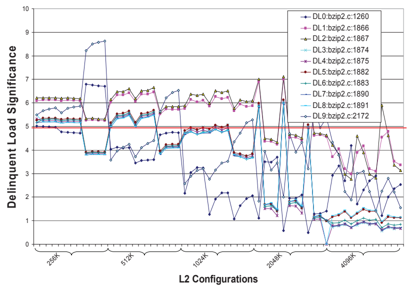
(e) BZIP2:ref



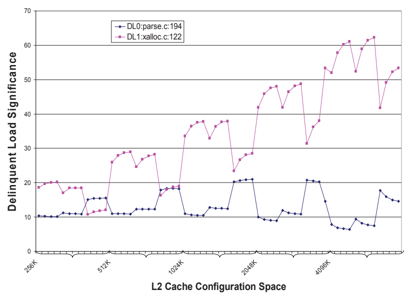
(b) MCF:train



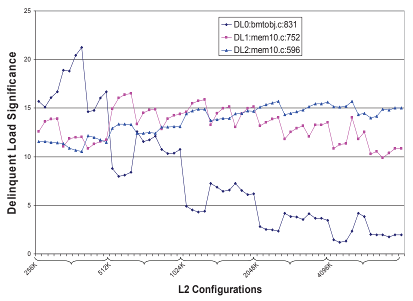
(d) VPR:train



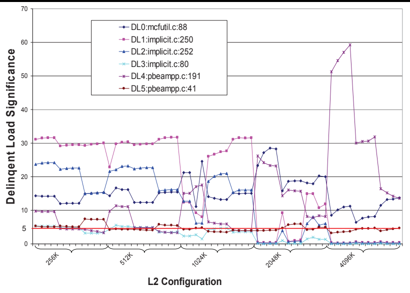
(f) BZIP2:train



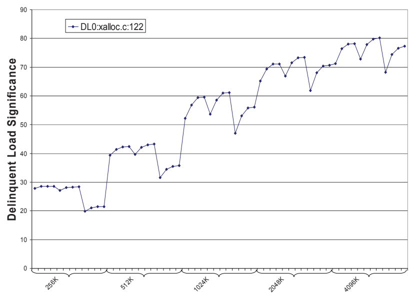
(g) PARSER:ref



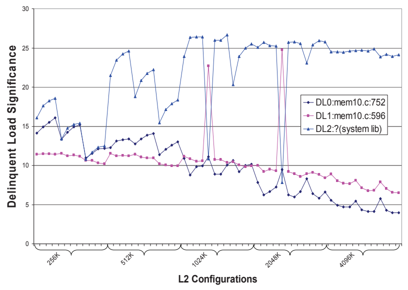
(i) VORTEX:ref



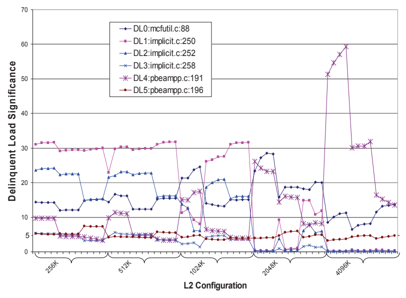
(k) MCF:ref,gcc4.0.4-O2



(h) PARSER:train



(j) VORTEX:train



(l) MCF:ref,icc10.1-O2

Figure 4: Persistence of DLs across: (a-j) architectures and benchmark inputs; and (k-l) compiler vendors.

5 Tolerating Delinquent Loads with Speculative Execution

In this section we propose two techniques that leverage compiler-based fine-grained checkpointing to tolerate DLs, namely data and control speculation. For such single-threaded speculation, we must make a prediction about the

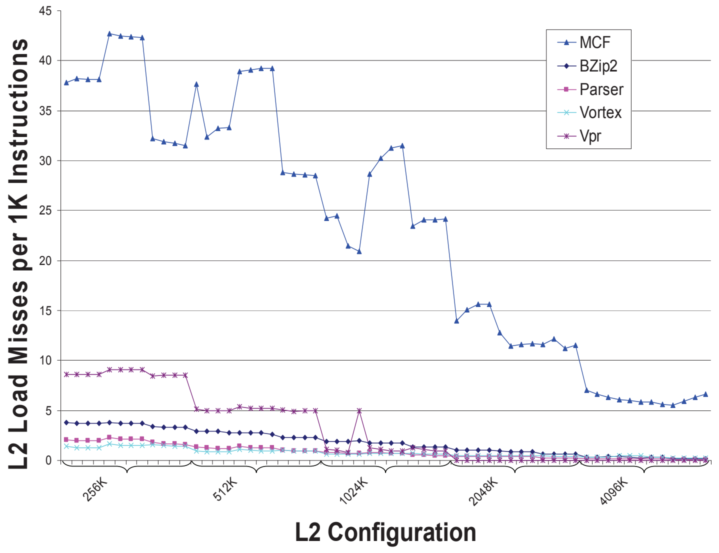


Figure 5: Number of L2 cache load misses per 1K instructions, across different cache configurations (described in Table 2).

resulting value of a DL and execute code that uses that prediction to make progress rather than awaiting the DL result value from off-chip; this approach exploits the parallelism provided by a wide-issue superscalar processor that can execute instructions in parallel with memory references. Ideally the latency of the DL is hidden when the prediction is correct, but execution can be rewound and re-executed using the correct DL value should the prediction be incorrect. We evaluate the proposed techniques using MCF, the benchmark with the most prominent DLs according to our study in the previous section.

5.1 Overview

Figure 7(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()`) likely depends on the DL's result value (`x`). Figure 7(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 computation 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 7(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 7(c), then we must rewind the checkpoint and then 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 Prediction

The effectiveness of speculation depends on the data prediction accuracy, since frequently-inaccurate prediction will result in an overwhelming amount of failed speculation. Also, the complexity of the predictor itself is a source of overhead—a cost that must be overcome by the benefits of tolerating the DL latency to produce speedup.

We implement and evaluate two of the simplest previously-proposed value predictors [6, 22, 39, 47]: including a last value predictor and a stride-based predictor. The last-value predictor simply predicts that the next value will be the same as the last value observed, requiring only a single variable for storage. The stride predictor computes the difference between each consecutive pair of values and predicts that the difference will be constant: hence the stride predictor must store the last value as well either the current value or the difference *stride* between them.

```
while( node != root ){
     while( node ){
          if( node->orientation == UP )           // DL0
               node->potential = node->basic_arc->cost + node->pred->potential;
          else{
               node->potential = (node->pred)->potential -node->basic_arc->cost;
               checksum++;
          }
  …
}
```

(a) DL0: mcfutil.c:86

```
while( arcin ){
      tail = arcin->tail;                               // DL 1

      if( tail->time + arcin->org_cost > latest ){ // DL 2
            arcin = (arc_t *)tail->mark;
            continue;
      }
      …
}
```

(b). DL1: implicit.c:250, DL2: implicit.c:252

```
cost_t compute_red_cost( cost_t cost, node_t *tail, cost_t head_potential )
     cost_t cost; node_t *tail; cost_t head_potential;
{
     return (cost - tail->potential + head_potential);   // DL3
}
```

(c) DL3: mcfutil.c:80

```
for( ; arc < stop_arcs; arc += nr_group )
{
      if(arc->ident > BASIC ) { // DL4
          red_cost = bea_compute_red_cost( arc );
          …
      }
}
```

(d) DL4: pbeampp.c:191

```
cost_t bea_compute_red_cost( arc_t *arc ){
      return( arc->cost - arc->tail->potential + arc->head->potential); // DL5
}
```

(e) DL5: pbeampp.c:41

Figure 6: Significant DL locations in MCF

Figure 8(a) shows the accuracy of last value and stride predictors on the DLs in MCF. DL1, DL2, and DL5 have close to 0% prediction accuracy and hence cannot benefit from the form of speculation that we propose. DL4 has a high prediction accuracy of 91.3% using the last-value predictor, and hence is our best candidate for speculation. A closer look at the value distribution for DL4 given in Figure 8(b) shows that the value 1 is extremely common, while only two other values are observed (for the `ref` input)—obviating while a last-value predictor does well for this DL. The stride predictor achieves 81.84% accuracy for this DL, but predicting a stride of zero (i.e., not really capitalizing on the stride predictor's ability). DL0 and DL5 have only a 22.85% and 43.7% prediction accuracy respectively and will likely suffer from too much misprediction to enjoy a speedup from speculation. For these DLs we find that the last-value predictor out-performs the stride predictor in every case, in addition to the stride predictor being slightly higher overhead. Hence we focus on implementations of speculation based on the last-value predictor.

To try to reduce mispeculation and improve prediction accuracy for the DLs we tried two things. First, we also measured previously-proposed context-based predictors that can predict fixed-length sequences of arbitrary values, but we expect that the storage and computation complexity of such predictors would be prohibitive. However, our initial studies showed that even aggressive context-based predictors did not significantly improve prediction accuracy

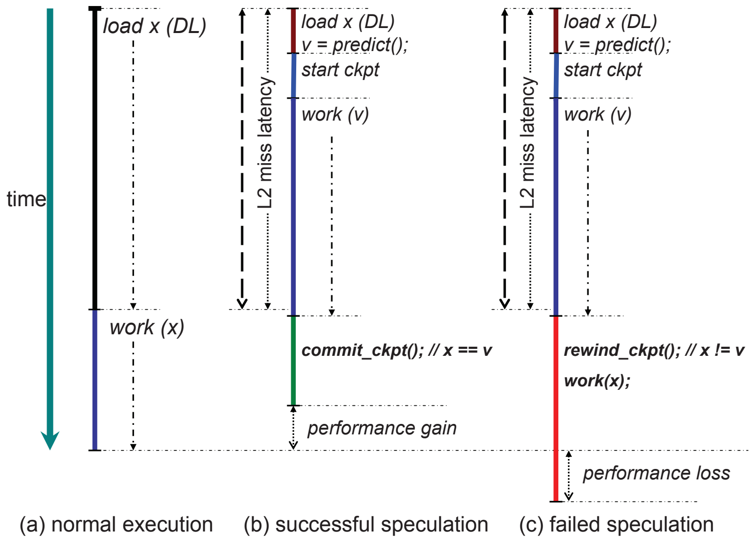


Figure 7: Overview of tolerating a DL with speculative execution.

for MCF, hence we do not discuss them further here. Second, a method of reducing the rate of costly misprediction is through an implementation of *confidence*: an n-bit saturating counter which tracks the recent accuracy of prediction, such that future predictions are only actually made when confidence is above a certain threshold. However, we found that a variety of confidence counters (ranging from 1-bit to 3-bit) did not improve the misprediction rate significantly.

5.3 Data Speculation

The first method of tolerating DL latency that we evaluate is *data speculation* (*DS*) where we predict the result data value of the DL and use it to continue execution speculatively, as illustrated in Figure 9. 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*).

5.4 Control Speculation

Whenever a the result value of a DL is used *solely* within a conditional control statement, as shown in Figure 10(a), 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.

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.

	Type	Last Value Predictor (%)	Strider (%)
DL0	CS	22.85	18.56
DL1	DS	0.0	0.0
DL2	CS	0.0	0.0
DL3	DS	1.75	1.72
DL4	***CS***	***91.3***	***81.84***
DL5	DS	43.67	30.84

(a) MCF DLs' Value Prediction Accuracy

Value	Distribution Percentage
1	***94.66***
0	3.65
2	1.69

(b) MCF DL4' Value Distribution

Figure 8: MCF DL (a) prediction accuracy and the type of speculation each DL is amenable to, data speculation (DS) or control speculation (CS), and (b) value distribution for DL4.

6 Performance Modeling and Evaluation

In this section, we give both theoretical performance modeling and practical evaluation of the proposed speculative techniques on real machines. We 1st present mathematical analysis of the implicit DL memory overlapping model and give theoretical upper-bound predictions of potential performance gains. We show that the theoretical model predicts 50%+ relative speedup. We then apply this model on synthetic benchmarks running on real machines and demonstrate that the relative performance gain of the micro benchmark closely matches the theoretical prediction. We finally conduct a detailed study for applying the model on a real-world DL-intensive application with software speculation enabled.

6.1 Theoretical Performance Modeling

Figure 11 illustrates the ideal timing model for overlapping execution with DLs. Figure 11(a). is the normal sequential model where the total execution time is the sum of both DL cycles and the overlapped work cycles. This represents the condition where the DL value is immediately needed to continue execution. While under the overlapped model (Figure 11(b)), the total execution time is the *maximum* of the two. This models the cases of either the DL value is not immediately needed or the DL is used to make a predictable control-flow decision thus its precise value is less important.

Let *CL* denote to the cycles of a cache miss and let *C* denote to the cycles of overlapped work, we have

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

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

Let *S* denote to 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 speedup for only overlapping with L1 cache is

```
...
work(P->a); // DL
...
```

(a) original code

```
1:  t = P->a;         // issue DL
2:  v = predict();    // value prediction
3:  start_ckpt();     // start ckpt
4:  work(v);          //speculative execution

5:  if( t == v ){     // check prediction
6:  commit_ckpt();
    }
    else{
7:  rewind_ckpt();
8:  work(t);          // normal re-execute
    }
```

(b) with data speculation

Figure 9: Tolerating a DL via data speculation.

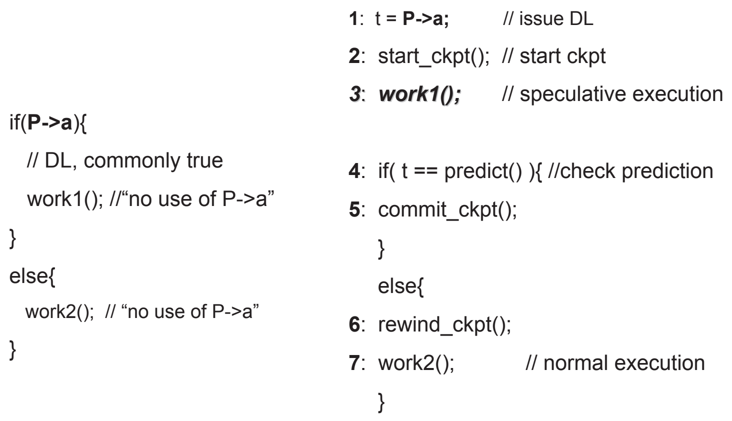
$$
\begin{aligned}
S &= \frac{CL_1 + C - \max(CL_1, C)}{CL_1 + C} \\
&= \begin{cases} \frac{CL_1+C-CL_1}{CL_1+C}, \text{if} & C < CL_1 \\ \frac{CL_1+C-C}{CL_1+C}, & \text{if } C \geq CL_1 \end{cases} \\
&= \begin{cases} \frac{C}{CL_1+C}, & \text{if } C < CL_1 \\ \frac{CL_1}{CL_1+C}, & \text{if } C \geq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

And the ideal theoretical speedup for only overlapping with L2 cache is

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

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

$$
S = \begin{cases} \frac{C}{CL_1+C} + \frac{C}{CL_2+C}, & \text{if } 0 \leq C < CL_1 \\ \frac{CL_1}{CL_1+C} + \frac{C}{CL_2+C}, & \text{if } CL_1 \leq C < CL_2 \\ \frac{CL_1}{CL_1+C} + \frac{CL_2}{CL_2+C}, & \text{if } C \geq CL2 \end{cases}
$$



```
if(P->a){
  // DL, commonly true
  work1(); //"no use of P->a"
}
else{
  work2();  // "no use of P->a"
}
```

```
1: t = P->a;        // issue DL
2: start_ckpt();  // start ckpt
3: work1();        // speculative execution

4: if( t == predict() ){ //check prediction
5: commit_ckpt();
   }
   else{
6: rewind_ckpt();
7: work2();          // normal execution
   }
```

(a) original code (b) with control speculation

Figure 10: Tolerating a DL via control speculation.

Figure 12 presents three theoretical speedup curves for overlapping with L1 cache only, with L2 cache only, and overlapping with combined L1-and-L2 cache. It shows both the overall similarity and individual differences. For ease of comparison, we fix the L1 cache miss cycles to 20 (CL1) and L2 cache miss cycles to 500 (CL2).

The curve that overlaps with L1-only workload goes sharply to its peak from 0 to CL1 (20) cycles in the beginning. Since the L1-miss-and-L2-hit cycles are relatively short, it has only limited room to stretch before reaching its theoretical maximum, which is predicted to be 50% when the overlapped cycles (C) equals to L1-miss-and-L2-hit cycles. 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 (CL2) and its theoretical performance upperbound is also 50%. Given ideal workloads, the two theoretical speedups can further combine and generate an aggregrated effect that can cross the 50% threshold, presented as the CL2-centered triangle-like area in Figure 12.

6.2 Benchmarks

With theoretical speedup predictions, it comes to implement the predicted speculative techniques and realize the performance premium in real workloads. We propose two set of application suites for evaluation. The 1st set is a group of synthetic micro benchmarks, including linklist, binary search tree, B-tree, red-black tree, avl tree, etc. They behave in a similar way that accessing to dynamically allocated data structures results frequent cache misses (DLs). The 2nd set contains real-world applications that expose data structures and memory access patterns more complex than synthetic benchmarks. At the same time, they need to exhibit extensive DL behaviors that are suitable for our DL study. For performance evaluation, we use linklist as the representative among the group of synthetic benchmarks and we select MCF from Spec2000Int suite due to its frequent and intensive DL behaviors.

6.3 Micro Benchmark Performance

We construct the linklist benchmark such that the size of each node is larger than the size of the cache line on the machine it evaluates. To exacerbate the DL situation, we randomize the starting address of each node in the linkist. This helps to cripple the hardware prefetcher as it becomes difficult to predict the starting address of the next node in the linklist with randomization enabled. By adjusting the number of nodes in the linklist, we achieve the effect of either polluting only L1 cache (L1-DL), or polluting both L1-and-L2 cache (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 real-machine used for evaluating the benchmarks has a single-core 3.0GHz Pentium-IV CPU, with 16KB 4-way set-associative L1 data cache, 12KB 8-way set-associative L1 instruction cache, and 1MB 8-way set-associative

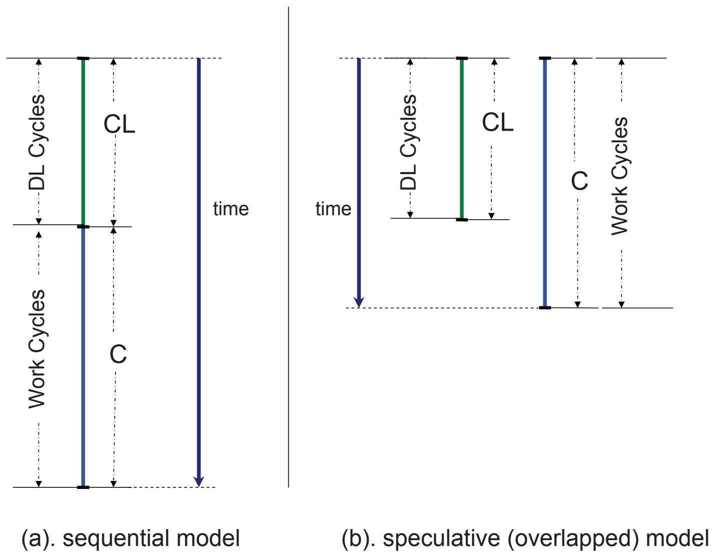


Figure 11: Ideal Timing Model

L2 cache. The cacheline size is consistent at 64B. Each measurement data point is the arithmetic average of 5 independent runs.

Figure 13 shows the relative speedup of overlapping with L1 DL using linklist. The workload to overlap with DL is a loop performing accumulation of integer adds (shown on x-axis), while y-axis gives the relative speedup. Figure 13 represents close similarity of the theoretical prediction of L1 speedup curve given in Figure 12. It reaches its maximum of 45% with overlapping roughly 70 INT-ADDs.

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

Figure 14 shows the relative performance result when overlapping with both L1 and L2 cache DLs on a real machine. In stage 1, the curve reaches around 35% speedup at roughly 70 INTADDs. This agrees with our own measurement given in Figure 13 and it is the effect of mostly overlapping L1 DL. In stage 2, the curve maintains stableness over 35% with top gain reaching very close to the 50% theoretical peak. This closely matches the L1-and-L2 prediction given in Figure 12 where a wide cap of 35%+ relative performance is expected after Stage 1.

We give theoretical predictions on performance gain which overlaps with various level of cache. We verify this claim with macro benchmarks that can reach very close to the theoretical peak. These results are obtained under ideal conditions that i). 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 produce 100% prediction accuracy. However, such ideal situations may not hold under non-synthetic benchmarks on real machines.

6.4 Initial Results for MCF

In this section we investigate the potential for compiler-based data and control speculation to tolerate DL latency, focusing on the DLs in MCF. Any beneficial data speculation has to satisfy a critical condition: the data value must be highly predictable. Any low data value prediction accuracy effectively renders data speculation unattractive due to the overwhelming expense of failed speculation. Of the three DLs best-suited to data speculation in MCF (DL1, DL3, and DL5), unfortunately all three have prediction accuracies that are too low to exploit. For control speculation, MCF's DL0 and DL2 are also too unpredictable to exploit; however DL4 presents an interesting case. This DL has only 3 different result values, where both the numerical values and their distributions are given in Figure 8(b). It is easy to see that a static branch predictor that always predicts taken (true) will yield 96.35% accuracy.

The 100% accuracy case (pred+ckpt+nousepred) represents one extreme where prediction and checkpointing are both enabled and aggressively optimized, but the predicted value is not actually used—hence this case measures our overheads without allowing any speculative overlap. This case results in only a tiny slowdown of 0.14%, emphasizing

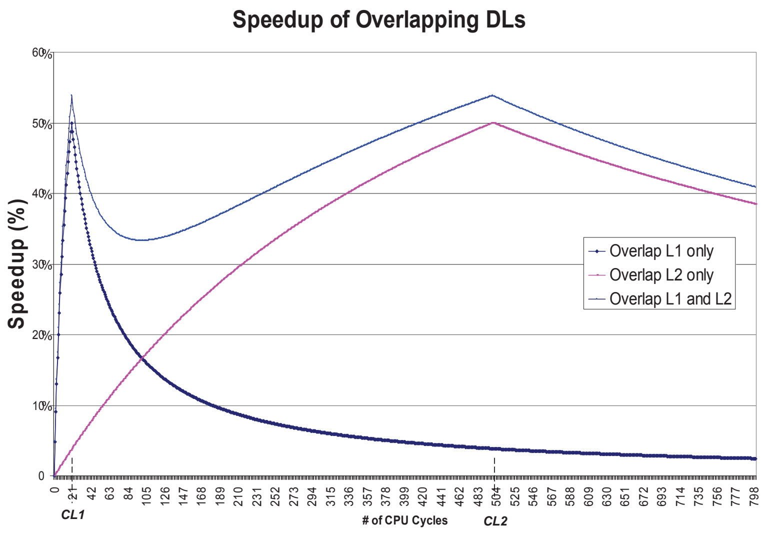


Figure 12: Relative Speedup of Ideally Overlapping Execution with DLs on Various Levels of Cache

Predictor	Accuracy	Speculative Perf
pred+ckpt+nousepred	100%	-0.14%
branch always taken	96.35%	-0.51%
const value(1)	94.66%	-1.33%
last value	91.3%	-1.36%
const value(0)	3.65%	-1.49%
const value(2)	1.69%	-2.43%
always predict wrong	0.0%	-2.45%

Table 3: Prediction accuracy and performance for MCF:DL4 for several predictors—the negative speedup percentages are all slowdowns.

the efficiency of our checkpointing framework. Various predictors yield different accuracies, ranging from 96.35% for a static taken branch predictor (always predicts true) to 1.69% of a const value predictor (always predicts the value 0). These are fully functional speculation executions with error checking and failure recovery in place. Their respective performance shows steady degradations with the ever decreasing prediction accuracy. This matches our expectation that low prediction accuracy triggers failed speculations that are overwhelmingly expensive.

A small overall slowdown of 2.43% (with 1.69% prediction accuracy) is derived from a much larger slowdown factor on regional granularity. The row with 0.0% prediction accuracy is achieved by constantly predicting a data value of -1, which is neither in the distribution of available values (Figure 8(b)) for value prediction, nor contributes to any success in branch prediction (Figure 6, DL4 case). Thus the global slowdown of 2.45% represent the performance lower bound in the worst case that a forever failing speculation transformation can cause.

A few critical conditions need to be satisfied simultaneously in order for control speculation to work. This includes *(*1*)* highly-biased branch prediction toward selected speculative region; *(*2*)* overlapping code region that is coarse-grain enough to closely match the DL latency and compensate checkpointing overhead; *(*3*)* no control-flow terminating instructions within the overlapping code region which can prematurely terminate speculative execution, *(*4*)* no reuse of DL's value within the speculative region, and *(*5*)* no hidden DLs in the speculative region that are covered by a leading DL. Among the three control-speculation cases (DL0, DL2 and DL4), DL4 is the only one that yields prediction accuracy high enough to proceed further. Unfortunately, DL4 has only a small computation to potentially overlap with (see Figure 6): this code region is too fine-grain to completely hide the long-latency DL while tolerating the software checkpointing overhead. In addition, DL4 is a *leading* DL, which means that it covers other memory loads whose code distances to DL4 are within the cache line size. Speculative execution on DL4 breaks its delinquent

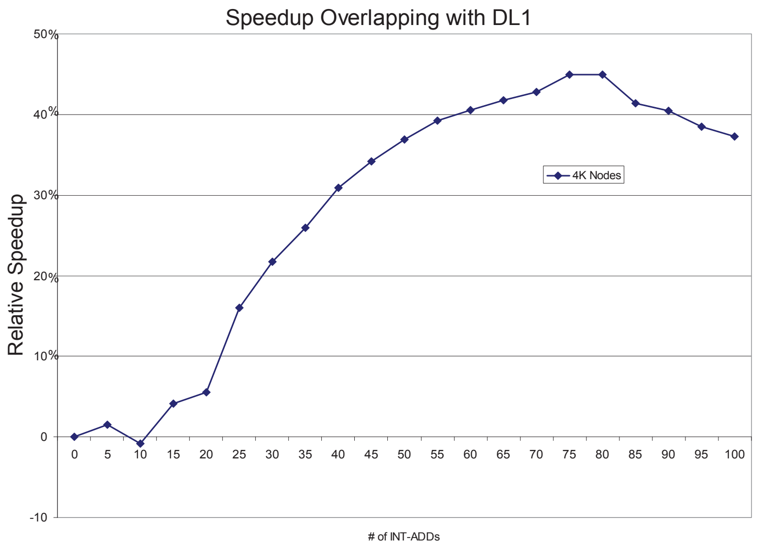


Figure 13: Relative Speedup Overlapping with L1-only DL on Real Machine

nature, but exposes DLs that used to be hidden and covered by the leading DL. As a result, control speculation on DL4 gives no positive performance despite its high prediction accuracy.

6.5 Summary

We present theoretical analysis of speculative fine-grain overlapping with DLs and predict that the relative speedup will reach 50%+ given combined L1-and-L2 DL cache effects. We measure the performance gains of representative synthetic benchmarks on real machine and verify that the macro benchmark delivers performance close to theoretical peak under ideal conditions. We continue our detailed study of speculative execution using MCF and demonstrates that MCF has balanced DL cases in control speculation and data speculation. We find that the DL data values are not always predictable; however, a simple last-value predictor or static branch predictor offers the best overall accuracy for those that are predictable. We find that MCF's data speculation cases are generally unsuitable for software speculation due to their low prediction accuracy. We identify a control speculation case in MCF that has highly attractive prediction accuracies. We implement compiler transformations to leverage on this DL and overlap with sequential execution. We believe that further success of speculative techniques will also heavily depend on the nature of the overlapping code region which needs to be coarser in granularity.

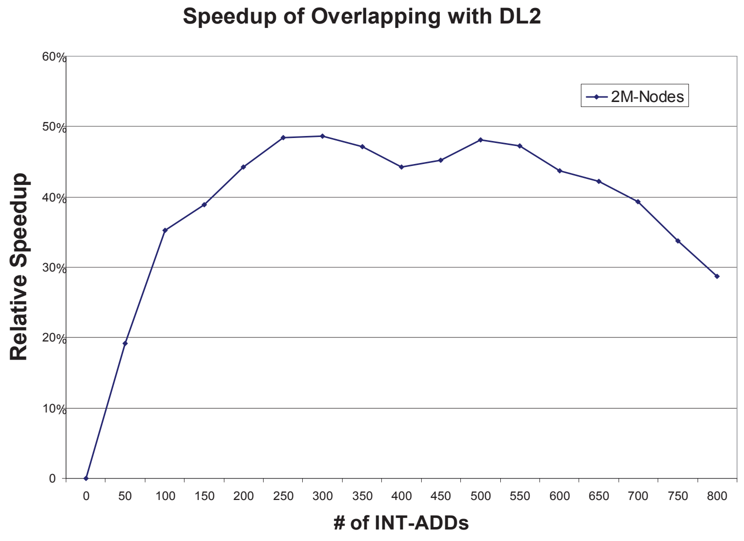


Figure 14: Relative Speedup Overlapping with L1-and-L2 DL on Real Machine

7 Related Work

Our techniques are based on a wide spectrum of existing work in related areas, including prefetching, multithreading, prediction, checkpointing, speculation and identifying DLs.

Prefetching Prefetching alleviates an application suffering from frequent cache misses [9, 24, 25, 32, 33]. Memory prefetching copies data from memory to cache in anticipation of near-future uses. Prefetching helps to hide memory access latency because execution needs no stall when requesting data that used to reside off chip. Prefetch non-DL locations provide little benefit, but with the guaranteed expense of wasted CPU cycles, increased usage of memory bandwidth and potential cache pollution. The effectiveness of prefetching depends on precisely identified DL locations.

Multithreading Long-latency memory access can be handled by multithreading [12, 38, 45, 46]. On a system with multiple threads in ready state, any running thread with a blocking memory access will be context switched out immediately. The processor keeps busy by executing threads not being blocked. It hides the long memory latency as the blocking thread will never execute before its memory access request has been satisfied. Multithreading greatly improves CPU utilization rate and throughput of a multi-program system, but has little benefit for the thread that frequently blocks on DLs.

Prediction Prediction is a mechanism to identify the outcome of a future action. This includes predict the branch taken or non-taken *(branch prediction)* [8, 43, 51, 52], predict a target address that a branch will transfer control to *(target address prediction)* [7, 47], and predict the value from a load instruction or a function call return *(value prediction)* [6, 22, 39, 42, 47]. Majority of existing prediction research focuses on obtaining high prediction accuracy, while the expense and complexity of building predictors that achieving such accuracy is often considered less important. In our software-only speculative scheme prediction contributes to overhead, hence we aim to achieve the same level of prediction accuracy while employing the simplest solution that has the lowest overhead.

Speculation Speculation is a form of optimistic program execution whose result might not be needed [10, 14, 16, 29]. Speculation handles control or data uncertainties that can't be statically proved. Steffan at al. [14, 44, 53] studied a speculative multi-core architecture that extends cache coherence protocol to include speculative states and buffers speculative change in cache. Most existing work is based on speculative hardware support with limited buffering capacity, which limits the granularity of the program region that can be speculated. In contrast we explore a software-only speculative approach that has no hardware dependency. We rely on both accurate and efficient software prediction as well as lightweight software checkpointing to enable speculation. We further leverage on aggressive compiler optimizations for speculative overhead reduction.

Checkpointing A checkpointing [13, 20, 21, 34, 40, 48, 49] enabled program copies data to its backup storage under designated request, in preparation for unexpected program errors and facilitate recovery by restoring the backup data. While hardware-based checkpointing solutions [3, 30] deliver desired performance, they come with a price premium and suffer from lack of availability and support in commercial systems. Software-only checkpointing solutions [20, 21, 37, 49] don't have inherited hardware dependency, but often carry prohibitive overhead through copying of coarse-grain memory blocks. This often prevents them from board adoption. In contrast with existing checkpointing work on coarse-granularity, we develop a lightweight software-only solution that works on a per-variable granularity. It is a compiler-based scheme that leverages on static program analysis and targets aggressive overhead reduction.

Identifying DLs Panait at el. [36] investigated techniques to identify DLs statically. They examine code on assembler level, categorize memory load instructions into various groups, and calculate a final weight based on profiling info 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, non-trivial to recognize on source level. We identify DLs through an efficient software cache simulator based on PIN [5, 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. Zhao at el. [54] 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.

8 Conclusion

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. Our in-depth study of the DLs in MCF finds that the DL result values are not always predictable; for those that are, a simple last-value predictor or static branch predictor is sufficient to give the best overall experience. We show that speculative overhead can be aggressively optimized to have only negligible impact on overall application performance and further success of software speculation also highly depends on the nature of the application and the availability of sufficient computation to overlap with the DL.

8.1 Future Work

In this work we found that the overheads of compiler-based checkpointing relative to the potential for speculative overlap for DLs (at least in MCF) were too prohibitive. However, with the appearance of hardware support for transactional memory [27] we may be able 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-grained checkpointing such as debugging support, and possibly as part of an optimized software transactional memory (*STM*) [18, 41].

References

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

[2] Intel c++ compiler user's guide. 2008.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

[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] 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.