Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors

Ilya Ganusov and Martin Burtscher
Computer Systems Laboratory
Cornell University
$\{$*ilya, burtscher*$\}$*@csl.cornell.edu*

Abstract

This paper proposes a new hardware technique for using one core of a CMP to prefetch data for a thread running on another core. Our approach simply executes a copy of all non-control instructions in the prefetching core after *they have executed in the primary core. On the way to the second core, each instruction's output is replaced by a prediction of the likely output that the n^{th} future instance of this instruction will produce. Speculatively executing the resulting instruction stream on the second core issues load requests that the main program will probably reference in the future. Unlike previously proposed thread-based prefetching approaches, our technique does not need any thread spawning points, features an adjustable lookahead distance, does not require complicated analyzers to extract prefetching threads, is recovery-free, and necessitates no storage for the prefetching threads. We demonstrate that for the SPECcpu2000 benchmark suite, our mechanism significantly increases the prefetching coverage and improves the primary core's performance by 10% on average over a baseline that already includes an aggressive hardware stream prefetcher. We further show that our approach works well in combination with runahead execution.*

1 Introduction

The cores of modern high-end microprocessors deliver only a fraction of their theoretical peak performance. One of the main reasons for this inefficiency is the long latency of memory accesses. Often, load instructions that miss in the on-chip caches reach the head of the reorder buffer before the data is received, thus stalling the processor. As a result, the number of instructions executed per unit time is much lower than what the CPU is capable of handling.

Prefetching techniques have been instrumental in addressing this problem. Prefetchers attempt to guess what data the program will need in the future and fetch them in advance of the actual program references. Correct prefetches can thus reduce the negative effects of long memory latencies. While existing methods have proven effective for regular applications, prefetching techniques developed for irregular codes typically require complicated hardware that limits the practicality of such schemes.

This paper proposes a new approach to hide the latency of cache misses in both regular and irregular applications using relatively modest hardware support. We call our approach *future execution* (FE). The design of the FE mechanism was inspired by the observation that most cache misses are cased by repeatedly executed loads with a relatively small number of dynamic instructions between consecutive executions of these loads. Moreover, the sequence of executed instructions leading up to the loads tends to remain similar. Hence, for each executed critical load, there is a high probability that the same load will be executed again soon. Therefore, whenever a load instruction executes, we issue a copy of that load on another core of the same CMP with the address for the n^{th} next instance to perform a prefetch into the shared L2 cache.

One could use a value predictor to determine the likely address each load is going to reference in the future. However, many important load addresses are not directly predictable. Fortunately, even if a missing load's address exhibits no regularity and is thus unpredictable, it is often possible to correctly predict the input values to its dataflow graph (backward slice) and thus to compute a prediction for the address in question. Since the same sequence of instructions tends to be executed before each critical load, the dataflow graph stays largely the same. Exploiting this property, future execution predicts all predictable values in the program and then speculatively computes all values that are reachable from the predictable ones in the program dataflow graph, which greatly increases the number of instructions for which an accurate prediction is available.

Our mechanism uses available idle cores in a CMP to perform the future execution. We propose to use an entire core instead of a specialized execution engine because it simplifies the design, allows to put idle cores to good use, and leaves a fully functional extra core in case there are non-speculative threads to run.

The FE mechanism works as follows. The original unmodified program executes on the first core. As each instruction commits, it updates the value predictor with its current result. Then a prediction is made to obtain the likely value the instruction is going to produce during its n^{th} next execution. After that, the committed instruction is sent to the second core along with the predicted value, where it is injected into the dispatch stage of the pipeline. Instructions are injected in the commit order of the first core to preserve the program semantics. Since we assume that the same sequence of instructions will execute again in the future, the second core essentially executes n "iterations" ahead of the non-speculative program running in the first core. The execution of each instruction in the second core proceeds normally utilizing the future values. Loads are issued into the memory hierarchy using speculative addresses and instructions commit upon reaching the head of the ROB ignoring all exceptions. Section 3 provides more detail.

This paper makes the following main contributions. First, future execution presents a new approach on how to create prefetching threads in a cheap and efficient manner without specialized instruction stream analyzers, identification of thread spawn points or trigger instructions, or storage for prefetching treads. Second, we demonstrate a simple technique to compute accurate predictions for instructions that are not directly value predictable. Third, our technique allows to adaptively change the prefetching distance through a simple adjustment in the predictor. Fourth, unlike some other thread-based prefetching techniques, future execution is recovery-free, meaning the results generated by the prefetching thread do not need to be verified for correctness. As a consequence, the prefetching thread never has to be restarted. Fifth, the critical path in the main core is unlikely to be affected by our mechanism and no hardware is needed to inject pre-execution results into the main program. Sixth, our approach has a very low thread startup cost and can instantly stop or resume execution of the prefetching thread depending on the availability of an idle thread context. Finally, FE requires no changes to the operating system or the instruction set architecture and needs no programmer or compiler support. As such, it can speed up legacy code as well as new code.

The next section discusses the implementation of the FE micro-architecture. We start by presenting a quantitative analysis of the observations that inspired our design. Then we focus on the hardware support necessary to implement FE. We made an effort to minimize the complexity and to move most of the added hardware out of the core. Next, we show that our simplest implementation delivers a geometric-mean speedup of 25% on SPECcpu2000 programs relative to a conventional superscalar core. Compared to a baseline with an aggressive hardware stream prefetcher, FE still provides an average speedup of 10%. Finally, we demonstrate that future execution is complementary to prefetching based on runahead execution and that both approaches exhibit significant synergy when used together, bringing the geometric-mean speedup to 20%.

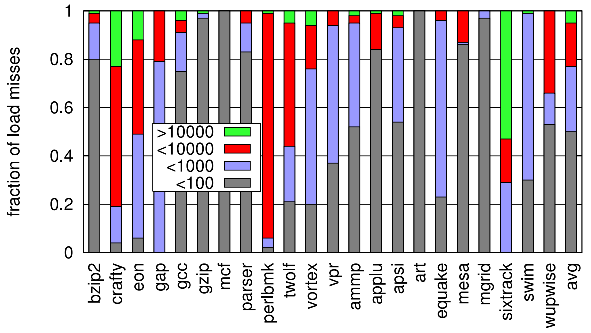


Figure 1. Execution distance in instructions between the loads that result in an L2 cache miss and the previous dynamic execution of the same static loads

2 Motivation

This section presents a quantitative analysis of the common program properties that are exploited by future execution. All the results are obtained using the benchmark suite and baseline microarchitecture described in Section 4.

One of the main program properties exploited by FE is that most load misses occur in 'loops' with relatively short iterations. In this paper, we define loop as any repeatedly executed instruction, be that because of an actual loop, recursion, function calls, or something else. Figure 1 presents the breakdown of the distance between the load instructions that cause an L2 cache miss and the previous execution of the same load instruction. The bars are broken down by distance: fewer than 100, between 100 and 1000, between 1000 and 10000, and over 10000 instructions. The taller the bar, the more often that range of instruction distances occurred. The data show that on average 80% of the misses occur in loops with iterations shorter than 1000 instructions. This observation suggests a prefetching approach in which each load instruction triggers a prefetch of the address that that same load is going to reference in the n^{th} next iteration. Since in most cases the time between the execution of a load instruction and its next dynamic execution is relatively short, the risk of prefetching much too early and polluting the cache is small.

Analyzing the instructions in the dataflow graphs of the problem loads, we found that while problem load addresses might be hard to predict, the inputs to their dataflow graphs often are not. Therefore, even when the miss address itself is unpredictable, it is frequently possible to predict the input

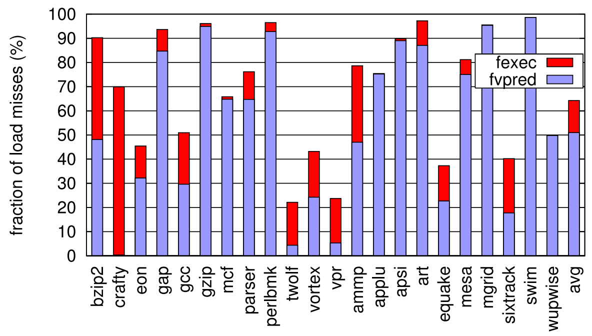
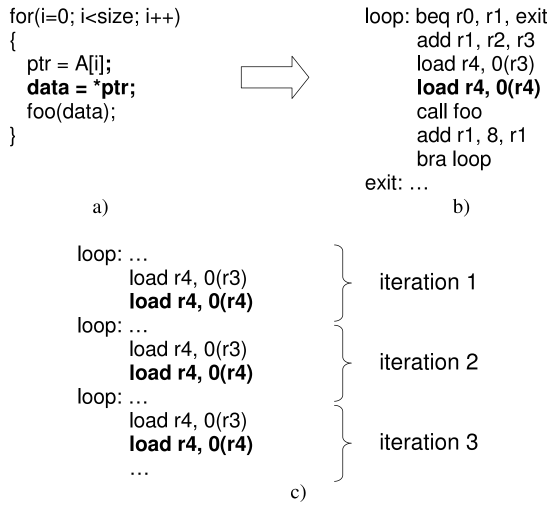


Figure 2. Distribution of the cache miss addresses that can be correctly predicted by a future value predictor (*fvpred*) and using future execution (*fexec*)



a)
```
for(i=0; i<size; i++)
{
  ptr = A[i];
  data = *ptr;
  foo(data);
}
```

b)
```
loop: beq r0, r1, exit
      add r1, r2, r3
      load r4, 0(r3)
      load r4, 0(r4)
      call foo
      add r1, 8, r1
      bra loop
exit: ...
```

c)
```
loop: ...
      load r4, 0(r3)
      load r4, 0(r4)      iteration 1
loop: ...
      load r4, 0(r3)
      load r4, 0(r4)      iteration 2
loop: ...
      load r4, 0(r3)
      load r4, 0(r4)      iteration 3
      ...
```

Figure 3. Code example

values of the instructions leading up to the problem loads and thus to compute an accurate prediction by executing these instructions.

Figure 2 shows the breakdown of the load miss addresses in the SPECcpu2000 programs that can potentially be predicted and prefetched by future value prediction and by future execution. The lower portion of each bar represents the fraction of misses that can be predicted by a stride-two-delta (ST2D) value predictor [15]. The upper bar shows how many of the miss addresses that are not predictable by the ST2D predictor can be correctly obtained by predicting the inputs of the instructions in the dataflow graphs of the missing loads and computing the addresses. The height of the stacked bar indicates the total fraction of misses that can potentially be correctly predicted. To measure the prediction coverage of future execution, we reconstruct the dataflow graph of each problem load whenever a cache miss occurs, compare it to the dataflow graph of the same static load during its previous execution, extract the part of the dataflow graph that is the same, and then check if the values provided by the future value predictor during the previous execution would allow to correctly compute the load address referenced by the load instruction in the current iteration. We limit the size of the dataflow graph that we analyze to 64 instructions.

Figure 2 illustrates that while value prediction alone is quite effective for some applications, future execution can significantly improve the fraction of load miss addresses that can be correctly predicted and prefetched. Over half of the SPECcpu2000 programs experience a significant (over 10%) increase in prediction coverage when future execution is employed in addition to value prediction.

Figure 3a shows a code example that exhibits the program properties discussed above. We extracted it from the program *mcf* and simplified it. An array of pointers A is traversed, each pointer is dereferenced and the resulting data are passed to the function "foo". Assume that the memory data referenced by the elements of array A are not cache-resident. Further assume that there is little or no regularity in the values of the pointers stored in A. Under these assumptions each execution of the statement `data=*ptr` will cause a cache miss. As shown in Figure 3b, in machine code this statement translates into a single load instruction `load r4, 0(r4)` (highlighted in bold).

A conventional predictor will not be able to predict the address of the problem instruction since there is no regularity in the address stream. However, the address references of instruction `load r4, 0(r3)` are regular because each occurrence of this instruction loads the next consecutive element of array A. Therefore, it is possible to use a value predictor to predict the memory addresses for this instruction, speculatively execute it, and then use the speculatively loaded value to prefetch the data for the problem load instruction. Since the control flow leading to the computation of the addresses of the problem load remains the same throughout each loop iteration (Figure 3c), a value predictor can provide predictions for the next iterations of the loop and the addresses of the problem load will be computed correctly. Therefore, sending the register-writing instructions to the second core in commit order and future predicting them makes it possible to compute the addresses that will be referenced by the main program during the next iterations of the loop.

3 Implementation of Future Execution

Our implementation of future execution is based on a conventional chip multiprocessor. A high-level block diagram of a two-way CMP supporting FE is shown in Figure 4. Both microprocessors in the CMP have a superscalar

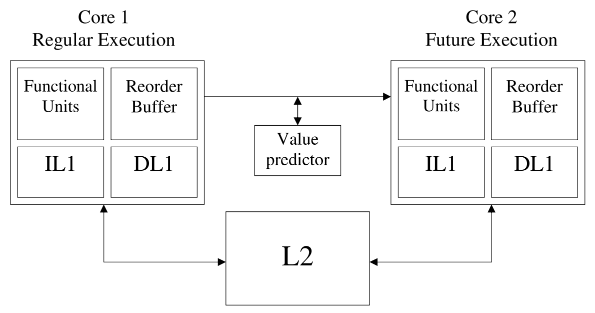


Figure 4. The FE architecture

execution engine with private L1 caches. The L2 cache is shared between the two cores. The conventional program execution is performed on the “left” core while the future execution is performed on the “right” core. To support FE, we introduce a unidirectional communication link between the cores, with a future value predictor attached to it. Both the communication link and the predictor are not on the critical path and should not affect the performance of the first core in a negative way. We also make slight modifications to the execution engine to assist the future execution. The following subsections describe the necessary hardware support and the operation of FE in greater detail.

3.1 Overview of Operation

Each register-writing instruction committed in the regular core is sent to the second core via the communication link. The data that need to be transferred to the second core include the instruction’s decoded opcode, result value, and a partial PC to index the value predictor. Stores, branches, jumps, privileged instructions, and system calls are not transmitted. If the communication link’s send buffer is full, further committed instructions are dropped and not transmitted to the second core. Each instruction sent from the regular core passes through the value predictor, updates it with the current output and requests a prediction of the value it is likely to produce in the n^{th} next iteration. Each prediction is accompanied by a confidence estimation [7].

When the instructions along with their predicted outputs reach the second core, they are inserted into the pipeline. Since instructions are transmitted in a decoded format, they can bypass the fetch and decode stages. Instruction dispatch proceeds as normal - each instruction is renamed and allocated a reservation station (RS) and a ROB entry if these resources are available. Whenever the input values for the instruction are ready, it executes, propagates the produced result to the dependent instructions and updates the register file.

The main difference between executing the instruction in FE mode versus normal mode is when and how the register values are determined to be ready. In normal mode, an instruction’s output is ready only after the instruction has executed. In FE mode, when an instruction is dispatched, the result field of the RS and ROB entries are immediately filled with the predicted future value and the result of this instruction is marked as ready if the confidence of the predicted output is above the threshold. Therefore, all subsequent instructions that depend on a predictable instruction can instantly read the corresponding input and execute. Instructions with ready results are allowed to retire without having to execute. Note, however, that we force all loads, including the ones with a predictable result, to execute so that they can issue prefetches. As in normal execution mode, non-ready FE instructions have to wait for their input values to become available before they can execute and write their result into the corresponding result field and forward it to the dependent instructions.

By the time an instruction reaches the ROB head in normal execution mode it is guaranteed that all of its inputs are ready. Therefore, if an instruction has not executed by that time, it is certain that it will execute as soon as the necessary execution resources become available. However, in future execution mode the instruction at the head of the ROB is not guaranteed to ever have its inputs ready. This happens, for example, when the input values could not be predicted with high confidence. Therefore, if an instruction reaches the head of the ROB in FE mode and its inputs are not ready, it is forced to retire without writing the corresponding result register. As a consequence, the execution of subsequent instructions that try to read this non-ready register will be similarly suppressed. If the instruction at the head of the ROB is a long latency load, it is forced to retire after a timeout period that equals the latency of an L2 cache hit. This approach significantly improves the performance of FE as it avoids stalling the pipeline. Forced retirement is disabled when the core is being used in non-FE mode.

3.2 Hardware Support

The value prediction module resides between the two CMP cores. We use a relatively simple, PC-indexed stride-two-delta predictor [15] with 4,096 entries. The predictor estimates the confidence of each prediction it makes using 3-bit saturating up-down counters. The confidence is incremented by one if the predicted value was correct and decremented by four if the predicted value was wrong. The particular organization of the value predictor is not essential to our mechanism and a more powerful predictor (e.g., [3]) may lead to higher performance.

Since we model a two-way CMP with private L1 caches, we need a mechanism to keep the data in the private L1 caches of the two cores consistent. In this work, we rely on an invalidation-based cache coherency protocol to maintain consistency. Therefore, whenever the main program executes a store instruction, the corresponding cache block in the private cache of the future core is invalidated. Since store instructions are not sent to the future core, future execution never incurs any invalidations.

4 Evaluation Methodology

We evaluate future execution using an extended version of the SimpleScalar v4.0 simulator [6]. The baseline is a two-way CMP consisting of two identical four-issue dynamic superscalar cores that are similar to the Alpha 21264 (Table 1). In all modeled configurations we assume that one of the cores in the CMP is idle and can be used for future execution. We also simulate a configuration with an aggressive hardware stream prefetcher between the shared L2 cache and main memory [11]. The stream prefetcher tracks the history of the last 16 miss addresses, detects arbitrary-sized strides, and applies a stream filtering technique by only allocating a stream after a particular stride has been observed twice. It can simultaneously track 16 independent streams, and prefetch up to 8 strides ahead of the data consumption of the processor. Our implementation of the future execution mechanism employs a stride-two-delta (ST2D) value predictor that predicts values 4 iterations ahead. Predicting four iterations ahead does not require extra time in case of the ST2D predictor. We simply modified the predictor hardware to add the predicted stride four times, which is achieved by a rewiring that shifts the predicted stride by two bits. The communication link between the two cores can buffer up to 40 instructions, has the latency of 5 cycles, and provides communication bandwidth that corresponds to the commit width (4 instructions/cycle).

Table 1. Simulated processor parameters

Processor	
Fetch/issue/commit width	4/4/4
I-window/ROB/LSQ size	64/128/64
Int/FP registers	184
LdSt/Int/FP units	2/4/2
Execution latencies	similar to Alpha 21264
Branch predictor	16K-entry bimodal/gshare hybrid
RAS entries	16
Memory Subsystem	
Cache sizes	64KB IL1, 64KB DL1, 1MB L2
Cache associativity	2-way L1, 4-way L2
Cache latencies	2 cyc L1, 20 cyc L2
Cache line sizes	64B L1, 64B L2
MSHRs	64 L1, 128 L2
Main memory latency	minimum 400 cycles
Hardware stream prefetcher	between L2 and main memory, 16 streams max. prefetch distance: 8 strides
Future Execution Hardware Support	
Future value predictor	4K-entry ST2D, 3bc conf. estimator
Communication link latency	5 cycles
Communication link bandwidth	4 instructions/cycle
Communication link buffer size	40 instructions

Table 2. Benchmark suite details (for the simulated interval of 500M instructions)

App.	NoPref IPC	loads (M)	L1 miss rate (%)	L2 miss rate (%)	Perf L2 speedup (%)
bzip2	1.27	142.16	1.63	19.02	45.57
crafty	1.73	156.37	0.85	1.51	2.92
eon	1.60	148.66	0.13	0.29	0.14
gap	1.11	127.02	0.37	35.03	53.17
gcc	0.98	179.25	2.58	9.57	56.90
gzip	1.61	99.62	5.23	0.71	2.39
mcf	0.04	209.74	25.41	73.32	1312.29
parser	0.76	121.98	2.47	20.65	97.38
perlbmk	1.38	157.11	0.27	11.57	3.71
twolf	0.53	142.56	4.99	14.57	125.24
vortex	1.52	129.11	0.83	14.48	51.03
vpr	0.54	164.96	3.39	17.50	107.34
ammp	0.69	134.07	4.36	30.65	135.23
applu	0.96	113.81	2.11	66.26	167.04
apsi	1.52	123.52	2.71	63.48	40.97
art	0.29	156.61	19.97	56.32	500.32
equake	0.26	235.05	7.44	54.04	552.11
mesa	1.84	129.52	0.34	28.00	19.49
mgrid	0.89	183.11	2.42	47.81	193.14
sixtrack	2.23	96.81	0.23	72.33	15.82
swim	0.41	123.09	9.04	59.74	513.41
wupwise	1.27	115.01	1.08	72.67	97.80

We use all integer and floating-point programs from the SPECcpu2000 benchmark suite [4] for this study with the exception of the four Fortran 90 programs for which we have no compiler. The programs are run with the SPEC-provided reference inputs. If multiple reference inputs are given, we simulate the corresponding programs with up to the first three inputs and average the results from the different runs. The C programs were compiled with Compaq's C compiler V6.3-025 using "–O3 –arch host –non_shared" plus feedback optimization. The C++ and Fortran 77 programs were compiled with g++/g77 V3.3 using "–O3 –static". We used the SimPoint toolset [16] to identify representative simulation points. Each program is simulated for 500 million instructions after fast-forwarding past the number of instructions determined by SimPoint.

Table 2 provides information about the benchmarks used. The cache miss rates shown are local. Out of the 22 programs used in this study, four integer programs are not memory-bound since they obtain less than 5% speedup with a perfect L2 cache. The perfect-cache speedup for the rest of the programs varies greatly and reaches up to 1312% for *mcf*. This large speedup is explained by an exceptionally large number of L1 misses and a very high L2 cache miss rate that reaches 73%.

5 Experimental Results

In this section, we experimentally determine the effectiveness of our proposed mechanism. In Section 5.1, we

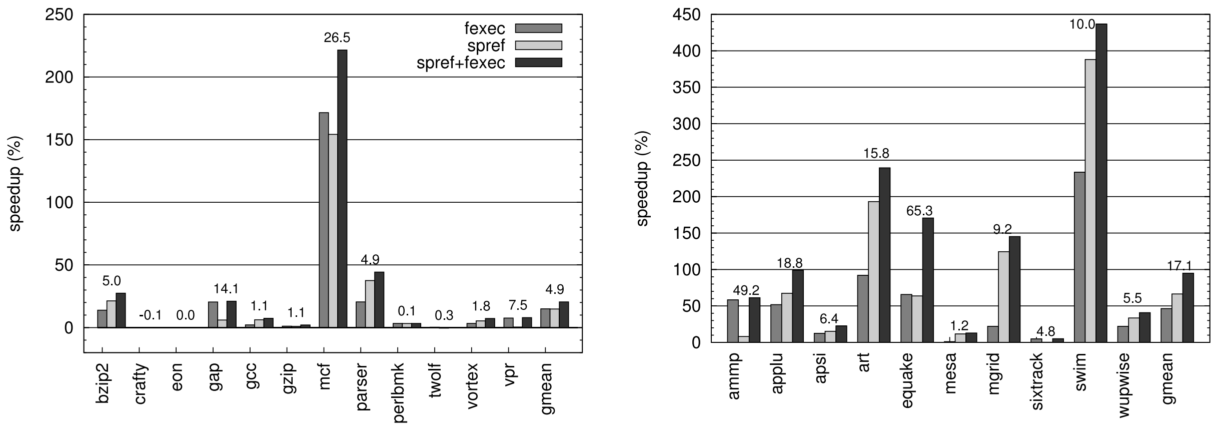


Figure 5. Execution speedup

measure the performance of prefetching based on future execution and compare its speedups with that of aggressive stream prefetching. In Section 5.2, we take a closer look at prefetching itself and gain additional insight by measuring the prefetching accuracy and coverage as well as the timeliness of prefetches. In Section 5.3 we compare our future execution technique to prefetching based on runahead execution and show that the two techniques are complementary to each other.

5.1 Execution Time Speedup

In this section, we measure the performance impact of our prefetching technique compared to a stream-based hardware prefetcher. The base machine for this experiment is described in Table 1. It represents an aggressive superscalar processor without hardware prefetching. We model three configurations: the baseline with prefetching based on future execution (*fexec*), the baseline with an aggressive hardware stream prefetcher between the shared L2 cache and main memory [11] (*spref*), and the baseline with stream prefetching as well as future execution (*spref+fexec*). Figure 5 presents speedups for individual programs as well as the geometric mean over the integer and the floating-point programs (integer programs are shown in the left panel, floating-point programs in the right panel). Note that the scale of the y-axis for SPECint and SPECfp programs is different. The percentages on top of the bars are the speedups of future execution combined with stream prefetching (*spref+fexec*) over stream prefetching alone (*spref*).

The results show that the hardware stream prefetcher used in our study is very effective, attaining significant speedups for the majority of the programs, with a peak of 387% for *swim* and 152% for *mcf*. The average speedup over the SPECint programs is 14.8%, while the SPECfp applications experience an average speedup of 66.5%. Note that we tuned the parameters of the stream prefetcher to maximize the prefetching timeliness and to minimize cache pollution on our benchmark suite. We use the same parameters on all programs.

When the model with only future execution is compared to the model with only stream prefetching, future execution outperforms stream prefetching on four programs, while stream prefetching is better on eleven. The remaining seven programs show about the same performance in both models. As the following section will show, in many cases the stream prefetcher can prefetch fewer load misses than future execution, but it provides timelier prefetching and hence larger performance improvements. The timeliness of the prefetches issued by future execution can be improved by adjusting the prediction distance of the future value predictor, but we use the same prediction distance throughout this paper to make the results comparable. Nevertheless, the *fexec* model provides significant speedup (over 5%) for 12 programs, with an average speedup of 15% for the integer and 46.4% for the floating-point programs, with a maximum of 233% on *swim*.

The model with the best performance is the one that combines the stream prefetcher and future execution. On average, this model has a 50% higher IPC than the model without prefetching. Moreover, this model has a 10% higher IPC than the baseline with stream prefetching. Out of the 22 programs used in our study, 14 significantly benefit (over 4.7% improvement) from future execution when it is added to the baseline that already includes a hardware stream prefetcher. If we look at the behavior of the integer and floating-point programs separately, adding future execution to the baseline with a stream prefetcher increases the performance of SPECint and SPECfp by 4.9% and 17.1%, respectively. This indicates that future execution and stream prefetching interact favorably and complement each other by prefetching different classes of load misses.

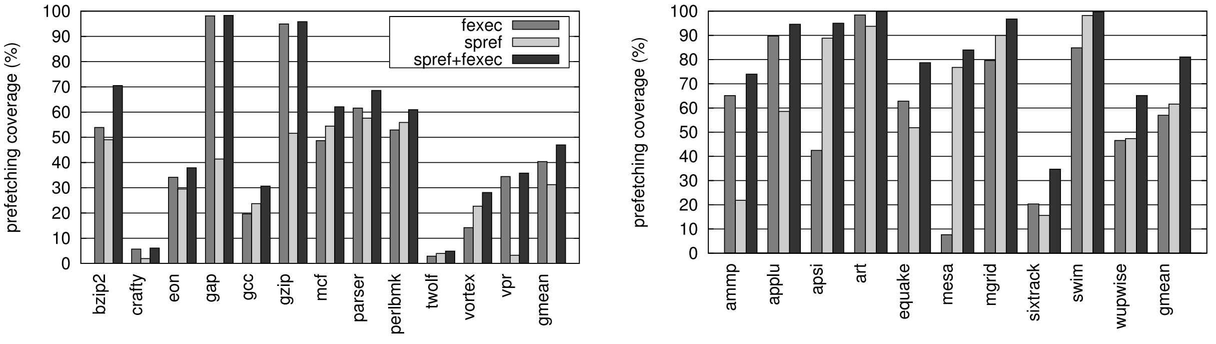


Figure 6. Prefetch coverage

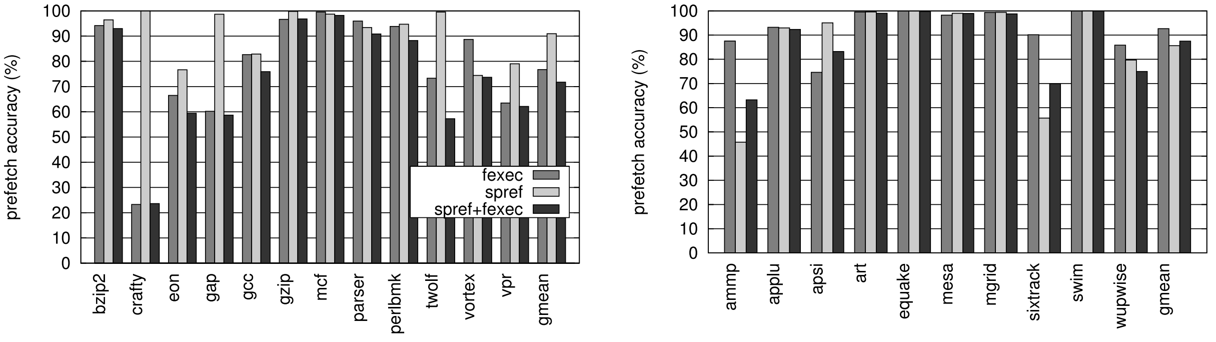


Figure 7. Prefetch accuracy

Overall, the results in this section demonstrate that future execution is quite effective on a wide range of programs and works well alone and in combination with a stream prefetcher.

5.2 Analysis of Prefetching Activity

This section provides insight into the performance of prefetching based on FE by taking a closer look at the prefetching activity. We begin by presenting the prefetch coverages obtained by different prefetching techniques. We define the prefetch coverage as the ratio of the total number of useful prefetches (i.e., the prefetches that reduce the latency of the cache missing memory operations) to the total number of misses originally incurred by the application.

Figure 6 shows the prefetch coverages for different prefetch schemes, illustrating significant coverage, especially for SPECfp. On roughly half of the programs the coverage achieved by future execution is higher than that achieved by the stream prefetcher. The other half favors stream prefetching. This result is somewhat surprising because future execution employs a more accurate form of value prediction than the stream prefetcher does. Hence, one would expect the coverage of future execution to be at least as high as that of a good stream prefetcher. This conundrum is explained by the following observation. The value predictor that assists the future execution makes predictions based on the local history of values produced by a particular static instruction, while the stream prefetcher observes only the global history of values. Therefore, the two techniques exploit different kinds of patterns, akin to local and global branch predictors, and in different applications different types of patterns dominate.

When stream prefetching is combined with future execution, the two techniques demonstrate significant synergy. In twelve programs (*bzip2*, *gcc*, *mcf*, *parser*, *vortex*, *ammp*, *apsi*, *equake*, *mesa*, *mgrid*, *sixtrack*, and *wupwise*) the coverage is at least 5% higher than when either technique is used alone. Overall, future execution increases the prefetching coverage from 31% to 47% on the integer and from 62% to 80% on the floating-point programs.

Next, we analyze the accuracy of our prefetching scheme by comparing the number of useful prefetches issued by the two prefetching mechanisms to the total number of prefetches issued. Figure 7 illustrates that the vast majority of the prefetches issued are useful in both the SPECint and the SPECfp programs with an average accuracy of over 70% for both techniques. There are a few interesting cases where stream prefetching has a much higher accuracy than future execution. They occur in the integer programs *crafty*, *gap*, and *twolf*. In all three cases useless prefetches occur in loops where many loads depend on the values of a loop-carried dependency passed through memory that is not pre-

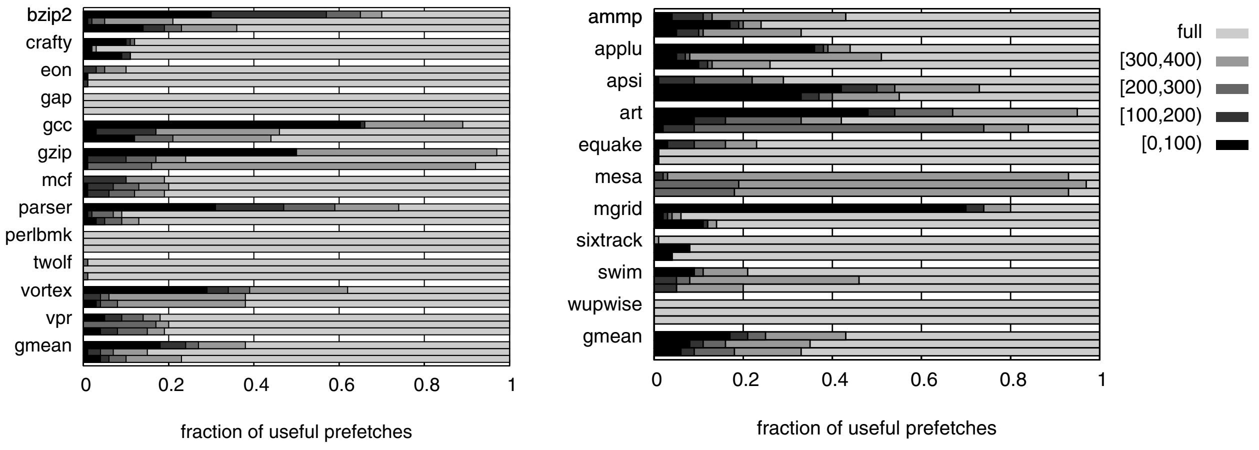


Figure 8. Timeliness of the prefetches

served by future execution. This results in computing the wrong addresses for load instructions and the fetching of useless data. However, even though the accuracy of the stream prefetcher is higher in those cases, the corresponding coverage is quite low, meaning that the higher accuracy does not translate into improved performance.

Finally, we investigate the prefetch timeliness of the different schemes. The prefetch timeliness indicates how much of the memory latency is hidden by the prefetches. The results are presented in Figure 8. For each program, the upper bar corresponds to the *fexec* model, the middle bar to the *spref* model, and the lowest bar represents the *spref+fexec* model. Each bar is broken down into five segments corresponding to the fraction of the miss latency hidden by the prefetches: less than 100 cycles (darkest segment), between 100 and 200 cycles, between 200 and 300 cycles, between 300 and 400 cycles, and over 400 cycles (lightest segment). Therefore, the lightest segment represents the fraction of prefetches that hide the full memory latency.

Both future execution and stream prefetching are quite effective at hiding the memory access latency. In case of future execution, 65% of the prefetches in SPECint and 55% of the prefetches in SPECfp are completely timely, fully eliminating the associated memory latency. For both the integer and the floating-point programs, only 25% of the prefetches hide less than 100 cycles of latency (one quarter of the memory latency). The timeliness of future execution prefetches can be improved by adjusting the prediction distance of the future value predictor. For example, increasing the prediction distance from 4 to 8 (results not shown) increases the number of completely timely prefetches for most of the programs with a low prefetch timeliness by at least 15%.

Overall, this section demonstrates that prefetching based on future execution is quite accurate, significantly improves the prefetching coverage over stream prefetching, and provides timely prefetches, which may be further improved by dynamically varying the prediction distance.

5.3 Comparison with Runahead Execution

The previous subsections showed that prefetching based on future execution is quite effective and provides significant speedups over the baseline with an aggressive stream prefetcher. In this section we compare our mechanism to runahead execution, another recently proposed execution-based prefetching technique.

The concept of runahead execution was first proposed for in-order processors [2] and further extended to perform prefetching for out-of-order architectures [10]. The runahead architecture "nullifies" and retires all memory operations that miss in the L2 cache and remain unresolved at the time they get to the ROB head. It starts by taking a checkpoint of the architectural state. Then it retires the missing load and the processor enters runahead mode. In this mode the instructions proceed largely normally except for two major differences. First, the instructions that depend on the result of the load that was "nullified" do not execute but are nullified as well. They commit an invalid value and retire as soon as they reach the head of the ROB. Second, store instructions executed in runahead mode do not overwrite data in memory. When the original "nullified" memory operation completes, the processor rolls back to the checkpoint and resumes normal execution. All register values produced in runahead mode are discarded.

We implemented a version of runahead execution similar to the one described by Mutlu et al. [10]. Runahead execution prefetches data into the L1 cache. To make a fair

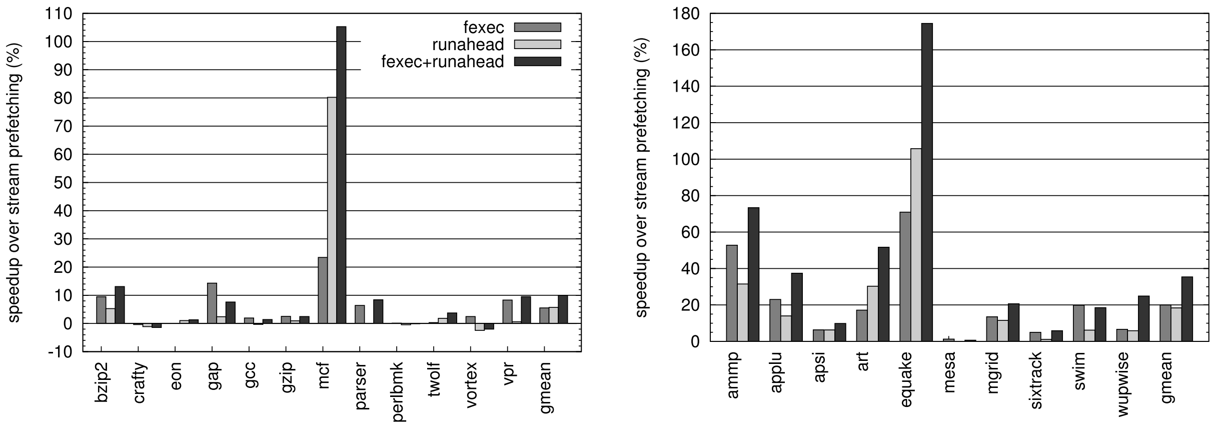


Figure 9. Comparison with runahead execution

comparison, we changed our implementation of future execution so that it also prefetches into the L1 cache. To do so, we modified the L2 cache controller to deliver the cache blocks requested by the L1 cache of the future core to the private L1 caches of both cores.

Figure 9 shows the execution speedup of different techniques relative to the *spref* baseline. Overall, more applications benefit from future execution than from runahead execution, but the geometric-mean speedups provided by the two techniques when they are applied separately are about the same. On average, both techniques provide a speedup of 5% on SPECint and around 19% on SPECfp.

When both techniques are employed together, their cumulative effect is quite impressive. The average speedups rise to 10% and 35% for the integer and the floating-point programs, respectively. The interaction between future execution and runahead execution is especially favorable with *mcf*, *ammp*, *applu*, *art*, *equake*, *mgrid*, and *wupwise*, where the speedups are from 6% to 70% higher than when either of the techniques is used alone. There is one case where adding runahead execution on top of future execution results in a lower speedup than when future execution is used alone (*gap*). We suspect this is caused by the corruption of the value prediction tables while the processor is in runahead mode, which causes the FE mechanism to issue incorrect prefetches when the processor exits runahead.

6 Related Work

Hardware prefetching techniques based on outcome prediction typically use various kinds of value predictors (e.g., [7, 11, 15]) and/or pattern predictors to dynamically predict which memory addresses to prefetch from. Unlike previous approaches, FE employs value prediction only to provide initial predictions. These initial predictions are then used to compute all values reachable from the predictable nodes in the program dataflow graph to obtain predictions for otherwise unpredictable values. We demonstrate that our approach significantly improves the prediction coverage relative to conventional value prediction.

Similar to future execution, Zhou and Conte [22] used value prediction to speculatively compute unpredictable values of instructions currently held in the instruction window and speculatively issue load instructions. However, our mechanism provides better latency tolerance due to the use of future prediction and delivers a higher prediction coverage since the speculation scope is not limited by the number of instructions in the instruction window.

Thread-based prefetching techniques [9, 13, 14] typically use additional execution pipelines or idle thread contexts in a multithreaded processor to execute helper threads that perform dynamic prefetching for the main thread. Helper threads can be constructed statically or dynamically by specialized hardware structures. If a static approach is used, the prefetching threads are constructed manually [23] or are generated by the compiler [8]. Future execution is a hardware-only approach and does not rely on a programmer or on compiler assistance. Nevertheless, we believe our approach and software-controlled helper threads to be complementary.

If helper threads are constructed dynamically, a specialized hardware analyzer extracts execution slices from the dynamic instruction stream at run-time, identifies trigger instructions to spawn the helper threads, and stores the extracted threads in a special table. Examples of this approach include slice-processors [9] and dynamic speculative precomputation [1]. Even though future execution is also a hardware mechanism, it needs essentially no specialized hardware to create the prefetching thread, it does not require any storage for prefetching threads, and it works without thread triggers. The only storage structure necessary for future execution is the prediction table, which is modest in complexity and size and can, in fact, be shared with other performance-enhancing mechanisms such as predictor-directed stream buffers [17] or checkpointed early load retirement [5].

Many thread-based software and hardware techniques propose to use the register results produced by the speculative helper threads. Examples include the multiscalar architecture [18], threaded multiple path execution [20], thread-level data speculation [19], speculative data-driven multithreading [14], and slipstream processors [12]. Even though the idea to reuse already computed results sounds appealing, it introduces additional hardware complexity and increases the design and verification costs. The results produced by future execution are used solely for prefetching, which eliminates the need for any mechanism to integrate the results in the original thread. As a consequence, there is no need to verify the results produced by the future execution, making our mechanism completely recovery-free.

Runahead execution is another form of prefetching based on speculative execution [2], [10]. In runahead processors, the processor state is checkpointed when a long-latency load stalls the head of the ROB, the load is allowed to retire and the processor continues to execute speculatively. When the data is finally received from memory, the processor rolls back and restarts execution from the load. Future execution does not need to experience a cache miss to start prefetching, requires no checkpointing support or any other recovery mechanism and, as we demonstrate is this paper, works well in combination with runahead execution.

Similar to FE, the dual-core execution paradigm (DCE) [21] proposed by Zhou utilizes idles cores of a CMP to speed up single-threaded programs. Instead of using the idle core for prefetching, DCE uses it to extend the effective instruction window size of a single core by distributing the state of a single thread over the two cores. The main difference between FE and DCE is that the former technique tries to eliminate cache misses while the latter improves the ability to tolerate them.

7 Conclusion

This paper presents *future execution* (FE), a simple technique to hide the latency of cache misses in both regular and irregular applications using moderate hardware and no OS, ISA, programmer, or compiler support. FE harnesses the power of a second core in a CMP to prefetch data for a thread running on a different core of the same chip. It dynamically creates a prefetching thread by sending a copy of all committed, register-writing instructions to the second core. The innovation is that on the way to the second core, a future value predictor replaces each instruction's result in the prefetching thread with the result the instruction is likely to produce during its n^{th} next execution. Future execution then leverages the second core's execution capabilities to compute the prefetch addresses that could not be predicted with high confidence, which we found to greatly increase the prefetching coverage. FE requires only simple hardware and small chip area additions. Unlike previously proposed approaches, future execution does not need any thread triggers, features an adjustable lookahead distance, does not use complicated analyzers to extract prefetching threads, requires no storage for prefetching threads, is recovery free, and works on legacy code as well as new code. Overall, FE delivers a geometric-mean speedup of 10% over a baseline with an aggressive stream prefetcher for SPECcpu2000 applications. Furthermore, future execution is complementary to runahead execution and the combination of these two techniques raises the average speedup to 20%.

References

[1] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 306–317, 2001.

[2] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, pages 68–75, 1997.

[3] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential fcm: increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 207–218, 2001.

[4] http://www.spec.org/osg/cpu2000/.

[5] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Checkpointed early load retirement. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 16–27, 2005.

[6] E. Larson, S. Chatterjee, and T. Austin. Mase: a novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the The Second International Symposium on Performance Analysis of Systems and Software*, pages 1–9, 2001.

[7] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[8] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual International Symposium on Computer Architecture*, pages 40–51, 2001.

[9] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *Proceedings of the 15th International Conference on Supercomputing*, pages 321–334, 2001.

[10] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture*, pages 129–140, 2003.

[11] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual International Symposium on Computer Architecture*, pages 24–33, 1994.

[12] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 269–280, 2000.

[13] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, 1998.

[14] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 37–49, 2001.

[15] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, 1997.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[17] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 42–53, 2000.

[18] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[19] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the The Fourth International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[20] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th annual International Symposium on Computer Architecture*, pages 238–249, 1998.

[21] H. Zhou. Dual-core execution: building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[22] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the 17th annual International Conference on Supercomputing*, pages 326–335, 2003.

[23] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th annual International Symposium on Computer Architecture*, pages 2–13, 2001.