Fast Track:
A Software System for Speculative Program Optimization

Kirk Kelsey, Tongxin Bai, Chen Ding
Department of Computer Science
University of Rochester
Rochester NY, USA
{*kelsey,bai,cding*}*@cs.rochester.edu*

Chengliang Zhang
Microsoft Corporation
Redmond WA, USA
chengzh@microsoft.com

***Abstract*—Fast track is a software speculation system that enables unsafe optimization of sequential code. It speculatively runs optimized code to improve performance and then checks the correctness of the speculative code by running the original program on multiple processors.**

We present the interface design and system implementation for Fast Track. It lets a programmer or a profiling tool mark fast-track code regions and uses a run-time system to manage the parallel execution of the speculative process and its checking processes and ensures the correct display of program outputs. The core of the run-time system is a novel concurrent algorithm that balances exploitable parallelism and available processors when the fast track is too slow or too fast. The programming interface closely affects the run-time support. Our system permits both explicit and implicit end markers for speculatively optimized code regions as well as extensions that allow the use of multiple tracks and user defined correctness checking. We discuss the possible uses of speculative optimization and demonstrate the effectiveness of our prototype system by examples of unsafe semantic optimization and a general system for fast memory-safety checking, which is able to reduce the checking time by factors between 2 and 7 for large sequential code on a 8-CPU system.

I. Introduction

The shift in processor technology toward multi-core, multi-processors opens new opportunities for speculative optimization, where the unsafely optimized code marches ahead speculatively, while the original code follows behind to check for errors and recover from mistakes. In the past, speculative program optimization have been extensively studied both in software and hardware as an automatic technique. The level of improvement, although substantial, is limited by the ability of static and run-time analysis. In fact, existing techniques target mostly single loops and consider only dependence and value based transformations.

In this paper we describe a programmable system we call *fast track*, where a programmer can install unsafely optimized code while leaving the tasks of error checking and recovery to the underlying implementation. The system uses software speculation, which needs coarse-grain tasks to amortize the speculation overhead but does not require special hardware support.

```
while (...) {
  ...
  if (FastTrack ()){
   /* unsafely */
   /* optimized */
   fast_fortuitous();
  }
  else {
   /* safe code */
   safe_sequential();
  }
  EndDualTrack();
  ...
}
```

Figure 1. Unsafe loop optimization using fast track. Iterations of `fast_fortuitous` will execute sequentially. Iterations of `safe_sequential` will execute in parallel with one another, checking the correctness of the fast iterations.

```
...
if (FastTrack ())
   /* optimized */
   fast_step_1();
else
   /* safe code */
   step_1();
...
if (FastTrack ())
   /* optimized */
   fast_step_2();
else
   /* safe code */
   step_2();
```

Figure 2. Unsafe function optimization using fast track. Routines `fast_step_2` and `step_2` can start as soon as `fast_step_1` completes. They are likely to run in parallel with `step_1`.

The programming interface allows a programmer to optimize code at the semantic level, select competing algorithms at run time, and insert on-line analysis modules such as locality profiling or memory-leak detection. Figures 1 and 2 show example use of fast track to enable unsafely optimized loop and function execution. If the fast tracks are correct, they will constitute the critical path of the execution; the original loop iterations and function executions, which we refer as normal tracks, will be carried out in parallel, "on the side". Thus, fast track allows multiprocessors to improve the speed of even sequential tasks.

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 the normal tracks. Without fast track, the next normal track cannot start until the previous one fully finishes. With fast track, the next one starts once the fast code for the previous normal track finishes. In other words, although the checking is as slow as the original code, it is now done in parallel. If

the fast code has an error or occasionally runs slower than the normal code, the program would execute the normal code sequentially and will not be delayed by a strayed fast track.

Speculative optimization has been widely studied. Most software techniques focus on loop or region based parallelization [1]–[5] not improving inherently sequential code. Hardware-based speculation support has been used to enable unsafe optimization of sequential code [6]–[9]. Fast track supports speculative program optimization in software, and because of its software nature, it provides a programming interface for invoking and managing the speculation system. The software implementation poses interesting concurrency problems, and the features of the programming interface strongly influences the system support.

Fast track relies on speculation, which, if it fails, wastes system resources including electric power. It may even delay useful computations by causing unnecessary contention. On the other hand, it enables inherently sequential computation to utilize multiple processors. The programming interface enables a new type of programming—programming by suggestions. A user can suggest faster implementations based on partial knowledge about a program and its usage.

Our implementation is based on processes rather than threads. It uses conventional hardware and is independent of the memory consistency model, making fast-track programs easily portable. The run-time system manages the parallelism transparently and programmers need no parallel programming or debugging. Fast-track code is added at the source level, allowing the combined code to be fully optimized by conventional compilers. The interface may also be used by compilers and profiling tools, which can leverage the existing techniques to fully automate the process.

II. Programming Interface

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

Any region of code whose beginning dominates the end can be made a dual-track region, and nesting is 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. We prohibit statements with side effects that would be visible across the processor boundary, such as system calls and file input and output inside a dual-track region. We limit the memory that a fast instance may allocate, so an incorrect fast instance will not stall the system through excessive consumption.

Figure 1 in the previous section shows an example of a fast track that has been added to the body of a loop. The dual-track region can include just a portion of the loop body, multiple dual-track regions can be placed back-to-back in the same iteration, and a region can be used in straight-line code. 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.

III. System Design

A. Compiler Support

The fast-track system guarantees that it produces the same result as 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 of the normal and fast instances. When both instances finish, it checks whether the changes they made are identical.

Program data can be divided into three parts: global, stack, and heap data. The stack data protection is guaranteed by the compiler, which identifies the set of local variables that may be modified through inter-procedural MOD analysis [10] and then inserts checking code accordingly. Imprecision in compiler analysis may lead to extra variables being checked, but the conservative analysis does not affect correctness.

The global and heap data are protected by the operating system's paging support. At the beginning of a dual-track instance, the system turns off write permission to global and heap data for both tracks. It then installs custom page-fault handlers that record which page has been modified in an access map and re-enables write permission.

B. Run-time Support

1) Program Correctness: To guarantee that our speculative execution is correct, we compare the memory state of the fast and normal tracks at the end of the dual track region. If the fast track reached the same state as the normal track, then the initial state of the next normal track must be correct. Typically, the next normal track was started well before its predecessor finished and know only in hindsight that it is correct.

Memory state comparison is performed by each normal track once it has finished the dual track region. During execution, memory pages are protected so that any write access will trigger a segmentation fault. Both the fast and normal tracks use catch these faults, and record the access in a bit map. Before spawning a normal track, the fast track allocates a shared memory space for two access map so the normal track can compare them upon reaching the end of the dual track region. Once the normal track determines that the write set of the two tracks is identical, is can compare the writes themselves.

In order to compare the memory modifications of the two track, the fast track must provide the normal track with a copy of any changes it has made. At the end of each

dual track region, the fast track evaluates its access map to determine what pages have been modified. Each page flagged in the access map is pushed over a shared pipe, and consumed by the normal track, which then compares the data to its own memory page.

If the two pages are the same then the two tracks have made identical changes to the same memory locations. From that point forward, the execution of the two track will be identical; the normal track is unnecessary and aborts. If there are conflicts in either the write set or the actual data, the normal track aborts the fast track and takes its place.

Because the fast track may spawn multiple normal tracks, which may then run concurrently, each normal track must know when all of its logical predecessors have completed. Before a normal track terminates, it waits on a flag to be set by its predecessor, and then signals its successor when complete. If there is an error in speculation, the normal track uses the same mechanism to lazily terminate normal tracks that are already running.

To ensure that the output of a program running with Fast-Track support is correct, we ensure output is produced only by the known-correct slow-track. Until a normal track has confirmed that it was started correctly (that previous speculation was correct), it buffers all terminal output and file writes. Once the normal track is the oldest, we can be certain that its execution is correct and any output it produces will be the same as the sequential program. The fast track never produces any output to the terminal nor does it write to any regular file.

At the end of any single dual track region we have no way to know whether another dual track region will follow, or whether we will soon reach the end of the program. As a result we must capture every program exit point in the fast track, and wait there for all normal tracks to finish. The signaling mechanism used to order the completion of normal tracks is also used by the fast track in these cases. The final normal track indicates its completeness in the usual way, without knowing whether another normal track follows it.

The fast track knows that the normal tracks are ordered, and can finalize the program's execution once they have all completed their correctness checks. This system allows for the program to exit from within a dual track region; as long as both the fast and normal track reach that exit point they will agree on the result and terminate cleanly. Note that the normal track does not need to reach the end of program execution in order for the fast track to be assured of correct execution.

2) Processor Utilization: For a system with p available processors, the Fast Track system reserves one processor for the fast track (unless it is too fast) and the remaining processors for running normal tracks in parallel.

Activity Control: The system uses $p-1$ tokens to limit the number of active normal tracks. The first $p-1$ normal tracks start without waiting for a token but will finish by passing a token. Each later normal track waits for a token before executing and releases the token after finishing.

Multiple slow-track processes may be waiting for a token at the same moment. We want to activate them in order. When fast track creates a normal track, it pushes the identifier of its floodgate into a *ready queue*, which is also implemented by a pipe. The newly created normal track then waits before its floodgate (by reading from the pipe).

Each time a normal track finishes, it fetches the next floodgate from the ready queue and opens it (by writing to the pipe) to unleash the next waiting process. Since fast track creates normal tracks sequentially, their flood gates follow the same sequence in the ready queue. Therefore, the activation of normal tracks is strictly chronological regardless of the order in which earlier normal tracks finish. Since the system does not require normal tracks to finish in order, it permits normal tracks to have different sizes and to execute on heterogeneous hardware. The steady state of the system is shown by the top diagram in Figure 3. The speed of fast track is the same as the combined speed of $p-1$ normal tracks. When their speeds do not match, the ready queue may become empty or may keep growing, as we will discuss shortly.

In theory we need one floodgate for each normal track. However, when a normal track is activated, its floodgate can be reused by a later normal track. In fact, we need as many flood gates as the maximal number of normal tracks in the ready queue. The size of ready queue is bounded by $2p$ (which we will show when explaining fast-track throttling), we need only $2p$ flood gates. Since any normal track may activate any later normal track, the flood gates need to be visible to all tracks. For p processors, we allocate $2p$ flood gates at the beginning of the program execution.

Fast-track Throttling: The fast track, if unconstrained, may scurry ahead arbitrarily far, which is undesirable for two reasons. First, the resource demand for the ready queue and the waiting normal tracks can be in the worst case proportional to the length of the execution. Second, if fast track has an error, the longer it runs after the error, the more processor time would be wasted on useless speculation. The goal of fast-track throttling is to keep the fast track just far enough ahead to keep processor utilization high, without wasting processor resources.

The protocol of throttling is to stop fast track and give the processor to a normal track, as shown by the middle diagram in Figure 3. When the *next* normal track finishes, it re-activates fast track. The word "next" is critical for two reasons. First, only one normal track should activate fast track when it waits, effectively returning the processor after "borrowing" it. Second, the time of activation must be exact and cannot be one track early or late.

Consider a system with p processors running fast track and $p-1$ normal tracks until the fast track becomes too fast and suspends execution giving the processor to a waiting

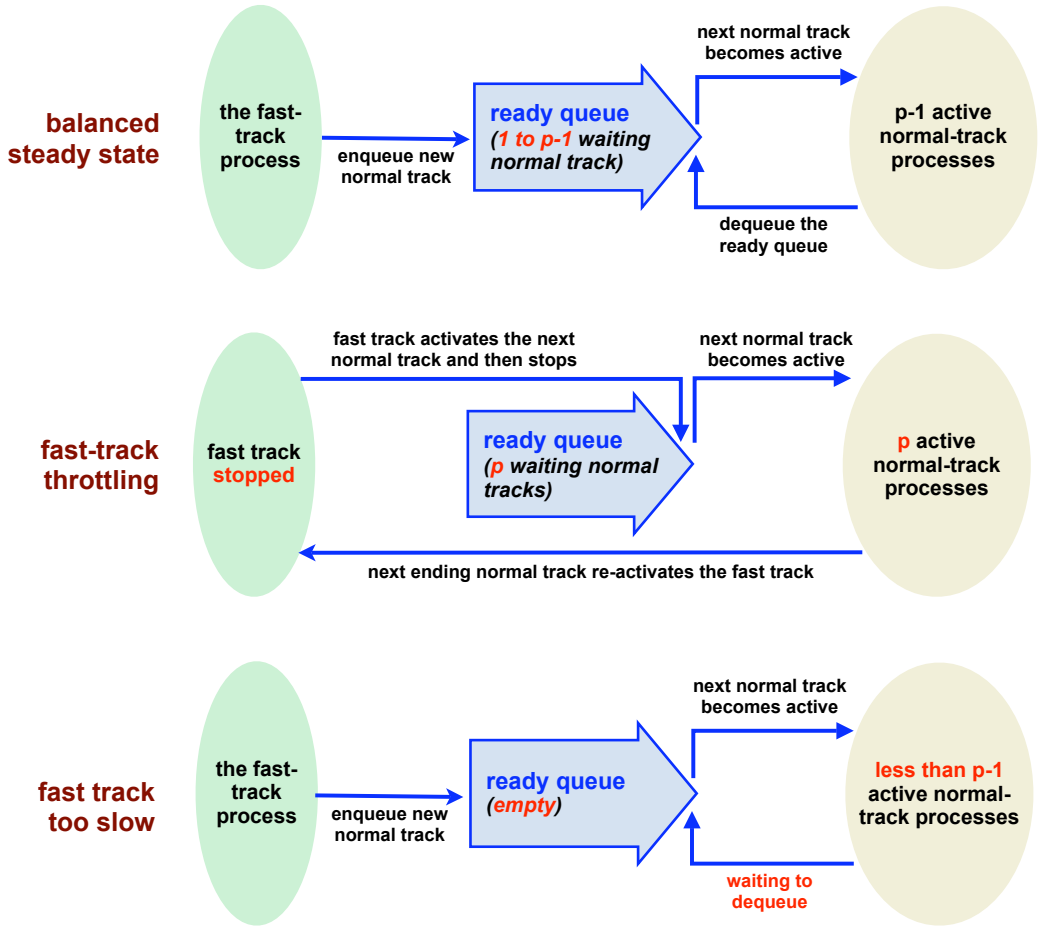


Figure 3. The three states of fast track: balanced steady state, fast-track throttling when it is too fast, and slow-track waiting when fast track is too slow. The system returns to the balanced steady state after fast-track throttling.

normal track. Suppose that three normal tracks finish in the order nt_1, nt_2, and nt_3, and fast track suspends after nt_1 and before nt_2. The proper protocol is for nt_2 to activate fast track so that before and after nt_2 we have p and only p processes running concurrently. Activation before and after nt_2 would lead to less than or more than p processes.

Since the suspension of the fast track needs to be timed exactly with respect to the completion of the normal tracks, we use a critical section for synchronization. The critical section maintains a counter and a flag: `waitlist_length` and `ft_waiting`. The former is the number of normal-track processes waiting in the ready queue. When the number exceeds p, fast track is considered too fast. It would yield its processor by activating the next waiting process in the ready queue, setting the `ft_waiting` flag, and then suspending execution. When a normal track finishes, it enters the critical section and determines which process to activate based on the flag: if `ft_waiting` is on, it activates the fast track and resets the flag; otherwise, it activates the next normal track and updates the value of `waitlist_length`.

A problem arises when there are no slow tracks waiting to start, which can happen when the fast track is too slow. If a normal track waits inside the critical section to start its successor, then the fast track cannot enter to add a new track to the queue. The bottom graph in Figure 3 shows this case, where one or more normal track processes are waiting for fast track to fill the queue. Concurrency control becomes tricky since a normal track may wait for work in the critical section yet block fast track from generating more work, thus creating a deadlock.

The solution is to add a back door to the critical section. When fast track finishes, it checks the size of the ready queue and enters the critical section only if the queue is too long. The check is not synchronized with the normal tracks for simplicity. If we assume that a program can be suspended for an arbitrarily long interval by the operating system after checking the length and before entering the critical section, a deadlock may happen. The simple solution should be adequate since a process should not be suspended long if we run p processes on p processors. Even if deadlock

occurs, a normal track can reset the system and resume the execution. Finally, the deadlock can be removed by adding a semaphore to protect `waitlist_length`. Then the back-door check by fast track is synchronized with normal tracks. Specifically, the fast track acquires the semaphore, checks the length and enters the critical section before releasing the semaphore.

With activity control and fast-track throttling, the available processors are utilized as much as possible. Incomplete utilization happens only due to a lack of parallelism (when the fast track is too slow). When there is enough parallelism, the fast track is constrained to minimize the potentially useless speculative computation.

Resource Contention: We rely on the operating system implementation of copy-on-write, which lets processes share memory pages to which they do not write. 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 a 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. Earlier work has shown that on-line monitoring can effectively adapt memory usage by monitoring the page-fault indicators from Linux [11], [12]. Our test cases have never indicated that memory expansion will be a problem, so we do not consider memory resource further in this paper.

Running two instances of the same program would double demand for off-chip memory bandwidth, which is a limiting factor for modern processors, especially chip multiprocessors. In the worst case if a program is completely memory bandwidth bound, no fast track can reduce the overall memory demand or improve program performance. However, our experience with small and large applications on recent multi-core machines, which we will detail later, is nothing but encouraging. In fast track, the processes originate from the same address space and share read-only data. Their similar access patterns help to prefetch useful data and keep it in cache. For the two large test applications we used, multiple processes in fast track ran almost the same speed as that of a single process. In contrast, running multiple separate instances of a program always degrades the per-process speed.

IV. Analysis

We use the following notation to represent the basic parameters of our system:

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

A. 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 $T(r_i)$. In the following, we assume $T(r_i) = t_c$ for all 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, all fast instances are correct ($\alpha = 1$) and the machine has unlimited resources $p = \infty$. Each time the fast track finishes an instance, a normal track is started. Thus, the active normal tracks form a pipeline, if we consider only dual-track instances (the component $T(E_r)$ in $T(E)$). The first fast instance is verified after t_s. 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. If we use the superscript to indicate the number of processors, the average time and the overall speedup are

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

$$speedup^\infty = \frac{original\ time}{fast\ track\ time} = \frac{nt_c + T(E_u)}{n\bar{t_f^\infty} + T(E_u)}$$

In the steady state $\frac{t_c}{t_f^\infty}$ 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 $t_f^\infty = t_s = (1+q_e)t_c+q_c$, and consequently $speedup = \frac{nt_c+T(E_u)}{n((1+q_e)t_c+q_c)+T(E_u)}$. While this is slower than the original speed ($speedup \leq 1$), the worst-case time is bounded only by the overhead of the system and not by the quality of fast-track implementation (factor x).

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}. This is equivalent to a pipeline flush. Each failure adds a cost of $t_s - t_f^\infty$, so the average time with a success rate α is $(1-\alpha)(t_s - t_f^\infty) + t_f^p$.

We now consider the limited number of processors. For the sake of illustration we assume no fast-track throttling first and will add it 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 (assuming $p-1$ divides n) is

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

When $\alpha < 1$, the cost of restarting has the same effect as in the infinite-processor case. The average time and the overall speedup are

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

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

Finally we consider fast-track throttling. 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, fast-track throttling 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 overall speedup (bounded from below and $n \geq 2$) is as follows, where all the basic factors are modeled.

$$speedup^p = \max(\frac{nt_c + T(E_u)}{nt_s + q_c + T(E_u)}, \frac{nt_c + T(E_u)}{n\bar{t}_f^p + T(E_u)})$$

B. 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, fast-track throttling 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 model simplifies the effect of fast-track system in terms of four parameters, which we have not validated with experiments on a real system. On the other hand, if the four parameters are the main factors, they can be efficiently

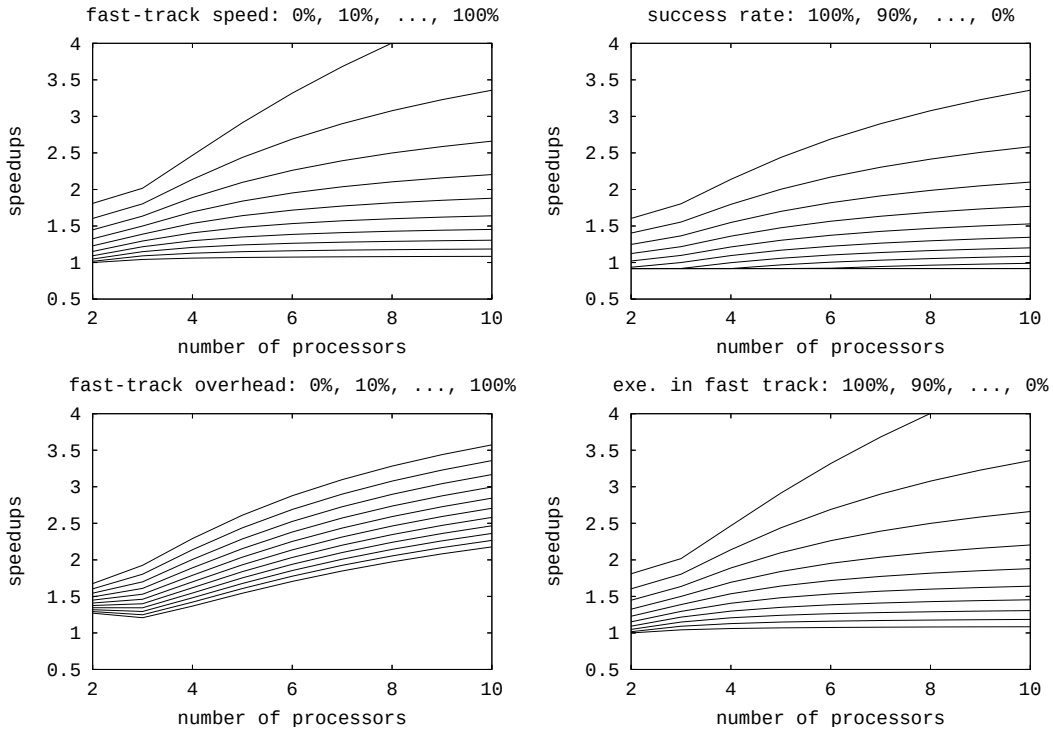


Figure 4. Analytical results of the fast-track system where the speed of the fast track, the success rate, the overhead, and the portion of the program executed in dual-track regions vary. The order of the parameters in the title in each graph corresponds to the top-down order of the curves in the graph.

monitored at run time, and 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.

V. Uses of Fast Track

A. Unsafe Program Optimization

In general, the fast code can be any optimization inserted by either a compiler or a programmer; for example memoization, unsafe compiler optimizations or manual program tuning. The performance of the system is guaranteed against slow or incorrect fast track implementations. The programmer can also specify two alternative implementations and let the system dynamically select the faster one. We discuss four types of optimizations 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 [13] show dramatic performance benefits, for example to speed up table look-up in transcoding programs [14]. Memoization be conservative about side-effects and is unsuitable for generic use in C/C++ program. With fast track, memoization does not have to be correct in all cases and therefore can be more aggressively used to optimize the common case.

Semantic optimization Often different implementations exist at different levels, from basic data structures such as hash tables to the choice of algorithms and their parameters. A given implementation is often more general than necessary for a program. The current programming languages do not provide a general interface for a user to experiment with an unsafely simplified algorithm or to dynamically select the best among alternative solutions.

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 to verify in terms of correctness or ensuring good performance on other inputs. As a result, many creative solutions go unused because an automatic compiler cannot possibly achieve them.

Program monitoring and safety checking A program can be instrumented to collect run-time statistics such as frequently executed instructions or accessed data, or to report memory leaks or out-of-bound memory access. The normal, uninstrumented code can serve as the fast track, and the instrumented code can run in parallel to reduce the monitoring overhead.

B. Parallel Memory-Safety Checking of Sequential Code

To test fast track on real-world applications, we use it to parallelize a memory-safety checking tool called *Mudflap*. Mudflap is bundled with GCC and widely used. It adds checks for array range (over or under flow) and validity of pointer dereferences. Common library routines that perform string manipulation or direct memory access are also guarded. Checks are inserted at compile time and require that a runtime library be linked into the program. The Mudflap compilation has two passes: *memory recording*, which tracks all memory allocation by inserting `__mf_register` and `__mf_unregister` calls, and *access checking*, which monitors all memory access by inserting `__mf_check` calls and inlined operations. The recording cost is proportional to the frequency of data allocation and recollection, and the checking cost is proportional to the frequency of data access.

To fast track the Mudflap checking system we introduced a new compiler pass that clones all functions in the program. The second Mudflap pass is instructed to ignore the clones while instrumenting the program. The result is an executable with the original code fully checked while the clone just records data allocation and free. The instrumentation of the clones is necessary to maintain the same allocation and meta data of memory as those of the original code. We create a Fast Track version of programs by using the fully checked version of the program to verify the memory safety of the unchecked fast track.

VI. Experimental Results

A. Implementation and Experimental Setup

We have implemented the compiler support in Gcc 4.0.1. The main transformation is converting global variables to use dynamic allocation, so the run-time support can track them and set appropriate access 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. The indirection causes only marginal slowdown because most global-variable accesses have been removed or converted to (virtual) register access by earlier optimization passes.

B. Parallel Memory Safety Checking

We have generated a Fast Track version of Mudflap for the C-language benchmarks *hmmer*, *mcf*, and *sjeng* from the SPEC 2006 suite, and *bzip2* from the SPEC 2000 suite. They represent computations in pattern matching, mathematical optimization, chess playing, and data compression. The number of program lines ranges from a few thousand to over ten thousand. We expect to see similar effect in other programs.

All four programs show significant improvement, up to a factor of 2.7 for *Bzip2*, 7.1 for *Hmmer*, and 2.2 for *Mcf* and *Sjeng*. The factors affecting the parallel performance are the coverage of fast track and the relative speed of fast track as discussed in our analytical model. For lack of space we omit detailed discussion of these benchmarks. One factor not tested here is the overhead of correctness checking and error recovery.

The running times with and without Mudflap overhead, as given in the captions in Figure 5, show that memory-safety checking delays the execution by factors of 5.4, 15.0, 8.6, and 67.4. By utilizing multiple processors, fast track reduces the relay to factors of 2.0, 2.1, 3.7, and 28.8, which are more tolerable for long-running programs.

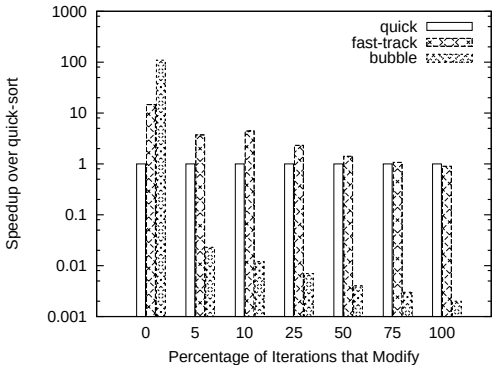


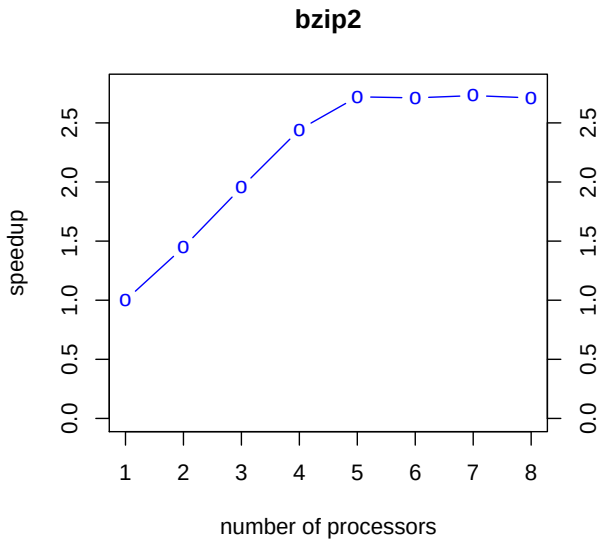
Figure 6. Sorting time with quick sort, bubble sort or the Fast-Track of both

C. Results of Sort and Search Tests

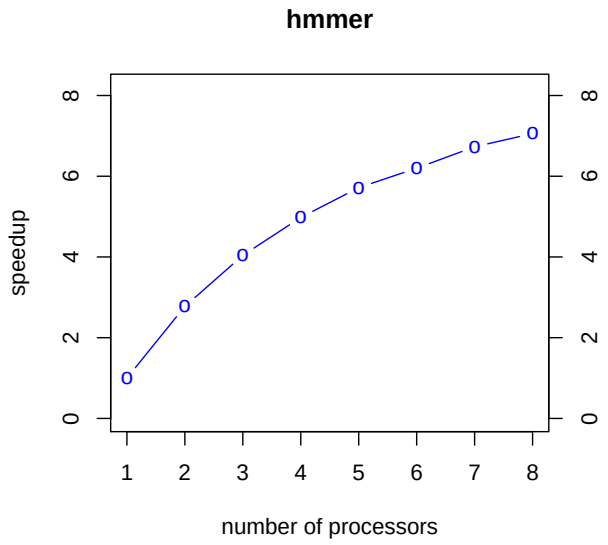
To use Fast-Track to support unsafe optimization we implemented two tests and measured performance on a machine with two Intel dual-core Xeon 3 GHz processors. The first is a simple sorting program. We repeatedly sort an array of 10,000 elements that is randomized in some fraction of cases. We sort the array with either a short-circuited bubble sort, quick sort, or by running both in a Fast-Track environment. The results of these tests are shown in Table 6. The quick sort performs consistently and largely independent of the input array. We see that the bubble sort quickly detects when array is sorted, but in other cases performs poorly.

The Fast-Track approach is able to out-perform either of the individual sorting algorithms. These results illustrate the utility of Fast-Track in cases where both solutions are correct, but it is unknown which is actually faster. In cases where the array is always sorted or always unsorted, the overhead of using Fast-Track will cause it to lose out. While we do not mean to suggest that Fast-Track is a better solution than explicitly parallel sort, we hope this example motivates the utility of automatically selected the faster of multiple sequential approaches.

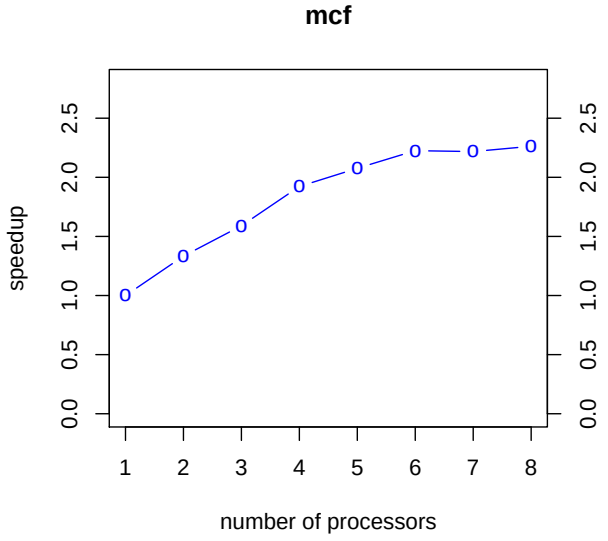
The second program is a simple search, which we use to test the effect of various parameters. The basic algorithm is given in Figure 7. The program repeatedly updates some



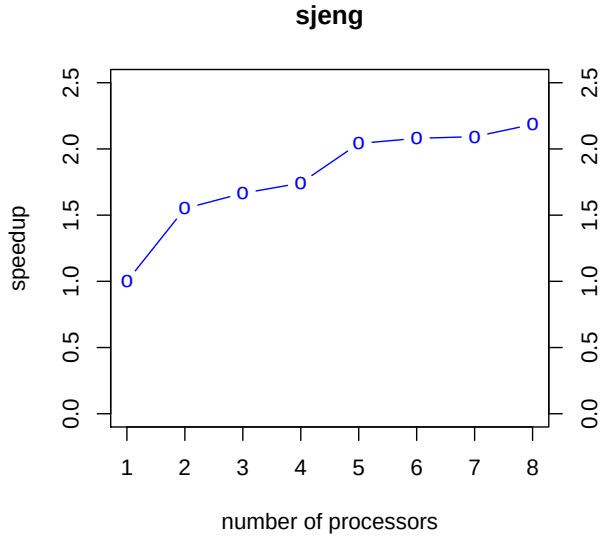
(a) The checking time of *Bzip2* is reduced from 24.5 seconds to 9.0 seconds. The base running time, without memory safety checking, is 4.5 seconds.



(b) The checking time of *Hmmer* is reduced from 235 seconds to 33.2 seconds. The base running time is 15.6 seconds.



(c) Fast track reduces the checking time of *Mcf* from 56 seconds to 24.8 seconds. The base running time is 6.7 seconds.



(d) Fast track reduces the checking time of *Sjeng* from 33.7 seconds to 14.4 seconds. The base running time is 0.5 seconds.

Figure 5. The effect of fast-track Mudflap on one SPEC 2000 and three SPEC 2006 benchmarks.

elements of a vector and finds the 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. Figure 8(a) shows the speedups over the base sequential execution, which takes 3.33 seconds on a 4-CPU machine. We time each run three times. The variation between times of different runs is always smaller than 1 millisecond.

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

```
initialize N numbers in a vector
for T iterations do
  normal track
   apply computation to each number
   identify the largest sample
   store it in the output vector
  fast track
   sample S random vector elements
   apply computation to each sample
   identify the largest sample
   store it in the output vector
  end dual-track

  modify N1 random elements
  repeat the loop
end loop
output the result vector
```

Figure 7. Pseudo code of the synthetic search program

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

(a) Effect of fast-track success rates on 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

(b) The speedup due to fast-track tuning of the synthetic benchmark

Figure 8. Fast track on the synthetic benchmark

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. 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 Figure 8(b) show interesting trade-offs 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 Figure 8(a).

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.

VII. Related work

Recently three software systems use multi-processors for parallelized program profiling and correctness checking. All use heavyweight processes, and all are based on Pin, a dynamic binary rewriting tool [15]. Shallow profiling creates replica processes and instruments them to collect and measure a sample segment of execution [16]. SuperPin uses a signature-checking scheme and strives to divide the complete instrumented execution into time slices and executing them in parallel [17]. Although fully automatic, SuperPin is "not foolproof" [17] since in theory the slices may overlap or leave holes in their coverage. Speck uses Pin with custom OS support to divide an execution into epochs, replay each epoch (including system calls), and check for security problems [18]. An earlier system used a separate shallow process for memory-safety checking [19]. These systems automatically analyze the full execution, but they are not designed for speculative optimization as fast track is with the programming interface for selecting program regions, the ability for a checking process to roll back the computation from the last correct point, and the throttling mechanism for minimizing useless speculation. Shallow profiling and SuperPin are designed for parallel profiling and have no need for speculation. Speck speculates but aborts the execution upon a security error.

Fast track is not designed for fully automatic program analysis, although it can be programmed so with some manual effort as we have shown with Mudflap. It guarantees the complete and unique coverage in parallel checking. The programming interface allows selective checking, which is useful when checking programs that contain unrecoverable operations on conventional OS. Fast track is program level, so it requires source code and cannot instrument externally or dynamically linked libraries. On the other hand, it benefits from full compiler optimization across original and instrumented code. This is especially useful for curbing the high cost of memory-safety checking. For example it takes a minute for GCC to optimize the instrumented code of *sjeng*, and the optimized code runs over 20% faster in typical cases.

Fast track is closely related to several ideas explored in hardware research. One is thread-level speculative parallelization, which divides sequential computation into parallel tasks while preserving their dependences. The dependences may be preserved by stalling a parallel thread as in the Superthreaded architecture [20] or by extracting dependent computations through code distilling [6], compiler scheduling for reducing critical forwarding path [7], and compiler generation of pre-computation slices [8]. These techniques

aim to only reorganize the original implementation rather than to support any type of alternative implementation. Fast track is not fully automatic, but it is programmable and can be used by both automatic tools and manual solutions. The run-time system checks correctness differently. The previous hardware techniques check dependences or live-in values, while fast track checks result values or some user-defined criterion.

Another related idea used in hardware systems is to extract a fast version of sequential code to run ahead while the original computation follows. It is used to reduce memory load latency (run-ahead code generated either in hardware [21] or software [22]) and recently to reduce hardware design complexity [23]. Unlike fast track, run-ahead threads accelerate rather than parallelize the execution of sequential code. A third, more recent idea is speculative optimization at fine granularity, which does not yet make use of multiple processors [9]. All of these techniques require modification to existing hardware. Similar special hardware support has been used to parallelize program analysis such as basic block profiling, memory checking [24], data watch-points [25], and recently Mudflap memory-safety checking [26].

Our system uses software speculative optimization at the source level and can run on commodity hardware. The design and implementation for coordinating parallel processes is completely different. Being programmable and implemented in software, our system is most suitable for selecting and optimizing coarse-grain tasks that are too large for loop-level fine-grained threads considered in the hardware-based studies. Coarse-grain tasks are more likely to have unpredictable sizes. We have addressed these issues in a concurrent run-time control system implemented on existing hardware.

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 [27]–[29]. Run-time sampling based on data [30] or code [31], [32] are efficient but does not monitoring all program accesses. For correctness, fast track uses page-based data monitoring, which has been used for supporting distributed shared memory [33], [34] and other purposes including race detection [28].

VIII. Summary

We have described fast track, a new system that supports unsafely optimized code and can also be used to off-loaded safety checking and other program analysis. The key features of the systems include a programmable interface, compiler support, and a concurrent run-time system that includes correctness checking, output buffering, activity control, and fast-track throttling. We have implemented a complete system including compiler and run-time support and used the system to parallelize memory safety checking for sequential code, reducing the overhead by up to a factor of seven for four large size applications running on a multi-core PC.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 fast-track regions. We have used our system and model in speculatively optimizing a sorting and a search program. Both analytical and empirical results suggest that fast track is effective at exploiting today's multi-processors for improving program speed and safety.

Acknowledgement: This research is supported in part by the National Science Foundation (Contract No. CNS-0834566, CNS-0720796, CNS-0509270), an IBM CAS Fellowship, two grants from Microsoft Research, and an equipment donation from IBM.The last author Zhang participated in the study when he was a graduate student at Rochester. The work was presented earlier as a PACT 2007 poster and at the CDP workshop at CASCON 2008. The authors wish to thank comments from Michael Scott and Michael Huang and the feedback from the anonymous reviewers in particular the suggestion to improve the presentation of the analytical model.

References

[1] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[2] M. Gupta and R. Nim, "Techniques for run-time parallelization of loops," in *Proceedings of SC'98*, 1998.

[3] M. H. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 562–576, 2005.

[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior-oriented parallelization," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, USA, 2007.

[5] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, "Copy or discard execution model for speculative parallelization on multicores," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 2008.

[6] C. B. Zilles and G. S. Sohi, "Master/slave speculative parallelization." in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 2002, pp. 85–96.

[7] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler optimization of scalar value communication between speculative threads," in *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, 2002, pp. 171–183.

[8] C. Quinones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2005, pp. 269–279.

[9] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. B. Zilles, "Hardware atomicity for reliable software speculation," in *Proceedings of the International Symposium on Computer Architecture*, 2007, pp. 174–185.

[10] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.

[11] 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.

[12] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski, "Isla vista heap sizing: Using feedback to avoid paging," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2007, pp. 325–340.

[13] D. Michie, "Memo functions and machine learning," *Nature*, vol. 218, pp. 19–22, 1968.

[14] Y. Ding and Z. Li, "A compiler scheme for reusing intermediate computation results." in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[15] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.

[16] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri, "Shadow profiling: Hiding instrumentation costs with parallelism," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2007, pp. 198–208.

[17] S. Wallace and K. Hazelwood, "Superpin: Parallelizing dynamic instrumentation for real-time performance," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2007, pp. 209–220.

[18] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, 2008, pp. 308–318.

[19] H. Patil and C. Fischer, "Efcient run-time monitoring using shadow processing," 1995, presented at AADEBUG'95.

[20] J.-Y. Tsai, Z. Jiang, and P.-C. Yew, "Compiler techniques for the superthreaded architectures," *International Journal of Parallel Programming*, vol. 27, no. 1, pp. 1–19, 1999.

[21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance." in *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, 2000, pp. 257–268.

[22] S.-W. Liao, P. H. Wang, H. Wang, J. P. Shen, G. Hoflehner, and D. M. Lavery, "Post-pass binary adaptation for software-based speculative precomputation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002, pp. 117–128.

[23] A. Garg and M. Huang, "A performance-correctness explicitly-decoupled architecture," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 2008.

[24] J. T. Oplinger and M. S. Lam, "Enhancing software reliability with speculative threads," in *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, 2002, pp. 184–196.

[25] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, "iwatcher: Efficient architectural support for software debugging," in *Proceedings of the International Symposium on Computer Architecture*, 2004, pp. 224–237.

[26] S. Lee and J. Tuck, "Parallelizing Mudflap using thread-level speculation on a CMP," 2008, presented at the Workshop on the Parallel Execution of Sequential Programs on Multi-core Architecture, co-located with ISCA.

[27] J. Mellor-Crummey, "Compile-time support for efficient data race detection in shared memory parallel programs," Rice University, Tech. Rep. CRPC-TR92232, September 1992.

[28] D. Perkovic and P. J. Keleher, "A protocol-centric approach to on-the-fly race detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 10, pp. 1058–1072, 2000.

[29] 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*, June 1993.

[30] 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.

[31] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[32] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[33] K. Li, "Shared virtual memory on loosely coupled multiprocessors," Ph.D. dissertation, Dept. of Computer Science, Yale University, New Haven, CT, Sep. 1986.

[34] 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.