

Fast Track: Supporting Unsafe Optimizations with Software Speculation

Kirk Kelsey, Chengliang Zhang and Chen Ding
Computer Science Department
University of Rochester
{*cding,kelsey,zhangchl*}@*cs.rochester.edu*

Abstract

The use of multi-core, multi-processor machines is opening new opportunities for software speculation, where program code is speculatively executed to improve performance at the additional cost of monitoring and error recovery. In this paper we describe a new system that use software speculation to support unsafely optimized code. We open a fast, unsafe track of execution but run the correct code on other processors to ensure correctness. We have developed an analytical model to measure the effect of major parameters including the speed of the fast track, its success rate, and its overheads. We have implemented a prototype and verified the correctness and performance using a synthetic benchmark on a 4-CPU machine.

1 Introduction

The increasing problems of power, heat dissipation, and design complexity have caused a shift in processor technology to favor multi-core, multi-processors. Most programs, however, cannot yet fully utilize all CPUs in a system. The redundant computing resource opens new opportunities for software speculation, where program code is speculatively executed to improve speed at the cost of having to monitor for errors and the risk of having to re-execute code when an error happens.

In this paper we describe a new use of software speculation in a system we call *fast track*. We are motivated by the observation that there are many unsafe ways of improving the performance of a program (as we will discuss shortly). We can open a fast track of execution by using some unsafely optimized code. To ensure correctness, we execute the correct code on another processor and use it to check the correctness of the fast execution.

Figure 1 shows an example loop that has two tracks in its loop body: a fast, unsafe implementation, and the normal, safe code. At each loop iteration, the fast code and normal code are run in separate Unix processes (hence the return value check in the example). After the fast code finishes and stores the changes it makes, the next iteration

```

while (...) {
    ...
    if (FastTrack()==1) {
        fast_fury_fortuitous(); /* unsafely optimized */
    }
    else {
        slow_safe_sequential(); /* normally optimized */
    }
    EndDualTrack();
    ...
}

```

Figure 1: An example loop on fast track

starts, including both fast and normal tracks. When the normal code finishes, it verifies the changes made by the fast code, cancel the fast execution and continue the execution upon an error. It is considered an error for the fast code to finish after the normal code does.

One may question the benefit of this setup: suppose the fast code gives correct results, wouldn't we still need to wait for the normal execution to finish to know it is correct? The reason for the speed improvement is the overlapping of loop iterations. Without fast track, the next iteration cannot start until the previous one fully finishes. With fast track, the next iteration starts when the fast code for the previous iteration finishes. In other words, although the checking is as slow as the original code, it is now done in parallel.

The performance of the system is guaranteed against bad or incorrect fast track implementation. If the fast code has an error, falls into an infinite loop, or if the fast code sometimes runs slower than the normal code, the program would execute the normal code sequentially and will not be delayed by a strayed fast track.

In general the fast code can be any optimization inserted by a compiler or a programmer. We discuss three types that are good fits for fast track: they may lead to great performance gains but their correctness and profitability are difficult to ensure.

Memoization For any procedure, we may remember the past inputs and outputs. Instead of re-executing it, we can retrieve the old result when seeing the same input. Studies dated back to at least 1968 [26] show dramatic performance benefits. A common example is recursive Fibonacci number calculation, for which memoization changes the asymptotic complexity from $O(1.6^n)$ to $O(n)$. Memoization is difficult for C programs because the side effect of large procedures is hard to determine. It may slow down an execution if the cost of storing inputs and retrieving results outweighs the benefit.

Unsafe compiler optimization A major source of the overhead in modern programs comes from supporting dynamic features such as polymorphism, exceptions, other forms of late binding, and concurrency. As a result, the compiler has to insert expensive operations such as virtual table lookup on the one hand and cannot perform the most effective optimizations such as instruction scheduling

```

...
if (FastTrack()==1)
    fast_step_1();
else
    step_1();
EndDualTrack();
...
if (FastTrack()==1)
    fast_step_2();
else
    step_2();
EndDualTrack();
...

```

Figure 2: Two example function calls on fast track

and register allocation on the other hand. If a compiler is freed of correctness concerns, optimizes aggressively based on partial information, or specialize for specific inputs, it may generate much faster code that is potentially unsafe.

Manual program tuning A programmer can often identify performance problems in large software and make changes to improve the performance on test inputs. However, the most radical solutions are often the most difficult for verifying the correctness or ensuring good performance on other inputs. As a result, we waste many creative solutions that an automatic compiler cannot possibly do.

Most existing software speculation techniques focus on loop or region based parallelization [6, 9, 14, 29, 32] not improving inherently sequential code. The hardware techniques such as branch prediction, value prediction, and speculative loads use speculation for the same purpose as we do but only for one instruction. In this paper we present the design and evaluation where large, possibly programmer-inserted code regions are executed in fast track.

2 The Fast-Track System

2.1 Programming Interface

A fast-track region contains the beginning branch `if (FastTrack()==)`, the two tracks, and the ending statement `EndDualTrack()`. An execution of the code region is called a *dual-track instance*. The two tracks are the *fast instance* and the *normal instance*. A program execution consists of a sequence of dual-track instances and computations before, between, and after these instances.

Any region of code whose beginning dominates the end can be made a dual-track region. Nested regions are allowed. When a inner dual-track region is encountered, the outer fast track will take the inner fast track, while the outer normal track will take the inner normal track. At present, we do not allow abnormal control flows entering into and leaving from the middle of a region except for a program exit. We also prohibit

statements with side effects such as system calls and file input and output inside a dual-track region. Dynamic memory allocation and reclamation are permitted. We set a threshold for the maximal size that a fast instance may allocate, so an incorrect fast instance will not stall the system by taking an excessive amount of memory.

Figure 1 in the last section shows an example of adding a fast track to the body of a loop. The dual-track region can include only a part of the loop body, and multiple dual-track regions can be placed side by side in the same iteration or in any other code sequence. Figure 2 shows the use of fast track on two procedural calls. The . . . means other statements in between. The dual-track regions do not have to be on a straight sequence. One of them can be in a branch and the other can be in another loop.

2.2 Compiler and Run-time Support

Dual-track system guarantees that the same result is produced as in the sequential execution. By using Unix processes, fast track eliminates any interference between parallel executions through the replication of the address space. During execution, it records which data are changed by each pairs of normal and fast instances. When both instances finish, it checks whether the changes are identical.

Program data consists of global, stack, and heap data. The stack data is protected by the compiler, which identifies the set of local variables that may be modified through interprocedural MOD analysis [7] and then inserts checking code accordingly. Imprecision in compiler analysis leads to more variables being checked but it does not affect correctness since the analysis is conservative.

The global and heap data are protected by OS paging support. At the beginning of a dual-track instance, the system turns off write permission for both tracks for all global and heap data. It installs customized page-fault handlers that open the permission for write upon the first write access. At the same time, the handler records which page has been modified in an access map.

Figure 3 shows an illustration to help explain the run-time support. In part (a), three normal instances of one or more dual-track regions are run sequentially. In part (b), fast-track uses a single process to execute fast instances sequentially. At the beginning, the two instances of the first dual-track region are started at the same time as two processes. When the fast instance finishes, the process stores its data changes, including the modified stack variables and global and heap pages. It then continues to run the next fast instance and at the same time starts the next normal instance in a separate process, as shown in part (b).

When the first normal instance finishes, its process compares the changed data with that of the fast instance. If the changes are identical, the process quits. If the changes are different, or if the fast instance has not yet finished, the normal instance aborts the process running the fast instances, and starts the next pair of dual-track instances in two processes (by forking a process for running the fast instance). The cycle of the execution continues until the end. The process running the fast instances treats a program exit as it does for errors, I/O, and other system calls—it will stall, waiting to be killed (if it reaches the exit by mistake) or waiting for a normal instance to officially reach the program exit.

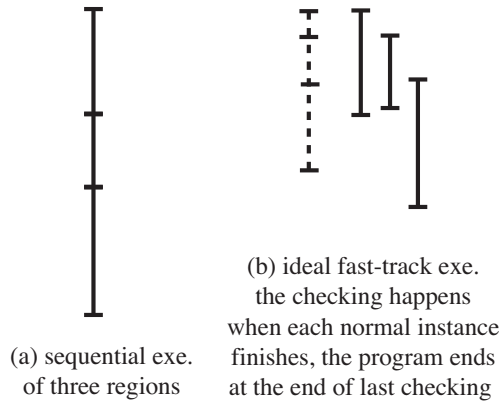


Figure 3: Illustration of a fast-tracked sequential execution

For implementation we use Unix signals to effect a message-passing implementation. No meta data is shared except for the access maps, which are never concurrently accessed. During execution, each process marks the page accesses in its own access map. At the end, the maps are compared by one process (the process running the normal instance). Currently the data are communicated through a pipe with a bounded capacity. If the size of changed data is too large, the system simply abandons the fast instance.

2.3 Dealing with Resource Constraints

Dual-track executions require additional processors and memory. To avoid over committing available resources, we may wait before starting new dual-track executions. We define the *depth* of fast-track execution as the maximal number of concurrent dual-track instances at run time.

We assume that the operating system implements copy-on-write, which lets processes share memory pages that are not accessed or read-only. Still in the worst case where every dual-track instance modifies every data page, the system needs d times the memory needed by the sequential run, where d is the fast-track depth. We may control the memory overhead in two ways. First, we abandon a fast instance if it modifies more pages than an empirical constant threshold h . This bounds the memory increase to be no more than dhM , where M is the VM page size. Second, we adjust the threshold based on the available memory in the system. Memory usage is difficult to estimate since it depends on the operating system and other running processes. In our earlier work we developed an on-line monitoring method based on reading the number of page faults reported by Linux [35]. Similar resource-based methods can be used. Currently, our test cases use far less memory than that available in our test system, so we do not consider memory resource further in this paper.

For the processor resource, we consider two control schemes. In the first, we wait until two processors are available before we start the next dual-track instance. We call

it *synchronous dual-track execution* because the two tracks are always started together. Often the fast instance finishes before the normal instance, and we have one processor available but not two. The second scheme starts the normal instance of the next dual track instance, waits for the next available processor, and then starts the fast instance. We call the latter scheme *asynchronous dual-track execution*. It makes full use of all available processors at all times.

3 Analysis

We use the following symbols to represent the basic parameters including the length of the program execution, the number of available processors, and the overheads and success rate of fast track.

- The original program execution $E = u_0 r_1 u_1 r_2 \dots r_n u_n$ is a sequence of dual track region instances separated by intervening computations.
- The function $T()$ gives the running time for a part of the execution.
- p ($p > 1$) is the number of available processors.
- A fast instance takes a fraction x ($0 \leq x \leq 1$) of the time the normal instance takes and the success rate of fast track is α ($0 \leq \alpha \leq 1$).
- The dual-track execution has a time overhead q_c ($q_c \geq 0$) per instance and is slowed down by a factor of q_e ($q_e \geq 0$) because of the monitoring for modified pages.

3.1 An Analytical Model

The original execution time is $T(E) = T(u_0) + \sum_{i=1}^n T(r_i u_i)$. By reordering the terms we have $T(E) = \sum_{i=1}^n T(r_i) + \sum_{i=0}^n T(u_i)$. We name the two parts $E_r = r_1 r_2 \dots r_n$ and $E_u = u_0 u_1 \dots u_n$. The time $T(E_u)$ is not changed by fast-track execution because any u_i takes the same amount of time regardless of whether it is executed with a normal or a fast instance. We now focus on $T(E_r)$, in particular the average time taken per r_i , $t_r = \frac{T(E_r)}{n}$, and how the time changes as a result of fast track.

Since we would like to derive a closed formula to examine the effect of basic parameters, we consider a regular case where the program is a loop with n equal length iterations. A part of the loop body is a fast-track region. Let $T(r_i) = t_c$ be the (constant) original time for each instance of the region. The analysis can be extended to the general case where the length of each r_i is arbitrary and t_c is the average. The exact result would depend on assumptions about the distribution of $T(r_i)$.

With fast track, an instance may be executed by a normal instance in time $t_s = (1 + q_e)t_c + q_c$ or a fast instance in time t_f^p , where q_c and q_e are overheads.

In the best case when all fast instances are correct ($\alpha = 1$) and the machine has unlimited resources $p = \infty$, the first fast instance is verified after t_s (in $T(E_r)$ we consider only dual-track instances) and since the rest $n - 1$ instances finish at a rate

of $t_f^\infty = (1 + q_e)xt_c + q_c$, where x is the speedup by fast track and q_c and q_e are overheads. The average time and the overall speedup are

$$\bar{t}_f^\infty = \frac{(t_s + (n-1)t_f^\infty)}{n}$$

$$\bar{x}^\infty = \frac{\text{original time}}{\text{fast track time}} = \frac{nt_c + T(E_u)}{n\bar{t}_f^\infty + T(E_u)}$$

In the steady state $\frac{t_s}{t_f^\infty}$ dual-track instances are run in parallel. For simplicity the equation does not show the fixed lower bound of fast track performance. Since a fast instance is aborted if it turns out to be slower than the normal instance, the worst-case is $t_f^\infty \leq t_s$, $\bar{t}_f^\infty \leq t_s$, and consequently $\bar{x} \geq \frac{nt_c + T(E_u)}{n((1+q_e)t_c + q_c) + T(E_u)}$. The worst case time is bounded only by the overhead of the system and not by the quality of fast-track implementation.

We now consider the limited number of processors. For the sake of illustration we give the formula for the synchronous dual track and add the effect of the asynchronous dual track later. With p processors, the system can have at most $d = \min(p-1, \frac{t_s}{t_f^\infty})$ dual-track instances running concurrently, where d is the *depth* of fast track execution. It is an average so it may take a value other than an integer. When $\alpha = 1$, $p-1$ dual-track instances take $t_s + (p-2)t_f^\infty$ ($p \geq 2$) time. Therefore the average time and the overall speedup (assuming $p-1$ divides n) are

$$\bar{t}_f^p = \frac{t_s + (d-1)t_f^\infty}{d}$$

$$\bar{x}^p = \frac{nt_c + T(E_u)}{n\bar{t}_f^p + T(E_u)}$$

As a normal instance for r_i finishes, it may find the fast instance incorrect, cancel the on-going parallel execution, and restart the system from r_{i+1} . The restarting has the same effect as stalling for the lack of processors as just discussed. Every failure adds a cost of $t_s - t_f^\infty$, so the average time with a success rate α is $(1-\alpha)(t_s - t_f^\infty) + t_f^p$. Redefining t_f^p we have

$$\bar{t}_f^p = (1-\alpha)(t_s - t_f^\infty) + \frac{t_s + (d-1)t_f^\infty}{d}$$

Finally we consider asynchronous dual track. As $p-1$ dual-track instances execute and when the last fast instance finishes, the system start the next normal instance instead of waiting for the first normal instance to finish (and start the next normal and fast instances together). Effectively it finishes $d + (t_s - dt_f^\infty)$ instances, hence the change to the denominator. Augmenting the previous formula we have

$$\bar{t}_f^p = (1-\alpha)(t_s - t_f^\infty) + \frac{t_s + (d-1)t_f^\infty}{d + t_s - dt_f^\infty}$$

After simplification, asynchronous dual track may seem to increase the per instance time rather than decreasing it. But it does decrease the time because $d \leq \frac{t_s}{t_f^\infty}$. The

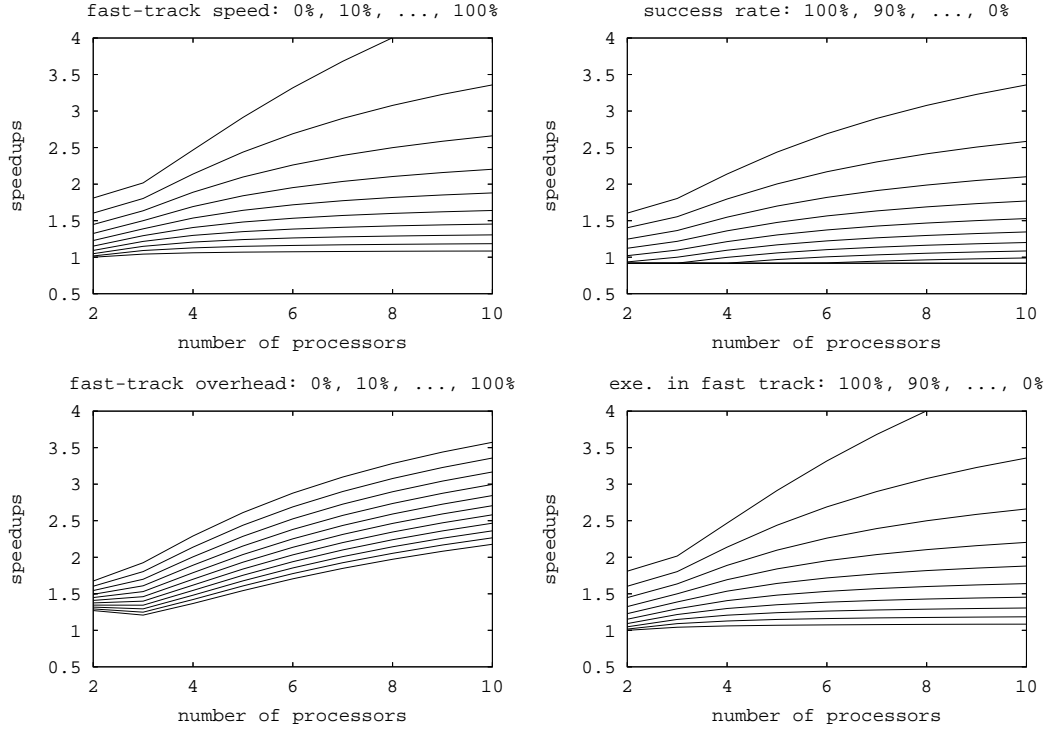


Figure 4: Analytical results of the fast-track system where the speed of the fast track, the success rate, the overhead, and the portion of the program executed in dual-track regions vary. The order of the parameters in the title in each graph corresponds to the top-down order of the curves in the graph.

overall speedup (bounded from below and $n \geq 2$) is as follows, where all the basic factors are modeled.

$$\bar{x}^p = \max\left(\frac{nt_c + T(E_u)}{n((1 + q_e)t_c + q_c) + T(E_u)}, \frac{nt_c + T(E_u)}{nt_f^p + T(E_u)}\right)$$

3.2 Simulation Results

We translate the above formula into actual speedup numbers and examine the effect of major parameters: the speed of the fast track, the success rate, the overhead, and the portion of the program executed in dual-track regions. The four graphs in Figure 4 show their effect for different numbers of processors ranging from 2 to 10 in a step of 1. The fast-track system has no effect on a single-processor system.

All four graphs include the following setup where the fast instance takes 10% the time of the normal instance ($x=0.1$), the success rate (α) is 100%, the overhead (q_c and q_e) adds 10% execution time, and the program spends 90% of the time in dual-track

regions. The performance of this case is shown by the second highest curve in all but the top-right graph, in which it is shown by the highest curve. Fast-track improves the performance from a factor of 1.60 with 2 processors to a factor of 3.47 with 10 processors. The maximal possible speedup for this case is 3.47.

When we change the speed of the fast instance to vary from 0% to 100% the time of the normal instance, the speedup changes from 1.80 to 1.00 with 2 processors and from 4.78 to 1.09 with 10 processors, as shown by the top-left graph. When we reduce the success rate from 100% to 0%, the speedup changes from 1.60 to 0.92 (8% slower because of the overhead) with 2 processors and from 3.47 to 0.92 with 16 processors, as shown by the top-right graph. Naturally the performance hits the worst case when the success rate is 0%. When we reduce the overhead from 100% to 0% of the running time, the speedup increases from 1.27 to 1.67 with 2 processors and from 2.26 to 3.69 with 16 processors, as shown by the bottom-left graph. Note that with 100% overhead the fast instance still finishes in 20% the time of the normal instance, although the checking needs to wait twice as long. Finally, when the coverage of the fast-track execution increases from 10% to 100%, the speedup increases from 1.00 to 1.81 with 2 processors and from 1.08 to 4.78, as shown by the bottom-right graph.

If the analytical results are correct, it is not overly difficult to obtain a 30% improvement with 2 processors, although the maximal gain is limited by the time spent outside dual-track regions, the speed of the fast instance, and the overhead of fast-track. The poor scalability is not a surprise given the program is inherently sequential to begin with.

We make two final observations from the simulation results. First, asynchronous dual-track execution is clearly beneficial. Without it there can be no improvement with 2 processors. It often improves the theoretical maximum speedup, although the increase is slight when the number of processors is large. Second, the benefit of fast-track system highly depends on the four parameters. We believe that the four parameters can be efficiently monitored at run time, so the analytical model may be used as part of the on-line control to adjust the depth of fast-track execution with the available resources.

4 Experimental Results

4.1 Implementation and Experimental Setup

We have implemented the compiler support in Gcc 4.0.1, in particular, in the intermediate language, GIMPLE (based on static-single assignment [8] similar to SIMPLE form [16]), so the transformation is applied after high-level program optimization passes but before machine code generation. The main transformation is converting global variables to use dynamic allocation, so the run-time support can place them for appropriate protection. The compiler allocates a pointer for each global (and file and function static) variable, inserts an initialization function in each file that allocates heap memory for variables (and assigns initial values) defined in the file, and redirects all accesses through the global pointer. All initialization functions are called at the beginning of the main function. The indirection causes only marginal slowdown because

most global-variable accesses have been removed or converted to (virtual) register access after the GIMPLE passes [32].

For data protection, we have not implemented the compiler analysis for local variables. Stack data is not checked, but global and heap variables are protected. The run-time system is implemented as a statically linked library. Shared memory is used only for storing access maps. The design guarantees forward progress, which means no deadlocks or starvation provided that the OS does not permanently stall any process.

The test machine has two Intel dual-core Xeon 3 GHz processors with 4MB L1 cache, 4GB Memory, and no hyper-threading. We use our modified Gcc compiler to compile all programs with “-O3” flag.

4.2 Results of a Synthetic Test

To test the effect of various parameters, we have devised a simple program. The basic algorithm is given in the pseudo code as follows.

```
initialize N numbers in a vector
for T iterations do
  normal track    perform C computations on each number
                  find the largest result in the process
                  store it in the output vector

  fast track
  sample S random vector elements
  perform C computations on each sample
  find the largest result in the process
  store it in the output vector
end dual-track

change N1 numbers randomly by a new value
repeat the loop
end loop
output the result vector
```

The program repeatedly updates some elements of a vector and finds their largest result from certain computations. By changing the size of the vectors, the size of samples, and the frequency of updates, we can effect different success rates by the normal and the fast instances. Table 1 shows the results we observed on the 4-CPU machine. The numbers are speedups over the base sequential execution, which takes 3.33 seconds. We time each run three times. On our machine, the time difference of different runs is always smaller than 0.01 second.

The sampling-based fast instance runs in 2.3% the time of the normal instance. When all fast instances succeed, they improve the performance by a factor of 1.73 on 2 processors, 2.78 on 3 processors, and 3.87 on four processors. When we reduce the frequency of updates, the success rate drops. At 70%, the improvement is a factor of 2.09 on 3 processors and changes only slightly when the fourth processor is added. This is because that the chance for four consecutive fast instances to succeed is only 4%. When we reduce the success rate to 30%, the chance for three consecutive successful fast tracks drops to 2.7% and no improvement is observed for more than 2 processors.

Table 1: Effect of fast-track success rates in the synthetic benchmark

success rate	number processors			
	1	2	3	4
100%	1	1.73	2.78	3.87
70%	1	1.47	2.09	2.15
30%	1	1.29	1.29	1.29
0%	1	0.94	0.94	0.94

Table 2: The effect of fast-track tuning in the synthetic benchmark

sample size	number processors			
	1	2	3	4
100	1	1.48	2.09	2.15
200	1	1.71	2.64	2.97
300	1	1.70	2.71	3.78
400	1	1.68	2.69	3.74

The speedup from 2 processors is 1.29. In the worst case when all fast instances fail, we see that the overhead of forking and monitoring the normal track adds 6% to the running time.

The results in Table 2 show interesting tradeoffs when we tune the fast track by changing the size of samples. On the one hand, a larger sample size means more work and slower speed for the fast track. On the other hand, a larger sample size leads to a higher success rate, which allows more consecutive fast tracks succeed and consequently more processors utilized. The success rate is 70% when the sample size is 100, which is the same configuration as the row marked “70%” in Table 1.

The best speedup for 2 processors happens when the sample size is 200 but more processors do not help as much (2.97 speedup) as when the sample size is 300, where 4 processors lead to a speedup of 3.78. The second experiment shows the significant effect of tuning when using unsafely optimized code. Our experience is that the automatic support and the analytical model have made tuning much less labor intensive.

5 Related work

The concept of data dependence was developed for parallelization (vectorization) by Lamport in the Parallelizer system, by Kuck and his colleagues in Paraphrase, and by Kennedy and others in Parallel Fortran Converter (PFC) [1]. Control dependence was developed by Ferrante et al [1, 8]. Static dependence checking can be overly conservative when two statements are mostly but not always independent and when the independence is too difficult to prove, especially between large code regions.

Early dynamic checking techniques developed for array-based scientific programs include the inspector-executor for dynamic parallelization [30] and the lazy privatizing *doall* (LPD) test for speculative parallelization [29]. The LPD test has two separate phases: the marking phase executes the loop and records access to shared arrays in

a set of shadow arrays, and the analysis phase then checks for dependence between any two iterations. Later techniques speculatively privatize shared arrays (to allow for non-flow dependences) and combine the marking and checking phases [6, 9, 14]. The technique of array renaming is generalized in Array SSA [21]. Inspection is used to parallelize Java programs at run-time [4]. Dynamic dependence checking is also used for improving cache performance [33]. Our system increases rather than reduces the total amount of work in a program. The new overhead is traded for parallelism as it can then overlap the sequential execution of dual-track regions. Fast-track provides general support for unsafe optimizations including speculative parallelization or locality optimization. The fast track is programmable and can be either given by a programmer or generated by a compiler.

Ding and Li developed compiler support for memoization in C programs [11]. The technique uses profiling to identify code fragments that tends to produce a small set of values and then replace the code with a table lookup. The compiler ensures correctness by re-wiring all inputs and outputs of the original fragment. For a set of media benchmarks and a computer game on a Compaq IPAQ, the technique improves program speed by 7% to a factor of two. This scheme is not speculative and therefore it is free of the checking and re-execution overheads. However, it does not exploit unsafe optimizations that is correct most of the times but not all the times.

Intel Itanium architecture supports for data (and control) speculation where a load can be issued early and checked later. Compiler techniques have been developed to generate speculative and recovery code, for example, for SSA-based partial redundancy removal [23].

Hardware-based thread-level speculation is among the first to automatically exploit loop- and method-level parallelism in integer code. Interested readers may download www.ics.uci.edu/~akejariw/SpeculativeExecutionReadingList.pdf (pages 9–16) for a list of 95 papers on this subject between 1991 and 2005. In a recent study, Kejariwal et al. classified existing loop-level techniques as control, data, or value speculation and showed that the maximal possible speedup is 12% on average for SPEC2Kint, assuming no speculation overhead and unlimited computing resources [19]. The result suggests that for speculation to make more use of multi-processor machines, we need to look for larger code regions and for ways to seek hints from the programmer to target the right program and use the right speculation for the appropriate code region.

Speculative execution is closely related to methods of nonblocking concurrency control. Transactional memory was originally proposed as a hardware mechanism to support nonblocking synchronization (by extending cache coherence protocols) [18]. It is rapidly gaining attention because of its potential to be a general and easy to use solution for concurrency [13]. Various software implementations rely on transactional data structures and primitive atomic operations available on existing hardware [15, 17, 31] (see [24] for a survey). Many hardware-based TM systems have also been developed. Run-time dependence checking is an efficient (but not necessary) solution to ensure serializability (i.e. the Bernstein conditions) [1, 3], which is NP-hard in the general case [27]. In particular, fast-track verifies that the result is the same as the execution of dual-track regions in their program order, while transactional memory verifies that the result is the same as some sequential execution of transactions. Both need data monitoring to identify conflicts.

For large programs using complex data, per-access monitoring causes slowdowns often in integer multiples, as reported for data breakpoints and on-the-fly data race detection, even after removing as many checks as possible by advanced compiler analysis [25, 28, 34]. Run-time sampling based on data [10] or code [2, 5] are efficient but does not monitoring all program accesses. Page-based data monitoring was used for supporting distributed shared memory [20, 22] and then for many other purposes including race detection [28] and read barrier implementation in a garbage collector [12].

Fast-track combines page-based monitoring and software speculation to make use of unsafely optimized code without compromising correctness. As far as we know, this use of speculation support is novel. In addition, our system differs from previous software speculation schemes in that (1) the fast-track system is programmable; (2) it uses OS-paging support, handles general C/Fortran programs, and runs on existing hardware; (3) it uses redundant computation to bound the worst-case performance; and (4) with asynchronous dual-track execution, it improves performance with just two processors. Redundant computation is used in our earlier work but in a different context—software speculative parallelization [32].

6 Summary

We have described fast track, a new system that supports unsafely optimized code while guaranteeing correctness and the worst-case performance. The key features of the systems include a programmable interface, page-based monitoring to support general programs on existing systems, redundant computation to bound the worst-case slowdown, and asynchronous dual-track execution to improve performance with just two processors. We have developed an analytical model that shows the effect from major parameters including the speed of the fast track, the success rate, the overhead, and the portion of the program executed in dual-track regions. We have implemented a prototype including compiler and run-time support and tested a synthetic benchmark on an Intel machine with two dual-core processors. Both analytical and empirical results suggest that fast track can obtain significant performance improvement for inherently sequential applications on a small number of processors.

Acknowledgments

We are grateful to IBM Research for making the Jikes RVM system available under open source terms. The MMTk memory management toolkit was particularly helpful.

This research is supported by the National Science Foundation (Contract No. CNS-0509270, CCR-0238176), IBM CAS Fellowship, two grants from Microsoft Research, and an equipment donation from IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of above named organizations.

References

- [1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.
- [2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.
- [3] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, Oct. 1966.
- [4] B. Chan and T. S. Abdelrahman. Run-time support for the automatic parallelization of java programs. *Journal of Supercomputing*, 28(1):91–117, 2004.
- [5] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
- [6] M. H. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, 2005.
- [7] K. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2003.
- [8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
- [9] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. Technical report, CS Dept., Texas A&M University, College Station, TX, 2002.
- [10] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.
- [11] Y. Ding and Z. Li. A compiler scheme for reusing intermediate computation results. In *Proceedings of the IEEE / ACM International Symposium on Code Generation and Optimization*, 2004.
- [12] J. R. Ellis, K. Li, and A. W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report Research Report 25, HP/DEC Systems Research Center, 1988.
- [13] D. Grossman. Software transactions are to concurrency as garbage collection is to memory management. Technical Report UW-CSE 2006-04-01, Dept. of Computer Science and Engineering, University of Washington, 2006.

- [14] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. In *Proceedings of SC'98*, 1998.
- [15] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Anaheim, CA, Oct. 2003.
- [16] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B.Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of LCPC. Lecture Notes in Computer Science No. 457*, 1992.
- [17] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th PODC*, pages 92–101, Boston, MA, Jul. 2003.
- [18] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, May 1993.
- [19] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *Proceedings of ACM International Conference on Supercomputing*, June 2006.
- [20] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Conference*, Jan. 1994.
- [21] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Proceedings of Symposium on Principles of Programming Languages*, San Diego, CA, January 1998.
- [22] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dept. of Computer Science, Yale University, New Haven, CT, Sept. 1986.
- [23] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
- [24] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, Department of Computer Science, University of Rochester, June 2004.
- [25] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared memory parallel programs. Technical Report CRPC-TR92232, Rice University, September 1992.
- [26] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

- [27] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of ACM*, 26(4), 1979.
- [28] D. Perkovic and P. J. Keleher. A protocol-centric approach to on-the-fly race detection. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1058–1072, 2000.
- [29] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
- [30] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.
- [31] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, Feb. 1997.
- [32] X. Shen. *Large-Scale Program Behavior Analysis for Adaptation and Parallelization*. PhD thesis, Computer Science Dept., Univ. of Rochester, July 2006.
- [33] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
- [34] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: design and implementation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.
- [35] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *Proceedings of the International Symposium on Memory Management*, Ottawa, Canada, June 2006.