Future Execution: A Prefetching Mechanism that Uses Multiple Cores to Speed up Single Threads

ILYA GANUSOV and MARTIN BURTSCHER
Cornell University

This paper describes future execution (FE), a simple hardware-only technique to accelerate individual program threads running on multicore microprocessors. Our approach uses available idle cores to prefetch important data for the threads executing on the active cores. FE is based on the observation that many cache misses are caused by loads that execute repeatedly and whose address-generating program slices do not change (much) between consecutive executions. To exploit this property, FE dynamically creates a prefetching thread for each active core by simply sending a copy of all committed, register-writing instructions to an otherwise idle core. The key innovation is that on the way to the second core, a value predictor replaces each predictable instruction in the prefetching thread with a *load immediate* instruction, where the immediate is the predicted result that the instruction is likely to produce during its nth next dynamic execution. Executing this modified instruction stream (i.e., the prefetching thread) on another core allows to compute the future results of the instructions that are not directly predictable, issue prefetches into the shared memory hierarchy, and thus reduce the primary threads' memory access time. We demonstrate the viability and effectiveness of future execution by performing cycle-accurate simulations of a two-way CMP running the single-threaded SPECcpu2000 benchmark suite. Our mechanism improves program performance by 12%, on average, over a baseline that already includes an optimized hardware stream prefetcher. We further show that FE is complementary to runahead execution and that the combination of these two techniques raises the average speedup to 20% above the performance of the baseline processor with the aggressive stream prefetcher.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design, Performance

Additional Key Words and Phrases: Future execution, prefetching, memory wall, chip multiprocessors

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

Authors' address: Ilya Ganusov and Martin Burtscher, Computer Systems Laboratory, Cornell University, Frank H.T. Rhodes Hall, Ithaca, NY 14853; email: {ilya,burtscher}@csl.cornell.edu.

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 consequence, 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 prediction-based prefetching methods have proved 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 caused 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 load, there is a high probability that the same load will be reexecuted soon. Therefore, whenever a load instruction is executed, we issue a copy of that load in another core of the same CMP with the address for the *n*th next instance to perform a prefetch into the shared memory hierarchy.

Value predictors can be used to determine the likely address each load is going to reference in the *n*th next "iteration." 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 data-flow 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 data-flow 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's data-flow graph, which greatly increases the number of instructions for which an accurate prediction is available.

Our mechanism uses a second core in a CMP to perform the future execution. We propose to use an idle core instead of a specialized execution engine, because it simplifies the design and allows the idle execution resources of the second core to be put to good use. Whenever additional threads need to be run on the CMP, the prefetching activity is canceled so that the second core can be utilized by another thread for nonspeculative computation.

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. A prediction is then made to obtain the likely value the instruction is going to produce during its *n*th next execution. If the confidence of the prediction is high, the instruction is replaced with a *load-immediate* instruction, where the immediate is the predicted result. Instructions with a low prediction confidence remain unmodified. After that, the processed stream

of instructions is sent to the second core, 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 reexecute in the future, the second core essentially executes n "iterations" ahead of the nonspeculative 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.

All major high-performance microprocessor manufacturers have announced or are already selling chips with at least two cores. Future generations of these processors will undoubtedly include more cores. While multiple cores are beneficial in multiprogrammed environments, the performance of individual computation threads does not improve and may even suffer a penalty because of increased contention for shared-memory hierarchy resources. In many cases, a programmer might have to manually parallelize applications in order to get a benefit from multiple cores, which increases the software complexity and cost. In light of this trend, architectural techniques that smooth the transition from single to multicore computing are becoming very important [Rattner 2005]. We believe that future execution provides a way for multicore processors to provide immediate benefits and presents a relatively simple yet effective architectural enhancement to exploit additional cores to speed up individual threads without the need for any programmer intervention or compiler support.

Future execution presents a novel way of generating prefetch addresses for loads, producing irregular sequences of addresses that are not directly predictable. Unlike previous proposals on prefetching via preexecution [Moshovos et al. 2001; Roth and Sohi 2001; Roth et al. 1998], FE does not require explicit program data-flow analysis to extract prefetching threads, either in software or hardware. In addition, future execution can prefetch dynamically distant misses and cover very long load latencies. Furthermore, it allows to adaptively change the prefetching distance n through a simple adjustment in the predictor. Finally, FE requires no live-in register variables from the target thread. As a consequence, it has a low thread startup cost (no hardware context needs to be copied) and can instantly stop or resume the execution of the prefetching thread depending on the availability of an idle core.

The next section discusses the implementation of the FE microarchitecture. We start by presenting a quantitative analysis of the observations that inspired our design. We then 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 simple implementation delivers an average 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 12%. Finally, we demonstrate that future execution is complementary to prefetching based on runahead execution and that both approaches exhibit significant synergy when used together.

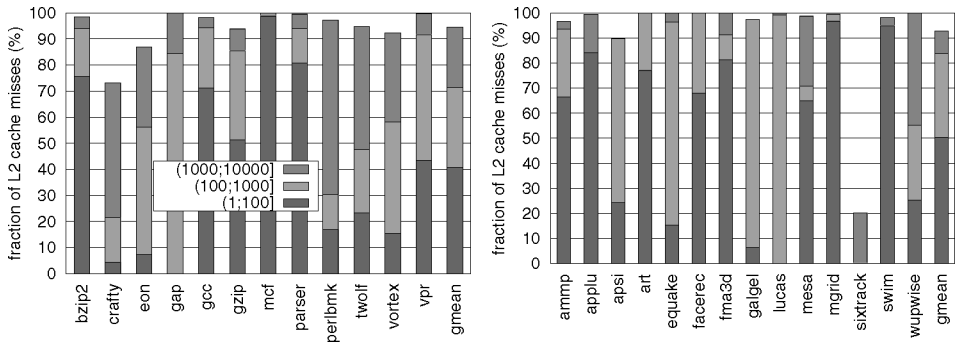

Fig. 1. Execution distance measured in number of instructions between the loads that result in an L2 cache miss and the previous dynamic execution of the same static loads.

2. MOTIVATION

In this section we present a quantitative analysis of the common program properties that are exploited by future execution. All 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. Note that we call any repetitively executed instruction a loop instruction and that FE is completely unaware about the location of loops in a program. 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, and between 1000 and 10,000 dynamic instructions. The taller the bar, the more often that range of instruction distances occurred. The total height of the bar represents the fraction of L2 cache misses that occur in loops with less than 10,000 instructions per iteration.

The data show that, on average, from 70 to 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 the same load is going to reference in the nth next iteration. Since, in most cases, the time between the execution of a load instruction and its next dynamic execution is relatively short, this approach is unlikely to prefetch much too early.

Analyzing the instructions in the data-flow graphs of the problem loads, we found that while problem load addresses might be hard to predict, the inputs to their data-flow graphs often are not. Therefore, even when the miss address itself is unpredictable, it is frequently possible to predict the input 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 SPEC-cpu2000 programs that can potentially be predicted and prefetched by future value prediction and by future execution one iteration ahead of the main

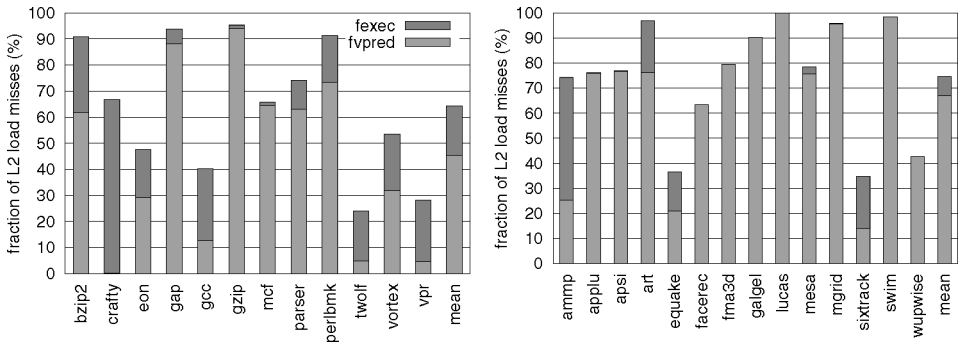

Fig. 2. Distribution of cache-miss addresses that can be correctly predicted directly by a future value predictor (*fvpred*) and using future execution (*fexec*).

program execution. The lower portion of each bar represents the fraction of misses that is directly predictable by a stride-two-delta (ST2D) value predictor [Sazeides and Smith 1997]. The upper bar shows how many miss addresses that are not predictable by the ST2D predictor can be correctly obtained by predicting the inputs of the instructions in the data-flow graph of the missing loads with the ST2D predictor and computing the resulting address. The height of the stacked bar indicates the total fraction of misses that can potentially be correctly predicted. To measure the potential prediction coverage of future execution, we reconstruct the data-flow graph of each problem load whenever a cache miss occurs, compare it to the data-flow graph of the same static load during its previous execution, extract the part of the data-flow graph that is the same, and then check if the values provided by the future value predictor during the previous execution would have allowed to correctly compute the load address referenced by the load instruction in the current iteration. We limit the size of the data-flow graph that we analyze to 64 instructions. This potential study ignores the effects of unpredictable loop-carried dependencies passed through memory, i.e., all load instructions with predictable addresses are assumed to fetch correct data one iteration ahead.

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. One-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. An array of pointers A is traversed, each pointer is dereferenced, and the resulting data are passed to the function "foo." Assume that the 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).

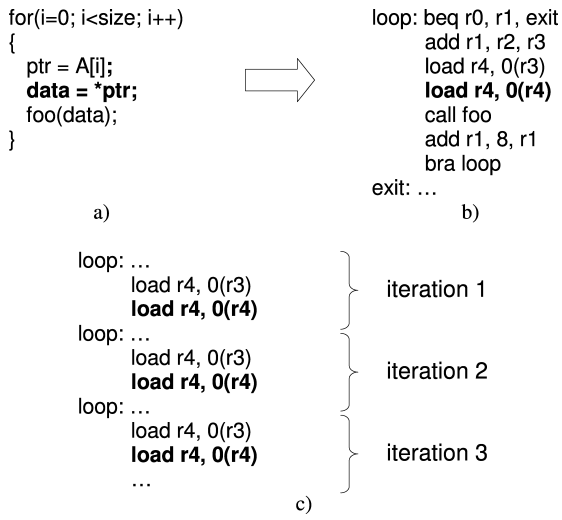

Fig. 3. Code example.

A conventional predictor will not be able to predict the address of the problem instruction since there is no regularity in the address stream for this instruction. However, the address references of instruction `load r4, 0(r3)` are regular because each instance 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 this instruction, 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 two load instructions to the second core in commit order and future predicting the first instruction makes it possible to compute the addresses of the second load that will be referenced by the main program during the next iterations.

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 execution engine with private L1 caches. The L2 cache is shared between the two cores. Conventional program execution is performed on the "left" core while future execution is performed on the "right" core. To support FE, we introduce a unidirectional communication link between the cores with a 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 either core in a negative way. The following subsections describe the necessary hardware support and the operation of FE in greater detail.

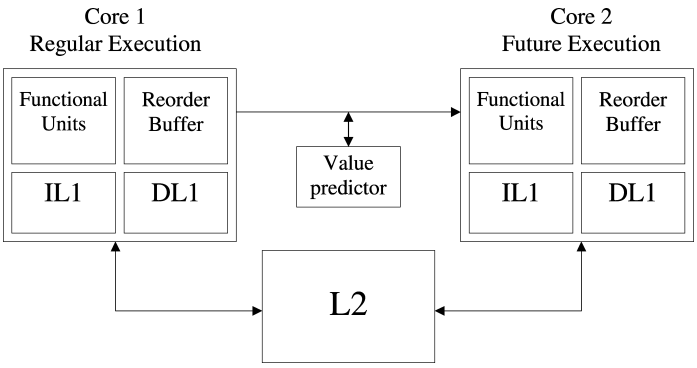

Fig. 4. The FE architecture.

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 decoded instruction, result value, and a partial PC to index the value predictor table. Stores, branches, jumps, calls, returns, privileged instructions, system calls, and arithmetic floating-point instructions are not transmitted. If the communication link's buffer is full, further committed instructions are dropped and not transmitted to the second core. Each sent instruction passes through the value predictor, updates the predictor with its current output, and requests a prediction of the value it is likely to produce in the nth next dynamic instance. Each prediction is accompanied by a confidence estimation [Lipasti et al. 1996].

If the confidence of the prediction is high, the corresponding instruction is replaced by a *load-immediate* instruction, where the immediate is the predicted result. If the predicted instruction is a memory load, an additional nonbinding prefetch instruction for that load's address is generated right before the *load-immediate* instruction. This allows the future core to prefetch this data without stalling the pipeline if the memory access misses in the cache. All instructions with a low prediction confidence remain unmodified.

After that, the processed stream of committed instructions is sent to the second core, where it is injected into the dispatch stage of the pipeline. Since instructions are transmitted in decoded format, they can bypass the fetch and decode stages. Instruction dispatch proceeds as normal—each instruction is renamed and allocated a reservation station 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. 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 the FE mechanism as it avoids stalling the pipeline. Timed-out instructions set the invalid bit in the corresponding result register. The execution of instructions that depend on the invalidated result is suppressed.

When entering FE mode, i.e., before the prefetching thread starts being injected into the available idle core, all registers of that core are invalidated. This flash invalidation of all registers occurs only once before the future execution thread is launched. The invalid register bits are gradually reset by the executed future instructions that have valid inputs. For example, *load-immediate* instructions always make their target registers valid since they do not have input dependencies. This register-invalidation policy suppresses the execution of all further instructions whose inputs cannot be predicted or computed with high confidence.

Note that the implementation of future execution in this paper differs from the implementation used in previous studies [Ganusov and Burtscher 2005; Ganusov 2005]. This study simplifies the implementation so that all instruction transformations take place outside the core. For example, our previous implementation required special logic in the dispatch stage of the pipeline to fill in the result field of the ROB and RS entries of FE instructions with predicted values. The implementation presented in this paper is more intuitive, requires no additional dispatch logic to support future execution, and features a simpler way to suppress the execution of unpredictable instructions.

3.2 Hardware Support

Future execution requires additional hardware support to transmit decoded instructions, their result values, and partial PCs to the value predictor between the cores. Depending on the microarchitecture, the ROB may have to be augmented to hold the necessary data until instruction retirement. In this study, the communication bandwidth corresponds to the commit width (four instructions/cycle), which we believe to be a reasonable bandwidth for a unidirectional on-chip point-to-point link. Since it is rare for a microprocessor to fully utilize its commit bandwidth for a long period of time, and because not all instructions need to be transmitted to the second core, it may be possible to curtail the bandwidth without significant loss of performance. For example, Section 5.4 demonstrates that the communication bandwidth can be reduced to two instructions/cycle with little effect on the efficiency of the FE mechanism.

The value-prediction module resides between the two CMP cores. We use a relatively simple, PC-indexed stride-two-delta predictor [Sazeides and Smith 1997] with 4096 entries. The predictor estimates the confidence of each prediction it makes using 2-bit saturating up–down counters. The confidence is incremented by one if the predicted value was correct and decremented by one 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., a DFCM predictor [Goeman et al. 2001]) may lead to higher performance.

To support the execution of the future instruction stream, a multiplexer has to be added in front of the dispatch stage of the pipeline. In FE mode, the multiplexer directs instructions to be fetched from the receive buffer of the communication link. In normal mode, instructions are fetched by the processor's front end.

Table I. Simulated Processor Parameters

Processor	
Fetch/dispatch/commit width	4/4/4
I-window/ROB/LSQ size	64/128/64
Physical registers	184
LdSt/Int(IntMult)/FP units	2/4(2)/2
Branch predictor	16k-entry bimodal/gshare hybrid
RAS entries	16
BTB	2k entries, 2-way
Branch misprediction penalty	Minimum 12 cycles
Memory Subsystem	
Cache sizes	64kB IL1, 64kB DL1, 2MB L2
Cache associativity	2-way L1, 8-way L2
Cache load-to-use latencies	3 cyc L1, 12 cyc L2
Cache line sizes	64B L1, 64B L2
Cache MSHRs	16 L1, 24 L2
Main memory latency	Minimum 400 cycles
Main memory bus	Split-trans., 8B-wide, 4:1 frequency ratio, contention, queuing, bandwidth modeled between L2 and main memory, 16 streams, max. prefetch distance: 8 strides
Hardware stream prefetcher	
Future Execution Hardware	
Future value predictor	4k-entry ST2D, 2bc conf. estimator
Prediction distance	4 strides ahead
Intercore communication link	5-cycle latency, 4 insns/cycle bandwidth
Communication link buffer size	64 instructions

The processor's register file may have to be extended to accommodate an invalid bit for each physical register. Only one extra bit per register is needed. Many modern microprocessors already include some form of dirty or invalid bits associated with each register that could be utilized by the FE mechanism.

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 for this purpose. 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 [Larson et al. 2001]. The baseline is a two-way CMP consisting of two identical four-wide dynamic superscalar cores that are similar to the Alpha 21264 (Table I). The minimum memory latency for the baseline processor is 400 cycles. We model bandwidth and contention on the memory bus and limit the number of outstanding bus transactions to 32. We used CACTI 3.2 [Shivakumar and Jouppi 2001] to determine the simulated cache access latencies.

The communication latency between the two cores is five cycles and the communication bandwidth corresponds to the commit width (four

instructions/cycle). Note that FE is not very sensitive to the communication latency (see Section 5.4). In all modeled configurations, we assume that one of the cores in the CMP can be used for future execution. We also simulate a configuration with an aggressive hardware stream prefetcher [Palacharla and Kessler 1994] between the shared L2 cache and main memory. The stream prefetcher tracks the global 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 [Sazeides and Smith 1997] that predicts values four 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.

We use all 26 integer and floating-point programs from the SPECcpu2000 benchmark suite [http://www.spec.org/osg/cpu2000/]. 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 only exception to this rule is the program *vpr*, which has two reference inputs. We simulate only one of the reference inputs (routing), because SimpleScalar could not simulate *vpr* correctly with the second input (placement). The C programs were compiled with Compaq's C compiler V6.3-025 using "–O3 –arch ev67 –non_shared" plus feedback optimization. The C++ and Fortran 77 programs were compiled with g++/g77 V3.3 using "–O3 –static." The Fortran 90 programs were compiled with Compaq's f90 compiler V5.3-915.

We use the SimPoint 3.1 toolset [Sherwood et al. 2002] and SimpleScalar's *sim-safe* simulator to identify representative simulation points. Each program is simulated for 500 million instructions after fast-forwarding past the number of instructions determined by SimPoint. Note that we use simulation points that are different from the simulation intervals use in previous studies on future execution [Ganusov and Burtscher 2005; Ganusov 2005]. Those studies use SimPoint 1.1 and the ATOM binary instrumentation and analysis tool [Srivastava and Eustace 1994] to produce the simulation points.

Table II provides information about the benchmarks used. The cache miss rates shown are local. Out of the 26 programs used in this study, four integer and two floating-point 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 1400% 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 48.6%. Note that for several memory-bound programs (e.g., *mcf*, *art*, *equake*, and *swim*), the perfect L2 cache speedup cannot be obtained even with a perfect prefetching scheme because of memory bus bandwidth limitations.

Table II. Benchmark Suite Details (for the Simulated Intervals of 500 M Instructions)

App.	NoPref IPC	Loads (M)	L1 Miss Rate (%)	L2 Miss Rate (%)	Perfect L2 Speedup (%)
bzip2	1.56	143.94	1.47	0.70	24.55
crafty	1.92	155.85	0.82	0.07	2.10
eon	1.75	148.32	0.12	0.00	0.20
gap	1.44	127.02	0.36	1.22	24.62
gcc	1.33	180.30	2.56	1.10	30.38
gzip	1.69	113.80	3.52	1.77	3.35
mcf	0.04	209.74	23.11	48.62	1399.55
parser	0.84	125.79	2.56	2.40	103.48
perlbmk	1.77	146.69	0.30	0.09	7.42
twolf	1.29	144.63	5.04	0.05	1.84
vortex	2.09	130.30	0.75	0.39	34.55
vpr	0.54	165.19	3.10	3.29	120.41
ammp	1.44	132.32	3.90	1.35	31.92
applu	0.97	114.27	2.10	6.64	198.38
apsi	2.43	120.67	1.52	0.78	10.81
art	0.69	148.46	19.72	6.97	183.48
equake	0.26	234.53	7.50	24.40	675.56
facerec	1.15	123.90	2.40	4.83	178.51
fma3d	0.80	150.17	3.01	7.10	217.41
galgel	2.49	184.71	2.94	0.29	2.00
lucas	0.42	80.69	7.91	23.68	431.63
mesa	1.92	129.16	0.32	0.55	23.04
mgrid	0.91	183.09	2.42	19.02	203.19
sixtrack	2.62	96.83	0.23	0.18	4.77
swim	0.42	123.65	8.51	19.11	689.75
wupwise	1.30	114.31	1.15	3.60	134.67

5. EXPERIMENTAL RESULTS

In this section, we experimentally measure the effectiveness of our proposed mechanism. Section 5.1 evaluates the performance of prefetching, based on future execution and compares the speedups with those of 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 several variations of runahead execution and show that the two techniques are complementary to each other. Finally, in Section 5.4 we study the sensitivity of future execution to several parameters of the baseline microprocessor, such as the minimum memory latency, the intercore communication delay/bandwidth, and the prefetch distance.

5.1 Execution Speedup

In this section, we compare the performance impact of our prefetching technique to a stream-based hardware prefetcher. The base machine for this experiment is described in Table I. It represents an aggressive superscalar processor without hardware prefetching. We model three processor configurations:

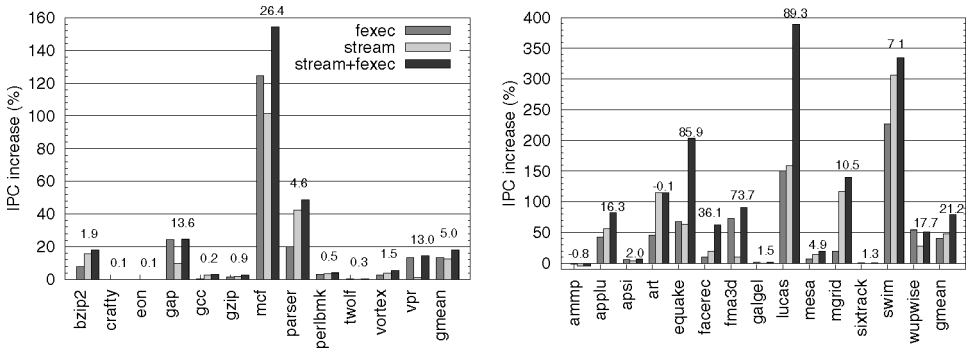

Fig. 5. Execution speedup.

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 (*stream*) [Palacharla and Kessler 1994], and the baseline with stream prefetching as well as future execution (*stream* + *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 the SPECint and the SPECfp benchmarks is different. The percentages on top of the bars are the speedups of future execution combined with stream prefetching (*stream* + *fexec*) over stream prefetching alone (*stream*).

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 peaks of 306% for *swim* and 159% for *lucas*. The average speedup over the SPECint programs is 13%, while the SPECfp applications experience an average speedup of 48%. Note that we tuned the parameters of the stream prefetcher to maximize the prefetching timeliness and to minimize cache pollution on our benchmark suite.

When the model with only future execution is compared to the model with only stream prefetching, future execution outperforms stream prefetching on five programs, while stream prefetching is better on nine. The remaining twelve programs achieve about the same performance with 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 more timely 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 (except for Section 5.4) to make the results comparable. Nevertheless, the *fexec* model provides significant speedup (over 5%) for 12 programs, with an average speedup of 13% for the integer and 40% for the floating-point programs, and a maximum of 227% 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 with no prefetching. Moreover, this model has a 10% higher

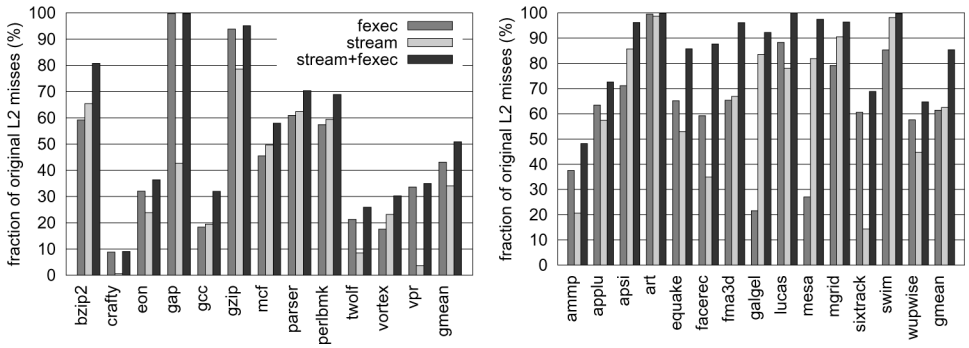

Fig. 6. Prefetch coverage.

IPC than the baseline with stream prefetching. Out of the 26 programs used in our study, 12 significantly benefit (over 5% 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 5 and 21%, respectively. This indicates that future execution and stream prefetching interact favorably and complement each other by prefetching different classes of load misses.

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

In this section, we provide insight into the performance of prefetching based on future execution 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 a cache missing memory operation) 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 one-half of the programs, the coverage achieved by future execution is higher than that achieved by the stream prefetcher. 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 the global history of values. Therefore, the two techniques exploit different kinds of patterns, akin to local and global branch predictors.

When stream prefetching is combined with future execution, the two techniques demonstrate significant synergy. In eleven programs (*bzip2*, *gcc*, *perlbmk*, *ammp*, *applu*, *apsi*, *equake*, *facerec*, *fma3d*, *lucas*, and *mesa*) the coverage is at least 10% higher than when either technique is used alone. Overall,

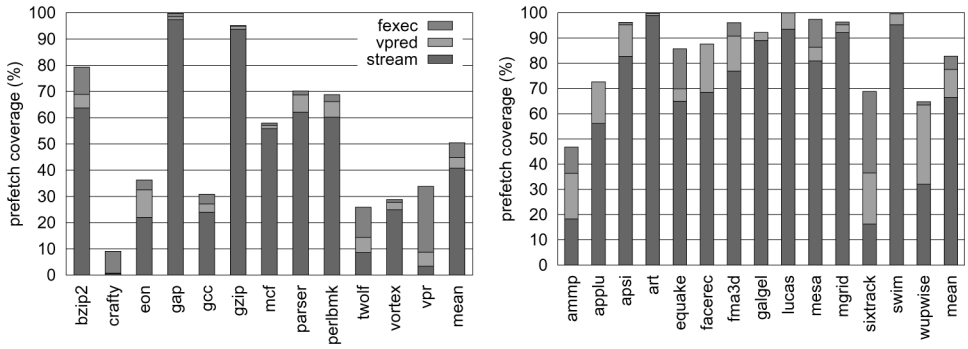

Fig. 7. Distribution of cache misses that were prefetched by a stream prefetcher (*stream*), based on value predictions (*vpred*), and using future execution (*fexec*).

future execution increases the prefetching coverage from 34 to 51%, on the integer, and from 63 to 85%, on the floating-point programs.

Figure 7 shows the breakdown of the prefetch coverage for the case when stream prefetching is combined with future execution (*stream* + *fexec* configuration). The lower segment of each bar corresponds to the prefetches initiated by the stream prefetcher. The middle portion shows how many prefetches were issued based on predictions provided by the value predictor. The upper part of each bar represents the portion of issued prefetch addresses that required the execution of instructions to compute the correct prefetch address. The height of the stacked bar indicates the total fraction of misses that were prefetched. Note that the height of the bar is sometimes slightly less than reported in Figure 6, because the origin of some of the prefetches could not be determined. In this study, if multiple prefetch mechanisms issued prefetches for the same memory location, the mechanism that issued the earliest prefetch is given credit for that prefetch request.

Figure 7 illustrates that future value prediction and future execution provide a significant coverage increase over the stream prefetcher (over 10%) for 16 out of the 26 applications used in our study. Out of these 16 applications, 6 programs benefit mostly from future execution, 8 programs owe most of the coverage increase to future value prediction, and 2 programs benefit roughly equally from value prediction and future execution. Note that in seven programs the addition of future execution makes the stream prefetcher more effective. For example, *gap* gets almost all of its misses prefetched by the stream prefetcher, but the data in Figure 6 demonstrate that stream prefetching can prefetch less than a one-half of the cache misses without future execution. We suspect that this is caused by favorable interactions between the loads issued from the future core and the stream prefetcher, where future loads enable more precise and earlier identification of important streams that are then further prefetched by the stream prefetcher. On average, future execution increases the coverage provided by the stream prefetcher from 34 to 41%, on the integer applications, and from 63 to 66%, on the floating-point programs.

We next analyze the accuracy of our prefetching scheme by comparing the number of useless prefetches issued by the prefetching mechanisms to the total

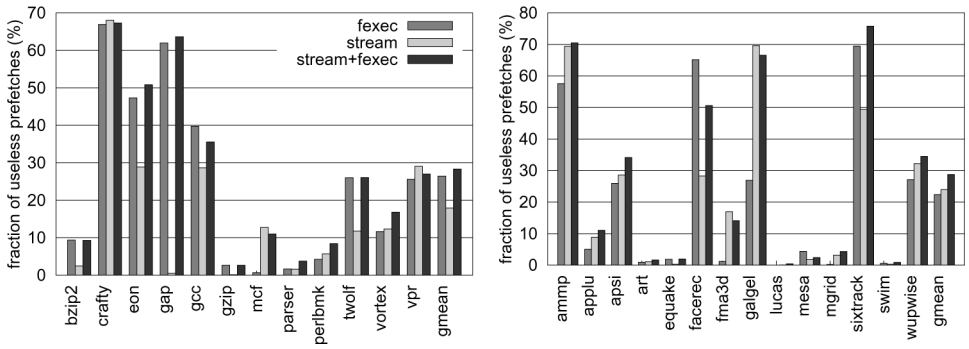

Fig. 8. Percentage of useless prefetches issued by different prefetching mechanisms relative to the total number of prefetches issued.

number of prefetches issued. We categorize a prefetch request as useless if the prefetched data is evicted from the cache before being used by the main thread. Figure 8 shows the percentage of useless prefetches associated with the two prefetching schemes. The results illustrate that a large majority of the prefetches issued are useful in both the SPECint and the SPECfp programs with over 70% of useful prefetches for both techniques. There are a few interesting cases where stream prefetching causes much fewer useless prefetches than future execution. They occur in the programs *eon*, *gap*, *twolf*, *facerec*, and *sixtrack*. We find that useless prefetches occur in loops where many loads depend on the values of a loop-carried dependency passed through memory that is not preserved by future execution. This results in computing the wrong addresses for load instructions and the fetching of useless data. We suspect that in *sixtrack* many prefetches are issued too far in advance and get evicted from the cache before being used. However, even though the accuracy of the stream prefetcher is higher in those cases, the coverage for many of the programs is quite small, meaning that the higher accuracy does not noticeably improve the 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 9. For each program, the upper bar corresponds to the *fexec* model, the middle bar to the *stream* model, and the lowest bar represents the *stream* + *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 minimum 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

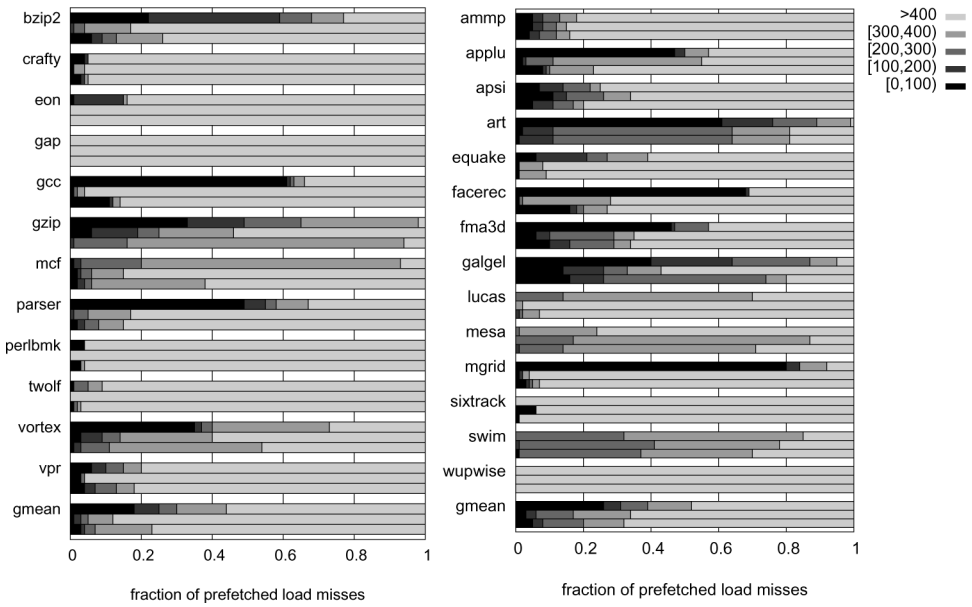

Fig. 9. Timeliness of the prefetches.

prefetches can be improved by adjusting the prediction distance of the future value predictor. For example, increasing the prediction distance from 4 to 8 increases the number of completely timely prefetches for most of the programs, with a low prefetch timeliness by at least 15%, resulting in significant increase in performance (see Section 5.4).

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 several variations of runahead execution, a recently proposed execution-based prefetching technique.

The concept of runahead execution was first proposed for in-order processors [Dundas and Mudge 1997] and then extended to perform prefetching for out-of-order architectures [Mutlu et al. 2003]. The runahead architecture "nullifies" and retires all memory operations that miss in the L2 cache and remain unresolved at the time they reach the ROB head. It starts by taking a checkpoint of the architectural state and retiring the missing load before the processor enters runahead mode. Once in runahead mode, instructions execute 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 result 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. [2003]. Runahead mode is triggered by load instructions that stall for more than 30 cycles. Store data produced in runahead mode is retained in a runahead cache, which is flushed upon the exit from runahead mode.

In addition to this conventional version of runahead execution, we implement and evaluate two extensions. First, we employ value prediction to supply load values for the long-latency load instructions. When such loads time out, a stride-two-delta value predictor provides a predicted load value and a prediction confidence. If the confidence is above threshold, the predicted value is allowed to propagate to the dependent instructions. If the confidence is below threshold, the result of the load instruction that timed out is invalidated in the same way loads are invalidated in the conventional runahead mechanism.

Second, we implement the checkpointed early load-retirement mechanism (CLEAR) [Kirman et al. 2005], which attempts to avoid squashing the correct program results produced in runahead mode. The CLEAR mechanism utilizes value prediction to provide values for the load instructions that time out and is similar in spirit to checkpoint-assisted value prediction, as proposed by Ceze et al. [2004]. While the conventional runahead mechanism checkpoints the processor state only once before entering runahead mode, CLEAR checkpoints the processor state before every prediction that is made in runahead mode. If the value provided by a value predictor was incorrect, the processor state is rolled back to the checkpoint corresponding to that value prediction. However, if the prediction was correct, the corresponding checkpoint is released and the processor does not have to roll back after the long-latency memory operation completes.

Note that both runahead extensions that we evaluate in this study share value prediction and confidence estimation tables with the future execution mechanism. When runahead is used without future execution, only load instructions update the value predictor. When runahead and future execution are used together, every committed instruction updates the value predictor with the exception of stores, branches, and system calls. Our implementation of CLEAR assumes an unlimited number of available checkpoints.

Figure 10 shows the execution speedup of different techniques relative to the *stream* baseline. First, we compare the performance of future execution without runahead execution, shown in Figure 5, to the performance of different runahead schemes when they are used without future execution. Overall, the geometric-mean speedups of different runahead techniques and FE are similar when they are applied separately. On average, conventional runahead, runahead with value prediction, and CLEAR provide performance improvements of 2, 6.4, and 7.2% on SPECint and around 17% on SPECfp. FE provides 5% speedup on SPECint and 21% on SPECfp applications.

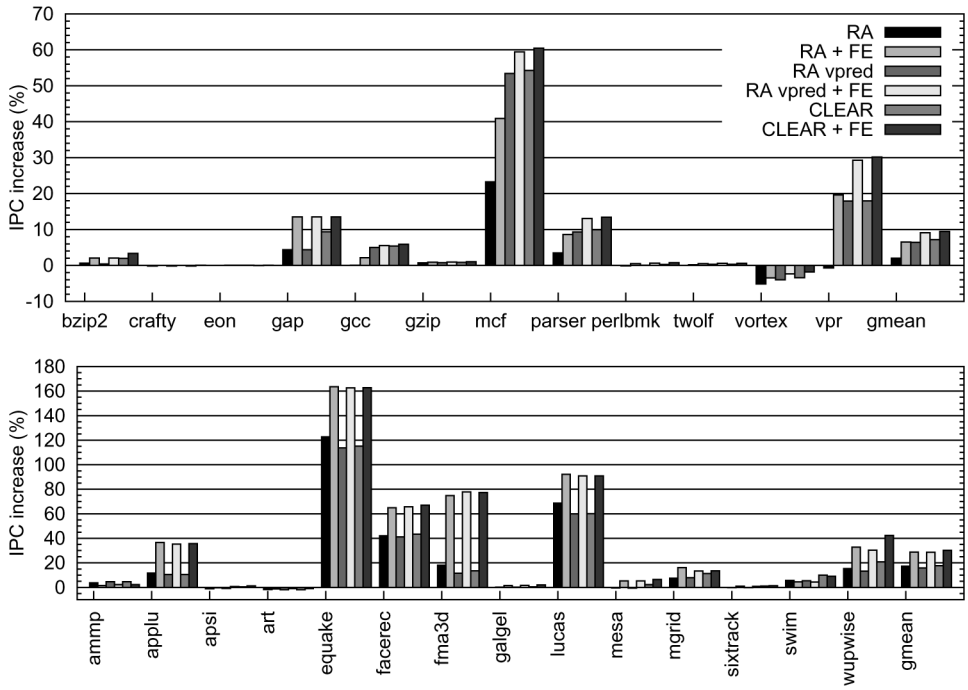

Fig. 10. Comparison with runahead execution.

When runahead execution and future execution are employed together, their cumulative effect is quite impressive. The average speedups for the conventional runahead implementation rise to 6.5 and 29% for the integer and the floating-point programs, respectively. The average speedups for runahead with value prediction also exhibits a significant boost with future execution, increasing from 6.4 to 9.1% for SPECint and from 16 to 29% for SPECfp applications. The CLEAR mechanism demonstrates similar performance improvements. The interaction between future and runahead execution is especially favorable with eight programs (*mcf*, *vpr*, *applu*, *facerec*, *fma3d*, *equake*, *mgrid*, and *wupwise*), where the speedups are from 5 to 50% higher than when either of the techniques is used alone.

Runahead allows prefetching cache misses that are within close temporal proximity of the long-latency load instructions that initiated runahead mode. Therefore, even though runahead execution obtains significant prefetch coverage while the processor is in runahead mode, its potential is limited by the length of the runahead period. On the other hand, FE prefetches can generally hide more latency than runahead prefetches, because the FE mechanism issues memory requests several iterations ahead of the current execution point. In spite of the better prefetch timeliness, FE's coverage is sometimes limited by the value prediction coverage and the regularity of the address-generating slices. The combination of runahead execution and future execution allows to exploit the strengths of both approaches, thus resulting in symbiotic behavior.

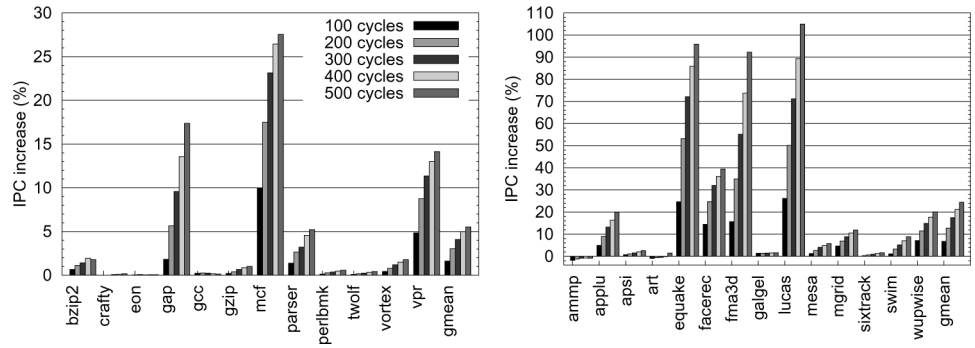

Fig. 11. Sensitivity of future execution to the main memory latency.

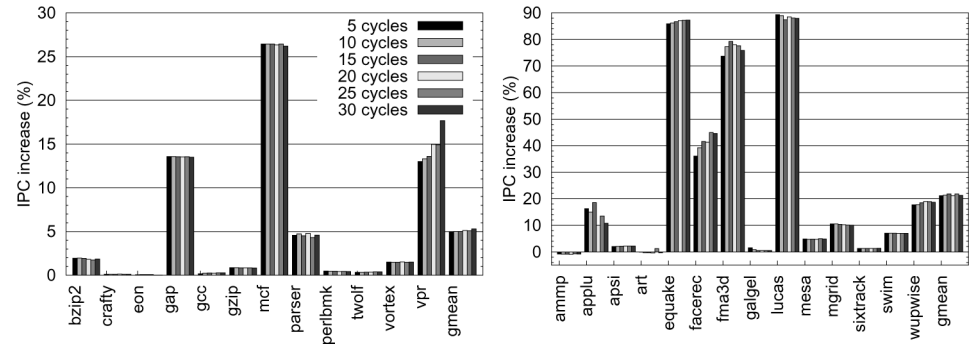

Fig. 12. Sensitivity of future execution to the intercore communication delay.

5.4 Sensitivity Analysis

In this subsection we evaluate the effectiveness of future execution when several hardware parameters are varied. First, we investigate the effect of the memory latency on FE's performance. Second, we compare the performance of FE configurations with different intercore communication latencies and bandwidth capabilities. Finally, we analyze the performance benefits provided by FE with different prefetch distances. All results in this subsection show the speedup provided by FE relative to the *stream* baseline.

Figure 11 shows the speedup provided by FE on processors with five different memory latencies, ranging from 100 to 500 cycles. Overall, the performance benefit for both integer and floating-point programs steadily increases with increasing memory latency. The SPECint speedup ranges from 1.6% for a relatively short 100-cycle memory latency to almost 5.5 for a 500-cycle latency. The average speedup for the SPECfp programs increases from 6.7 to 24.4%.

Figure 12 demonstrates how the communication latency between the cores affects the performance improvements provided by FE. As we increase the communication latency from 5 to 30 cycles, most of the applications show no significant changes in the amount of speedup obtained by future execution. The speedup changes by more than 5% only for four programs (*vpr*, *applu*, *facerec*, and *fma3d*). We observe that longer communication latencies seem to hurt the

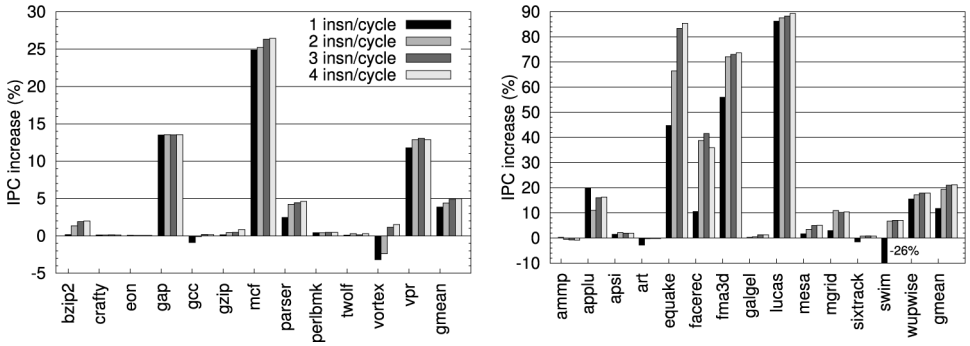

Fig. 13. Sensitivity of future execution to the intercore communication bandwidth.

performance benefit in *fma3d*. Future execution in *vpr* and *facerec* generally becomes more effective with the increasing communication delay, while *applu* demonstrates no correlation between the speedup and the communication latency. The geometric mean speedups for the SPECint and SPECfp applications change by less than 0.5%. FE is not very sensitive to the communication delay because of two main reasons. First, prefetching four iterations ahead hides the full memory latency for many of the applications and delaying a prefetch request by an additional 5–25 cycles still results in a timely prefetch. Second, even if the delayed prefetch does not hide the full memory latency, the communication delay constitutes only a small fraction of the total memory latency. Increasing the latency of a prefetched memory request by a few percent does not have a significant performance impact.

Figure 13 shows the sensitivity of future execution to the communication bandwidth between the cores. We vary the communication bandwidth from four to one instructions/cycle. The results show that decreasing the bandwidth from four to three instructions/cycle has almost no impact on the effectiveness of FE. If we cut the communication bandwidth in half to two instructions/cycle, only three programs experience a significant (over 5%) performance degradation (*vortex*, *applu*, and *equake*), while the geometric mean speedups stay practically unchanged. However, further reduction of the bandwidth to one instruction/cycle often makes the bandwidth insufficient for the effective operation of FE. In particular, six programs experience a significant performance degradation. Compared to the case where the communication bandwidth corresponds to the processor commit width (four instructions/cycle), the geometric mean speedup for integer programs decreases from 5 to 3.8%, while the IPC speedup for the floating-point applications drops from 21 to 12%. In most cases, this degradation is caused by a large number of instructions that are dropped because of the lack of space in the communication buffer. As a result, some prefetch addresses are never computed, while others are computed incorrectly and pollute the cache or the stream prefetcher's miss history (e.g., *swim*).

Next, we analyze the impact of the prefetch distance on the performance of future execution. We vary the prediction distance of the value predictor from one to ten iterations ahead and show the corresponding speedups in Figure 14. The

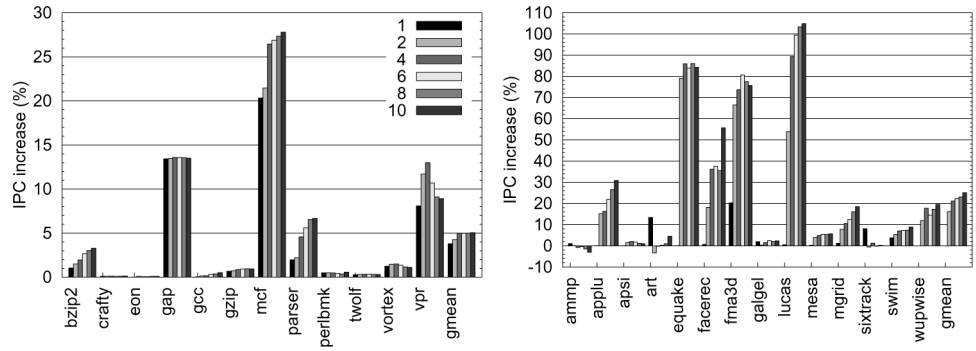

Fig. 14. Sensitivity of future execution to the future value prediction distance.

results show that most of the programs benefit from an increased prefetch distance. As one might expect, the prefetch coverage generally decreases slightly for larger lookaheads, but the reduction in coverage is compensated for by the improved timeliness of the prefetch requests. *Vpr* and *ammp* are the only programs where the decreasing prefetch coverage dominates the improved timeliness. Surprisingly, some programs (e.g., *bzip2*, *applu*, and *fma3d*) exhibit a growing prefetch coverage with increasing prefetch distance. This phenomenon occurs due to a favorable interaction between future execution and the stream prefetch engine. As we increase the prediction distance, the memory requests issued by future execution indirectly increase the prefetch distance of the stream prefetcher and thus make the stream prefetches more timely. Therefore, fewer loads time out in the future core and fewer address-generating slices are invalidated, enabling more future load addresses to be computed. We found that prefetching more than 10 strides ahead does not improve FE performance.

On average, increasing the future prediction distance from 1 to 10 iterations ahead increases the geometric mean speedup for integer applications from 3.2 to 5%, while the IPC speedup for the floating-point applications increases from 11 to 30% over the baseline with aggressive stream prefetching. These results suggest that future execution may greatly benefit from a dynamic mechanism to adjust the prediction distance.

Overall, this section demonstrates that prefetching based on future execution obtains performance improvements across microprocessor configurations with different memory latencies, intercore communication delay/bandwidth parameters, and future value prediction distances.

6. RELATED WORK

Hardware prefetching techniques based on outcome prediction typically use various kinds of value predictors [e.g., Palacharla and Kessler 1994; Sazeides and Smith 1997; Lipasti et al. 1996] and/or pattern predictors to dynamically predict which memory references should be prefetched. One of the first hardware prefetchers based on outcome prediction was the concept of stream buffers proposed by Jouppi [1990]. Subsequently, a number of other outcome prediction-based prefetching techniques was introduced. Examples include

stride prefetching [Fu et al. 1992], the Markov prefetcher [Joseph and Grunwald 1997], content-directed prefetching [Cooksey et al. 2002], tag-correlating prefetching [Hu et al. 2003], and dead-block-correlating prefetching [Lai et al. 2001].

The advantage of prefetching schemes based on outcome prediction is the ability to implement the schemes in the cache controller so that other parts of the microprocessor do not need to be modified. This way the implementation of the prefetching scheme can be decoupled from the design of the execution core, significantly lowering the complexity and the verification cost. The downside of such prefetching schemes is their limited coverage and ability to capture misses that exhibit irregular behavior.

Unlike previous approaches, future execution 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 data-flow graph, i.e., obtain predictions for otherwise unpredictable values. We demonstrate that our approach significantly improves the prediction coverage relative to conventional value prediction. Since future execution requires relatively simple modifications to the execution core of the microprocessor, we believe it provides a reasonable tradeoff between the implementation cost and the resulting performance improvement.

Similar to future execution, Zhou and Conte [2003] 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 because of the use of future predictions and delivers a higher prediction coverage because the speculation scope is not limited by the number of instructions in the instruction window.

Prefetching techniques based on preexecution [Moshovos et al. 2001; Roth and Sohi 2001; Roth et al. 1998] 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. Typically, these techniques create preexecution helper threads (PEHT) by extracting program slices that compute critical data addresses. Then they insert triggers for these helper threads into the original program. The execution of the helper threads at runtime precomputes critical data addresses ahead of the original program and issues prefetch requests. Helper threads can be constructed dynamically by specialized hardware structures or statically. If a static approach is used, the prefetching threads are constructed manually [Zilles and Sohi 2001] or generated by the compiler [Luk 2001] or a trace analysis tool [Roth and Sohi 2002]. If PEHTs are constructed dynamically, a 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 [Moshovos et al. 2001] and dynamic speculative precomputation [Collins et al. 2001].

FE has the following advantages over preexecution prefetching. First, FE allows to dynamically change the prefetching distance through a simple adjustment in the predictor. Second, since a PEHT is only able to execute once all inputs to the thread are available, it runs a higher risk of prefetching

late. FE, on the other hand, does not need all inputs to initiate prefetching. Third, if any load with dependent instructions in a PEHT misses in the cache, the prefetching thread will stall, preventing further prefetching. FE often breaks data-flow dependencies through value prediction and thus can avoid stalling the prefetch activity. Compared to software PEHT approaches, FE does not require recompilation or binary rewriting and thus can speedup legacy code.

On the other hand, prefetching based on preexecution can potentially provide a higher prefetching coverage than future execution since it is not limited by the predictability of the program data. In addition, PEHTs typically need to execute fewer instructions than FE and as such can operate profitably on the same core together with the main thread. Finally, preexecution requires no value prediction table and software approaches need no hardware support at all. Notwithstanding, we believe our approach may well be complementary to software-controlled pre-execution helper threads.

Slipstream prefetching [Ibrahim et al. 2003] is another form of a software-controlled preexecution that targets distributed shared-memory (DSM) applications. Slipstream prefetching threads represent a reduced version of the target computation threads. This reduced version dynamically skips the execution of shared memory stores and synchronization primitives and thus is able to run ahead of the target thread and generate an accurate address stream. As a result, slipstream prefetching can provide higher prefetching coverage than FE. However, the proposed approach targets only DSM applications and cannot speed up single-threaded programs.

Runahead execution is another form of prefetching based on speculative execution [Dundas and Mudge 1997; Mutlu et al. 2003]. 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 in this paper, works well in combination with runahead execution.

Similar to FE, hardware-only techniques, such as the minimal dual-core speculative multithreading architecture [Srinivasan et al. 2004] (SpMT) and the dual-core execution paradigm (DCE), proposed by Zhou [2005], utilize idle cores of a CMP to speed up single-threaded programs. The DCE approach effectively proposes to launch a runahead execution thread whenever a long-latency load instruction fully stalls the execution of a thread. The runahead core then tries to follow the program path and execute all instructions that do not depend on the results of load instructions that miss in the cache while invalidating all instructions that stall the execution. The regular core reexecutes all instructions committed by the runahead core. If the regular core detects that the runahead core deviated from the correct control path, it flushes the runahead core's pipeline and restarts runahead execution. In contrast to DCE, our FE technique never requires to check the results produced by the speculative threads and is recovery-free. In addition, DCE requires the nonspeculative core to redirect

the instruction fetch engine upon reaching the speculation point, while in FE the nonspeculative computation is not even aware that prefetching is taking place.

The SpMT approach spawns speculative threads on procedure calls, loop boundaries, or cache misses and executes them on another core. Speculative threads prefetch important data, precompute branch outcomes, and perform some useful computation that can later be integrated into the nonspeculative thread. The main difference between SpMT and future execution is that the former needs mechanisms to control the execution of the speculative threads by tracking the violation of memory and register dependences. Future execution generates prefetching threads that are completely decoupled from the nonspeculative execution, which eliminates the need for any checking mechanism and makes it recovery-free. In general, SpMT requires more complex hardware support, but may provide more performance benefit, since it allows the reuse of speculatively computed results.

7. CONCLUSION

This paper presents *future execution* (FE), a simple technique to hide the latency of cache misses using moderate hardware and no ISA, programmer, or compiler support. FE harnesses the power of a second core in a multicore microprocessor 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 value predictor replaces each predictable 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 significantly increase the prefetching coverage. FE requires only small chip area additions. Unlike previously proposed approaches, our mechanism 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, and works on legacy code, as well as new code. Overall, FE delivers a geometric-mean speedup of 12% over a baseline with an aggressive stream prefetcher on the SPECcpu2000 applications. Furthermore, future execution is complementary to runahead execution and the combination of these two techniques raises the average speedup to 20%.

REFERENCES

CEZE, L., STRAUSS, K., TUCK, J., RENAU, J., AND TORRELLAS, J. 2004. Cava: Hiding l2 misses with checkpoint-assisted value prediction. *IEEE Comput. Archit. Lett. 3*, 1, 7.

COLLINS, J. D., TULLSEN, D. M., WANG, H., AND SHEN, J. P. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. 306–317.

COOKSEY, R., JOURDAN, S., AND GRUNWALD, D. 2002. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 279–290.

DUNDAS, J. AND MUDGE, T. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*. 68–75.

FU, J. W. C., PATEL, J. H., AND JANSSENS, B. L. 1992. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. 102–110.

GANUSOV, I. 2005. Hardware prefetching based on future execution in chip multiprocessor architectures. M.S. thesis, Department of Electrical and Computer Engineering, Cornell University, Ithaca, New York.

GANUSOV, I. AND BURTSCHER, M. 2005. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. 350–360.

GOEMAN, B., VANDIERENDONCK, H., AND DE BOSSCHERE, K. 2001. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA'01)*.

HTTP://WWW.SPEC.ORG/OSG/CPU2000/.

HU, Z., MARTONOSI, M., AND KAXIRAS, S. 2003. Tcp: Tag correlating prefetchers. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. 317.

IBRAHIM, K. Z., BYRD, G. T., AND ROTENBERG, E. 2003. Slipstream execution mode for cmp-based multiprocessors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. 179.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 252–263.

JOUPPI, N. P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. 364–373.

KIRMAN, N., KIRMAN, M., CHAUDHURI, M., AND MARTINEZ, J. F. 2005. Checkpointed early load retirement. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. 16–27.

LAI, A.-C., FIDE, C., AND FALSAFI, B. 2001. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. 144–154.

LARSON, E., CHATTERJEE, S., AND AUSTIN, T. 2001. Mase: A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the 2nd International Symposium on Performance Analysis of Systems and Software*. 1–9.

LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. 1996. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. 138–147.

LUK, C.-K. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. 40–51.

MOSHOVOS, A., PNEVMATIKATOS, D. N., AND BANIASADI, A. 2001. Slice-processors: an implementation of operation-based prediction. In *Proceedings of the 15th International Conference on Supercomputing*. 321–334.

MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. 129.

PALACHARLA, S. AND KESSLER, R. E. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. 24–33.

RATTNER, J. 2005. Multi-core to the masses. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. 3.

ROTH, A. AND SOHI, G. S. 2001. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. 37.

ROTH, A. AND SOHI, G. S. 2002. A quantitative framework for automated pre-execution thread selection. In *MICRO 35: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. 430–441.

ROTH, A., MOSHOVOS, A., AND SOHI, G. S. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. 115–126.

SAZEIDES, Y. AND SMITH, J. E. 1997. The predictability of data values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. 248–258.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 45–57.

SHIVAKUMAR, P. AND JOUPPI, N. P. December 2001. Cacti 3.0: An integrated cache timing, power, and area model. Tech. Rep. WRL-2001-2, Compaq Western Research Laboratory.

SRINIVASAN, S. T., AKKARY, H., HOLMAN, T., AND LAI, K. 2004. A minimal dual-core speculative multithreading architecture. In *Proceedings of the IEEE International Conference on Computer Design*. 360–367.

SRIVASTAVA, A. AND EUSTACE, A. 1994. Atom: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. ACM Press, New York. 196–205.

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

ZHOU, H. AND CONTE, T. M. 2003. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the 17th Annual International Conference on Supercomputing*. 326–335.

ZILLES, C. AND SOHI, G. 2001. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. 2–13.

Received November 2005; revised April 2006; accepted May 2006