System Support for Implicitly Parallel Programming

Matthew I. Frank
Coordinated Science Laboratory
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Abstract

Implicit parallelization involves developing parallel algorithms and applications in environments that provide sequential semantics, *e.g.*, the C programming language. System tools convert the parallel algorithms into a set of threads partitioned appropriately for a particular parallel machine organization. The resulting parallel programs are easier and faster to develop, debug and maintain, because the programmer can request a meaningful and well defined program state at any point of execution.

The contribution of this paper is a case study of a video encoding application. We show that error checking code, code reuse, and variable scoping interfere with parallelization. We suggest that system tools must perform reactive and speculative transformations if they are to reduce this tension between application robustness and parallelization.

1 Introduction

Processor performance gains are now due almost entirely to the incorporation of ever larger numbers of cores onto commodity chips. Multi-core and multi-threaded designs deliver peak instruction throughput that scales with Moore's law and they provide better power/performance tradeoffs than monolithic superscalar designs given the same number of transistors. Additionally, multi-core designs leverage replication to amortize design and verification costs. Semiconductor manufacturers find multi-core processors easier and cheaper to design than monolithic designs of similar size or peak performance.

While multi-core designs allow peak performance to track Moore's law, they introduce new challenges for software developers. Multi-core and multi-threaded designs require multi-threaded software. While there may be domains where finding adequate threads to run concurrently is "easy," (perhaps in web services, database query processing, scientific computing, graphics, gaming, or signal processing), multi-threaded programs, in general, take longer to develop than sequential programs with equivalent functionality. Multi-threaded programs are non-deterministic, making it difficult to reproduce bugs, and have more bugs (race conditions, livelocks, deadlocks) than sequential programs with the same functionality. When coupled with the scope of code change required to achieve desired execution throughput, as dictated by Amdahl's law, the increased time for testing, debugging and verification make explicitly multi-threaded models unattractive.

This creates a new set of challenges for the semiconductor and software industries. There will be few new "killer apps" to take advantage of the new computing power until an entire generation of millions of programmers learns to write programs that can leverage the parallelism on these new chips. Without observable increases in functionality, there will be little reason for consumers to invest in new hardware or software.

Motivated by previous work in automatically parallelizing compilers and speculative multi-threading, I propose a set of runtime tools that will help programmers express parallel programs in a way that is more natural, allowing them to use the abstraction and information hiding tools that they need in order to deliver robust programs in a timely manner. The programmer writes an *implicitly parallel* program. That is, the programmer designs a parallel algorithm, but expresses it in a conventional sequential programming language. The programmer annotates the program to indicate where the tools should look for parallelism. The program is compiled and run on the multi-core system. The compiler and run-time system are responsible for actually transforming the program to express the latent parallelism.

Building tools that transform a parallel algorithm into an explicitly parallel representation is difficult because there is no silver bullet that will solve the problem. The work proposed here builds on forty years of work by the computer systems community on automatic parallelization, and twenty years of work on speculative parallelization. Programmers, for software

Center for Reliable and High-Performance Computing Technical Report CRHC-07-06, October 8, 2007.

engineering and information hiding reasons, prefer to combine together sequential portions of code with potentially data-parallel code. Thus the system must support generalized methods of loop distribution to "tease apart" the sequential and parallel portions of code. In addition, programmers, again for software engineering reasons, often need to include error checking code, special case code, and optional but rarely used features. Many of these application features introduce true sequential dependencies that defeat conservative compiler analysis. Thus the system must support approximate forms of analysis and transformations that can be undone by the runtime system.

The system consists of three main components. The first is a *coarse-grain checkpoint repair* system. This allows the programmer to write code that contains error, and special case, handling code that might otherwise impede the expression of parallelism in the common case. A runtime *distiller*, directed by feedback about the common paths in the program, then extracts a speculative version of the program where uncommon paths are replaced by traps to recovery code. This removes any error handling code from the critical path, and also specializes the code for the particular command line options and inputs that it was invoked with. Finally an *on-line queue converter* takes the streamlined parallel loop and performs scalar expansion and loop distribution. This exposes the latent parallelism in the loop by extracting any remaining sequential code into its own loops, to be run separately from the parallel code. The resulting implicitly parallel programming environment allows programmers to express parallel algorithms, but to do so in a deterministic, reproducible and portable way [50].

2 The Conflict between Maintainability and Parallelism

The problem of parallelizing a task seems, unfortunately, to conflict with the primary goals of software engineering, including minimizing the time to deployment of a robust product [66]. It is already difficult to engineer robust, reusable and maintainable modules. Parallelism makes the problem harder. In this section I consider several examples that demonstrate these conflicts to motivate the need for implicit parallelization tools.

Consider, for example, the H.264 video encoding reference implementation included in the SPEC 2006 benchmark suite. H.264 video encoding would seem, at first glance, to be an application that ought to be easy to parallelize. In fact, it may be relatively easy to produce a parallelized kernel [61].[1] Unfortunately, even though the implementation included in SPEC 2006 is only about 50,000 lines long (i.e., small compared to any significant application), we discovered multiple places where we had to dramatically change the software structure or algorithm in order to create a version that could run in parallel [87].

Briefly, H.264 is a recent international video encoding standard that is used for high quality digital television. It achieves both good picture quality and excellent compression by exploiting the fact that portions of background images tend to be shared between frames, giving the video stream redundancy from frame to frame.

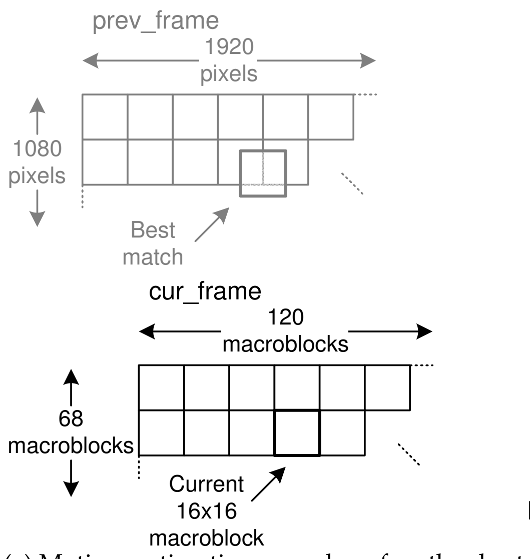
The algorithm divides each frame of the movie into 16×16 *macroblocks*. As shown in Figure 1, for each macroblock in a frame, the application successively performs (a) motion estimation (the most compute intensive step), followed by (b) frame-differencing, (c) a discrete cosine transform (DCT), (d) quantization of the resulting transform (the lossy step), and (e) bitstream encoding (a task that seems to be fundamentally sequential). In order to avoid accumulating errors at the decoder output, the encoder keeps track of the picture that will be reconstructed by the decoder. That is, the encoder runs the decoder on the encoded result of the current frame and uses it as the next previous frame when encoding the next frame. The decoder's work is reconstructed by (f) dequantization and (g) running an inverse discrete cosine transform (IDCT).

2.1 Error checking

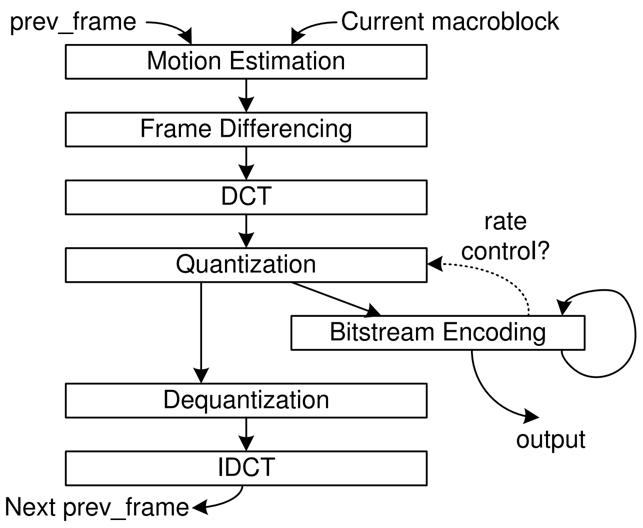
Of course, good software engineers check for errors religiously. Even errors that can "not possibly happen" ought to be checked for, because the "proof" of impossibility often depends on invariants that are invalidated in a future revision of the code. In the case of the H.264 reference implementation, for example, the main loop calls a function `SetModesAndRefframeForBlocks()`, that checks for, and handles, invalid arguments. Of course, the common case is (hopefully) an input with no errors, so the validity check should rarely fail.

The exception handlers that restore an application's valid state, allowing it to continue operating after an error is detected, present a barrier to parallelization. The exception handlers usually restore a valid state by modifying shared data structures. The shared data structure modifications create interdependences with

[1] A *kernel* is the innermost loop of an application or algorithm, stripped of all error checking and handling, options, generalizations, and information hiding.



(a) Motion estimation searches for the best matching 16×16 block in the previous frame.



(b) The body of the H.264 application main loop is applied to each macroblock in the current frame.

Figure 1: The H.264 video encoding reference implementation from the Spec 2006 benchmark suite. Each video frame is made of 8,160 16×16 pixel *macroblocks.* Frames are processed sequentially because the computationally intensive motion estimation step needs to search the previous frame for a close match to the current macroblock. Except for the bitstream encoding stage, the macroblocks within a frame can be processed in parallel.

the rest of the application, forcing conservative compiler transformations to serialize the code. So the parallelization system must both handle special cases like these, and yet provide parallelism in the common case that there is no error.

In the ILP domain problems like these have been effectively attacked with checkpointing and speculation. Fisher's Trace Scheduler [34] used profiling to select likely paths (traces) through the code, and then speculatively scheduled those paths assuming that none of the intervening branches would leave the path. Since then, numerous ILP techniques have successfully used speculative optimizations, both in software [48, 16, 88, 73] and in hardware [49, 80, 106]. The key is to use checkpointing to implement precise exceptions, *predict* that certain invariants will hold, and raise an exception if the prediction turns out to be incorrect.

For the implicit parallelization problem I propose to use a *coarse-grain* checkpoint repair mechanism to provide a form of precise exception handling during parallel execution. With the assistance of the run-time compiler, system state is checkpointed at regular intervals during parallel execution. Error and special case handling code that rarely executes is identified using feedback from previous runs of the program. The runtime compiler rewrites error and special case code that might run during parallel execution, so that they will raise exceptions that (a) force rollback to the most recent checkpoint and (b) roll forward on the sequential version of the code until the special case code is executed, and then (c) checkpoint and resume parallel execution. The load-time compilation and runtime-system support required for this is discussed in Section 3.1.

2.2 Command-line Parameters

The H.264 reference implementation includes many command line parameters so that the encoder can exercise various options of the standard. For example, the user of the application may choose whether or not to turn on *rate control*. If rate control is turned on then the bitstream encoder monitors the compression rate and may adjust the parameters to be used by the quantization stage on the subsequent frame. If rate control is off, the quantization stage can run ahead of the bitstream encoding stage. If rate control is on, the quantization stage needs to run in lock-step with the bitstream encoding stage. Thus, how the program is restructured for parallelization depends on how this command line parameter is set.

In both cases, with rate control on or off, the rest of the encoding algorithm is the same. So that the rest of the code may be reused (rather than, for example, cut and pasted into two different files for the two different options), the tests for the rate control option are embedded into the main body of the code. As the number of options to a program grows, this form of code reuse becomes increasingly critical: while there are exponentially growing possible dynamic paths through the code, options allow the static amount of code that must be maintained to grow considerably more slowly.

We are attacking this problem using *feedback-directed program distillation*. When the program is loaded the run-time compiler makes a best guess at the common path through the parallel code based on statistics from previous runs. Paths that are deemed uncommon are rewritten, like special case code, to raise exceptions. As the program runs, if a particular exception path is taken repeatedly (if, for example, a run-time parameter is requested that was not requested in previous runs), the run-time compiler is reinvoked to remove the exception and to try to reparallelize the code based on the new common-case paths. The required profiling and runtime-system support for feedback-directed program distillation is described in Section 3.2.

2.3 Local Variables

The H.264 main loop, as written, is not actually parallel. There is a loop carried dependence from each iteration of the loop to the next through the bitstream encoding step. The bitstream encoder used in H.264, as with most variable rate and lossless predictive encoders, needs to maintain some state about the encoding that it has already done, and change that state as it outputs each encoded macroblock. Because the encoding of each macroblock depends on the encodings performed for the previous macroblocks, the bitstream encoding for all the macroblocks within a frame must be performed sequentially.[2]

When designing sequential code, programmers often follow the tactic of placing some sequential work inside an otherwise parallelizable loop because doing so makes the code easier to maintain. In this case, with the bitstream encoding step inside the loop the quantized macroblock can be stored in a variable declared local to the scope of the loop body, rather than in a variable exposed to other parts of the code. If the bitstream encoding is moved outside the loop (as is required for parallelization of the rest of the loop) then all the quantized macroblocks must be stored in a queue data structure.

Since the C programming language doesn't natively provide a queue data type, code for such a data structure would need to be written, tested and maintained. In addition, because of the rate control flag, discussed above, if the bitstream encoder is queueing its data then the quantizer needs to work from a queue as well. Finally, if the quantizer is working from a queue, the dequantization phase also needs to be moved out of the loop. To parallelize the program, the main body of the program needs to be completely rewritten.

Streaming library frameworks and programming languages have been designed to assist in this kind of rewriting [17, 56, 40, 25, 42], elevating the notion of communication queues to a program structuring technique. This conversion of variables into queues is both fundamental to parallelization and comes at a cost. Scalar values that were previously communicated through registers are now communicated through memory references to queue data structures. Thus, one of the main objectives of compilers for streaming languages is to transform the stream communication back to register communication [59].

We find it preferable to leverage the coarse-grain checkpointing and run-time recompilation tools, introduced above, to implement *on-demand* scalar expansion and loop distribution. *Scalar expansion* is the process of converting a particular scalar variable in a program to *dynamic* single assignment form [58, 24, 33, 53]. *Loop distribution*, or loop fission, is the process of turning one loop, containing both parallelizable and sequential statements, into multiple loops each containing either just parallelizable or just sequential statements [58, 54, 47]. I call the combination of scalar expansion and loop distribution *scalar queue conversion* [36]. This process is described in more detail in Section 3.3.

3 Background and Overview

The goal of the work proposed here is to design a set of implicit parallelization tools that help alleviate the tension, described in the previous section, between the desire to keep software robust, reusable, testable and maintainable, and the transformations required to actually expose the parallelism in the code. An overview of the proposed system support is shown in Figure 2. The programmer writes implicitly parallel code (parallel algorithms in a sequential programming language). The programmer annotates the code with directives that tell the runtime system which portions of the code it should try to parallelize. The system compiler generates a traditional sequential binary from the code, and passes the programmer annotations as hint instructions in the binary.

The first time the code runs, the run-time distiller makes a best guess at which paths through the code

[2]One can also parallelize bitstream encoding of different *frames*, at the cost of some loss in compression, by clearing the prediction tables between frames. The more compute-intensive motion estimation phase, however, needs to be parallelized at the macroblock level, so parallelizing the bitstream encoding stage would also require the kind of restructuring discussed below.

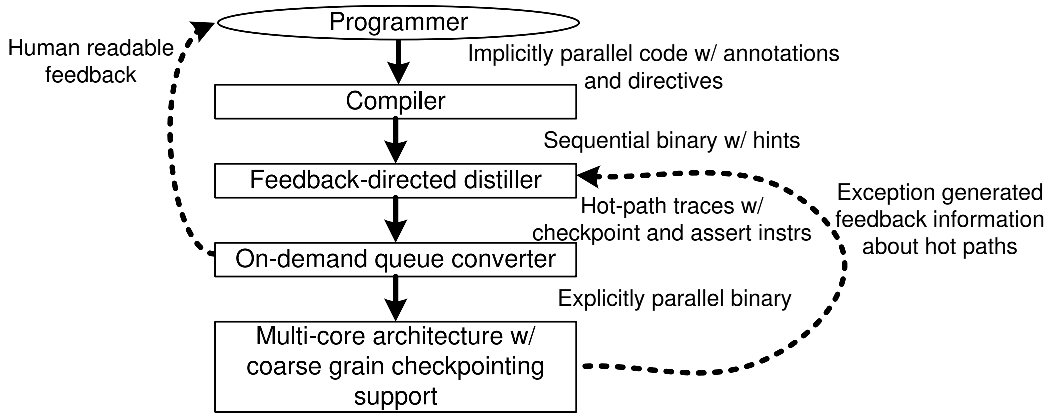

Programmer
Human readable feedback
Implicitly parallel code w/ annotations and directives
Compiler
Sequential binary w/ hints
Feedback-directed distiller
Hot-path traces w/ checkpoint and assert instrs
Exception generated feedback information about hot paths
On-demand queue converter
Explicitly parallel binary
Multi-core architecture w/ coarse grain checkpointing support

Figure 2: Proposed software tool chain.

are to be executed, and generates *distilled* code for the programmer-annotated sections. The distilled code is a second version of the code that contains checkpoint instructions that will execute at regular intervals and then replaces cold-path code with special trap instructions that, if ever executed, roll machine state back to the most recent checkpoint, and then roll forward with the original, sequential version of the code.

The distilled code is then passed to the on-demand queue converter, which performs queue conversion on the distilled code, producing a sequence of parallel and sequential loops that communicate through queues. The resulting code is then run on the multi-core architecture. In the common case it is hoped that very few trap instructions from cold-path code will cause rollbacks. The system collects statistics about cold-path traps that cause rollbacks. If rollbacks occur at a significant rate the distiller is reinvoked to choose a new set of hot paths, and the cycle iterates.

3.1 Coarse-grain Checkpointing and State Repair

Checkpointing across windows of several hundred instructions can be achieved microarchitecturally by checkpointing registers only at points likely to require rollback, and queueing speculative stores until commit [49, 105, 65, 3]. Checkpointing across larger windows can be achieved, for example, by updating memory in place, and then keeping a log, in virtual memory, of the previous contents of each memory location overwritten. Wu et al [104] used an idea like this to support multiprocessor error recovery. More recent examples of logging to support multiprocessor error recovery [82, 91], use the directory controller to log the previous contents to virtual memory in the (rare) case that an error occurs and rollback is required.

The Software UnDo System (SUDS) used update in place and a log in virtual memory to support speculative parallelization [37, 35, 36]. Similar ideas have been proposed recently for virtualizing transactional memory support [81, 5, 83]. The LogTM system [71], in particular, uses in-place updates and history logging to optimize the common case in transactional memory.

If one follows the path of supporting coarse-grain checkpointing at the directory controller, a*checkpoint* instruction becomes a directive to (a) save the currently live registers to the stack frame, (b) store the program counter of the exception handling routine corresponding to this checkpoint in a well-known location, (c) reset the history buffers from the previous checkpoint and (d) clean all the caches, ensuring that all cache lines are in state shared.

The first time any cache asks for exclusive access to a line after the checkpoint, the directory controller will save a clean copy of that cache line in a history buffer. Storage for the history buffer is made of physical DRAM pages that the operating system has allocated to the directory controllers [104, 81, 91, 5, 83, 71]. The directory controller then sets a bit in the state for that line that says that it has been logged.

As program execution rolls forward each thread of execution commits store instructions to memory, as they normally would. If the next checkpoint is reached without requiring a rollback then the caches are again cleaned, the directory controllers unset the logged bit on the cache lines that they control, and the history buffers are emptied, inexpensively, by resetting head and tail pointers.

A rollback is initiated by a cold-path trap instruction, and is handled, in software, by the operating system. The processors involved in the computation are all interrupted. Then each processor rolls back the log associated with one of the directory controllers. This is effected by copying each cache-line copy from the log over the corresponding (incorrectly written) line in main memory.

Note that the checkpointing and state repair system specifically does *not* perform any memory renaming. This is because previous work we have done in this domain has shown that memory renaming is rarely necessary [95, 96]. In cases where memory renaming is necessary, simple tricks can eliminate the need to do it dynamically. For example, most store-after-store dependences occur because the traditional stack-based frame allocation policy leads adjacent procedure calls to use the same stack memory locations for completely different values. If one uses a freelist-based frame allocation policy [92], instead of a stack-based policy, these dependences are eliminated [76]. Similarly, in languages with nested variable scoping (e.g., C, C++, C#, Java) arrays and structs that are private to a loop iteration can be declared in the scope of the loop body [36], eliminating even the need for array privatization analyses [33, 62, 67, 100].

3.2 Distiller

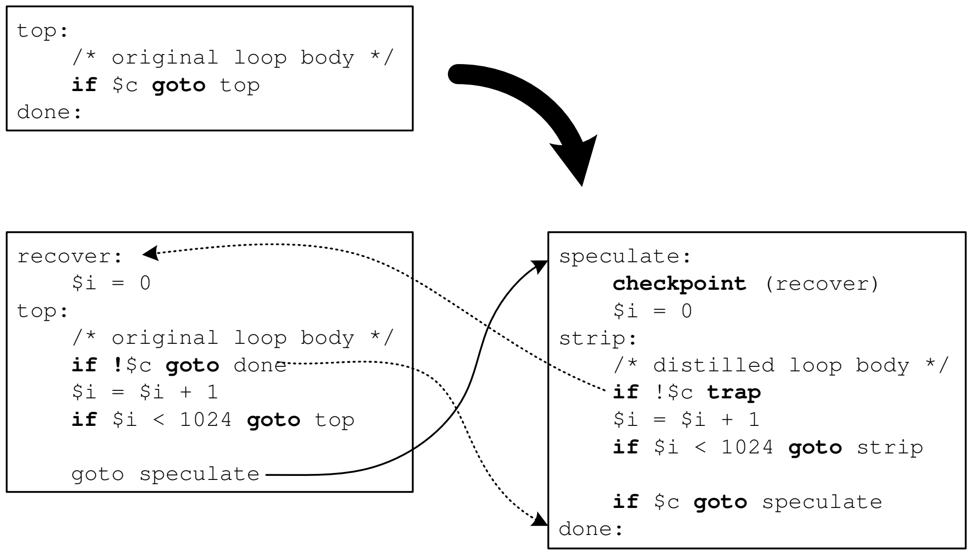
The distiller stage can be viewed as a feedback-directed program specializer [41] that leverages checkpointing to permit optimistic, and speculative, optimizations. The design of the distiller stage is influenced by trace scheduling compilers [34, 48], the main difference here being that we are proposing to use the technique to enable turning loop iterations into threads rather than parallelizing across individual instructions. While the original trace scheduling compilers used feedback from profile runs, more recent versions have been online compilers that can make use of feedback from the currently running program [9, 31, 10, 68, 30, 26, 102].

The checkpointing interface, discussed above, that is leveraged by the distiller stage is most directly influenced by the rePLay interface [80, 32]. RePLay introduced two primitives. The first indicates to the microarchitecture where it should commit the previous checkpoint and start a new one. The second, an *assertion* instruction tests a condition, and rolls back to the most recent checkpoint if the condition fails. When an assertion fails, execution rolls back to the checkpoint and then moves forward on the original (unoptimized) code. While rePLay's trace optimizer was implemented in hardware, a similar interface has been leveraged more recently by the runtime compiler in a Java virtual machine [73].

The distiller in our system will be applied only to the loops that the programmer has identified as desirable to parallelize. The distiller will choose an appropriate checkpointing interval by *speculatively strip mining*. In Figure 3 the distiller has strip mined the loop at the top so that a checkpoint occurs once per *strip* of 1024 iterations. The actual number of iterations chosen will depend on the application, the number of cores, and the configuration of the caches (more iterations will force the queue converter to create larger buffers). The body of the strip mined loop contains arbitrary control flow, but the distiller can decide to speculatively remove any code paths from the body that feedback tells it are not taken often enough to be relevant. The distiller transforms the branches to these rarely taken paths into conditional trap instructions. It is the intention that it is the 1024 iteration strip that will be parallelized by the ondemand queue converter described in Section 3.3.

The distiller also leverages ideas from 20 years of research in speculative parallelization [55, 98, 38, 101, 90, 84, 28, 99, 94, 45, 64, 57, 51, 2, 81, 79, 27, 85, 77, 18, 23, 1], and control independence [86, 20, 22, 46, 4]. Although there is some recent evidence to the contrary [52] these systems have shown that one can improve parallelism further by speculating on invariants in addition to branch direction. In particular, it seems worthwhile to speculate on memory dependences. This has also been observed in the ILP domain [74, 39, 21]. In my own previous work on PolyFlow [1], for example, we have observed automatic parallelization speedups on dusty deck, Spec 2000 *integer*, benchmarks of between 10% and 133%, with an average of 53%, as shown in Figure 4. Careful examination of the loops parallelized shows that they contain *true* memory dependences (loads in one thread that occasionally depend on a store in a different thread), but that these dependences rarely, or never, manifest themselves. Mock et al have similarly observed that points-to sets measured during profiling are significantly smaller than the points-to sets calculated by static analysis [69], and attribute the difference, in part, to these potential, but unexpressed dependences.

So that our system may parallelize across these real (in a conservative sense) but rare memory dependences the distiller must also be able to test for cross iteration dependences and trap if they manifest. Relatively small tables and hashing structures, similar to the ALAT in the IA64 can be leveraged to effect these dependence tests efficiently [39, 8]. For example Ceze's Bulk mechanism [18] simplifies dependence testing hardware by keeping *signatures* of the sets of addresses accessed by a thread. These signatures are approximate, but can be made probabilistically accurate by leveraging techniques from Bloom filters [14]. Knight also noted that hashing could be used to do approximate dependence testing [55]. Dependence testing does not need to be a particularly low latency operation. For example, it does not need to be done as mem-



```
top:
    /* original loop body */
    if $c goto top
done:
```

```
recover:
    $i = 0
top:
    /* original loop body */
    if !$c goto done
    $i = $i + 1
    if $i < 1024 goto top

    goto speculate
```

```
speculate:
    checkpoint (recover)
    $i = 0
strip:
    /* distilled loop body */
    if !$c trap
    $i = $i + 1
    if $i < 1024 goto strip

    if $c goto speculate
done:
```

Figure 3: Speculative strip mining turns the loop on the top into the distilled loop on the right that checkpoints once every 1024 iterations. When an early exit, or any other exceptional condition, occurs in the distilled loop, system state is restored to the most recent checkpoint and the sequential recovery loop on the left is run to get past the exception.

benchmark	bzip2	crafty	mcf	parser	twolf	vpr.place	vpr.route
speedup (%)	10	60	10	44	75	120	133

Figure 4: Memory dependence speculation on dusty deck (spec 2000 integer) benchmarks can yield parallelism improvements as high as 133% when running on a 4-core processor.

ory accesses occur, but may be delayed until just before committing a checkpoint [84].

While the latency of dependence testing and profiling information for feedback directed distillation is not particularly important, the resource utilization of these operations is a critical performance determinant. Eventually we hope that the distiller can move performance counters off the critical path, into exception handlers. In the short run our system needs to support relatively efficient basic-block counting [60, 63]. In the longer term it will also be necessary to get feedback about specific memory dependences [69, 72]. There will need to be an interplay between the static compiler and the instrumentation tool in order to make this satisfactorily efficient.

3.3 On-demand Queue Converter

The final piece of the proposed system is the *on-demand queue converter*. Some kind of scalar renaming, or dynamic single assignment form seems to be a necessity for all parallel programming systems. Parallelizing compilers have always required scalar expansion and loop distribution [58, 78, 24, 33, 54, 47]. Similarly, many explicitly parallel programming languages are based on dynamic single assignment (functional programming) as their primary method for expressing parallelism [43, 70, 15, 12]. In these languages the procedure activation (stack frame) is the primary unit for expressing renaming [53, 7].

Because of the requirement to have many copies of scalar values live simultaneously, many research parallel architectures have provided specific hardware support for synchronizing on these values [6, 25, 89, 44, 97]. The support varies from fine-grain multithreading combined with full-empty bits on memory locations [89, 44, 75], to support for streams of structures [6, 25], to explicit, fine grain, message passing interfaces [90, 97]. It is an open research question whether commodity multi-core processors, with their straight-forward shared memory implementations contain adequate support for scalar expansion, fine-grain multi-threading or streaming.

The on-demand queue converter in our system is responsible for performing scalar expansion and loop distribution. There are a number of phases required. First the queue converter must collect dependence information and form a value-flow graph [93]. Cycles in the value flow graph indicate portions of the graph that must be serialized [54]. Arcs of the value-flow

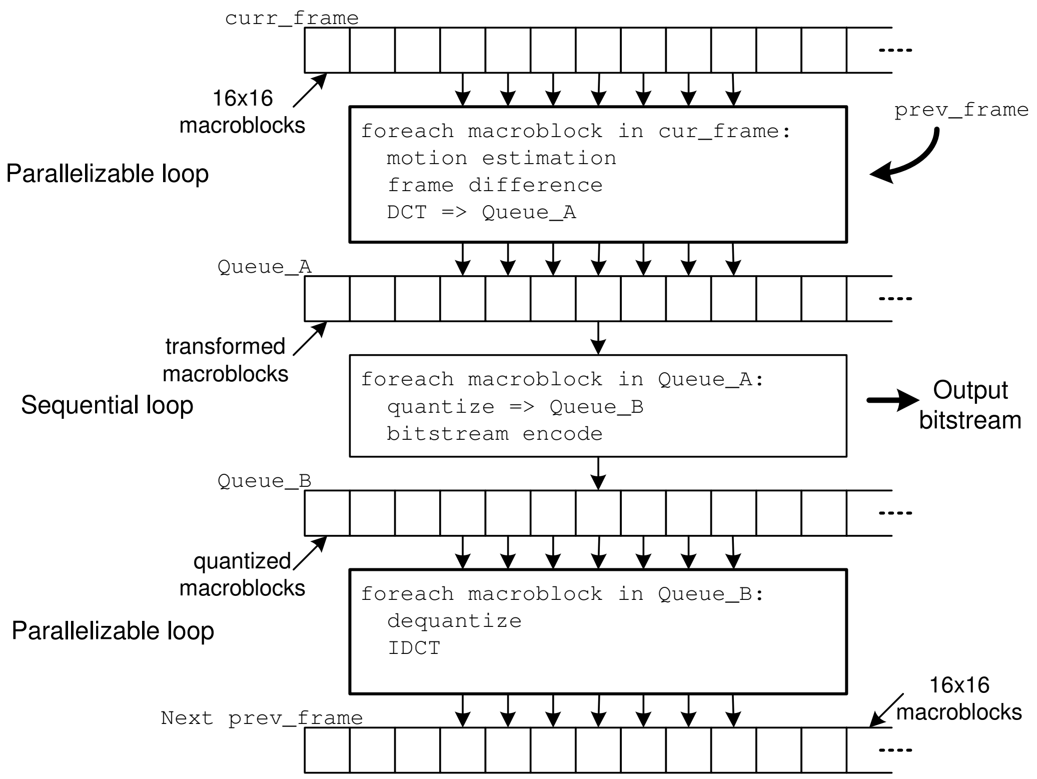
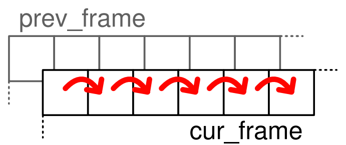


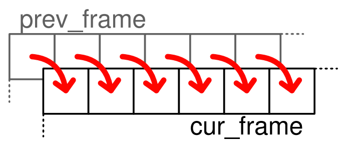
Figure 5: H.264 main loop restructured by on-demand queue conversion.

```
for (i = 0; i < num_blocks; i++)
  Vector guess = i ?
    (cur_frame[i-1].best_vector) :
    (0,0);
  cur_frame[i].best_vector =
    GetMatch(cur_frame[i],
             prev_frame, i, guess);
```



(a) Guess vectors are obtained from the previous macroblock of the current frame.

```
for (i = 0; i < num_blocks; i++)
  Vector guess =
    prev_frame[i].best_vector;
  cur_frame[i].best_vector =
    GetMatch(cur_frame[i],
             prev_frame, i, guess);
```



(b) Guess vectors are obtained from the corresponding macroblock in the previous frame.

Figure 6: H.264 Encoder Motion Estimation Example and Dependence Visualization. The algorithm on the left is sequential because every iteration depends on the `best_vector` generated in the previous iteration. The loop on the right can be parallelized because this dependence has been removed.

graph that enter or leave cycles represent variables that need to be scalar expanded [36]. Next the code for each thread must be transformed to make the communication operations explicit. Finally, the resulting parallelizable loops must actually be converted into the native thread interface of the underlying architecture.

The result of this process performed on the H.264 main loop is the three loops shown in Figure 5. The first loop is parallelizable across macroblocks, and contains the motion estimation, frame differencing and DCT operations. The output of the first loop is a queue that contains the results, across many macroblocks, of the DCT phase. The second loop is sequential and contains the quantization and bitstream encoding operations. The sequential loop reads in the queue produced by the first loop, and outputs both an encoded bitstream and a new queue that contains quantized macroblocks. The final loop is parallelizable and contains the dequantization and IDCT operations. It reads in quantized macroblocks produced by the sequential loop, dequantizes and inverse-transforms each one, and writes the result into the Next `prev_frame` image. Queues are required to communicate between the three loops, but no additional overhead is added to communicate between operations inside one loop.

3.4 Additional Compiler Optimizations to Support Concurrency

Traditional vectorizing and parallelizing compilers typically improve code concurrency with several transformations in addition to loop distribution. It is likely that these transformations would also be beneficial in this context. Examples of such transformations include reduction reassociation and forward propagation. Reduction reassociation identifies long spines of dependent operations that are associative, and turns those spines into trees or rakes that have smaller dependence depth [58, 103, 19, 11, 36]. Forward propagation "undoes" any redundancy eliminations that created additional dependence spines. Forward propagation may result in the program doing more work, but can also eliminate dependence chains that constrain concurrency.

4 Algorithm Choice

If parallelization is to succeed, the programmer must choose a parallel algorithm, rather than a sequential algorithm. For example, radix sort contains fewer cross-iteration dependences than does quicksort [13, 29], so a programmer developing a parallel application should know to call a radix sort routine rather than a quicksort (and the system library should provide a radix sort in addition to, or instead of, quicksort).

In the case of H.264 there are also important choices to be made in algorithm design. Motion estimation is the most compute intensive part of the application, and therefore the part of the application that it is most desirable to parallelize. The motion estimation stage for each macroblock works roughly as follows. Each macroblock represents a 16×16 pixel square of the current frame. The frame preceding the current frame is searched for a 16×16 pixel square that is most similar to the current macroblock.

The H.264 standard permits this search to be heuristic (rather than an actual optimization), and so the motion estimation stage is where vendors distinguish their products in terms of compression rate versus computational efficiency. An optimal compression algorithm would calculate the similarity between the current macroblock and the 16×16 block at every position in the previous frame and choose the most similar block. In practice this would be far too computationally intensive, so the heuristic algorithm will instead search inside a relatively small disc that surrounds the initial `guess` and stop sooner if it finds a block that matches the current macroblock closely enough.

Many motion estimation heuristics have been proposed and two are shown in Figure 6. In both the heuristics shown here, the motion estimation starts with a `guess` vector that represents a heuristic guess about the most likely point for the best match in the previous frame. The heuristic in Figure 6(a) chooses a `guess` vector based on the insight that objects tend to be larger than a single macroblock, so it is likely that whatever motion vector was calculated for the macroblock to the left is likely to be a pretty good guess for the motion of the current macroblock. The heuristic in Figure 6(b) chooses a `guess` vector based on the insight that physical objects (including video cameras) tend to have inertia, and thus the motion of the scene in this frame is likely to be similar to the motion in the previous frame.

While the two motion estimation heuristics in Figure 6 seem similar on the surface, the heuristic in Figure 6(a) will completely serialize the motion estimation algorithm, while the heuristic in Figure 6(b) permits parallelization of the motion estimation algorithm for a frame. In the heuristic in Figure 6(a) the current macroblock can't start its search until the previous macroblock has finished finding its best match, which is then used as the guess vector for the current macroblock. In the heuristic of Figure 6(b), however, the motion vectors for all the macroblocks of the previous frame have already been produced (because we need the previous frame to compare to anyway), so the motion vectors for the macroblocks from the previous frame can be used without creating a dependence that will obstruct parallelism. If the compute-intensive motion estimation step is to be parallelized, the programmer must choose an appropriate parallel algorithm, in this case one like the heuristic in Figure 6(b).

Designing parallel algorithms is the hard intellectual work that requires human creativity. Parallelization often (as in the case of H.264 motion estimation) requires the programmer to understand tradeoffs that are difficult to communicate in code. In this case, different motion estimation heuristics change the compression rate and quality of the output. The programmer must evaluate the tradeoffs between parallel performance, compression and quality. The goal of my work in implicit parallelization is to automate as much of the error prone parallelization process as possible, so that the programmer can concentrate on the truly challenging issues that are at the heart of the matter.

Acknowledgments

I thank Shane Ryoo and Chris Rodriguez for helping me understand the H.264 reference implementation. I thank Steve Lumetta and Wen-mei Hwu for the numerous useful comments they provided on an earlier draft of this paper.

References

[1] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting postdominance for speculative parallelization. *Intl Symp High-Performance Comp Arch*, (HPCA-13), 2007.

[2] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. *Int'l Symp. Microarchitecture*, (MICRO-31):226–236, 1998.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO 36*, page 423, 2003.

[4] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary. Transparent control independence (tci). *Intl Symp Comp Arch*, (ISCA-34):448–459, 2007.

[5] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, pages 316–327, 2005.

[6] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. The StarT-Voyager parallel system. In *PACT*, page 185, 1998.

[7] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4), 1998.

[8] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *25th International Symposium on Computer Architecture (ISCA-25)*, pages 227–237, Barcelona, Spain, June 1998.

[9] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. *ACM Conf Prog Lang Design and Impl*, (PLDI):149–159, 1996.

[10] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM Conf Prog Lang Design and Impl*, (PLDI):1–12, 2000.

[11] G. E. Blelloch, S. Chatterjee, and M. Zagha. Solving linear recurrences with loop raking. *Journal of Parallel and Distributed Computing*, Feb. 1995.

[12] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

[13] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. *ACM Symp. Parallel Algorithms and Architectures*, (SPAA-3):3–16, 1991.

[14] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[15] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Nov. 1994.

[16] R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. *ACM Conf Programming Language Design and Implementation*, (PLDI):1–14, 1988.

[17] V. G. Bose. *Design and Implementation of Software Radios Using a General Purpose Processor*. PhD thesis, Massachusetts Institute of Technology Dept. of Electrical Engineering and Computer Science, June 1999.

[18] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. *Intl Symp Comp Arch*, (ISCA-33):227–238, 2006.

[19] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings of Supercomputing*, pages 666–675, New York, NY, Nov. 1990.

[20] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *MICRO 34*, pages 4–15, 2001.

[21] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *ISCA-25*, pages 142–153, June 1998.

[22] J. Collins, D. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *Int'l. Symp. Microarchitecture (MICRO 37)*, 2004.

[23] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Tolerating dependences between large speculative threads via sub-threads. *Int'l Symp Comp Arch*, (ISCA-33):216–226, 2006.

[24] R. Cytron and J. Ferrante. What's in a name? The value of renaming for parallelism detection and storage allocation. *Intl Conf Parallel Processing*, (ICPP-16):19–27, 1987.

[25] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. *ACM/IEEE Supercomputing Conf*, (SC2003):35, 2003.

[26] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. *Intl Symp Code Generation and Optimization*, (CGO-1):15–24, 2003.

[27] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. *ACM Conf Prog Lang Design and Implementation*, (PLDI):71–81, 2004.

[28] P. K. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grain multithreading. In *Int'l Conf. Parallel Architecture and Compiler Techniques (PACT)*, pages 109–121, 1995.

[29] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin. Fast parallel sorting under LogP: experience with the CM-5. *IEEE Trans Parallel Distributed Sys*, 7:791–805, Aug. 1996.

[30] K. Ebcioğlu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Trans. Comput.*, 50(6):529–548, 2001.

[31] K. Ebcioğlu and E. R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. *Intl Symp Comp Arch*, (ISCA-24):26–37, 1997.

[32] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. *Intl Symp Microarchitecture*, (MICRO-34):16–27, 2001.

[33] P. Feautrier. Array expansion. In *Proceedings of the International Conference on Supercomputing*, pages 429–441, July 1988.

[34] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, July 1981.

[35] M. Frank, W. Lee, and S. Amarasinghe. A software framework for supporting general purpose applications on Raw computation fabrics. Technical Memo LCS-TM-619, MIT Laboratory for Computer Science, July 2001.

[36] M. I. Frank. *SUDS: Automatic Parallelization for Raw Processors*. PhD thesis, Massachusetts Institute of Technology Dept. of Electrical Engineering and Computer Science, May 2003.

[37] M. I. Frank, C. A. Moritz, B. Greenwald, S. Amarasinghe, and A. Agarwal. SUDS: Primitive mechanisms for memory dependence speculation. Technical Memo LCS-TM-591, MIT Laboratory for Computer Science, Jan. 1999.

[38] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *19th International Symposium on Computer Architecture (ISCA-19)*, pages 58–67, Gold Coast, Australia, May 1992.

[39] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 183–193, San Jose, California, Oct. 1994.

[40] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. *Intl Conf Arch Support Prog Lang and Operating Sys*, (ASPLOS-X):291–303, Oct. 2002.

[41] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248:147–199, Oct. 2000.

[42] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. *Intl Symp Microarchitecture*, (MICRO-38):343–354, 2005.

[43] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.*, 7(4):501–538, 1985.

[44] R. H. Halstead, Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. *Intl Symp Comp Arch*, (ISCA-15):443–451, 1988.

[45] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *Arch Support Prog Lang Operating Sys*, (ASPLOS VIII):58–69, 1998.

[46] A. D. Hilton and A. Roth. Ginger: control independence using tag rewriting. *Intl Symp Comp Arch*, (ISCA-34):436–447, 2007.

[47] B.-M. Hsieh, M. Hind, and R. Cytron. Loop distribution with multiple exits. In *Proceedings Supercomputing '92*, pages 204–213, Minneapolis, MN, Nov. 1992.

[48] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, 1993.

[49] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. *Intl Symp Comp Arch*, (ISCA-14):18–26, 1987.

[50] W. W. Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. Implicitly parallel programming models for thousand-core microprocessors. *Design Automation Conf.*, (DAC-44), 2007.

[51] I. H. Kazi and D. J. Lilja. Coarse-grained speculative execution in shared-memory multiprocessors. In *International Conference on Supercomputing (ICS)*, pages 93–100, Melbourne, Australia, July 1998.

[52] 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. *Intl Conf Supercomputing*, (ICS-20):24–35, 2006.

[53] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, Jan. 1995.

[54] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. *ACM/IEEE Supercomputing Conf*, (SC-90):407–416, 1990.

[55] T. Knight. An architecture for mostly functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 88–93, Aug. 1986.

[56] E. Kohler, R. Morris, and B. Chen. Programming language optimizations for modular router configurations. *Intl Conf Arch Support Prog Lang and Operating Sys*, (ASPLOS-X):251–263, Oct. 2002.

[57] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *International Conference on Supercomputing (ICS)*, Melbourne, Australia, July 1998.

[58] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. *Symp Principles of Prog Lang*, (POPL-8):207–218, 1981.

[59] A. A. Lamb, W. Thies, and S. P. Amarasinghe. Linear analysis and optimization of stream programs. *Prog Lang Design and Impl*, (PLDI):12–25, 2003.

[60] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. *ACM Conf Prog Lang Design Impl*, (PLDI):291–300, 1995.

[61] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. *IEEE Int'l Symp on Workload Characterization*, (IISWC), 2005.

[62] Z. Li. Array privatization for parallel execution of loops. In *Intl Conf Supercomputing*, Washington, DC, July 1992.

[63] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Conf Prog Lang Design Impl*, (PLDI):190–200, 2005.

[64] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *Int'l Conf. Supercomputing*, pages 77–84, 1998.

[65] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. *Intl Symp Microarchitecture*, (MICRO-35), 2002.

[66] S. Matsuoka, K. Taura, and A. Yonezawa. Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. *Conf Object-Oriented Programming Systems, Languages, and Applications*, (OOPSLA-8):109–126, 1993.

[67] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data-flow analysis and its use in array privatization. In *Proceedings of the Symposium on Principles of Programming Languages*, volume POPL-20, pages 2–15, Charleston, SC, Jan. 1993.

[68] M. Mock, C. Chambers, and S. J. Eggers. Calpa: a tool for automating selective dynamic compilation. *Intl Symp Microarchitecture*, (MICRO-33):291–302, 2000.

[69] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, June 2001.

[70] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[71] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. *Intl Symp High Perf Comp Arch*, (HPCA-12):254–265, 2006.

[72] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *ISCA-24*, pages 181–193, 1997.

[73] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. *Intl Symp Comp Arch*, (ISCA-34), 2007.

[74] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Trans. Comput.*, 38(5):663–678, May 1989.

[75] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-machine multicomputer: An architectural evaluation. In *ISCA*, pages 224–235, 1993.

[76] T. R. Novak. Freelist-based stack frame allocation. Master's thesis, University of Illinois, Dept. of Electrical and Computer Engineering, May 2004.

[77] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In *Int'l Symp. Microarchitecture (MICRO 38)*, pages 81–92, 2005.

[78] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, Dec. 1986.

[79] I. Park, B. Falsafi, and T. N. Vijaykumar. Implicitly-multithreaded processors. In *ISCA-30*, pages 39–51, 2003.

[80] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Trans. Comput.*, 50(6):300–318, June 2001.

[81] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. *Intl Symp Comp Arch*, (ISCA-28):204–215, 2001.

[82] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. *Int'l Symp Comp Arch*, (ISCA-29), 2002.

[83] R. Rajwar, M. Herlihy, and K. K. Lai. Virtualizing transactional memory. In *ISCA*, pages 494–505, 2005.

[84] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *ACM Conf Prog Lang Design and Impl*, (PLDI):218–232, 1995.

[85] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *19th Int'l Conf. Supercomputing (ICS)*, pages 179–188, 2005.

[86] E. Rotenberg and J. E. Smith. Control independence in trace processors. In *International Symposium on Microarchitecture*, pages 4–15, 1999.

[87] S. Ryoo, S.-Z. Ueng, C. I. Rodrigues, R. E. Kidd, M. I. Frank, and W. W. Hwu. Automatic discovery of coarse-grained parallelism in media applications. *Trans. High-Performance Embedded Architectures and Compilers*, 1(3):194–213, 2006.

[88] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu. Field-testing IMPACT EPIC research results in Itanium 2. *Intl Symp Comp Arch*, (ISCA-31):26–37, 2004.

[89] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Real Time Signal Processing IV*, volume 298, pages 241–248. Soc. Photo-optical Instrumentation Engineers (SPIE), 1981.

[90] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA 22*, pages 414–425, June 1995.

[91] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. *Intl Symp Comp Arch*, (ISCA-29):123–134, 2002.

[92] G. L. Steele Jr. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The ultimate GOTO. In *Annual Conference of the ACM*, pages 153–162, 1977.

[93] B. Steensgaard. Sparse functional stores for imperative programs. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, Jan. 1995.

[94] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *The 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[95] S. S. Stone, K. M. Woley, and M. I. Frank. Address-indexed memory disambiguation and store-to-load forwarding. In *MICRO-38*, pages 171–182, 2005.

[96] S. S. Stone, G. Wu, and M. I. Frank. Multi-versioning in the store queue is the root of all store forwarding evil, June 2007. submitted for review.

[97] M. B. Taylor, W. Lee, J. E. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. I. Frank, S. P. Amarasinghe, and A. Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. *Intl Symp Comp Arch*, (ISCA-31):2–13, 2004.

[98] P. Tinker and M. Katz. Parallel execution of sequential scheme with ParaTran. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 40–51, July 1988.

[99] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proc. Int'l Conf. Parallel Architecture and Compiler Techniques (PACT)*, pages 35–46, 1996.

[100] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, Portland, OR, Aug. 1993.

[101] C.-P. Wen and K. Yelick. Compiling sequential programs for speculative parallelism. In *Int'l. Conf. Parallel and Distributed Systems*, Dec. 1993.

[102] D. Wentzlaff and A. Agarwal. Constructing virtual architectures on a tiled processor. *Intl Symp Code Generation and Optimization*, (CGO-4):173–184, 2006.

[103] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.

[104] K.-L. Wu, W. K. Fuchs, and J. H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Trans Parallel Distributed Sys*, 1(2):231–240, Apr. 1990.

[105] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[106] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *MICRO 35*, pages 85–96, 2002.