SYSTEM SUPPORT FOR IMPLICITLY PARALLEL PROGRAMMING

Matthew I. Frank

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign
 Implicit parallelization involves developing parallel algorithms and applications in environments that provide sequential semantics, like 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.
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
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]. 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 interdependencies with...
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 run-time 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 num-

---

![Diagram](image.png)

(a) Motion estimation searches for the best matching 16x16 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.
ber 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
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, 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, 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 on-demand 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 speculative 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-
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.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>bzip2</th>
<th>crafty</th>
<th>mcf</th>
<th>parser</th>
<th>twolf</th>
<th>vpr.place</th>
<th>vpr.route</th>
</tr>
</thead>
<tbody>
<tr>
<td>speedup (%)</td>
<td>10</td>
<td>60</td>
<td>10</td>
<td>44</td>
<td>75</td>
<td>120</td>
<td>133</td>
</tr>
</tbody>
</table>

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.

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 multi-threading 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...
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);

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);

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

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

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 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


[22] M. I. Frank


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.


