Fastpath Speculative Parallelization*

Michael F. Spear[2], Kirk Kelsey[1], Tongxin Bai[1], Luke Dalessandro[1], Michael L. Scott[1], Chen Ding[1], and Peng Wu[3]

[1] University of Rochester
{kelsey,bai,luked,scott,cding}@cs.rochester.edu

[2] Lehigh University
spear@cse.lehigh.edu

[3] IBM T. J. Watson Research Center
pengwu@us.ibm.com

Abstract. We describe Fastpath, a system for speculative parallelization of sequential programs on conventional multicore processors. Our system distinguishes between the *lead* thread, which executes at almost-native speed, and *speculative* threads, which execute somewhat slower. This allows us to achieve nontrivial speedup, even on two-core machines. We present a mathematical model of potential speedup, parameterized by application characteristics and implementation constants. We also present preliminary results gleaned from two different Fastpath implementations, each derived from an implementation of software transactional memory.

1 Introduction

As just about everyone knows by now, constraints on heat dissipation and available instruction-level parallelism dictate that mainstream processors will have multiple cores for the foreseeable future, and maximum performance will require that programs exhibit thread-level parallelism. Building such parallelism into new programs is difficult enough; retrofitting it into the vast repository of legacy code is a truly daunting task, particularly given that most programs are old enough that no available programmer understands them very well.

There exists a huge literature—and significant commercial tools—devoted to the *automatic* parallelization of legacy code. Developed largely for high performance scientific computing, parallelizing compilers work very well for programs with very regular data parallelism. Unfortunately, many programs don't fit the mold. Some simply cannot be parallelized without fundamental algorithmic restructuring. Others offer the tantalizing prospect of latent parallelism that is typically available at run time, but difficult (or impossible) to predict at compile time, because either (1) the program is too complex for current technology to analyze, or (2) the sections of code that we would like to run in parallel are usually, but not always, independent, due either to statistical properties of the program itself or to input dependences. These we target here.

* At the University of Rochester, this work was supported by NSF grants CNS-0615139, CCF-0702505, CSR-0720796, and CNS-0834566; by equipment support from Sun and IBM; and by financial support from IBM, Intel, and Microsoft.

Speculative parallelization aims to exploit dynamic thread-level parallelism. Blocks of code that are likely, though not certain, to be mutually independent are statically identified—in our case by the programmer, though in principle it could be by the compiler. Then, at run time, the language implementation runs the blocks concurrently, monitoring their executions for conflicts, and in the event of conflict preserves the execution that would have come first in a sequential execution, *undoing* the others.

Our system, Fastpath, works entirely in software within a single address space, making it suitable for existing multicore processors and for programs with potentially parallizable blocks on the order of a single function call in size. Drawing inspiration from hardware implementations of thread-level speculation, we distinguish between a *lead* thread, which has minimal overhead and is guaranteed to complete (in the absence of fatal program bugs), and one or more *speculative* threads, which have higher overhead and complete only after they have been shown not to conflict with earlier threads. This implementation strategy allows us to execute lead-thread code at essentially native speed. It also allows us to obtain nontrivial speedup (on the order of 25%) for important classes of programs on even a dual-core processor.

In Section 2 we present a mathematical model of potential speedup in Fastpath, parameterized by instrumentation overheads and program characteristics. Focusing on loop-based parallelization (our system could also make use of *futures* [1]), we consider both steady-state pipelined execution and the optimal partitioning of work across threads that start executing concurrently. Section 3 describes two implementations of our system. Both employ techniques originally developed for transactional memory. They differ in the types of instrumentation inserted in the program; one has larger per-block overhead, the other larger per-instruction overhead. Preliminary results appear in Section 4. Related work is described in Section 5. We conclude in Section 6 with current status and plans.

2 Mathematical Model

2.1 Steady State Model

The intuition for our model appears in Figure 1. We assume that loop iterations (or other speculative blocks) are approximately equal in size, mutually independent at run time (and thus concurrently executable), and significantly more numerous than worker threads. We assume the following parameters.

- w is the amount of work (sequential execution time) required by each loop iteration.
- p is the number of worker threads.
- r_f is the time required per unit of work when running in fast (lead thread) mode. This is likely to be a factor slightly larger than 1.
- r_s is the time required per unit of work when running in slow (speculative) mode. This will be larger than 1, but probably less than 8.

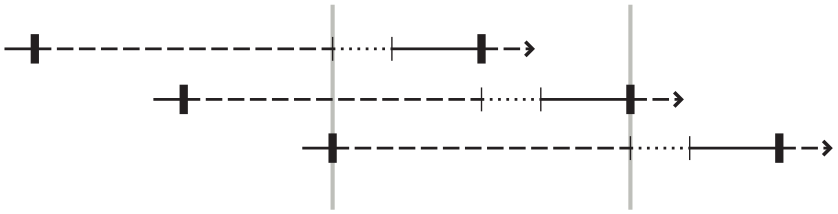

Fig. 1. Speculative pipelining. Each thread repeatedly executes part of a loop iteration in slow mode (dashed line), transitions to fast mode when signaled by its predecessor (dotted line), and completes its iteration in fast mode (solid line). The time spent in speculative mode (between the vertical gray lines) should be equal, in steady state, to the time required for transition and fast-path execution in the other threads.

- r_c is the time required per unit of work to convert from slow mode to fast mode. It covers validation of speculative reads (if this requires linear time) and write-back of speculative writes. It needs to be significantly less than 1, or speculative execution will be counterproductive.
- c_c is the constant cost component of the slow-to-fast transition.

Parameters r_f, r_s, r_c, and c_c are influenced by the choice of run-time system. Parameters w, r_f, and r_s are influenced by the choice of workload.

We introduce an unknown factor d that represents the fraction of w accomplished by each thread in slow mode, steady state. Using this factor, we can express the time t devoted to a single iteration in a single thread as

$$t = r_s dw + [r_c dw + c_c + r_f(1-d)w]$$

The part of the formula in square brackets lies on the critical path of the application. Assuming p is sufficiently small, the leading part of the formula is overlapped with critical path execution in other threads. So

$$r_s dw = (p-1)[r_c dw + c_c + r_f(1-d)w]$$

We can solve this equation for d, yielding

$$d = \frac{(p-1)(c_c + r_f w)}{w(r_s - (p-1)(r_c - r_f))}$$

For a value of d, driven by the choice of p, we calculate speedup (work/time) as

$$s = (wp)/(r_s dw + r_c dw + c_c + r_f(1-d)w)$$

As we increase p, threads spend more and more of their time in slow mode. In the limit, fast mode execution disappears entirely: each thread completes its iteration in slow mode, waits for its predecessor to finish, validates its reads, and writes back. This saturation limit is reached when $d = 1$ — that is, when

$$(p-1)(c_c + r_f w) = w(r_s - (p-1)(r_c - r_f))$$

We can solve this equation for p, yielding

$$p_{sat} = \frac{r_s w + c_c + r_c w}{c_c + r_c w} = \frac{slowpath + transition}{transition}$$

Substituting p_{sat} in the equation for s gives maximum achievable speedup s_{max}.

In Section 3 we describe two run-time systems, one based on "values" and the other on "signatures". The former has a larger r_c, the latter a larger c_c. If, for a given workload and system, $c_c \gg w$, p_{sat} will be very close to 1. On the other hand, if c_c is very close to 0, we will have $p_{sat} \approx (r_s + r_c)/r_c$. If, for a given workload and system, $r_s = kr_c$ for some k, we will have $p_{sat} \approx k + 1$.

All of this, of course, is predicated on the assumption that iterations (or other speculative blocks) are numerous and approximately equal in size. If this is not the case, we may want to assign different amounts of work to each thread.

2.2 Optimal Static Partitioning

Given a possibly parallel loop and a machine with p processors, static partitioning divides the loop iterations into p tasks for execution on the p processors. A static partition is optimal if the loop is executed no slower than it would be with any other static partition. For this part of the model, we assume that work can be divided into arbitrary chunks $w_1, \ldots, w_p$. Other parameters of the system are the same as in Section 2.1.

In static partitioning, all tasks start at the same time. The first task runs in fast mode. A later task i executes in slow mode until it finishes its work or until task $i-1$ finishes, whichever comes first. It then validates and, assuming the validation is successful, writes back the work it has completed and executes the rest of its work, if any, in fast mode.

We claim that given p processors and a sufficiently large amount of aggregate work W, a schedule (partition) in which task i completes its work (in slow mode) just as task $i-1$ finishes is strictly faster than any other schedule. Under this "tight packing" strategy, for $i > 1$, we have

$$t_i \;=\; t_{i-1} + \frac{t_{i-1}}{r_s} r_c + c_c \;=\; t_{i-1}\left(1 + \frac{r_c}{r_s}\right) + c_c \;=\; a t_{i-1} + c_c, \;\text{ where }\; a = 1 + \frac{r_c}{r_s}.$$

In closed form we have

$$t_i \;=\; a^{i-1} t_1 + c_c S_i, \;\text{ where }\; S_i = \sum_{1 \le j \le i-1} a^{j-1} = \frac{a^{i-1} - 1}{a - 1}. \tag{1}$$

Processor 1 will perform t_1/r_f work. For $i > 1$, processor i will perform work t_{i-1}/r_s, for an aggregate total of

$$W \;=\; \frac{t_1}{r_f} + \frac{1}{r_s} \sum_{1 \le i \le p-1} t_i$$

In closed form, after considerable rearranging, we have

$$W = t_1 \left(\frac{1}{r_f} + \frac{S_p}{r_s} \right) + \frac{c_c}{r_c}(S_p - p + 1) \quad (2)$$

(NB: Tight packing is feasible only if the work we need to accomplish is greater than the second of these terms. In the following discussion, we assume that this is always the case.)

Solving for t_1 in equation 2 and combining with equation 1, we can calculate all tight packing times t_i and work allotments w_i as functions of W and p.

Optimality: Tight packing is trivially optimal for $p = 1$. Suppose it is likewise strictly optimal (faster than all alternatives) for $p < n$ and sufficiently large W. Suppose further that strategy X is optimal for n processors, and that it gives processor n more work than tight packing does (and the other processors correspondingly less). The extra y units of work will be performed in fast mode after all previous processors have finished and processor n has validated and written back its results.

Consider what would happen if we move $y/2$ units of work back to processor $n-1$, to be performed there in fast mode. If processor n simply waits for this work to complete before performing its validation, write-back, and remaining fast-mode work, total execution time will be unchanged. While processor $n-1$ is doing the moved work, however, processor n has the option of doing $yr_s/2r_f$ units of work in slow mode. Then, after processor $n-1$ finishes, processor n will be able to validate and write back that slow mode work *in lieu of* executing it in fast mode. Since $r_c < r_f$, this is a net win, and we have reduced the running time achieved by supposedly optimal strategy X.

Suppose on the other hand that optimal strategy X gives processor n less work than tight packing does (and the other processors more), and that processor n sits idle for some amount of time u before processor $n-1$ finishes. During this idle time it could have performed u/r_s work in slow mode. Consider what would happen if we move $u/2r_s$ work from processor $n-1$ to processor n. The missing work in processor $n-1$ relieves it of the obligation to perform $ur_c/2r_s$ time units of validation and write-back after processor $n-2$ finishes, allowing processor $n-1$ to complete, and processor n to start its validation and write-back, that much sooner. At the same time, the extra $u/2r_s$ work in processor n obligates it to perform $ur_c/2r_s$ additional validation and write-back, leaving overall completion time unchanged.

Now, having moved some work from processor $n-1$ to processor n, consider whether the schedule of work on the first $n-1$ processors is optimal for the work completed ($W - w_n - u/2r_s$). If so, we have a non-tight-packing optimal schedule for $n-1$ processors, a violation of the inductive hypothesis. If not, there exists a shorter schedule on $n-1$ processors, and since there is still idle time in processor n's execution, it can begin its validation and write-back sooner, thereby finishing sooner, and violating the assumption that strategy X was optimal for

n processors. This covers all cases, and completes an inductive proof of the optimality of tight packing.

3 Implementation

Our Fastpath implementation strategy consists of two main components. First, we use per-access instrumentation to perform conflict detection, using two algorithms inspired by software transactional memory (STM). Second, we use a set of code transformations to guarantee minimal overhead for single-threaded code and pathological workloads. These transformations ensure that the fast path is as fast as possible. The code transformations include support for transitioning from slow to fast path at the earliest possible time.

While many STMs can avoid sandboxing, it appears fundamental to speculative parallelization with transactions. The difference stems from program semantics: STM is intended for explicitly parallel programs, in which transactions are unordered; each transaction transitions the system from a state in which any scheduled transaction can be executed safely to another state in which any scheduled transaction can be executed safely. When we parallelize sequential programs with STM, the resultant "transactions" do not preserve this property. Briefly, if iteration T_i initializes a value v that is used by iteration T_{i+1}, then there is a potential use before initialization error if T_{i+1} runs in parallel with T_i and accesses v first. If v is used as a divisor, a reference, or a function pointer, then a signal may be generated. The only appropriate action is to use sandboxing to recover from the fault and then restart T_{i+1}.

3.1 The "Value" Algorithm

Our first algorithm is based on the NOrec STM system [2], which in turn interits its conflict detection mechanism from JudoSTM [3]. It mandates that whenever a slow-mode thread reads a shared location, it logs the address and value read. To transition to fast mode, the thread iterates through the log, and ensures that all addresses still hold the values that they held when the log entries were created. In contrast to JudoSTM, an address may be logged multiple times, and there are no lookups into the log. These simplifications reduce overhead. Furthermore, they do not sacrifice correctness: we "sandbox" slow-path code to prevent it from performing any externally visible changes (even if it reads inconsistent values from memory due to conflicts with previous threads), and the transition to fast mode ensures that no inconsistent reads occurred.

With this approach, fast-mode threads require no per-access instrumentation whatsoever. They run at uninstrumented speed, modulo a branch at the beginning of the speculative region and an increment (to a shadow copy of the loop control variable—LCV) at the end of the region. Slow-mode threads must log an ⟨address, value⟩ pair for each read and each write. At the end of a slow-path execution, a thread first transitions to fast mode (by checking all reads, replaying all writes, handling any deallocations, and fast-clearing all logs), and then increments the shadow LCV.

Assuming that most computation accesses potentially shared data, which must be instrumented, this algorithm results in a slow-to-fast transition cost that is linear in the amount of work done prior to the transition. Thus appreciable speedup is unlikely unless there is uninstrumented work: the best outcome otherwise is that a thread's speculative execution serves to prefetch all values, so that the transition is slightly cheaper than it would otherwise have been.

In return for this overhead, the Value algorithm has two valuable characteristics. First, there are never false conflicts due to the granularity of global metadata. This stands in stark contrast to most STM algorithms, as well as the Signature algorithm described below. Second, it is possible to obtain opportunistic data forwarding from the fast-mode thread to slow-mode threads. Briefly, if fast-mode thread F performs a write to location L before slow-mode thread S reads L, then not only does the correct data reach S, but when S transitions to fast mode (as it ultimately must in order to complete), no conflict will be detected on location L. Since the fast-mode thread runs faster than slow-mode threads, the algorithm provides the appearance of opportunistic post/wait in a `do-across` loop [4] without explicit post and wait instructions.

3.2 The "Signature" Algorithm

Whereas the Value algorithm requires no global metadata apart from the shadow LCV, our Signature algorithm, which is based on the RingSTM algorithm [5], requires a small block of metadata shared by all threads. However, the metadata is far simpler and more compact than that proposed for RingSTM.

We maintain a flat array of write signatures for conflict detection: a single hash function maps each word of memory to a bit in the signature, much like Bloom filters [6]. All threads update their array entries on every write (even fast-mode threads). Furthermore, each slow-mode thread maintains a private signature to represent all the shared locations it reads.

Using signatures, the transition from slow to fast mode entails signature intersections, which can be accelerated via SIMD instructions on many machines. A slow-mode thread S must intersect its read set with the write set of every prior thread that was active when S started. S does not need to wait until its transition to fast mode, however, to perform all intersections. As in the Value algorithm, a fast-mode thread F indicates completion by incrementing the shadow LCV. Doing so effectively announces that the signature array entry corresponding to F's execution will no longer change. Any concurrent thread can intersect its read signature with F's write signature at this point. Indeed, doing so decreases the risk of conflicts: if F updates location L, and S reads L, then S conflicts with F only if the read of L was before F completed. If F completes, then S intersects its read signature with F's writes, and then S reads L, there is no conflict.

As in RingSTM, we can re-use array entries (the array is an ordered ring). However, the array can be quite small. The key observation is that all threads are ordered. When the array holds twice as many entries as threads, then each entry can be mapped to a thread based on the thread order (more specifically, $\mathsf{entry} = \mathsf{thread\_order} \ \% \ \mathsf{ring\_size}$.) The fast-mode thread never looks at older entries, since

it is logically oldest. However, when a thread is transitioning to fast mode, it may need to look at the previous $p-1$ entries (if, for example, it performed one shared read and then preempted while the other $p-1$ threads completed, then it would need to check its read signature against those $p-1$ signatures). When a thread is transitioning to fast mode, at most $p-1$ threads can be executing logically younger operations. Thus at most $2p-1$ entries are ever needed.

Unlike the Value algorithm, our Signature algorithm admits false conflicts. For small speculative regions, 1024-bit signatures should suffice, but with larger regions, it may be necessary to increase the signature size. There is no notion of data forwarding: if a slow-mode thread reads the value written by an in-progress fast-mode thread, then it will ultimately be forced to abort and restart. Furthermore, the algorithm puts overhead on the fast path, to set bits in the signature. Our hope is that superscalar cores can mask this overhead, since it can typically occur in parallel with program code. Finally, the algorithm requires that threads clear their public write signatures on both the slow and fast paths, and their private read signatures on the slow path.

By accepting these costs, the Signature algorithm obtains validation costs independent of the amount of work performed in a speculative region. The transition to fast mode should be faster for large regions, read set metadata has a constant size, and SIMD instructions can be used to further decrease metadata operations (particulary signature clearing and signature intersection).

3.3 Compiler Instrumentation

To maximize the benefit of our Fastpath algorithms, some compiler support is needed. Our current library implementation requires a small amount of code transformation, which we currently perform by hand. In future work, we plan to implement these transformations automatically, so that we can leverage static analysis to optimize the code more aggressively.

Cloning – A naïve instrumentation would not distinguish between fast mode and slow mode at the boundaries of a speculative region. This would result in a `setjmp` call at the beginning of every region, even if the region could be guaranteed to run in fast mode (as is the case when running in single-thread mode). Our first optimization is to clone code to avoid `setjmp` overhead, as depicted below:

```
Before:                  After:
for (...)                for (...)
   /* body */               if (fastpath())
                              /* instrumented body */
                           else
                              setjmp()
                              /* instrumented body */
```

Importantly, we provide a hint to the compiler that the added condition is highly likely to be true. This results in code generation in which optimizations

(e.g., register allocation) assume that the branch will be taken, adding any necessary overhead to the slowpath, but minimizing the overhead of fast execution.

Optimized Instrumentation – Loads and stores to shared memory within the speculatively parallelized body require instrumentation, much like the instrumentation required by STM. The simplest approach would be to test whether a region was executing in fast mode immediately after each load, and to branch immediately before each load or store depending on whether the thread was in slow or fast mode. Once the above transformation is applied, this instrumentation is excessive: on the fast path, the tests and branches are unnecessary, since the code is already running in fast mode. By cloning the loop body, we have the ability to instrument the code differently in the fastpath and slow-path cases.

Transitioning to Fast Path – When the oldest slow-mode thread determines that there is no older fast-mode thread, it may transition to fast mode. In order to do so, the thread first performs a test to ensure that its reads remain valid. After that test, the thread must replay its buffered writes to memory. In our current implementation, slow-path threads perform a check on every read to determine when there are no active older iterations.

When a thread determines that it is logically oldest, it immediately transitions to fastpath mode. However, it does not immediately jump into the cloned code with fastpath instrumentation. Instead, we place a test at the end of each basic block in the slow-path instrumented code: if the test succeeds, then the thread jumps to the bottom of the corresponding basic block in the fast path code. The same approach can be applied on return from any function call. For correctness, this optimization requires careful attention to stack frame layout. In return, after the slow-to-fast transition a thread has two fewer tests/branches per load and one fewer test/branch per store. In our current implementation, we implement this transition by hand, by inserting `goto` statements directly into the cloned source code.

4 Preliminary Results

We use a simple, list-based microbenchmark for preliminary testing of our two TLS systems. Each list node consists of a `next` node pointer, a `key`, and a `value`. We begin by generating a 256 element list. The outer loop (Figure 2) chooses a list key to search for, and executes the microbenchmark inner loop with that node as a target. The loop induction variable, `i`, is managed by OpenMP, and determines the order in which iterations must commit. The `workpernode` global is a command line parameter that allows us to control the ratio of shared to local operations within the inner loop.

The inner loop executes speculatively. It searches the list for the corresponding `key`, accumulating the `key`s that it encounters during the search, and writing the accumulated value into the `value` at the node it finds. The inner loop is

```
1  // Outer loop, managed by OpenMP
2  parallel for (i = 0; i < iterations; ++i)
3    key = random_key(i)
4    // Inner loop
5    FASTPATH_BEGIN(i)
6      accum = 0
7      current_node = list_front
8      while (current_node->key != key)
9        tmp = current_node->key
10       // controlled by the command line
11       for (j = 0; j < workpernode; ++j)
12         tmp = rand_r(&tmp)
13       accum += tmp
14       current_node = current_node->next
15     current_node->value += accum
16   FASTPATH_END(i)
```

Fig. 2. Microbenchmark code.

mostly parallel. Each iteration reads all of the nodes between `1` and the generated `key`, and completes by reading and writing the target node's `value`. This strategy minimizes but does not eliminate true conflicts in our execution.

In Figure 3 we have hand-instrumented the microbenchmark according to the algorithm described in Section 3.3. The primary transformation is straightforward: (1) clone and instrument the desired region, (2) add a branch to select the right path, and (3) add transitions from the instrumented code as desired. Much of the complexity in transforming larger regions is determining appropriate slow-to-fast transition points, a task which we defer to future work. Note that using `__builtin_expect` on Line 3 is crucial to single-threaded performance.

As described in Section 3, speedup is expected to depend critically on the fraction of instructions in a speculative region devoted to shared-memory loads and stores. We use the `workpernode` parameter to adjust this fraction by executing `workpernode` calls to `rand_r` per node. Each of these calls computes with local data only, with no instrumentation required. We report speedups for the resulting workload characteristics separately.

4.1 Test Platform

Our preliminary tests were performed on an quad chip, Intel Xeon platform. Each chip in this platform contains two, 2-way SMT processors. We refer to the resulting architecture as 4x2x2 in chips/core/SMT order. The underlying 64bit, x86 core is relatively powerful, an important consideration given the large instruction level parallelism of much of the transactional instrumentation we are adding. All benchmarks were compiled with `gcc 4.4.0` with `-fopenmp` and `-O3`.

```
1  parallel for (i = 0; i < iterations; ++i)
2    key = random_key(i)
3    if (__builtin_expect(FASTPATH_START(i)), true)
4  fast_loop:
5      // Uninstrumented loop, lines 6-15 in Figure 2
...
15   else
16     FASTPATH_SETJMP
17     accum = 0
18     current_node = FASTPATH_READ(list_front)
19     while (FASTPATH_READ(current_node->key) != key)
20       tmp = FASTPATH_READ(current_node->key)
21       for (j = 0; j < workpernode; ++j)
22         tmp = rand_r(&tmp)
23       accum += tmp
24       current_node = FASTPATH_READ(current_node->next)
25       if FASTPATH_SWITCH() goto fast_loop:
26     value = FASTPATH_READ(current_node->value)
27     FASTPATH_WRITE(current_node->value, value + accum)
28   FASTPATH_COMMIT(i)
```

Fig. 3. Hand-instrumented version of the microbenchmark code in Figure 2. The Signature algorithm would additionally require that the write on Figure 2, line 15 be instrumented similar to line 27.

4.2 Results

Figure 4 shows speedup for our microbenchmark. Points are an average of five trials of 10 million iterations, corresponding to actual execution times between 4 and 40 seconds. Speedups are relative to uninstrumented sequential code.

The first result from our tests is that single-threaded performance with Fastpath instrumentation is indistinguishable from uninstrumented code. This is important for two reasons: the overhead of fast mode execution is one of the fundamental factors that controls the system's scalability, and a single binary may be suitable for distribution in both single and multi-threaded environments on this particular Xeon system. This is not a general result—its powerful pipeline hides much of the instrumentation latency. Preliminary single-threaded results on a much less complex Sun Niagara2 processor show 15% and 20% slowdown for Value and Signature, respectively.

The scalability of both systems shows the importance of the characteristics of the workload being parallelized. Neither the Value nor the Signature implementation appears able to speed up workloads in which the ratio of uninstrumented to instrumented work is low. The intuition behind this result is that it does us little good to execute speculatively if "replaying" the speculative state takes as long as, or longer than, calculating it in the first place.

The Value system replays by validating its read locations, and writing its write buffer to memory. Validation is proportional to the number of instrumented

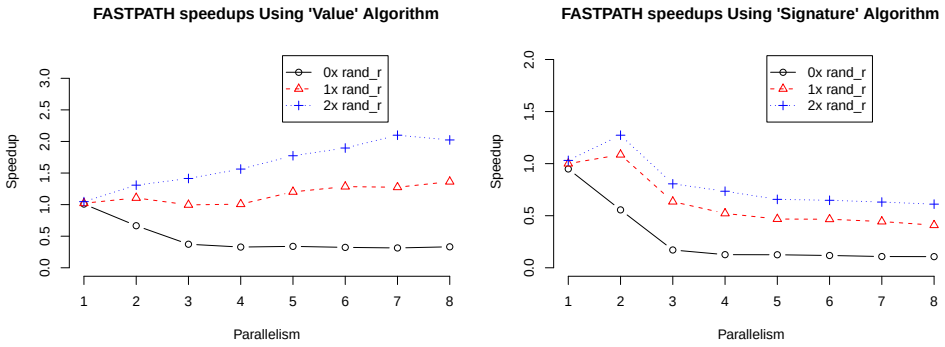

Fig. 4. Speedup for a 4 chip Intel Xeon server.

reads performed. Clearly, the lower the ratio of instrumented to uninstrumented work, the lower the cost of replay vs. the cost of sequential execution.

The Signature system has the same general tasks, but where the Value system's validation is proportional to the number of instrumented reads, the Signature system's validation is proportional to the number of active threads in the system (with a constant factor determined by signature size). This implies that the number of instrumented reads should not play a part in the scalability of the Signature system; rather, the Signature system requires that the amount of overall work available per iteration be enough to dominate the cost of validation. The one caveat to this conclusion is that Signature's accuracy is sensitive to load factor—the number of locations accessed (and thus inserted in the signature) as a fraction of the signature's size. If an iteration touches too many different locations, false conflicts may lead to unnecessary aborts. The size of the signatures can be adjusted dynamically if these conditions arise, at the cost of higher validation overhead.

The `0x rand_r` curves in both graphs in Figure 4 show that the simple list traversal neither contains enough work to overcome Signature's validation cost, nor provides a good enough ratio for Value to scale effectively.

On the other hand, adding one `rand_r` call per inner-loop execution shows promising results for Value, showing no slowdown at any of the thread levels tested, and scalability to a speedup of 1.36 at 8 cores. Two calls to `rand_r` per iteration results in a speedup of 2.10 at 7 cores. These results point to the importance of compiler analysis and optimization to reduce unnecessary instrumentation, an area we are investigating as future work.

Signature behaves differently. A single `rand_r` call can hide the overhead of one validation per iteration, resulting in a speedup of 1.09 at two cores, but not more. An additional `rand_r` boosts speedup to 1.27, with performance trailing off quickly with more cores.

The Signature algorithm requires that a speculative worker intersect the signature corresponding to its read set with all of the published write signatures from tasks that completed between its start and commit points. This is typically p signatures, with p being the maximum number of processors available. One

and two calls to `rand_r` apparently provide enough work to overcome the cost of validation with two active threads, but not with more.

We can minimize this problem somewhat. Our current Signature algorithm chooses to validate speculative workers as soon as they detect a committing writer. The writer's filter was generated generated on a nonlocal core, thus this eager validation results in guaranteed cache misses. Sandboxing provides speculators the opportunity to prefetch the remote signatures and validate at a later time. This should enable us to achieve continued speedup by overlapping much of the cache-miss latency with continued speculative execution, but may also result in delayed conflict detection and lower efficiencies. We expect to implement this heuristic and test its impact as part of future work.

5 Related Work

Loop-level software speculation was pioneered by the LRPD test [7]. It executed a loop in parallel, recorded data accesses in shadow arrays, and checked for dependences after the parallel execution. Later techniques speculatively privatized shared arrays (to allow for false dependences) and combined the marking and checking phases (to guarantee progress) [8–10]. The techniques supported parallel reduction [7, 8]. More general speculation support has been used in Java programs to enable safe futures [11], speculative return-value prediction [12], and speculative data abstractions [13] and in C programs through the copy-or-discard model (CorD) [14].

These techniques use metadata to precisely record the set of data being read and modified. They also use a single copy of the code for both lead and speculative threads. We present two different schemes: the Signature algorithm records the location of data access using an imprecise (but safe) Bloom filter, and the Value algorithm records the result of data access using a log. In both, the lead thread executes faster code.

The STMlite system of Mehrara et al. [15] also draws ideas from software transactional memory, but with a very different implementation, in which all validation checks are performed by a single bookkeeping thread. Other threads log their writes, build read and write signatures, and enqueue these structures for perusal by the bookkeeping thread. They then await permission to write back. Unlike our system, STMlite permits concurrent writebacks in separate threads so long as their write signatures are disjoint. Because the bookkeeping thread does no "real" work of its own, at least three threads are required in order to achieve any speedup, and read and write instrumentation appears in every worker thread.

In SpLIP [16], threads acquire STM-like ownership records for reading and writing, and then perform writes "in place.". Threads keep undo logs for rollback. The code is optimized for ordered speculation, and exploits architectural characteristics to avoid the need for atomic read-modify-write instructions. In the absence of conflicts, SpLIP tasks operate entirely in parallel, without write-back serialization. Rollback, however, is extraordinarily expensive: to achieve any

speedup, speculation must fail well under 1% of the time. Furthermore, after a misspeculation one loop iteration must execute sequentially to ensure progress.

Previous studies do not address the question of optimality for either static or dynamic loop scheduling in the presence of sequential commit. In this paper, we have analyzed the effect of not just sequential commit but also mixed mode speculation including the effect of path switching during execution.

Coarse-grain tasks can be implemented using processes, as in the *BOP* [17] and Grace [18] systems. It has the advantage of being able to start a speculation task anywhere in the code and to buffer speculative states and monitor speculative execution using paging support without instrumenting data access. However, the overhead of starting a process is too high for fine-grained parallelism in the innermost loop. In addition, it monitors heap data at page granularity, which may lead to false alerts.

6 Status and Plans

We have described a prototype implementation of Fastpath, a software system for speculative parallelization of semantically sequential code. Our system is distinguished by its single-address-space implementation, its use of conventional hardware, and its asymmetric instrumentation, in which the lead thread runs at near-native speed.

Our implementation is currently able to detect conflicts between threads using either value-based validation or Bloom filter signatures. These exhibit a tradeoff between per-read and per-region overhead. A simple analytical model allows us to predict achievable speedup based on these overheads and properties of the application. Experiments with a hand-instrumented microbenchmark demonstrate the feasibility of speedups on the order of 25% on two cores—and better on more—when speculative regions have significant private computation and few dynamic conflicts. Moreover they confirm the achievability of worst-case slowdowns below 1%. This is significantly better than reported for previous software-only systems.

We are currently constructing compiler infrastructure that will allow us to instrument programs automatically, once the programmer has identified regions of code for speculative execution. This infrastructure will enable us to experiment with large-scale applications.

References

1. Halstead, R.H.: Multilisp: A language for concurrent symbolic computation. ACM Trans. on Programming Languages and Systems **7**(4) (Oct. 1985) 501–538
2. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by abolishing ownership records. In: Proc. of the ACM Symp. on Principles and Practice of Parallel Programming (PPoPP), Bangalore, India (January 2010) To appear.
3. Olszewski, M., Cutler, J., Steffan, J.G.: JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In: Proc. of the Intl. Conf. on Parallel

Architectures and Compilation Techniques (PACT), Brasov, Romania (Sep. 2007) 365–375
4. Cytron, R.: Doacross: Beyond vectorization for multiprocessors. In: Proc. of the Intl. Conf. on Parallel Processing (ICPP), Saint Charles, IL (Aug. 1986) 836–844
5. Spear, M.F., Michael, M.M., von Praun, C.: RingSTM: Scalable transactions with a single atomic instruction. In: Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA). (June 2008) 275–284
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Comm. of the ACM **13**(7) (July 1970) 422–426
7. Rauchwerger, L., Padua, D.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. IEEE Trans. on Parallel and Distributed Systems **10**(2) (Feb. 1999) 160–199
8. Gupta, M., Nim, R.: Techniques for run-time parallelization of loops. In: Proc. of the Intl. Conf. for High-Performance Computing, Networking, Storage, and Analysis (SC), Orlando, FL (Nov. 1998) 12
9. Dang, F., Yu, H., Rauchwerger, L.: The R-LRPD test: Speculative parallelization of partially parallel loops. In: Proc. of the Intl. Parallel and Distributed Processing Symp. (IPDPS), Ft. Lauderdale, FL (Apr. 2002) 20–29
10. Cintra, M.H., Llanos, D.R.: Design space exploration of a software speculative parallelization scheme. IEEE Trans. on Parallel and Distributed Systems **16**(6) (June 2005) 562–576
11. Welc, A., Jagannathan, S., Hosking, A.L.: Safe futures for Java. In: Proc. of the Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), San Diego, CA (2005) 439–453
12. Pickett, C.J.F., Verbrugge, C.: Software thread level speculation for the Java language and virtual machine environment. In: Proc. of the Wkshp. on Languages and Compilers for Parallel Computing (LCPC), Hawthorne, NY (Oct. 2005) 304–318
13. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. Comm. of the ACM **52**(9) (Sep. 2009) 89–97
14. Tian, C., Feng, M., Nagarajan, V., Gupta, R.: Copy or discard execution model for speculative parallelization on multicores. In: Proc. of the Intl. Symp. on Microarchitecture (MICRO), Lake Como, Italy (Nov. 2008) 330–341
15. Mehrara, M., Hao, J., Hsu, P.C., Mahlke, S.A.: Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In: Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI), Dublin, Ireland (June 2009) 166–176
16. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA), Calgary, AB, Canada (Aug. 2009) 223–232
17. Ding, C., Shen, X., Kelsey, K., Tice, C., Huang, R., Zhang, C.: Software behavior oriented parallelization. In: Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI), San Diego, CA (June 2007) 223–234
18. Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: Safe multithreaded programming for C/C++. In: Proc. of the Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL (Oct. 2009)