*In Proceedings of the 11th International Symposium on High Performance Computer Architecture, February 2005*

# Multithreaded Value Prediction

Nathan Tuck  Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego
{ntuck,tullsen}@cs.ucsd.edu

**Abstract**

*This paper introduces a novel technique which leverages value prediction and multithreading on a simultaneous multithreading processor to achieve higher performance in a single threaded application. By allowing the value-speculative execution to proceed in a separate thread, this technique overcomes barriers that make traditional value prediction relatively ineffective for tolerating long latency loads. It shows that this technique can be as much as 2-5 times more effective than traditional value prediction, achieving more than 40% average performance gain on the SPEC benchmarks with realistic hardware parameters. These gains come from two effects: allowing greater separation between the stalled load and the speculative execution, and the ability to speculate on multiple values for a single load.*

## 1 Introduction

Memory latency is a significant challenge for microprocessor designers today. Although latency has been decreasing in absolute terms, aggressive microarchitectures, increasingly sophisticated circuit design, and performance oriented logic processes have combined to dramatically increase the number of clocks required to access memory. If current trends continue, we will see latencies of close to 1000 cycles in the not too distant future.

Traditional latency tolerance techniques (out-of-order execution, for example) are ineffective for latencies of this size. This motivates the use of more heavyweight measures to hide these long memory latencies. Multithreading [29, 3] is one technique that hides the latency of one thread by executing ready instructions from other threads. However, it is ineffective at solving the problem when only a single thread is running.

Value prediction [15, 10] can be an effective latency-tolerance solution. It allows execution to bypass the dependencies on the predicted instruction in the case where the prediction is correct. This allows dependent instructions to proceed in parallel with the latency-incurring instruction. However, current value prediction proposals are severely handicapped when trying to hide latencies of this magnitude. The processor typically holds all speculative state beyond the predicted instruction. Thus, the instruction window (e.g., the reorder buffer) quickly fills with speculative, uncommitted instructions.

Prior techniques proposed for value prediction suffer because they utilize a single thread context. This single context both buffers speculative state, in case misspeculation recovery is needed, and seeks to make forward progress with the predicted value. This approach has two limitations that make it less effective for tolerating long memory latencies. First, once a long latency load is value predicted, no instruction past the load can safely commit until the load returns. As a result, the ROB and instruction queues quickly fill with speculative state, as mentioned. Thus, the distance beyond the load which the thread can speculate is limited by the amount of state which can be buffered in a single thread; a processor with 1000-cycle memory access will likely be stalled long before the load completes. The second limitation of traditional value prediction is the inability to follow more than a single path – it may make a series of value predictions, but cannot make more than a single prediction for any single dynamic instruction. However, if the processor is waiting for a very long memory operation, it may have time and resources to profitably follow any predicted value that has sufficient likelihood of being correct.

This research proposes threaded value prediction, which takes advantage of resources available on a multithreaded processor, in this case a simultaneous multithreading [29, 28] (SMT) CPU, to overcome the drawbacks of traditional value prediction for hiding long memory latencies. In particular, we show that threaded value prediction can more effectively decouple the speculative stream from the non-speculative stream, allowing the speculative stream much greater progress and thereby creating the effect of a larger, or decoupled, instruction window. We also show the ability

to follow multiple predictions for a single load.

The decoupling is the same effect multithreading has always provided for parallel programs, allowing execution to occur in decoupled windows of execution arbitrarily separated [17]. In this case, however, the separation is enabled by the value prediction hardware, rather than the compiler or programmer. Other architectures (in different contexts) have used multithreading hardware to provide hardware support for decoupled execution [33, 7, 36]. More recently, checkpointing architectures [20, 2, 8] have also been proposed that add explicit state buffering support to enable this type of decoupling. The advantage of this architecture over those is the leveraging of multithreaded hardware to provide the decoupling, the ability to follow multiple paths of execution, and most critically, to allow execution beyond the stalled instruction to proceed in parallel despite dependences.

Additionally, this paper demonstrates a simplified version of the more general architecture which not only reduces the complexity of implementation along several dimensions, but also provides a performance boost.

Prior value prediction research has typically ignored floating point computation, because of the low gains exhibited. We show that those poor results are not the result of low locality, but the result of the value prediction model. While we confirm that disappointing performance with conventional value prediction, we show strong increases when multiple-thread value prediction is applied. The average gains for our best realistic predictor architecture are over 40%, far exceeding single-thread value prediction.

Initial results for multiple-value multithreaded value prediction indicate there is also significant potential for this feature of multithreaded value prediction.

This paper is organized as follows. Section 2 discusses relevant related works. In Section 3 we explain the architecture of our multithreaded value predicting processor. Section 4 explains the methodology we use for testing the effectiveness of our architecture. Section 5 outlines the performance of our architecture and investigate the sensitivity of that performance to possible variations in architectural parameters. Finally, Section 6 summarizes our contributions.

## 2 Background and Related Work

This research builds on prior work in multithreading and value prediction. Additionally, it shares some common aspects with other work that either executes a single thread within a large instruction window, or speculatively breaks execution into multiple windows, sometimes separated by checkpoints.

Simultaneous multithreading [29, 28, 11, 35] provides a base architecture for our design. Multithreading, applied to a single program (e.g., through traditional parallelism [17]), allows that program to proceed along multiple independent, decoupled paths. We apply the same phenomenon, in this case, to the non-speculative and speculative streams of execution.

Value prediction [16, 15] gives us the opportunity to break the dependence between a long-latency load and the following code, which in this research is followed in a separate thread. There have been several different types of value predictors proposed. Those based upon the Finite Context Method [23] (FCM) use past values to predict future behavior. Differential FCM (DFCM) uses the differences between prior values to predict a new value [22, 5]. Wang-Franklin [34] value prediction uses the pattern of previous value predictions to make the next prediction.

Research into overcoming long memory latencies has identified the size of the instruction window as a key to performance. Recent so-called *checkpoint* architectures have proposed breaking the instruction window into a sequence of checkpoints and a much smaller active issue window. Incorrect speculation outside the active window is resolved by rolling execution state back to a checkpoint.

The Cherry architecture [20] decouples instruction resource consumption from instruction retirement by using checkpoints to enable instructions to be removed early from the ROB and load/store queues. More recently Akkary, et al. [2] and Cristal, et al. [8] have extended this idea with microarchitectures that eliminate the ROB. These designs perform checkpoints at low-confidence branches, enabling a much larger window of instructions to be in flight.

This checkpointing work, then, shares similar goals with our research. They allow large separation between the oldest uncompleted instruction in the processor (often a load) and the youngest/most speculative. They accomplish it with what amounts to a single large window with differentiated rollback, whereas we do so with the multiple windows that a multithreaded architecture already supports. Because we leverage existing hardware more heavily, we minimize the extra resources and design risk (assuming that value prediction is also part of the baseline architecture). The biggest differentiator, though, is the fact that we not only allow the processor to execute far past a stalled load, but also increase parallelism in the process. In the checkpointing architectures, if much of the intervening computation depends on the load, the processor still only makes limited forward progress.

Speculative multithreading architectures [25, 18, 1, 26] speculatively separate execution into multiple windows of execution (threads). However, those architectures speculate on control flow between the threads, as well as register and memory values. We speculate on a single value and no control flow, and yet still achieve, in many cases, tremendous separation between the non-speculative and speculative threads.

Balasubramonian et al [4] propose introducing a *future* thread which expands the instruction window by computing ahead of the main thread and communicating predictions back to the main thread. Runahead Execution [21] avoids building a wide-window processor by checkpointing and then inserting a bogus result into a long-latency load and continuing the processor in a speculative mode thereby starting prefetching of other data which may be outside the processor's instruction window.

## 3 A Threaded Value Prediction Architecture

Much of the architectural support for threaded value prediction on top of traditional value prediction already exists in a multithreaded processor (including both SMT and chip multiprocessors) – namely, the ability to independently follow multiple streams of execution, each independently building up their own speculative and nonspeculative state. This section describes the additional hardware structures required to implement threaded value prediction.

We start by describing the mechanisms required for single threaded value prediction, then list additional hardware required when fully general multithreaded value prediction is to be supported. We also present simplifications possible from the general case with a flavor of multithreaded value prediction which we call *single fetch path multithreaded value prediction*.

### 3.1 Single Threaded Value Prediction

Effectively performing single threaded value prediction requires building up a value history to generate a predicted value, then deciding whether to use that value – based on confidence in the predicted value and the expected profitability of predicting that instruction [6]. Confidence in a particular prediction is typically built up by the value predictor. Profitability of a prediction can be estimated by using techniques from critical path prediction [31, 32, 9]. Both of these estimators can be combined, but in this work we choose to leave them separate so as to more easily show the effects of each upon the result.

Because our memory latency is large, we make a simplifying assumption that only loads are worth value predicting. This reduces the number of instructions which must be tracked by our value predictor and increases its accuracy. It also allows us to simplify our critical path predictor to merely predict the latency of the load. Typically, any load with significant latency will be critical, but we may miss other critical instructions (e.g., the predecessor or dependents of a long-latency load, if the load is not predictable).

The first piece of hardware required is a mechanism that generates value predictions for load instructions. We experiment with several predictors, including an oracle predictor, a hybrid predictor similar to that described by Wang and Franklin [34], and an improved third order DFCM predictor [5] with a confidence estimator. When the result of a predicted load returns, we compare it against the predicted value. If the load matches the predicted value, we can allow the load and successor instructions to commit normally. If the value returned by the memory subsystem does not match the predicted value, we must recover. We assume selective re-issue [14, 30] for the recovery mechanism (in the single-thread case) to maximize performance of the baseline.

### 3.2 Multiple Fetch Path Multithreaded Value Prediction

Threaded value prediction spawns a new thread to follow the speculative stream of execution following a value prediction. This new context is allowed to commit instructions from its instruction window. This allows the speculative context to possibly execute much further ahead; with a single thread, the load (with or without standard value prediction) would prevent commit from advancing the instruction window. When the load value returns through the memory subsystem to the main thread, it either kills the spawned thread or kills itself, depending on whether the prediction was incorrect or correct.

We still must buffer speculative state, because we still must treat the instructions in the predictive threads as speculative. However, because register writes in the speculative threads do not affect register state in prior threads, we need only buffer memory writes. Thus, when a speculative thread becomes nonspeculative (because a value prediction is confirmed), we need only release the store buffer to begin writing to the memory subsystem – single-threaded value prediction must still sequentially commit all of the speculative instructions.

This means that the speculation distance is only limited by the size of the speculative store buffer, rather than the instruction queues and reorder buffer.

The required hardware for threaded value prediction is precisely the same as that required for Threaded Multipath Execution [33, 12, 13], which must be able to replicate register state, track thread order, and track memory dependencies between threads – both techniques are simplified due to the fact that either the spawning thread or the spawned thread commits, never both. Thus, their register state need not be kept consistent with each other. The required mechanisms, then, are described in that prior research; we will discuss the specific hardware we assume here, but in less detail.

The threaded value predictor needs a mechanism for reproducing program state, mechanisms for recording the relationship between threads, and structures for buffering the speculative store data.

A spawned thread receives a flash copied register rename map with the load destination register substituted by the predicted value for the load. Because an SMT processor shares the physical register file [28, 19], it is only necessary to copy the register map to replicate register state. This is the primary piece that would be different in a CMP implementation of threaded value prediction, which would require a more expensive mechanism to copy state. It is also important to ensure that spawned threads cannot cause physical registers in the original thread to be recycled prior to confirmation of the value prediction. This requires an additional use counter, similar to the *pending counter* in Cherry, to be incremented at spawn time for every active register in the map being copied.

We also assume enough state is collected per context to maintain the tree of spawned threads, tracking which spawned which. In that way, the correct branch of the tree will be killed whenever a prediction is resolved.

The final piece of hardware required is a per context store buffer similar to that used on the PolyPath architecture [13]. This store buffer must be searched by every load and if a corresponding value is found in it, it must be used in preference to the value stored in memory. Maintaining these store buffers allows each thread to function independently and not to corrupt the values seen by other threads. The store buffer implementation can be either that of a unified store buffer with tags to distinguish between stores that apply to different threads, or a per-context store buffer. We chose to implement a separate store buffer per context.

### 3.3 Single Fetch Path Multithreaded Value Prediction

We consider a special restricted version of multithreaded value prediction which simplifies implementation. This is the case where each thread is only allowed to spawn a single speculative thread, where we stop fetching for the parent thread immediately (until the value prediction is confirmed), and do not support multiple-value prediction. We call this architecture *single fetch path multithreaded value prediction*.

Without this constraint, allowing both paths to proceed as far as possible, we both maximize speculative benefits and virtually eliminate prediction recovery costs. This modification of the architecture sacrifices some of the latter to simplify the implementation, and to maximize resources devoted to the former. Thus, when prediction accuracy is high, trading recovery time to get higher speculation performance is a win – the hardware savings is a bonus. Our results confirm this. In fact, the single fetch path architecture is our default configuration – the more aggressive option is evaluated in Section 5.5.

The first simplification that this architecture allows is in tracking thread descendants and antecedents. There is always a strict linear list ordering of threads rather than a tree structure. Therefore the per-thread child table requires only a single entry and instructions do not need to track the threads they have spawned.

The second set of simplifications made possible by single fetch path MTVP is in removing the communication requirement to the fetch stage of the pipeline to request it to start fetching for new threads. Since only one thread is a client for fetch at any one time and the spawned thread starts execution at the next sequential PC, the currently active thread can always use instructions which have already been fetched. From the fetch and decode units' perspectives, nothing has changed, only rename and below in the pipe know that they are delivering instructions to a different thread; there need be no fetch interruption whatsoever.

The third simplification is that copying registers from one register map to the next only needs to be performed as fast as the spawned thread can modify the rename map. By making the spawned thread owner of the parent's physical (but not logical) rename map and then performing copy-on-write update of the parent's rename map, the latency of the copy can be hidden with very little additional hardware and a fairly narrow mapping synchronization bus.

A final simplification made possible by single fetch path MTVP is in the implementation of the store buffer. With single fetch path MTVP, only a single store buffer needs to be maintained with a tag for each entry indicating which thread generated it. Searches through the store buffer are then a hit if the searching thread was spawned more recently than the owner thread.

## 4 Methodology

Our baseline processor is extrapolated from current trends and has parameters as shown in Table 1. Its long memory latency is twice that of any current uniprocessor design, but main memory latencies will continue to increase so long as main memory remains off chip. We use a moderate-length pipeline of 30 stages and increased parallelism and cache sizes from current mainstream designs.

We do all our testing using a modified version of the SMTSIM multithreading simulator [29]. All of our benchmark simulations use the SPEC CPU2000 benchmarks with simulation of 100 million instructions beyond the early single Simpoints of [24]. The early single Simpoints we use attempt to extract representative behavior for a particular SPEC reference input. As such, some benchmarks are represented multiple times. We present only the results we generated for different reference inputs when they produced markedly different behavior.

Our baseline design includes a very aggressive stride prefetcher and all results we present use it. Because our

| | |
|---|---|
| Pipeline Depth | 30 stages |
| Fetch Bandwidth | 16 total instructions from 2 cachelines |
| Branch Predictor | 2bcgskew |
| | 64K entry Meta and gshare |
| | 16K entry bimodal table |
| Stride Prefetcher | PC based, 256 entry with 8 stream buffers |
| ROB Size | 256 entry |
| Rename Registers | 224 |
| Queue Sizes | 64 entries each IQ, FQ, and MQ |
| Issue Bandwidth | 8 instructions per cycle |
| | up to 6 Integer, 2 FP, 4 load/store |
| ICache Size | 64 KB 2-way set associative |
| ICache Latency | 2 cycles |
| L1 size and latency | 64 KB 2-way set associative |
| | 2 cycles |
| L2 size and latency | 512 KB 8-way set associative |
| | 20 cycles |
| L3 size and latency | 4 MB 16-way set associative |
| | 50 cycles |
| Main Memory Latency | 1000 cycles |

**Table 1. Simulator Architectural Parameters**

memory latency is so high, we felt that it was very important to include other common architectural features to reduce memory latency and thereby not overstate the effect of our architectural optimizations. Indeed, we find that without a stride prefetcher the effect of multithreaded value prediction is greater and more consistent. However even with a stride prefetcher we find very significant speedups are possible with our techniques, and the mechanisms appear to be highly complementary.

## 5 Results

This section examines the performance of threaded value prediction in an environment with long memory latencies. It examines oracle predictors as well as realistic prediction mechanisms. It also examines the effect of varying hardware assumptions – thread spawn overhead, the size of the store buffer, and the activity of the main thread after a VP thread is spawned. Last, we examine the potential and the utility of multiple-value prediction.

### 5.1 Potential for Multithreaded Value Prediction

This section attempts to understand the limits of available performance for both threaded and non-threaded value prediction, and then to understand the inherent differences between the two approaches. To see this without the limitations of a real predictor, we initially evaluate these techniques with an oracle value predictor and several idealized conditions that are not used in the rest of the paper. The oracle predictor always predicts the correct value for any load it chooses to predict. The value predictor does not perform predictions when the processor is fetching down the wrong path. We further add the requirement that Fetch stalls on the spawning thread immediately, thereby allowing full fetch and execution resources to always be applied towards making maximal forward progress. We assume that the store buffer can hold any number of stores.

We make only two non-ideal assumption in this first study. We limit the total number of thread contexts available, and we assume that one cycle is required to flash copy the register map and spawn a thread.

To explore the tradeoffs in multithreaded value prediction, there are a variety of hardware configurations. We vary these along two axes, the number of threads available on the processor and the choice of criticality predictor for value prediction. We simulate one (baseline), two, four or eight threads on our processor assuming that all are available for execution by our single threaded program and that the processor is otherwise idle.

We also tested several different criticality predictors which behave as load-selectors. One predicted criticality based on the expected cache behavior of the load (which in the case of the oracle is known). It assumes that L3 misses are profitable to perform a multithreaded value prediction, and spawns a thread, if available, to follow the prediction. Further, it assumes that L1 misses are profitable for single threaded value prediction. We do not show results for this predictor in this paper, because on average the predictor was slightly inferior to the following, more realistic, predictor.

A second criticality predictor attempts to track the forward progress of threads and to determine when value predictions have been profitable. The method used to track forward progress is to count the number of instructions fetched from the time that the value prediction is made to the time that the prediction is confirmed. By keeping a running counter of forward progress made and cycles taken for the various outcomes (no value prediction made, single threaded value prediction made, multithreaded value prediction made) on a PC basis, the predictor allows value predictions of a certain type only if the average forward progress (measured in issued instructions) of that type is greater than the forward progress when no value prediction is made. Although it may seem that this requires three integer divides to implement, it is efficiently done in an imprecise manner by shifting down the forward progress counter by the largest integer power of two in the aggregate cycle count. This policy is labeled *ILP-pred* in the figures.

We also examined a third type of predictor similar to *ILP-pred* but which gauged forward progress based on committed rather than issued instructions. This predictor was generally comparable to *ILP-pred*, so we also do not present those results here.

In our simulations, we always assume that single-thread value predictions are allowed when the multithreaded ar-

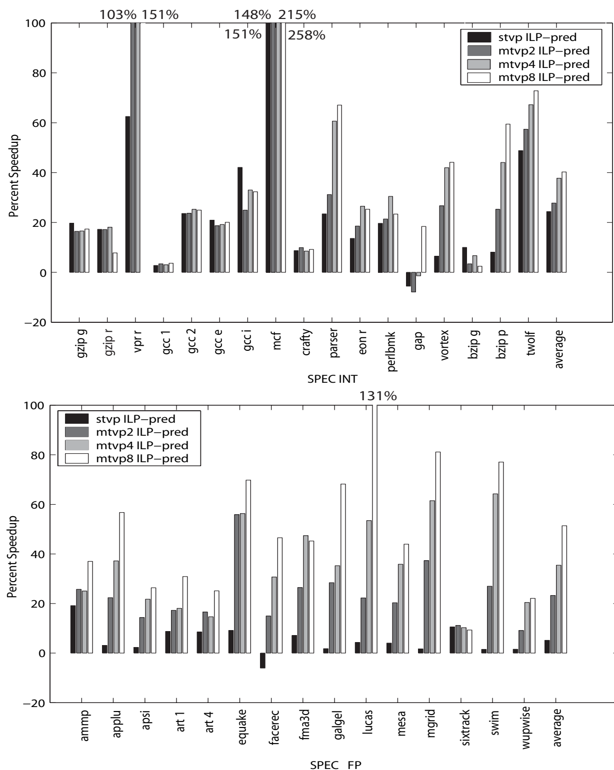


**Figure 1. Change in Useful IPC with Oracle Value Prediction. These results compare with a baseline that does not implement value prediction. The total number of threads (2, 4, 8) and the load selection policy (ILP-pred, oracle) are varied.**

chitecture lacks hardware contexts to initiate new threaded value predictions.

As can be seen in Figure 1, multithreaded value prediction (MTVP) has the potential to give us a substantial performance increase over single threaded value prediction in most cases. Although it must account for the same register dependencies as single threaded value prediction (STVP), the ability to speculatively commit instructions past a stalled load instruction allows us to achieve performance increases of over 100% in several cases and has a geometric mean speedup of 40% on integer benchmarks and 50% on floating point benchmarks. This compares favorably with single threaded value prediction speedups of 24% and 5% respectively.

We can also see from this graph that even with our aggressive assumptions, there are a few cases where the one cycle penalty of multithreaded value prediction spawns swamps performance gains, and there are even some cases where a prediction of any sort can be detrimental to performance.

In these results, we see that more threads is consistently better than fewer. We also found, although not shown here due to space constraints, that the implementable load selector, *ILP-pred*, consistently outperforms the unimplementable perfect load miss oracle. Unless specified otherwise, further results will continue to use the *ILP-pred* load selector.

The advantage of *ILP-pred* is partially due to a phenomenon exhibited in our results which we have not seen described elsewhere in the literature. Hardware prefetchers are generally trained when an issued load attempts to access the L1 cache and misses. If the prefetcher is PC based, loads with the same PC will generally train the prefetcher in program order. However, in microprocessors with large reorder buffers and queues it is possible that loads with the same PC will train the prefetcher out of order and thereby confuse or mistrain it. This tendency can be exacerbated by value prediction and multithreaded value prediction breaking dependencies. Our adaptive load selection mechanism (*ILP-pred*) becomes increasingly important as we inject more realism into our simulations, and as we speculate more aggressively. In part, this happens because it adapts not just to the effectiveness of the value prediction, but also to the combined effectiveness of the value predictions and the side effects of the predicted thread upon the prefetcher.

## 5.2 Varying Thread Spawning Penalties

Our earlier assumption of a single cycle penalty for spawning a new thread could be optimistic. One possible implementation would require a mapping synchronization bus similar to the TME work [33] capable of copying the entire register map from any thread to any other thread in a single cycle and simultaneously incrementing the usage counts of physical registers. Although the on-chip bandwidth required to do this is not unreasonable to consider, it is still desirable to perform a sensitivity analysis of the effect of longer latencies. In the case of single fetch path multithreaded value prediction, we only need to set up the copy on write, making the 1-cycle assumption reasonable, if not conservative.

This section considers the effect of spawning penalties of 1, 8 and 16 cycles. As we can see from Figure 2 our technique is in the best cases only somewhat sensitive to long latencies. We can still achieve speedups of approximately 40- 50% for integer and floating point programs with an 8 cycle copy latency. With a 16 cycle spawn latency, it becomes difficult for integer programs to beat single threaded value prediction, but floating point codes can still maintain almost 4 times the gain of single threaded value prediction (about 20% speedup). We also observe that as the latency

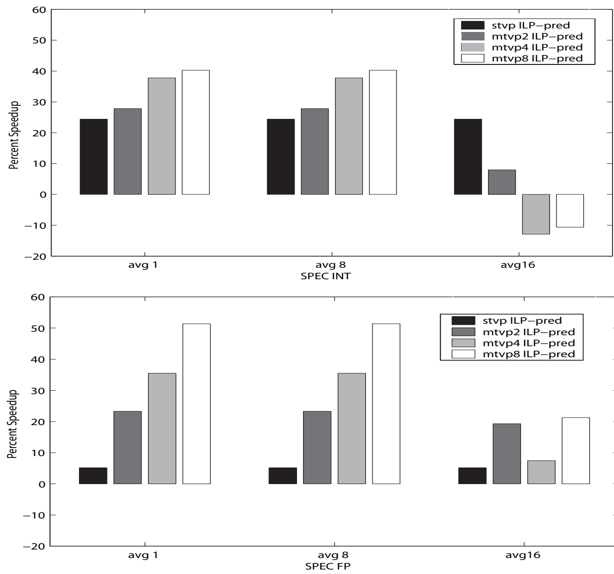


**Figure 2. Speedups from value prediction, varying latencies for spawning new threads and number of available threads.**

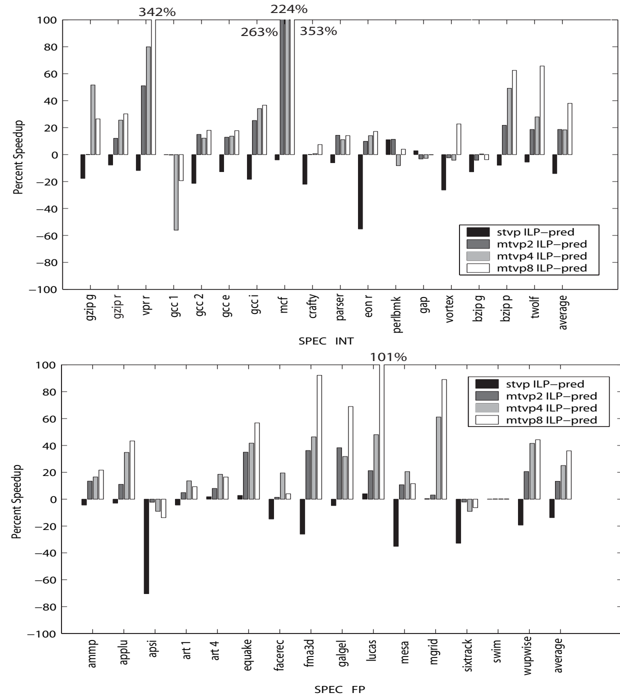


**Figure 3. Change in Useful IPC with a realistic value predictor based on Wang-Franklin.**

increases, our ability to make effective use of more thread contexts decreases.

We stick to 8 cycle spawn latencies in the remainder of our experiments, although that may be overly conservative, especially for the single fetch path case.

## 5.3 Varying Store Buffer Sizes

The size and structure of the store buffer is a critical factor in the design of the MTVP system. The store buffer must be small enough that it can be accessed in the same amount of time as a hit to the L1 cache (we assume that this allows us time to access a single 512 entry buffer). Also, whenever the store buffer is unable to hold a store, a thread must be stalled until the store buffer is flushed out sufficiently to allow the store. The maximum number of stores allowed in the store buffer determines the maximum *separation* (counted in stores) of a spawning thread and its child. A larger separation allows the child thread to accomplish more before the value prediction is confirmed. We don't show the specific results here, but we find that limiting the size of the store buffer can have a significant effect on the performance of multithreaded value prediction. Performance begins to tail off at 64 and below entries. However, a 128-entry buffer gets nearly the performance of the largest buffer we simulate.

## 5.4 Using a realistic value predictor

In this section we replace our oracle value predictor with realistic value predictors. We first use a hybrid predictor based on the Wang-Franklin value predictor. The Wang-Franklin value predictor has two tables. The first table is a value history table (or VHT) indexed by PC which contains the most recent values created by that PC, a last-value and stride for the stride predictor, and a pattern history (similar to a branch history) which is used to index the next table. The second table is the Value Pattern History Table or ValPHT and contains the confidence level for the values in the VHT. We also assume an 8 cycle spawn latency and a 128-entry store buffer.

In our processor, we use a Wang-Franklin value predictor with five learned values, a hardwired zero and one, and a stride value. We fix our VHT at 4K entries and our ValPHT at 32K entries. The total memory required to implement this is 160 KBytes for the ValPHT and approximately 244 KB for the VHT. These sizes are somewhat large by today's standards, but feasible to consider in future processor designs with feature sizes of 65 nm or less. The stride component of the value predictor is speculatively updated in the queue stage where the value predictor is used. Training and

replacement of other values is done when instructions commit. We assume that value confidence increases by 1 on correct predictions and decreases by 8 on incorrect predictions with a threshold of 12 and a maximum counter value of 32.

In Figure 3 we see that even with a realistic value predictor we can achieve substantial average speedups of about 40% on SPECfp and SPECint with eight threads. In specific instances, we see higher performance increments than with the oracle predictor (e.g., *vpr* and *mcf*). This happens for several reasons. First, there is a slight decrease in baseline performance caused by the limited store buffer size. Second, there is a marked decrease in prefetcher/value-predictor interactions for certain benchmarks because of the much lower value predict rate. Third, by making far fewer predictions the realistic value predictor allows each thread to have its own slightly larger instruction window (e.g., increased access to the instruction queue). Although there is a higher incidence of negative values, due to mispredictions, the average results are still quite high.

We also simulated an improved third order DFCM predictor with similar size based on Burtscher [5]. Our results with this predictor were not as good as our Wang-Franklin predictor so we do not present them here for space reasons. The primary reason for this was that it is in general a more aggressive predictor – making more correct predictions and more incorrect predictions. It is possible that more careful tuning of the confidence metrics for DFCM will yield a more beneficial result.

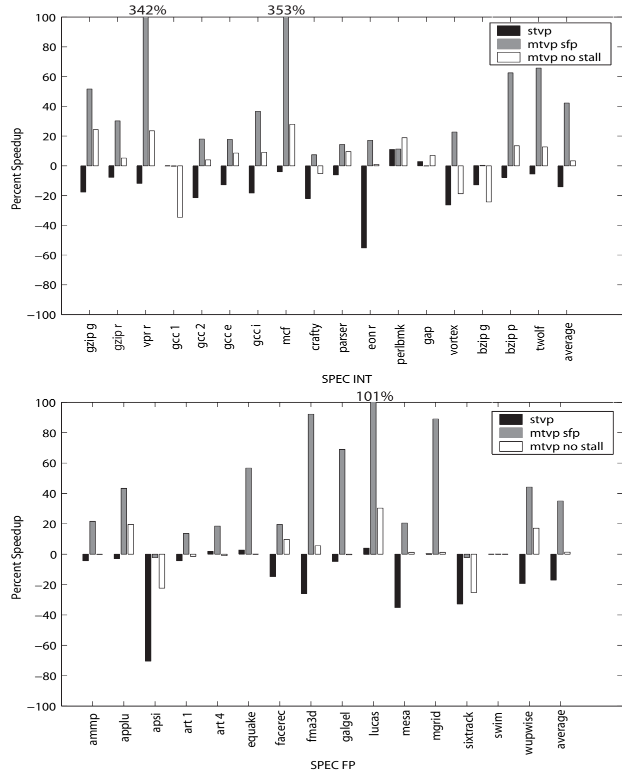


**Figure 4. Benchmark results allowing fetch to continue in the parent after a threaded value prediction has been made.**

## 5.5 Fetch Policies

All the results thus far have used the Single Fetch Path MTVP model. This model significantly eases the implementation complexity of MTVP, but the performance tradeoff is less clear – we evaluate that tradeoff in this section. In the single fetch path MTVP architecture, when a thread performs a multithreaded value prediction, it immediately stops fetching instructions. This is a conservative policy, and prevents the spawning thread from making significant progress (progress which is useful in the case of a value misprediction). This policy is effective because most predictions are correct, and even during long memory latencies there is competition for resources among the speculative and non-speculative threads. In this section we examine the effect of a more aggressive fetch policy, allowing the main thread to continue to fetch and execute instructions, using the ICOUNT policy to arbitrate between the threads.

As we can see in Figure 4, allowing fetch to continue after a threaded value prediction (no stall) is highly counterproductive. This figure shows the relative gain for MTVP using the more aggressive fetch policy. Comparing this figure with Figure 3, we see that the results are consistently lower than our baseline, which uses the single fetch path policy. What we observe is that competition for fetch and execution resources swamps any gains made by maximizing forward progress in the case of incorrect predictions. Also, when the load which has been value predicted is a long latency load, allowing that thread to continue to fetch (even at a reduced priority) will eventually fill the queue with instructions from that thread and stop forward progress [27] for all threads. A statically partitioned queue could ameliorate this effect but would limit the performance of the spawned threads.

## 5.6 Multiple Value Predictions

The second key advantage enabled by threaded value prediction, which we have not yet exploited, is the ability to follow multiple predictions for a single instruction. We would expect that during a very long latency load there would be sufficient idle resources to follow even less likely predictions. For example, a prediction that gave us a 30% chance of hiding a 1000-cycle latency may still be worth

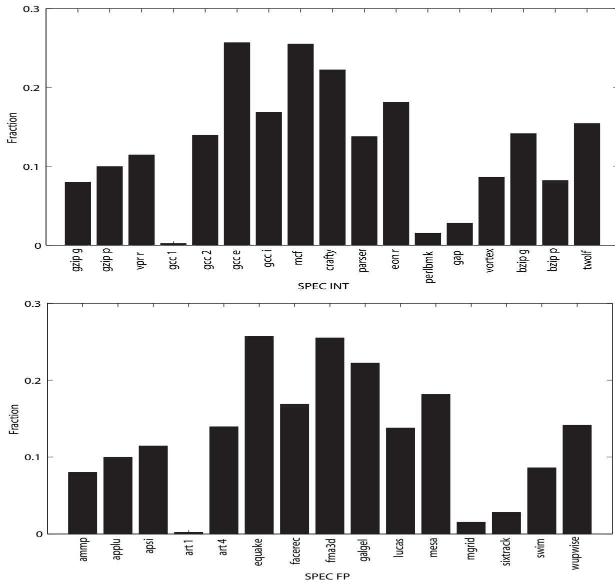


**Figure 5. Fraction of total value predictions for which prediction was incorrect but the correct value was present and over threshold**

pursuing, even if we are already pursuing a value having 60% probability.

In order to follow multiple predictions, we need a value predictor that is capable of providing multiple value predictions. Wang-Franklin value prediction can be easily adapted to perform this by performing predictions for all values which are over threshold in the pattern history table. However, our implementation of that predictor used so far makes it difficult for multiple values to be above threshold.

Our initial results show some promise. With a more liberal predictor but a more discriminating criticality measure (the oracle L3 miss predictor, described earlier, which allows the technique to focus on those loads most likely to benefit from multiple-value prediction), we have found a few promising results. More extensive experimentation is necessary to find parameterization that provides speedups across the board, but with these particular changes, *swim* and *parser* show speedups of 70% and 40% respectively, outperforming their single value multithreaded value prediction speedups of less than 1% and 14% respectively, given our best overall parameterization for single-value MTVP.

There is evidence that there do exist policies that will make multiple-value prediction profitable in the more general case. Figure 5 shows the percentage of total followed predictions for which the primary prediction was incorrect, but the correct value was in the predictor, and above the prediction threshold, even with a conservative threshold. Most of the benchmarks have this property to one degree or another, with some having as much as 25% of their loads being good candidates for multiple predictions.

The proper mechanisms and policies to fully exploit multiple value multithreaded value prediction remains an open question and is the subject of future research.

### 5.7 Comparison to Checkpoint Architectures

There is some overlap, at least in goals, with recent research in architectures that provide the effect of wide instruction windows [20, 2, 8] via checkpointing. In this section we compare the techniques. As described in Section 2, they present other mechanisms for allowing the processor to proceed well beyond a stalled load. Those solutions are more general, providing separation even in the absence of value locality. However, while those solutions increase the processor's ability to exploit available parallelism, the architecture we describe both exploits available parallelism across large distances, and also creates new parallelism. Thus, while those architectures bypass dependent computation to find independent computation, our architecture executes *and commits* both the independent computation and the dependent computation.

We measure the performance of our realistic value-predictor architecture against two other machines. The first is that of a machine with similar architectural parameters except for an 8192 entry ROB, unlimited registers and 8192 entry queues. This is an idealized version of a checkpointing architecture.

The second is a machine with the same architectural parameters as our architecture, but which spawns a new thread without performing a value prediction. This is called *spawn only* in the figure. This architecture can commit non-dependent instructions in its spawned threads beyond the stalled load, thereby creating a "split-window" effect, but without the value prediction gains. This allows us to decouple the gains we get from the two effects.

As we can see in Figure 6, our work can compare favorably with a highly idealized checkpoint architecture, for many of the benchmarks, especially the integer. In fact, the techniques appear more complementary than redundant. The specific results conform well to intuition in that when there is abundant parallelism (the floating point benchmarks, primarily), the wide window architectures are very successful at finding and exploiting it. The idealized wide-window machine beats threaded value prediction on nearly all the SPECfp benchmarks. For the SPECint benchmarks, where parallelism is harder to find, the story is very different. For those applications, the value speculation is indeed critical to enabling the separation between the forward edge and the trailing edge of the execution window. Our ar-

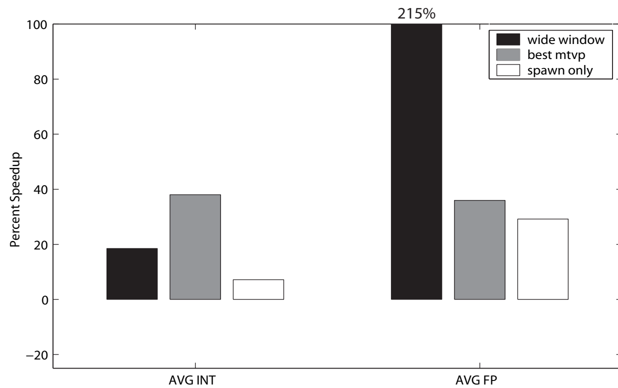


**Figure 6. Comparison of a wide window machine with 8K ROB entries and unlimited registers with our work and a checkpoint architecture similar to our work but with only window separation. Eight cycles are assumed for our work and the checkpoint architecture to make and spawn a thread**

chitecture provides more than 7 times the speedup of even the idealized checkpoint architecture for *vpr* and even more dramatic increases on *mcf*.

In comparison with our spawn-only results, in particular, we see that exploiting the combination of value prediction and thread spawning is critical to getting full performance, particularly for the integer benchmarks. It has already been shown that value prediction alone has limited effectiveness. Here we see that the decoupling effect also is quite ineffective alone (spawn only). In contrast, the combination of the two techniques (value prediction and the ability to continue past the load in a new thread) provides the dramatic gains we've been seeing in this paper.

Thus, we see that the use of multithreading to do value prediction is a powerful combination, allowing separation between the speculative computation and the nonspeculative. But key to that separation is the breaking of dependences provided by the value prediction. In this way, we overcome the limitations of previous attempts to exploit value prediction. We also see that this approach provides high gains where even the checkpoint architectures have been least effective.

## 6 Summary

Value prediction has not delivered on its promise, typically giving relatively low speedups, despite the abundance of available value locality. We show that this is at least partially due to the way it has been implemented. Using a single thread to contain both the architectural state of the non-speculative path, and the speculative state of the predicted path, severely limits the window of speculation.

Threaded value prediction attacks these constraints, by spawning a separate thread to execute the speculative path using the predicted value. This allows arbitrary separation between the predicted load and the leading thread (limited only by the size of the store buffer).

We find that our technique of multithreaded value prediction overcomes traditional barriers to the utility of value prediction, potentially making it 2-5 times more effective. We find that a realistic implementation of this technique yields speedups of over 100% on a few SPEC benchmarks and a geometric mean improvement in IPC of over 40% compared to a processor without any value prediction.

Additionally, we show that while traditional value prediction work has been seemingly ineffective at providing gains for floating point benchmarks, multithreaded value prediction is more effective for those programs than for integer programs.

Last, we show that another benefit of multithreaded value prediction is the ability to follow multiple predictions for a single load. Initial results show one case of a 70% gain from this technique, using a more liberal predictor, indicating there is potential to exploit this feature enabled by multithreaded value prediction.

A simplified version of multithreaded value prediction, single fetch path MTVP, is shown to significantly outperform the more general (also more aggressive and more complex) architecture, increasing the feasibility of this technique.

## Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF grant CCR-0311683, an IBM Faculty Award, and a grant from Intel.

## References

[1] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *31st International Symposium on Microarchitecture*, Nov. 1998.

[2] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *36th International Symposium on Microarchitecture*, 2003.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, June 1990.

[4] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically allocating processor resources between nearby and distant ilp. In *28th Annual International Symposium on Computer Architecture*. ACM Press, 2001.

[5] M. Burtscher. An improved index function for (D)FCM predictors. *Computer Architecture News*, 30(3), June 2002.

[6] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[7] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[8] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Tenth International Symposium on High-Performance Computer Architecture*, 2004.

[9] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *28th Annual International Symposium on Computer Architecture*, 2001.

[10] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, Nov. 1996.

[11] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[12] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *32nd International Symposium on Microarchitecture*, Nov. 1999.

[13] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath architecture. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[14] M. Lipasti. Value locality and speculative execution. Ph.D. Dissertation, Carnegie Mellon University, 1997.

[15] M. Lipasti and J. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, Dec. 1996.

[16] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[17] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, Aug. 1997.

[18] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *International Conference on Super-Computing*, July 1998.

[19] D. Marr, F. Binns, D. Hill, G. Hinton, K. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: a hypertext history. *Intel Technical Journal*, 1(1), Feb. 2002.

[20] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *35th International Symposium on Microarchitecture*, 2002.

[21] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Ninth International Symposium on High-Performance Computer Architecture*, 2003.

[22] Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical Report ECE-97-8, University of Wisconsin-Madison, December 1997.

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

[24] T. Sherwood, E. Perelman, G. Hammerley, and B. Calder. Automatically characterizing large-scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[25] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[26] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Fourth International Symposium on High-Performance Computer Architecture*, Jan. 1998.

[27] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[28] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[29] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[30] D. Tullsen and J. Seng. Storageless value prediction using prior register values. In *26th Annual International Symposium on Computer Architecture*, pages 270–279, May 1999.

[31] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Seventh International Symposium on High-Performance Computer Architecture*, pages 185–196, 2001.

[32] E. Tune, D. Tullsen, and B. Calder. Quantifying instruction criticality. In *11th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[33] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[34] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, Dec. 1997.

[35] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[36] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, June 2001.