HELIX-UP:
Relaxing Program Semantics to Unleash Parallelization

Simone Campanoni   Glenn Holloway   Gu-Yeon Wei   David Brooks

Harvard University

{xan,holloway,guyeon,dbrooks}@eecs.harvard.edu

Abstract

Automatic generation of parallel code for general-purpose commodity processors is a challenging computational problem. Nevertheless, there is a lot of latent thread-level parallelism in the way sequential programs are actually used. To convert latent parallelism into performance gains, users may be willing to compromise on the quality of a program's results. We have developed a parallelizing compiler and runtime that substantially improve scalability by allowing parallelized code to briefly sidestep strict adherence to language semantics at run time. In addition to boosting performance, our approach limits the sensitivity of parallelized code to the parameters of target CPUs (such as core-to-core communication latency) and the accuracy of data dependence analysis.

1. Introduction

Contrary to common assumptions, there is considerable latent parallelism, even in irregular sequential programs, e.g., between the iterations of a loop. When such iterations can run unfettered on multiple cores in a modern multicore processor, performance scales with the number of cores. Unfortunately, loop iterations often depend on one another and require strict ordering. Hence, compilers that strictly adhere to program semantics generate slow sequential code to guarantee correct results. However, for applications that can tolerate bounded distortion of results, there is an exciting opportunity to ignore some dependences and liberate parallelism. We show how to avoid bottlenecks by relaxing sequential consistency constraints in a disciplined manner, doing so only when it increases performance while incurring little or no output distortion.

Automatic generation of parallel code is hampered by the huge computational complexity of teasing out truly independent parallel threads at static compile time. Conventional compilers will conservatively keep apparent dependences unless provable otherwise. While this strict adherence to program semantics guarantees correctness, not all dependences are created equal. Some will slow down execution more than others. Some dependences have little to no impact on the outputs we actually care about. Lastly, these characteristics change depending on inputs and at run time. Therefore, the challenge is identifying these dependences that bottleneck parallel performance, understanding what impact if any there would be on the outputs by relaxing sequential requirements, and relaxing them at runtime, all guided by performance goals and tolerable limits set by the program's user (not the programmer).

Relaxing constraints in exchange for performance is not a new concept. Approximate computing has been studied and applied to a wide variety of applications that can tolerate bounded approximate outputs, for example, multimedia, machine learning, and pattern recognition [17, 26, 27]. This work seeks to combine approximate computing with automatic parallelization by providing user settable tuning knobs that trades output distortion for performance and/or energy gains. In this effort, we implemented the *unleashed parallelizer* (HELIX-UP), a co-design of profilers, a loop-parallelizing compiler, and a runtime system. The HELIX-UP compiler, built on top of a recently developed parallelizing compiler called HELIX [4], selectively relaxes strict adherence to language semantics to increase parallel scalability at run time. HELIX-UP's user provides a sequential program, representative training inputs, and a function that measures distortion of the program's outputs. The profiler uses this information to assess how much of the program semantics can be relaxed and the impact of doing so. Finally, the runtime system applies the relaxations judiciously and automatically based on the user's request.

In contrast to other work that combines approximate computing with automatic parallelization, HELIX-UP makes the following contributions. First, because it relaxes program semantics only long enough to overcome bottlenecks detected at run time, HELIX-UP delivers high performance with less output distortion. Second, it automatically tunes relaxation of program semantics to the characteristics of the target platform. Third, HELIX-UP is able to eliminate obstacles to parallelization because it recognizes which code segments cause them.

The remainder of this paper is organized as follows. Section 2 lays out the opportunities and justifications for applying approximate computing to automatic parallelization. We then describe the details of HELIX-UP, its constituent components and how they interact at runtime in Section 3. Section 4 then presents optimizations offered by relaxed semantics and Section 5 evaluates HELIX-UP compared to related approaches to parallelization and approximate computing. Finally, Section 6 describes the related prior work in detail.

2. Opportunity

Even complex, irregular, sequential programs have latent parallelism. Research over the last decade has developed ways to unlock that parallelism on commodity multicore microprocessors [4, 6, 23]. For example, HELIX is a parallelizing compiler that automatically speeds up SPEC benchmarks by an average of 2.3× on a 6-core processor [4]. However, the performance

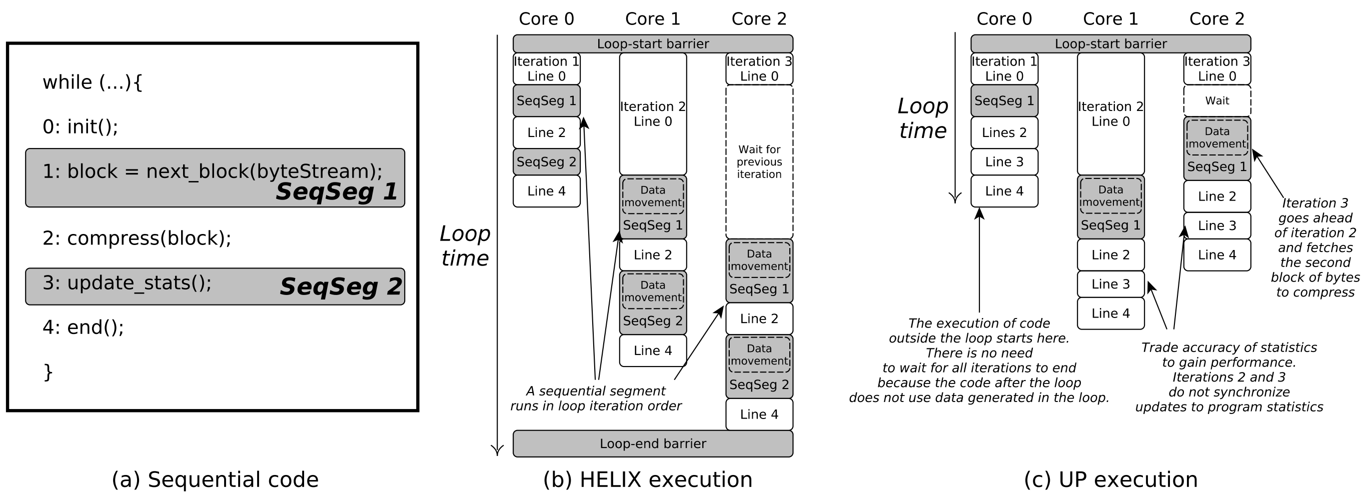


Figure 1: Possible improvements for semantics-relaxing code parallelization like HELIX-UP.

gains tend to saturate with higher core counts. Moreover, these gains are sensitive to hardware characteristics (such as core-to-core latency and bandwidth) and to the amenability of workloads to high-quality analysis.

Traditionally, parallelizing compilers are conservative. They preserve sequential evaluation order of programs unless the reordering is provably not observable. They respect all apparent dependences unless proven to not be actual dependences, guarantee parallelized versions will terminate if the original sequential programs do, and produce exactly the same results. In contrast, approximate computing suggests that for users who can characterize representative inputs of their programs and can accept small deviations from perfect outputs, this conservatism is blocking potential performance and/or energy benefits. There is an opportunity to realize those benefits by disciplined relaxation of strict adherence to program semantics (dependence satisfaction and therefore program order).

The HELIX-UP compiler builds on HELIX, which is a conservative parallelizing compiler that automatically generates parallel threads from a sequentially-expressed program and allows parallel loop iterations to run concurrently on separate cores in a commodity multicore processor. This approach resembles traditional DOACROSS parallelism [11]. To satisfy data dependences between loop iterations (i.e., *loop-carried dependences*), HELIX identifies segments of a loop's body—called *sequential segments*—that must execute in iteration order on the separate cores. Synchronization operations that surround each sequential segment guarantee loop-iteration order. HELIX also guarantees that different sequential segments are independent so that dynamic instances run in parallel to gain more performance [5].

Three main sources of overhead hinder performance improvements in traditional parallelizing compilers like HELIX that strictly adhere to program semantics. First, strict observance of a program's original sequential order leads to bottlenecks resulting from a long running iteration that stalls all other cores. Second, slow communication between cores amplifies the cost of inter-core data sharing and communication required to satisfy dependences. Lastly, distribution of shared data across multiple cores degrades locality. By carefully relaxing otherwise strict adherence to program semantics, HELIX-UP can alleviate all three of the aforementioned overheads with little to no impact on output correctness.

2.1 Example Comparing Execution of HELIX to HELIX-UP.

To better understand how HELIX-UP improves performance over a traditional parallelizing compiler like HELIX, we use an illustrative example. Figure 1a shows a snippet of sequential code that resembles the most important loop found in 256.bzip2 from SPEC CPU2000 after applying transformations (e.g., memory privatization and variable vectorization) performed by HELIX before parallelization to remove as many dependences as possible. The while loop compresses a stream of bytes (i.e., `byteStream`) read from a file where each loop iteration compresses a block of the stream (i.e., `block`). An iteration starts by initializing the memory required for its execution (`Line 0`). Then, it retrieves the next block from the byte stream (`Line 1`). Because this part of the iteration depends on itself across the loop boundary, HELIX creates a sequential segment shown as `SeqSeg 1`. `Line 2` compresses the block and `Line 3` computes some statistics (e.g., how many bytes saved by compression) updated by all iterations. Hence, HELIX creates another sequential segment `SeqSeg 2` to serialize their updates. Finally, `Line 4` cleans up the private memory used since the beginning of the iteration.

Figure 1b illustrates the execution of parallel code generated by HELIX. All instances of `Line 0` run in parallel on three cores. However, `Core 1` happens to take longer initializing its private memory, delaying execution not only in `Core 1` but also in `Core 2`. This is because program semantics dictate dynamic instances of sequential segments must execute in

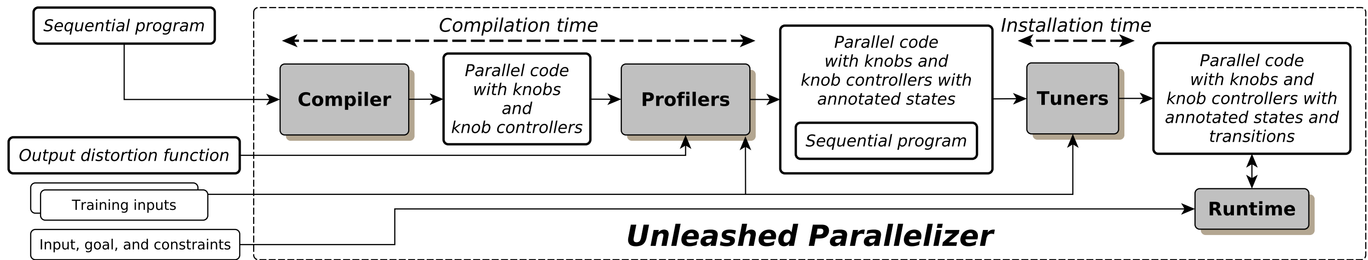


Figure 2: User provides a sequential program to parallelize, a function to measure program's output distortions, representative inputs, and goal and constraints. The compiler parallelizes the code, including in it both knobs and knob controllers. The profilers annotate knob controllers by using the training inputs. The tuners customize knob controllers for the target platform. While the program runs with the reference input, the runtime interprets knob controllers, adjusting knobs to alleviate performance bottlenecks, while respecting the constraints and goal set.

loop iteration order and only one instance of a sequential segment may run at a time.[1] This strict adherence requires stalling `Core 2` while waiting for the sequential segment in the prior iteration to complete. But this type of bottleneck of the parallel code can be measured at runtime. Relaxing constraints on program order just long enough to remove detectable bottlenecks substantially increases performance while minimizing the impact on program output. Said another way, if transformations that sidestep strict sequential semantics are judiciously applied only when needed and beneficial, HELIX-UP can relieve all three sources of overhead. Figure 1c shows how relaxing the treatment of sequential segments improves parallelization.

Distortion-free reordering. Sidestepping strict sequential semantics need not distort the output. In our example, `SeqSeg 1` arises from the actual dependence between `Line 1` and itself. The imposed ordering prevents separate iterations from picking the same block to compress, but serves no other purpose since blocks can be compressed in any order. Unfortunately, compilers cannot make such semantically-justified inferences, especially if they only hold during parts of the workload's execution. However, HELIX-UP can. Relaxing the ordering requirement removes this sequential bottleneck and allows the third iteration to now compress the second block instead of waiting, demonstrated in Figure 1(c). The reordering also reduces communication overhead by removing synchronizations. Moreover, the compiler is now free to reschedule iterations to preserve spatial locality.

Low-distortion reordering. Of course, there sometimes is a cost to breaking the semantics of the original sequential program, but this cost can be small and often ignored. Again looking at our example, HELIX executes `SeqSeg 2` sequentially to guarantee that loop statistics are computed in the same order as in the original sequential program; every instance of `SeqSeg 2` loads the value generated by the previous iteration, updates the shared value, and stores back the new value. If the program's user can tolerate some error in these statistics, the HELIX-UP compiler allows `SeqSeg 2` to execute in parallel most of the time. Breaking the dependence that gave rise to `SeqSeg 2` eliminates sequencing and communication overheads. It also improves data locality. Instead of loading statistics data from a remote private DL1 cache, as done in HELIX, data is kept in the private cache of a core while `Line 3` runs in parallel.

[1] Consequently, HELIX prefers many short sequential segments over few long segments.

3. Unleashed Parallelization

HELIX-UP provides scalable and robust performance of parallelized code automatically generated from sequential programs by sidestepping language semantics at runtime. HELIX-UP's runtime system relaxes the program order dictated by instruction dependences when beneficial. This allows HELIX-UP to improve TLP by reducing sequential code; to reduce the amount of communication among cores, by not sharing data and not synchronizing cores; and to reorder parallel loop iterations to improve data locality. HELIX-UP also allows the user to set the maximum amount of relaxation allowed, ranging from nothing (e.g., all dependences must be satisfied) to unbounded (e.g., no dependence need be satisfied). As in existing approximate computing systems, the user provides an extra function, for example $\sum \frac{\hat{o}-o}{\hat{o}}$ where $\hat{o}$ is the expected output and o is the output obtained [17], to compute distortion during the profiling phase based on user-provided inputs. Through this output distortion function, the user expresses the relative importance of program goals, albeit not in domain-specific terms. A user chooses between performance and energy savings and sets a maximum or a minimum performance, energy savings, and/or output distortion. Then, HELIX-UP automatically decides how much to relax the code at runtime to fulfill user requirements. The rest of this section describes the components of HELIX-UP and how they interact to relax program order.

Figure 2 provides an overview of HELIX-UP. Information inferred by the HELIX-UP compiler through traditional code analyses (e.g., data dependence analysis or induction variable analysis) is coupled with information collected by HELIX-UP profilers. Each profiler identifies dependences that have little or no impact on workload outputs when left unsatisfied. The HELIX-UP runtime uses this information while monitoring parallel execution. When the observed execution does not meet expectations, the runtime selectively relaxes program order based on profiler data to boost performance.

To control how much to relax program order, HELIX-UP provides a set of *knobs*, which are defined by the compiler,

calibrated off line by profilers, and then used by the HELIX-UP runtime. Each knob corresponds to a potential source of parallelism-dampening overhead for a parallelized loop. For example, a knob may describe how strictly the dynamic instances of a sequential segment like those in Figure 1 are sequenced, or whether the order of loop iterations is being maintained, or how rigorously communication between cores is being carried out. The setting of a knob starts from the most conservative configuration (e.g., program order is preserved completely, at some cost in performance) to the most aggressive one (e.g., order is sacrificed completely to maximize performance).

Creating knobs. The HELIX-UP compiler generates parallel code that includes knobs as well as one automaton per parallelized loop that serves as a *knob controller* (Figure 2). Each state of a knob controller represents a specific setting of the loop's knobs. State transitions represent the changes to knob settings that the runtime is allowed to make. Each transition is labeled with a condition, based on knob metrics, under which the corresponding knob changes can be made. Each knob has an associated *knob metric*, a measurement of the amount of overhead generated as a result of the knob's current setting. For example, for a knob that models a sequential segment, the knob metric represents the cumulative time spent by all threads waiting to execute this sequential segment.

Calibrating knobs. Knobs are calibrated based on training inputs provided by the user, and they can be adjusted by the runtime in response to changing behavior of the parallelized workload. The HELIX-UP runtime uses the knob metric to decide when to adjust the knob. Offline HELIX-UP profilers (Figure 2) use representative inputs to characterize the sensitivity of knobs during parallelization and measure the characteristics of the code (e.g., performance). The profilers then annotate all states of a knob controller with performance, energy saved, and output distortion generated by different knob settings. These measurements are averages over the available inputs, relative to the execution of the original sequential program (e.g., $3\times$ speedup, $2\times$ energy saved, 1% output distortion), and they are computed for the entire program execution (i.e., end-to-end evaluation). HELIX-UP's profilers rely on hardware performance counters of commodity platforms.

Tuning knob controllers. During installation, tuners define a triggering condition for each state transition of each controller, based on platform-specific characteristics. Tuners rely both on microbenchmarks and on the original sequential program. We run microbenchmarks to assess platform-specific characteristics like core-to-core communication latency. We run the original program to assess platform-specific behavior, such as its cache miss ratio, used in triggering conditions that respond to data locality loss. For example, if a platform has high core-to-core communication latency and cores begin to synchronize frequently, the synchronization-relaxing knob can lower communication overhead, albeit at the cost of some output distortion. But if a platform's inter-core communication links are fast enough, there is no benefit to making this compromise. After defining a controller automaton's transitions, the tuner minimizes states that do not improve performance, energy, or output distortion prior to installing it in the parallelized executable file.

Using knobs. The runtime is an independent thread created when the program starts. It uses knobs to control whether and for how long the parallelized program makes compromises to gain performance and save energy. Before initiating the compiled program, the runtime further optimizes each knob controller by removing states that fail to satisfy user constraints. The runtime starts each knob controller in its semantics-preserving state, i.e., with all knobs at the most conservative settings. After starting the compiled program, the runtime uses hardware performance counters to monitor the knob metrics associated with each parallelized loop as it executes. It wakes up periodically to check all conditions (defined by tuners at installation time) that might trigger a state change. Triggering conditions prompt the runtime either to increase or decrease aggressiveness of knob settings. When a transition is triggered, the runtime checks whether moving to the new state helps to meet the user's objective (e.g., performance). If so, the runtime makes the transition and adjusts the knob settings accordingly.

Assumptions and limitations. HELIX-UP assumes that the training inputs yield representative performance, energy, and output distortion. If that assumption is incorrect, HELIX-UP does not guarantee to bound output distortion or to avoid unrecoverable faults. In that case, the user of HELIX-UP risks unexpected output distortion to gain substantial performance or energy benefits. To overcome this problem, other systems rely on calibration periods at run time [26].

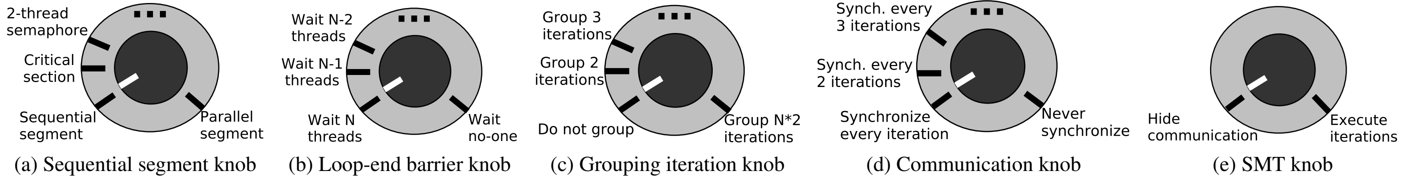
4. Optimizing for Adaptability

For a region that has an associated knob, the HELIX-UP compiler produces alternative implementations corresponding to the settings of that knob. At least one alternative reflects strict semantics. Others relax semantics to improve performance and/or power. The HELIX-UP runtime sets a knob by selecting which alternative to execute. The runtime itself never generates or reoptimizes code.

HELIX-UP's relaxed-semantics optimizations fall into three categories, according to the overhead they target: sequential bottlenecks, loss of data locality, and delay incurred by inter-core communication.

4.1 Optimizing Sequential Bottlenecks

We define a sequential bottleneck as the case where most of the cores running parallelized loops get stuck waiting for one core to finish its loop iteration. A bottleneck can occur because, strictly speaking, dynamic instances of each sequential segment within a loop (which arise from a loop-carried dependences between loop instructions) should evaluate in loop iteration order. Recall the example in Figure 1(b), `SeqSeg 1` is a sequential segment that guarantees dynamic instances of `line 1` execute in the order specified by the original sequential program, which leads to stalling core 2 and all subsequent iterations. Another source of sequential bottlenecks is the barrier at the end of a loop that prevents code beyond the loop from running until all iterations complete. Bottlenecks like these, which occur when loop iterations are out of balance, are an important



(a) Sequential segment knob (b) Loop-end barrier knob (c) Grouping iteration knob (d) Communication knob (e) SMT knob

Figure 3: Knobs automatically generated by the HELIX-UP compiler. Increasing a knob setting relaxes program semantics.

source of overhead for HELIX and similar parallelizers. The following subsections describe different optimizations HELIX-UP's runtime relies on to address the above bottlenecks. We describe each optimization with respect to the underlying *idea*, the associated *knob* settings, and the *metric* used to determine which setting to use at run time.

4.1.1 Relaxing Sequentiality

Idea. The consumption of data resulting from a loop-carried dependence is often more than one iteration away from its production [6]. In that case, or when failure to satisfy the dependence has a tolerable impact on program output, mutual exclusion of all subsequent instances of the corresponding sequential segment is unnecessary. Notice that, for a semantics-preserving compiler, inferring the number of iterations that separate the producing and consuming instructions of a dependence is challenging at best, and sometimes impossible, for non-trivial programs. Also, for some sequential segments, maintaining loop iteration order is unnecessary and their dynamic instances can run in any order. This is usually a consequence of an overly sequential implementation arising from a sequential programming language such as C. We saw in Figure 1(c) how HELIX-UP freed `SeqSeg 1` to run the third iteration on core 2 ahead of the second iteration on core 1. Again, a semantics-preserving compiler cannot readily infer this code property.

Knob. Figure 3a illustrates the knob automatically created by the HELIX-UP compiler and available to the HELIX-UP runtime for optimization. The compiler generates one knob per sequential segment and retains the most conservative setting to restrict execution of at most one thread per sequential segment and to preserve loop-iteration order (i.e., the HELIX policy for sequential segments). A more relaxed setting removes this order constraint while letting in one thread per sequential segment, making it a critical section. The next relaxed setting allows two threads per sequential segment, as though guarded by a 2-thread semaphore. Additional settings allow increasing numbers of cores to execute a sequential segment at a time, making it a parallel segment. Note that all settings other than the last require thread synchronization.

Metric. The metric for this knob that is related to a sequential segment is the idle time spent by cores waiting for its segment instances to execute sequentially. We estimate it with:

$$\textit{Pressure} = \frac{\sum_{i \in \text{Threads}} \text{Idle}_i}{\text{WallTime} \times (\text{Cores} - 1)}$$

where Idle_i is the time that thread i idles before it can execute the corresponding sequential segment. WallTime is the wall-clock time since the start of the parallelized loop.

4.1.2 Relaxing the Loop-End Barrier

Idea. Data generated by loops are not always used immediately. Therefore, the barrier placed at the end of the loop to resume the execution of code outside of it can be relaxed.

Knob. Figure 3b illustrates the knob corresponding to this optimization. The compiler generates one knob per loop. The most conservative setting waits for all threads to complete before resuming execution of the code outside the loop. A relaxed setting waits for all but one of the threads to complete. Subsequent settings wait for fewer and fewer threads.

Metric. The Pressure metric also applies here.

4.2 Optimize Data Locality

Distributing loop iterations across multiple cores typically degrades spatial and temporal locality compared to the original sequential code that runs on a single core. This can be an important source of overhead.

4.2.1 Grouping Loop Iterations

Idea. Rather than distributing individual iterations across cores, executing a few contiguous iterations on the same core as a group gains both temporal and spatial locality. However, this can create create a severe sequential bottleneck if all dependences must be satisfied. In other words, a sequential segment of the first iteration of a core has to wait for all prior iterations on an adjacent core to complete. To avoid this bottleneck, we generate the code such that only the first iteration of the group synchronizes with only the first iteration of the previous group running on the adjacent core. Of course, this comes with the risk of breaking some dependences that might lead to output distortions. However, because most loop-carried data dependences are not between adjacent loop iterations [6], the output distortion created by this transformation can be acceptably low for some loops.

Knob. Figure 3c shows the knob corresponding to this optimization, where the compiler generates one knob per loop. The most conservative setting is to not group iterations like for HELIX. The more relaxed setting groups two subsequent iterations together. Next settings increase the number of iterations per group. We set the maximum number of iterations within a group to be twice the number of cores of the platform.

Metric. The extra DL1 cache misses of all cores per cycle corresponds to the overhead due to grouping loop iterations together. We tried other cache levels and observed that DL1 best represents the locality lost. This metric is defined as:

$$\textit{LocLost} = \frac{\frac{(\sum_{c \in \text{Cores}} \text{Current}_c)}{\text{Cores}} - \text{Expected}}{\text{WallTime}}$$

where Current_c is the DL1 miss rate for the i-th core since the beginning of the parallelized loop. Expected is the average cumulative DL1 misses of the sequential version of the parallelized loop across all of its invocations, found by running the original sequential program. WallTime is the wall-clock time since the start of the parallelized loop being executed.

4.3 Optimize Core-to-Core Communication Cost

Inter-core communication overhead can also be optimized.

4.3.1 Relaxing Synchronization

Idea. As a side effect of other synchronizations, some sequential segments may get synchronized with prior iterations, thereby making some synchronizations redundant. However, this redundancy is difficult to predict and the conservativeness of traditional code analysis can easily miss it. This optimization relaxes synchronizations assuming some may be redundant.

Knob. Figure 3d illustrates the knob corresponding to this optimization. Again, the compiler generates one knob per sequential segment. The most conservative setting is to always synchronize with prior iterations. The relaxed settings decrease the frequency of synchronizations by skipping an increasing number of successive iterations.

Metric. On commodity platforms, threads can only synchronize via memory operations that invoke the cache coherence protocol. Hence, synchronization incurs an overhead of moving modified cache lines from the private DL1 cache of one core to the cache in a core requiring synchronization. This metric estimates how frequently cores communicate as follows: $Comm = \frac{\text{ModCacheLinesT}}{\text{Loads}}$, where ModCacheLinesT is the number of modified cache lines transferred between cores and Loads is the sum of all loads executed by any core used.

4.3.2 Multiple Uses of SMT

Idea. The simultaneous multithreading (SMT) feature in today's commodity platforms can ether hide the communication latency between cores or perform actual computation. The best use of SMT depends on the runtime characteristics of a parallelized loop. If the parallel code requires a relatively large amount of communication, then best to hide latency (as it is used by HELIX [4]). On the other hand, if cores communicate infrequently enough (e.g., thanks to relaxed-semantics optimizations previously described), it may be better to use SMT to execute more iterations.

Knob. Figure 3e illustrates the knob corresponding to this optimization. The compiler generates one knob per core and only has two settings: hide communication latency or execute iterations. Similar to grouping iterations (Section 4.2.1), SMT threads execute iterations without synchronizing with any other iterations to avoid severe sequential bottlenecks.

Metric. The Comm metric also applies here.

5. Evaluation

The HELIX-UP runtime temporarily relaxes strict program semantics, but only for long enough to alleviate performance bottlenecks. Doing so, HELIX-UP makes the performance of parallelized code robust and scalable while limiting output distortion. We now demonstrate the resulting benefits on a variety of today's commodity platforms, and we show that each component of HELIX-UP is important to its success, including the semantics-preserving code optimizations that come with HELIX. We conclude with an oracle study that quantifies remaining opportunities.

5.1 Experimental Setup

Using sequentially-designed benchmarks ranging from those commonly considered difficult to parallelize (SPEC CPU suite) to more regular ones (PARSEC), we evaluated HELIX-UP on real desktop and server processors from Intel and AMD.

Platforms. We label the platforms with their microarchitecture names. The first, *Nehalem*, includes six Intel® Core™ i7-980X cores, each operating at 3.33 GHz. The second, *Haswell*, includes four Intel® Core™ i7-4770K cores, each operating at 3.5 GHz. We disabled Turbo Boost for both platforms. Each has three cache levels. L1 (32KB) and L2 (256KB) are private to each core. With HyperThreading enabled, two threads can run per core, sharing the first two cache levels. In both Nehalem and Haswell, the last level cache (12MB and 8MB, respectively) is shared by all cores.

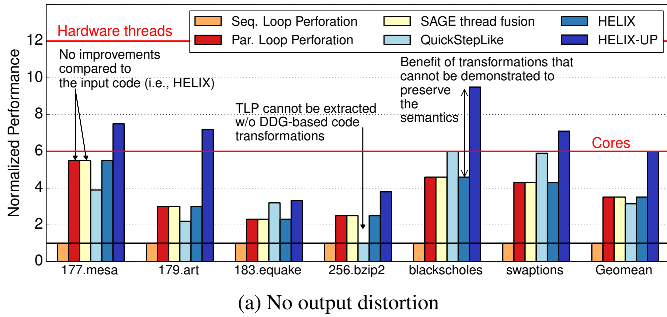
The third platform, *Bulldozer*, includes eight AMD Opteron 6380 cores, each operating at 2.5 GHz. This processor also has three cache levels. The 16KB first-level data cache is private to each core. Adjacent pairs of cores share a 2MB DL2 cache. The 8MB DL3 cache is shared by all cores. Each core runs one thread at a time.

On each of the three platforms, the cache coherence protocol is the only mechanism for synchronizing the cores of a single die and for sharing data.

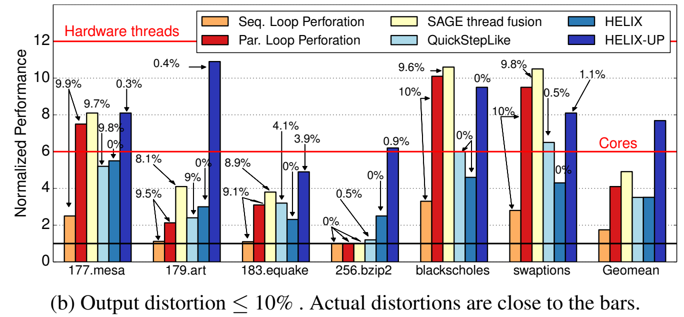
HELIX-UP Tuners. Transition rules are pre-defined and they rely on thresholds. Threshold values are either universal to all benchmarks (e.g., core-to-core communication latency of the target platform) or benchmark specific (e.g., DL1 cache miss rate on the target platform). The latter are computed by running the sequential form of the benchmark on the target platform during tuning at installation time.

In more detail, to define the triggering conditions for each state transition of knob controllers, tuners included in HELIX-UP measure the following platform characteristics with special-purpose microbenchmarks: core-to-core communication bandwidth (C_B), latency (C_L), and the difference (in clock cycles) between time to access the last level cache and that to access the DL1 cache ($DL1_L$). We found the following transition thresholds empirically. For a knob that uses the *LocLost* metric, exceeding the threshold $\frac{0.05}{DL1_L}$ causes the knob setting to increase by a step. For a knob that uses *Comm* metric, the corresponding threshold is $(0.20 \times \frac{C_B}{C_L})$. An observed metric lower than 0.001, on the other hand, causes either of those knobs to decrease by a step. For a knob that uses the *Pressure* metric, a value higher than 0.15 triggers a one step increase; a value lower than 0.002 decreases the setting by one step.

Compiler. We used the second version of the HELIX compiler, HCCv2 [6]. HCCv2 is based on the ILDJIT compilation framework [3]. We extended ILDJIT to use the latest available version of LLVM: 3.4.1. The sequential programs used as baseline were the unmodified versions of benchmarks, opti-



(a) No output distortion



(b) Output distortion $\leq 10\%$. Actual distortions are close to the bars.

Figure 4: HELIX-UP doubles the speedup obtained by the traditional parallelizing compiler HELIX. Platform used: Nehalem

mized (O3) and compiled by ILDJIT with LLVM 3.4.1 as the back end. On all platforms considered, this baseline outperforms GCC 4.8.2 on our benchmark set by 1% on average.

Benchmarks. Our benchmark set included four benchmarks from the SPEC CPU2000 suite and two from the PARSEC suite. In each case, we used the suite's training inputs for profiling and its reference inputs for evaluations. While these benchmarks are well known, we review each one in the context of HELIX-UP.

177.mesa is an OpenGL library. The input defines the use of the library. Execution of this code using the inputs included in the suite renders a 3D image. With HELIX-UP, approximation was applied to the rendering. The output distortion function compared the rendered image with the expected one.

179.art first trains a neural network on two objects and then uses the trained network to recognize these objects in a large image. Approximation was applied to both phases of the execution. The output distortion function compared the locations of the recognized objects.

183.equake simulates the propagation of waves. Approximation was applied to these propagations. The output distortion function checked all wave propagations.

256.bzip2 compresses and then decompresses an input file. For this study, we used only the compression phase, which accounts for (85%) of the execution time. Approximation was applied to the computation of statistics about the compression. The output distortion function validated the compressed file as well as the statistics of the compression.

blackscholes prices a portfolio of European options by solving a partial differential equation. Approximation was applied to the solution of this equation. The output distortion function checked the computed prices.

swaptions prices a portfolio of swaptions by using Monte Carlo simulation to solve a partial differential equation. Approximation was applied to the Monte Carlo simulation. The output distortion function checked the computed prices.

Measuring code executions. We used hardware performance counters to measure both performance and energy consumed. The PAPI library, version 5.3.0, includes support for RAPL performance counters. We used them to measure the energy consumed as implemented in HaPPy [34]. Finally, all measurements were cumulative over all loops parallelized, which cover the majority (≥ 80 %) of benchmark executions.

Prior work. We compare HELIX-UP with the HELIX parallelizing compiler [4] and with our implementations of three state-of-the-art approximate computing approaches: SAGE, loop perforation, and QuickStep. All three were implemented under the same compilation framework on which HELIX-UP is built.

SAGE [26] is designed for GPUs, but one of its semantics-relaxing transformations, thread fusion, is applicable to CPUs as well. Hence, we applied it to the parallel code generated by HELIX. Thread fusion merges adjacent threads (hence, adjacent loop iterations) and a copy of the output of the first becomes the output of the second.

Loop perforation [27] skips loop iterations. We applied it both to the sequential program (sequential loop perforation) and to the HELIX-generated code (parallel loop perforation).

QuickStep [22] distributes loop iterations across cores without relying on either code analyses or code transformations. We applied our implementation of QuickStep, called QuickStepLike, to the sequential versions of the benchmarks.

5.2 Performance Evaluation

Figure 4 compares HELIX-UP on the Nehalem platform against prior work when either 0% or 10% output distortion is acceptable. To evaluate thread fusion, loop perforation, and QuickStepLike, we swept their transformation parameters (e.g., which loop iterations to skip) and we kept the best results.

Using semantics-relaxing code transformations to *enhance* semantics-preserving ones is better than replacing them (as QuickStepLike does). Their difference in Figure 4 shows the value of sidestepping strict semantics sparingly—only when semantics-preserving optimizations fall short.

HELIX-UP limits the duration of departures from strict semantics by using the resulting code only long enough to erase the bottlenecks detected at run time. Doing so achieved at least the same performance as the other approaches with significantly less output distortion. Loop perforation applies its relaxation scheme for the entire execution, and we applied SAGE thread fusion in the same way.

No output distortion. Figure 4a plots the performance achieved when no output distortion is allowed. On the six-core Nehalem platform, HELIX-UP increases the average workload speedup to 6× (versus 3.5× for HELIX) *without* changing the output. This is the result of allowing transformations that cannot be proved correct at compile time, but that turn out to be correct at run time. One example is the transformation that changes the sequential segment `SeqSeg 1` in Figure 1 into a critical section.

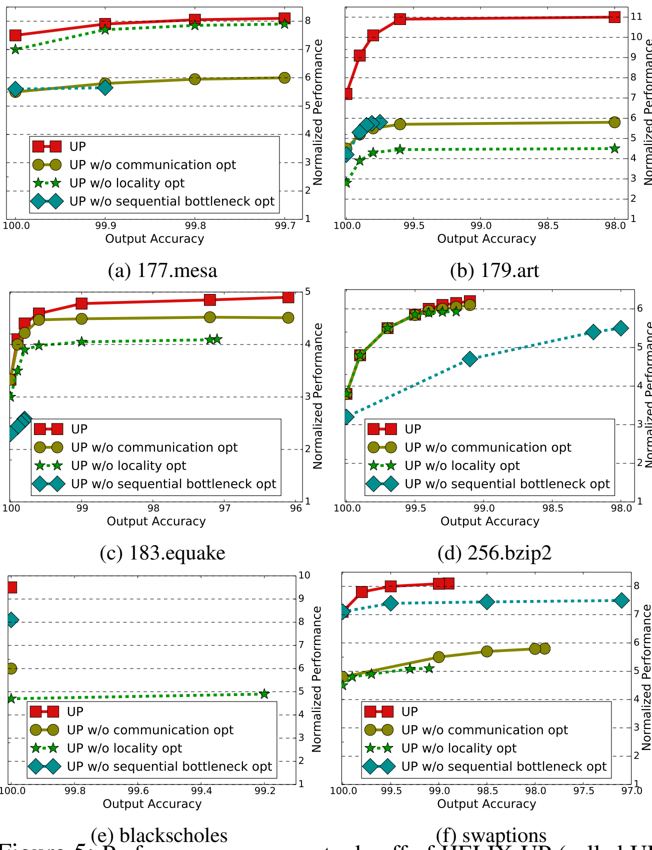


Figure 5: Performance-accuracy trade-off of HELIX-UP (called UP in these figures). Platform: Nehalem

HELIX-UP improves speedup beyond the number of cores for some benchmarks (e.g., 177.mesa). While HELIX always uses an available SMT thread to hide inter-core communication latency [4], the HELIX-UP runtime also uses an SMT thread for computation whenever advantageous.

HELIX-UP outperforms QuickStepLike because the latter does not incorporate semantics-preserving transformations (such as variable vectorization, method inlining, and code versioning) that enable additional parallelism. HELIX-UP applies semantics-preserving optimizations before using those that relax semantics. Section 5.6 further evaluates the importance of retaining semantics-preserving optimizations.

When no output distortion is allowed, neither loop perforation nor SAGE thread fusion improves the performance of its HELIX-parallelized input code. Each approach relies on skipping some computation (e.g., loop iterations), which necessarily changes the output for our six benchmarks. Hence, neither can increase performance over HELIX. The same reasoning applies to sequential loop perforation, the baseline.

Low output distortion. When up to 10% output distortion is allowed, HELIX-UP achieves higher performance for every benchmark (Figure 4b). SAGE thread fusion and loop perforation produced about the same speedups as HELIX-UP, but at the cost of significantly larger output distortion. The reason is that the loops selected by HELIX to maximize parallelism tend to have large iterations. Because thread fusion and loop per-

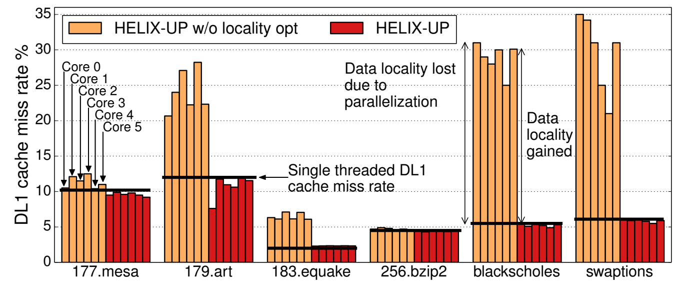


Figure 6: HELIX-UP brings back the data locality lost due to parallelization. Platform used: Nehalem

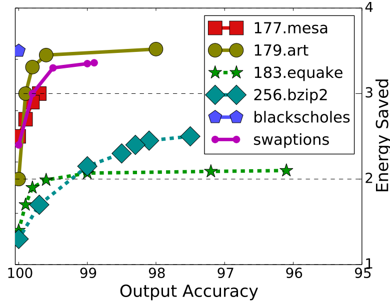


Figure 7: Energy-accuracy trade-off of HELIX-UP. Platform used: Haswell

foration apply to whole loop iterations, the coarse granularity limits their freedom to trade output accuracy for performance.

HELIX-UP significantly outperforms other approaches for 179.art. This is because HELIX-UP only slightly reduces the quality of the output (by sidestepping strict semantics only when necessary at run time). 179.art is particularly sensitive to distortion of the data generated by parallelized loops because poor accuracy during its second phase causes it to use more iterations to converge to a fixed point. This result highlights the importance of tightly controlling distortion during execution. We found that allowing more than 1.5% output distortion for 179.art led to extra iterations during its second phase, which reduced its overall speedup due to parallelization.

5.3 Performance, Energy, Output Accuracy Trade-offs

Figure 5 plots the tradeoff between performance and output accuracy for HELIX-UP.[2] To generate this data, we swept the constraints given as inputs to HELIX-UP while keeping performance as the goal. We ran these experiments on Nehalem.

Conventional wisdom holds that the most important bottleneck in a parallelized program is the sequential component. Figure 5 shows that this is not the case for some benchmarks. To highlight the most important bottleneck for a benchmark, we turned on and off each of the three classes of knobs used by HELIX-UP one at a time. For some benchmarks (e.g., 179.art), the locality lost through parallelization is the main bottleneck, which is orthogonal to the sequential code created by dependences. Other benchmarks follow the conventional wisdom (e.g., 183.equake).

[2] Accuracy is computed as 100 minus the output distortion.

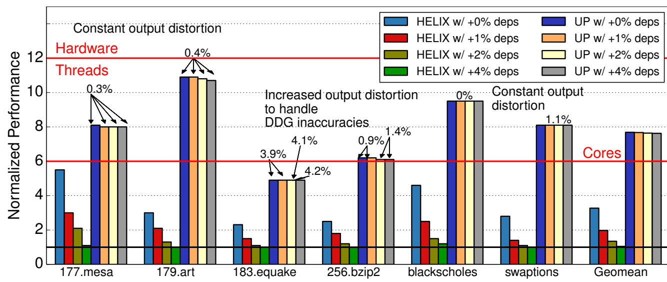


Figure 8: HELIX-UP (called UP in this figure) makes the system tolerant to DDG inaccuracies. This reduces the burden of designing accurate DDG analysis. Platform used: Nehalem

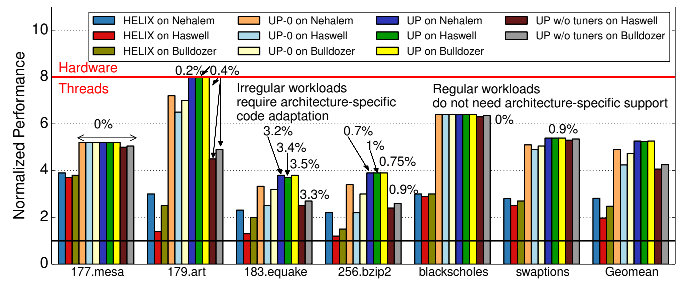


Figure 9: HELIX-UP (called UP in this figure) makes the performance gained consistent across platforms by trading output distortion for performance when necessary. Comparing bars with and without tuners shows the importance of adapting relaxation of program semantics to the characteristics of the target platform. Only eight hardware threads (all 3 platforms have them) are used for each platform.

To further understand the loss of data locality through parallelization, we analyzed the DL1 cache miss rate of all cores. Figure 6 compares the overall DL1 cache miss rate of each core against the baseline (the black line) when HELIX-UP does not optimize for data locality. This data is also representative of the code generated by HELIX. As Figure 6 shows, the DL1 cache miss rate increases significantly for some benchmarks (e.g., 179.art) when the code is parallelized. The figure also shows the DL1 cache miss rate for HELIX-UP when all knobs are used. Thanks to the knob that targets data locality, the DL1 cache miss rate decreases to the level for the baseline, confirming that HELIX-UP eliminates this source of overhead.

There are no redundant semantics-relaxing code transformations in HELIX-UP. Figure 5 shows that disabling any class of knobs decreases performance for at least some benchmarks.

Finally, HELIX-UP saves energy. Figure 7 shows the tradeoff between energy and output accuracy on the Haswell platform, the only one of our experimental processors with RAPL counters, which were used to compute energy consumption.

5.4 Tolerance of Dependence Analysis Weaknesses

HELIX-UP makes a strict parallelizing compiler like HELIX less sensitive to inaccuracies in data dependence analysis. We modeled a less accurate data dependence analysis, compared to the analysis used in HELIX, by adding random dependences to parts of the data dependence graph (DDG) to be parallelized by the compiler (either HELIX or HELIX-UP). Figure 8 plots speedups obtained when different numbers of additional dependences were added. We performed these experiments on the Nehalem platform.

The performance gained by HELIX noticeably drops even with a few additional dependences. In contrast, HELIX-UP's performance remains substantially constant, but output distortion for 183.equake and 256.bzip2 get worse as more dependences are added. This is because the additional dependences led the compiler to merge two sequential segments into one. Before being merged, the two segments had different requirements: one required loop iteration order and the other did not. After the merge, HELIX-UP was forced to relax loop iteration order for the merged sequential segment to preserve performance, which led to higher output distortion.

5.5 Performance on Different Architectures

HELIX-UP achieves consistent performance per core across commodity architectures. Figure 9 plots the performance of HELIX, HELIX-UP-0 (HELIX-UP with no output distortion), and HELIX-UP on our three target platforms. Each workload was compiled separately for the target on which it was tested.

Curiously, HELIX performance drops on Haswell compared to Nehalem. Using microbenchmarks, we found that the latency to move a CPU word between adjacent cores increases from 110 cycles in Nehalem to 190 cycles in Haswell, which accounts for the performance degradation. HELIX-UP, on the other hand, maintains consistent performance by trading off output accuracy to reduce inter-core communication overhead.

To further illustrate this tradeoff, Figure 9 also plots HELIX-UP's performance when constrained to avoid output distortion (HELIX-UP-0). Performance decreases, but not as much as for HELIX, thanks to semantics-relaxing transformations that turned out to be correct at run time.

Finally, it is important to tune the program for the platform on which it is been installed. To demonstrate that, Figure 9 includes performance results when HELIX-UP's tuners were not used and the parallel code was generated for Nehalem. Recall our microbenchmarks show Nehalem has lower core-to-core communication latency than Nehalem, resulting in higher thresholds to trigger relaxation. Hence, without tuners, HELIX-UP's performance on Haswell degrades because the HELIX-UP runtime was not able to adapt to running on Haswell.

5.6 Importance of Retaining Semantics-Preserving Transformations

HELIX-UP needs semantics-preserving code transformations to efficiently extract parallelism. Figure 10 compares HELIX-UP with the case in which code transformations included in HELIX (e.g., variable vectorization) were not used. As noted previously for QuickStepLike, HELIX-UP also loses a significant amount of performance if it cannot rely on code transformations designed to help extract parallelism. This is because semantics-relaxing transformations included in HELIX-UP do not effectively replace the semantics-preserving ones used by

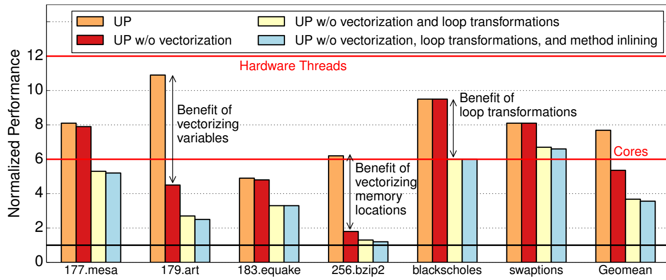


Figure 10: Importance of including both semantics-preserving and semantics-relaxing code transformations. HELIX-UP is called UP in this figure. Platform used: Nehalem

HELIX. For example, HELIX-UP cannot vectorize a variable to avoid dependences.

5.7 Remaining Opportunities

To estimate what remains to be improved, we built several oracles that decide when to relax strict semantics to gain performance. These oracles were computed using a brute force approach. Figure 11 plots the results when HELIX-UP uses these oracles. We first validate the effectiveness of the runtime. Then, we discuss whether the compiler could be improved to eliminate the runtime.

HELIX-UP's use of code produced by semantics-relaxing transformations is effective. As seen by the data labeled "UP w/ runtime oracle for increasing knobs" in Figure 11, replacing the runtime with an oracle only for relaxing knobs has small impact for most benchmarks. However, results for "UP w/ runtime oracle", obtained by using an oracle that entirely replaces the HELIX-UP runtime, exhibit a larger difference. This difference suggests there may be opportunities to increase computation accuracy by improving how the HELIX-UP runtime reverts back to strict mode after relaxing actions have served their purpose and are no longer needed.

Finally, the HELIX-UP runtime is needed to decide when to make compromises. Figure 11 plots results for two additional oracles: the static and "static per knob" oracles. For these two experiments, we removed the HELIX-UP runtime and chose the knobs settings statically. Knobs were set at the beginning of execution and remained unchanged until the end. The oracle "static" chooses only one setting for all knobs of the same type. The oracle "static per knob" chooses one setting for each knob. This last oracle represents the best results we can obtain at static time even if we were to implement advanced code analyses for deciding which knob settings to use. The gap between "static" and HELIX-UP is small only for regular benchmarks, like blackscholes. Hence, the runtime plays a crucial role for less regular benchmarks.

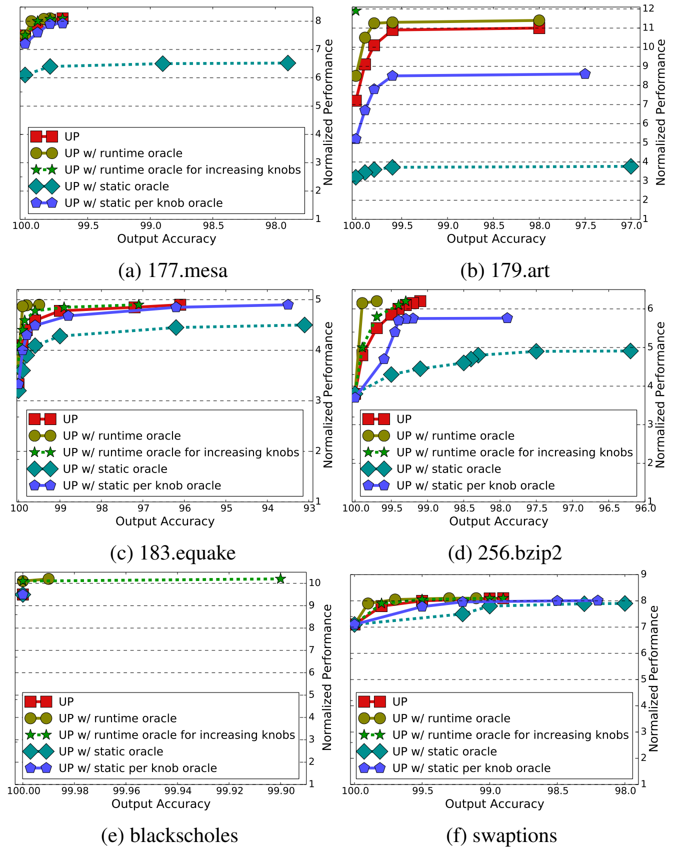


Figure 11: A runtime is needed to choose the best performance-accuracy settings. HELIX-UP is called UP in these figures. Platform used: Nehalem

6. Related Work

6.1 Semantics-Relaxing Code Transformations

Several code transformations have been proposed that relax program semantics for various purposes [7, 8, 17, 26, 27]. None of these extracts TLP from sequentially-designed programs and all of them work by skipping computation, whereas HELIX-UP temporarily ignores dependences when it is advantageous. Approaches that skip computation, like loop perforation [27] and SAGE thread fusion [26], could profitably be combined with HELIX-UP. When low output distortion is required, HELIX-UP provides higher performance because it relaxes the code only long enough to avoid performance bottlenecks. When high output distortion is acceptable, skipping computation provides higher performance by increasing relaxation.

6.2 Automatic Parallelization

Semantics-preserving approaches. Many strategies for extracting parallelism while preserving program semantics have been proposed, starting a half century ago [2, 4, 9, 10, 12, 14, 18, 20, 21, 23, 25, 29, 32]. These approaches rely only on semantics-preserving code transformations. While we have shown their value in HELIX-UP, this paper also shows the value of combining them with semantics-relaxing transformations when perfect output isn't crucial.

Speculation. Several parallelizing compilers rely on thread-level speculation to reduce the cost of dependences that turn out to be false at run time (i.e., apparent dependences) [13, 15, 16, 19, 24, 28, 30, 31, 33, 35]. All of these approaches are still designed to preserve the program semantics. Hence, if a dependence has been misspeculated (i.e., the dependence actually exists), then the execution rolls back, paying the correspondent overhead. While speculative approaches reduce the cost of ap-

parent dependences, they behave like traditional compilers for the actual ones. Instead, HELIX-UP not only erases the cost of apparent dependences, it also reduces the cost of those that are actual, but have low impact on program output.

Relaxing program semantics improves the performance obtained by a hypothetical speculative approach. Executions shown in Figure 4a produce 100% correct output, so no speculative approach could do better than the HELIX-UP bar in that figure. Finally, the increase in HELIX-UP's performance between Figures 4a and 4b is due to relaxing program semantics.

Semantics-relaxing approaches. Recently, a new way of extracting thread-level parallelism that embraces the semantics-relaxing approach has been proposed: QuickStep [22] and ALTER [1]. QuickStep first distributes loop iterations among cores without synchronizing them. Then, it adds synchronizations as needed based on profiling information. Unlike QuickStep, HELIX-UP uses semantics-relaxing transformations to enhance the semantics-preserving ones instead of replacing them. Section 5.6 shows the importance of this strategy.

ALTER distributes loop iterations among cores, deterministically allowing stale reads from a consistent snapshot of the global memory. The parallelization is based on source annotations that can be defined either manually or automatically. ALTER's semantics-relaxing transformations are chosen at compile time. HELIX-UP, on the other hand, observes program behavior at run time to engage semantically relaxed code only for long enough to avoid performance bottlenecks. Moreover, HELIX-UP includes nondeterministic computations with acceptable unsynchronized data races.

7. Conclusion

HELIX-UP is a fully-implemented parallelizing compiler that incorporates both semantics-relaxing and semantics-preserving code transformations. The user of HELIX-UP need not understand the implementation of the program being parallelized, but must be skilled enough to provide representative inputs and an output distortion function. Currently HELIX-UP offers no feedback about how well the properties of production input data have been covered by user-supplied training sets. In future work, we will use lightweight run-time instrumentation to dynamically assess and improve the quality of training inputs that can be used when a program is re-parallelized.

HELIX-UP shows that when representative inputs are available, enhancing a parallelizing compiler with semantics-relaxing transformations yields substantial performance gains. Moreover, future extension of HELIX-UP to include transformations that skip computation, such as loop perforation, will combine benefits of both: HELIX-UP produces high performance gains with low distortion, and skipping computation may boost performance even higher, but with higher distortion.

References

[1] U. Abhishek et al. ALTER: Exploiting breakable dependences for parallelization. In *PLDI*, 2011.

[2] A. Aiken et al. Perfect pipelining: A new loop parallelization technique. *ESOP*, 1988.

[3] S. Campanoni et al. A highly flexible, parallel virtual machine: Design and experience of ILDJIT. *SPE*, 2010.

[4] S. Campanoni et al. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *CGO*, 2012.

[5] S. Campanoni et al. HELIX: Making the extraction of thread-level parallelism mainstream. In *IEEE Micro*, 2012.

[6] S. Campanoni et al. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *ISCA*, 2014.

[7] M. Carbin et al. Detecting and escaping infinite loops with jolt. In *ECOOP*, 2011.

[8] M. Carbin et al. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[9] D-K. Chen et al. On effective execution of nonuniform DOACROSS loops. *IEEE TPDS*, 1996.

[10] D-K. Chen et al. Redundant synchronization elimination for DOACROSS loops. *IEEE PDS*, 1999.

[11] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. *ICPP*, 1986.

[12] K. Ebcioglu et al. A global resource-constrained parallelization technique. In *ICS*, 1989.

[13] S. J. Gregory et al. The STAMPede approach to thread-level speculation. In *ACM TCS*, 2005.

[14] Jialu H. et al. Decoupled software pipelining creates parallelization opportunities. *CGO*, 2010.

[15] L. Hammond et al. The Stanford Hydra CMP. In *IEEE Micro*, 2000.

[16] L. Han et al. Speculative parallelization of partial reduction variables. In *CGO*, 2010.

[17] H. Hoffmann et al. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[18] A. Hurson et al. Parallelization of DOALL and DOACROSS loops - a survey. *Advances in Computers*, 1997.

[19] T. A. Johnson et al. Speculative thread decomposition through empirical optimization. In *PPoPP*, 2007.

[20] D. Liu et al. Optimal loop parallelization for maximizing iteration-level parallelism. 2009.

[21] K. S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *ICS*, 1994.

[22] S. Misailovic et al. Parallelizing sequential programs with statistical accuracy tests. *ACM TECS*, 2013.

[23] G. Ottoni et al. Automatic thread extraction with decoupled software pipelining. *MICRO*, 2005.

[24] A. Raman et al. Speculative parallelization using software multi-threaded transactions. *ASPLOS*, 2010.

[25] E. Raman et al. Parallel-Stage decoupled software pipelining. *CGO*, 2008.

[26] M. Samadi et al. SAGE: Self-tuning approximation for graphics engines. In *MICRO*, 2013.

[27] S. Sidiroglou-Douskos et al. Managing performance vs. accuracy trade-offs with loop perforation. In *ESEC/FSE*, 2011.

[28] J. G. Steffan et al. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA*, 1998.

[29] G. Tournavitis et al. Towards a holistic approach to auto-parallelization. *PLDI*, 2009.

[30] C. Wang et al. Dynamic parallelization of single-threaded binary programs using speculative slicing. In *ICS*, 2009.

[31] L. Wei et al. POSH: A TLS compiler that exploits program structure. In *PPoPP*, 2006.

[32] C. Z. Xu et al. Time stamp algorithms for runtime parallelization of DOACROSS loops with dynamic dependences. *TPDS*, 2001.

[33] A. Zhai et al. Compiler and hardware support for reducing the synchronization of speculative threads. In *ACM TACO*, 2008.

[34] Y. Zhai et al. Happy: Hyperthread-aware power profiling dynamically. In *USENIX ATC*, 2014.

[35] H. Zhong et al. Uncovering hidden loop level parallelism in sequential applications. *HPCA*, 2008.