HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing

Simone Campanoni
Harvard University
Cambridge, USA
xan@eecs.harvard.edu

Timothy Jones
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

Glenn Holloway
Harvard University
Cambridge, USA
holloway@eecs.harvard.edu

Vijay Janapa Reddi
The University of Texas at Austin
Austin, USA
vj@ece.utexas.edu

Gu-Yeon Wei
Harvard University
Cambridge, USA
guyeon@eecs.harvard.edu

David Brooks
Harvard University
Cambridge, USA
dbrooks@eecs.harvard.edu

ABSTRACT

We describe and evaluate HELIX, a new technique for automatic loop parallelization that assigns successive iterations of a loop to separate threads. We show that the inter-thread communication costs forced by loop-carried data dependences can be mitigated by code optimization, by using an effective heuristic for selecting loops to parallelize, and by using helper threads to prefetch synchronization signals. We have implemented HELIX as part of an optimizing compiler framework that automatically selects and parallelizes loops from general sequential programs. The framework uses an analytical model of loop speedups, combined with profile data, to choose loops to parallelize. On a six-core Intel® Core™ i7-980X, HELIX achieves speedups averaging 2.25×, with a maximum of 4.12×, for thirteen C benchmarks from SPEC CPU2000.

1. INTRODUCTION

Now that multicore processors are commonplace, there is a need for compilers that produce multi-threaded code from ordinary, irregular, single-threaded sources. There have been exciting demonstrations that ordinary irregular programs can be speeded up by compiling loops to multiple threads that run in parallel on a chip multiprocessor [19, 23, 30, 32, 34]. Much of this work is based on decoupled software pipelining (DSWP), which uses the structure of a loop's dependence graph to divide it into threads without cyclic dependences [30]. With the aid of architectural extensions for inter-thread buffering, such thread pipelines have been shown to speed loops up substantially. However, DSWP is inherently constrained by the static structure of each loop's body, which limits the number of hardware threads that can usefully be applied to the loop [34]. Recent work has used DSWP to expose other loop parallelization opportunities [19, 23, 32], and these hybrid approaches produce significant speedups on real systems when applied to selected benchmarks by hand, but they have yet to be automated.

For loops with no data dependences between iterations, a straightforward parallelization approach that scales well is to implement each iteration as a separate thread. (This has been called DOALL parallelism.) But for general loops, the costs of inter-thread communication have discouraged compiler writers from simply allocating a thread per iteration. The loop-carried dependences between iterations not only mean that computed results must be forwarded between threads, but also that some sections of a loop's body must be executed in sequential order, which requires synchronization. However, as we will show, when the right loops are selected for parallelization, the cost of transferring computed results is a small fraction of total overhead. And the inter-thread signaling needed for synchronization can be minimized both by code optimization and by using simultaneous multi-threading (SMT). Based on these observations, we developed a new technique for loop parallelization called HELIX,[1] which distributes the iterations of selected loops across a pool of threads bound to a ring of cores. The simplicity of the HELIX concept enables the use of a simple but accurate analytical model to estimate the speedup obtained by parallelizing a single loop. A heuristic, based on that model and on feedback from profiling runs, provides an efficient and effective algorithm for selecting loops to parallelize. We have developed new code analyses and transformations that minimize the number of signals required. We further reduce signal latency through SMT by giving each iteration thread a helper thread [26] to ensure that inter-core transmission of each signal begins as soon as it is sent. Where dependences force loop segments to execute in iteration order, we transform the parallelized code to reduce the size of such *sequential segments* and to run distinct sequential segments in parallel whenever possible.

HELIX is a *fully automatic* technique; it does not depend on source code annotations or modifications, or other forms of human intervention. It can achieve significant speedups of regular and irregular workloads on a real multicore processor. HELIX is the first technique using thread-level parallelism (TLP) among loop iterations that is able to achieve significant speedups without constraints such as a loop nesting limit, regular memory accesses, or regular control flow [4]. For thirteen C language benchmarks from the SPEC CPU2000 suite, the geometric mean of the resulting speedups on our six-core CPU is 2.25×, with a maximum of 4.12×.

The main contributions of the paper are: (1) a new general purpose parallelization technique, including a new set of code analyses and transformations, that is able to significantly reduce both the number of signals sent between threads and the TLP loss due to sequential segments; (2) a new approach that exploits SMT technology in a real system to further reduce signaling overhead; (3) an effective feedback-directed heuristic for selecting loops to parallelize (also useful for parallelization frameworks other than HELIX); (4)

[1] Helical Execution of Loop Iterations across cores.

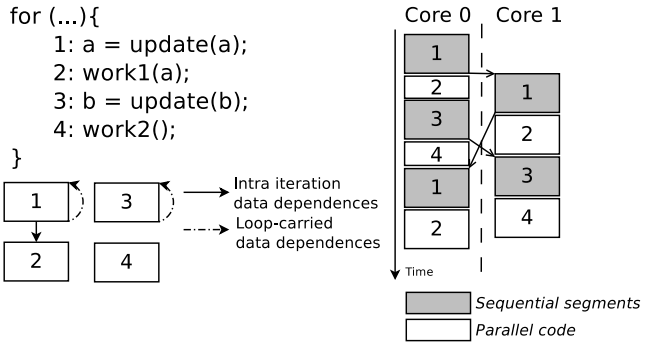

Figure 1: Execution of code produced by HELIX for a dual-core processor. Note that code blocks 1 and 3 must each be executed sequentially, but since they are independent, HELIX overlaps them in time.

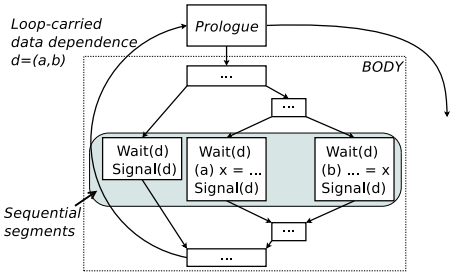

Figure 2: Insertion of $Wait(d)$ and $Signal(d)$ due to a RAW data dependence $d = (a, b)$. Empty sequential segments handle dependences that span multiple iterations.

a demonstration that the amount of data to forward between loop iterations is small when loops are carefully chosen; and (5) an experimental evaluation on a broad range of applications.

The rest of the paper is arranged as follows. We describe the HELIX transformations and how to choose the right set of loops to parallelize in Section 2. Section 3 presents our experimental setup and HELIX's speedups on the SPEC CPU2000 suite. Section 4 compares HELIX with related work. Finally, Section 5 presents our conclusions.

2. HELIX

Loops parallelized by HELIX run one at a time. The iterations of each parallelized loop run in round-robin order on cores of a single processor. HELIX applies code transformations to minimize the inefficiencies of sequential segments, data transfer, signaling, and thread management.

HELIX executes different sequential segments in parallel whenever possible, as shown in Figure 1, where sequential segments 1 and 3 overlap. To further reduce the effect of sequential segments on TLP, HELIX automatically chooses the most profitable loops to parallelize, which are usually those with the least code in sequential segments, and then it transforms the chosen loops to minimize their sequential code (see Step 5 of Section 2.1).

Transferring data between cores can be a significant source of overhead. However, as we show in Section 3, careful selection of the loops to parallelize keeps the amount of data needing to be forwarded between cores small, compared with the amount consumed within each iteration. To understand why, consider the code in Figure 2, which shows the structure of most sequential segments in our chosen loops. We show a read-after-write (RAW) dependence, the

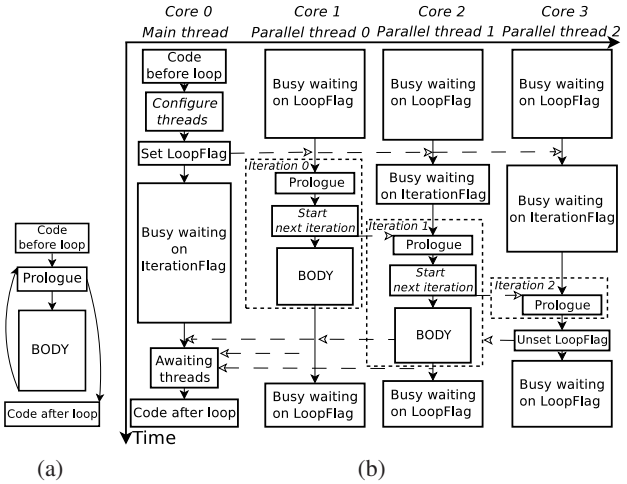

Figure 3: (a) Normalization of a loop so that it can be parallelized by HELIX. (b) Overview of the execution of two loop iterations. The main thread executes outside loop code and configures parallel and helper threads. These run the loop iterations in parallel and notify the main thread once finished. The main thread then executes the sequential code after the loop. Helper threads are not shown for simplicity.

only kind that can lead to a data transfer. Here we have three blocks of code that are executed sequentially because of a data dependence between instructions a and b. For successive loop iterations i and $i + 1$, running on cores c_i and c_{i+1} respectively, data needs to be transferred from c_i to c_{i+1} only if iteration i executes a and iteration $i + 1$ executes b. Assuming that each branch in the loop body has equal probability of being taken or not, this would happen only 6.25% of the time.

The next source of inefficiency is signaling threads to notify them of events that have occurred (e.g., the end of a sequential segment). This is the main source of overhead for HELIX, and it is addressed by the code analyses and transformations described below. The number of signals sent in the system is minimized by exploiting redundancy among them and by transforming the code. Also, HELIX reduces the perceived signal latency by exploiting the SMT technology of the underlying platform. It couples each thread with a *helper thread* running on the same core. Helper threads mask a significant fraction of signal latency by "prefetching" signals. Moreover, HELIX schedules code to support this dynamic prefetching mechanism by separating signals in time.

Finally, HELIX efficiently organizes threads so that one main thread executes the code outside of parallelized loops. Within parallelized loops the main thread is joined by parallel and helper threads which are used as a pool to execute the separate loop iterations and prefetch signals respectively. The main and parallel threads are called *iteration threads*. To avoid overhead, the mapping of all threads to available cores is performed when the application starts and does not change. Figure 3(b) shows the resulting execution of a two-iteration loop on a processor with four cores. After configuring all parallel and helper threads, the main thread acts like a parallel thread. In this example, however, there are fewer iterations than threads, so the main thread does not need to execute any loop iterations. Once the loop is complete, parallel and helper threads wait for the next parallel loop to execute, and the main thread continues sequential execution of the program. When targeting N cores, HELIX produces $2N$ threads composed of one main thread, $N - 1$ parallel threads and N helper threads.

2.1 Algorithm for Parallelizing One Loop

HELIX works on an intermediate representation of a program. It takes as input a loop L to be optimized and modifies it as a side effect. After putting L into a normal form, it inserts code to satisfy control and data dependences, creating the sequential segments. Sequential segments are optimized first by shortening them through code scheduling, and then by elimination of redundant signals between cores. HELIX inserts instructions to forward data values between cores and to implement signals for synchronization. It then couples each iteration thread with a helper thread to prefetch signals dynamically. To support this prefetching of signals, the code of the iteration thread is rescheduled to space its outgoing signals more uniformly in time. Finally, HELIX links the parallelized loop back into the original program.

Step 1: Normalizing the loop. HELIX transforms loops by putting them into the form shown in Figure 3(a). A normalized loop has two regions, the *prologue* and the *body*. The prologue is the minimum set of instructions that must be executed to determine whether the next iteration's prologue will be executed. Formally, the prologue includes loop instructions that are not post-dominated by the loop's back edge in the control flow graph (CFG). Exits from the loop can only originate in the prologue. In general, this is a non-trivial subgraph of the CFG. The body is the rest of the loop. Notice that the body is composed of both sequential segments and code that can run in parallel. The transformed code resembles a *while* loop in which there are no instructions in the body that jump outside the loop.

Step 2: Identifying data dependences to satisfy. To detect data dependences, HELIX applies interprocedural pointer analysis to the whole program [17]. In order to avoid unnecessary synchronization, HELIX satisfies only the necessary loop-carried data dependences. These are a subset of loop-carried data dependences, where false (i.e., write-after-write and write-after-read) dependences either through registers or the call stack are excluded, because each iteration is executed on a separate core with its own internal registers and call stack that are not exposed to other threads.

Data shared between threads includes live-in and live-out values (those produced outside the loop but consumed inside it and *vice versa*), and loop iteration live-in values (those produced by one loop iteration and consumed by another) [30]. Variables that hold these values are called *loop boundary live variables*. Memory locations of loop boundary live variables are kept within the allocation frame of the main thread. Accesses from parallel threads are performed through loads and stores and managed by synchronizing the loop-carried data dependences that involve them. Data dependences that result from reading and writing invariant or induction variables do not need to be synchronized. Invariant variables do not change between iterations and induction variables are locally computable from the iteration number and their values at the beginning of the loop [6]. D_{Data} is the set of dependences to synchronize.

Step 3: Starting next iterations. Iterations execute in a non-speculative manner, each starting in a parallel thread once the prologue of the preceding iteration has been executed. To satisfy the loop's control dependences, iteration $i + 1$ starts only when it is certain that its prologue (at least) will be executed. That is known at the beginning of the body of iteration i (because the body cannot contain exits). Therefore, instructions to start the next iteration are added to the beginning of the body, which results in overlapped execution of bodies from different loop iterations. Note that HELIX does not require advance knowledge about the number of loop iterations to execute. However, when the number of iterations is known, HELIX produces a prologue that requires neither signals nor data to be transferred from previous iterations.

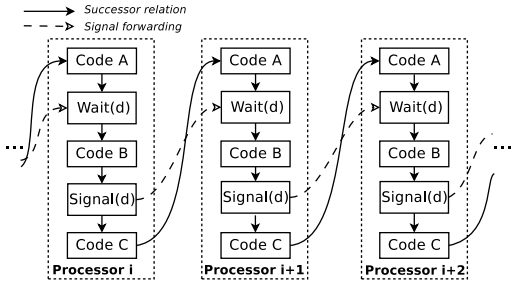

Figure 4: Signal forwarding across processors due to a loop-carried data dependence.

Step 4: Computing sequential segments. For every data dependence, $d = (a, b) \in D_{Data}$, HELIX inserts $Wait$ and $Signal$ operations to ensure that occurrences of a and b in separate loop iterations will execute in the correct order. The instruction $Wait(d)$ blocks execution of a thread until its previous thread sends a signal (i.e., a *data signal*), by executing $Signal(d)$. The implementation of $Wait$ and $Signal$ in our testbed is described below. The effect of these operations is that the code between $Wait(d)$ and $Signal(d)$ executes in loop iteration order, satisfying the data dependence d. For example, in Figure 4 the block *Code B* runs in loop iteration order while *Code A* and *Code C* execute in parallel.

For a data dependence $d = (a, b)$, HELIX inserts $Signal(d)$ at the earliest point along every path through the loop body at which it is provable that neither a nor b can be reached during the rest of the current iteration. These points are found using data flow analysis. Before either a or b is executed, HELIX ensures that all past iterations have already executed them, by inserting $Wait(d)$ before each of them. It also makes sure that $Wait(d)$ is executed before every $Signal(d)$ in the iteration, even if neither a nor b happens to be executed. That means that the next iteration is unblocked only if no previous iterations (adjacent or distant) have yet to execute either a or b. Therefore, data dependences between non-adjacent iterations are handled by sending signals via the threads for intervening iterations. Figure 2 shows typical sequential segments for a dependence between instruction a, which writes `x`, and instruction b, which reads `x`. Note that an actual data transfer occurs only when instruction b is reached, and then only if a was executed more recently than b. Compared with synchronization, data transfer happens infrequently.

Step 5: Minimizing sequential segments. HELIX applies a method inlining and code scheduling pass in order to keep sequential segments small. If there is a data dependence between a function call within a loop and another instruction in that loop, this pass inlines the function call, provided it is not contained in a subloop (since that would inhibit shrinking the sequential segment). This heuristic strikes a good balance: it is conservative enough to avoid code blow-up that could thwart speedups in applications like the SPEC benchmarks, but it is liberal enough to optimize all the loops we parallelized with HELIX as tightly as they could be with unconstrained inlining.

After method inlining, the code is scheduled. For every sequential segment containing data dependence $d = (a, b) \in D_{Data}$, instructions that are not directly or indirectly dependent on either a or b are moved after the segment. This reduces the problem shown in Figure 5. In this example, the parallel code is not well balanced across loop iterations, which causes unnecessary stalls if the sequential segment is kept at the end of the loop. For the same reason, the remaining instructions belonging to the sequential segment are

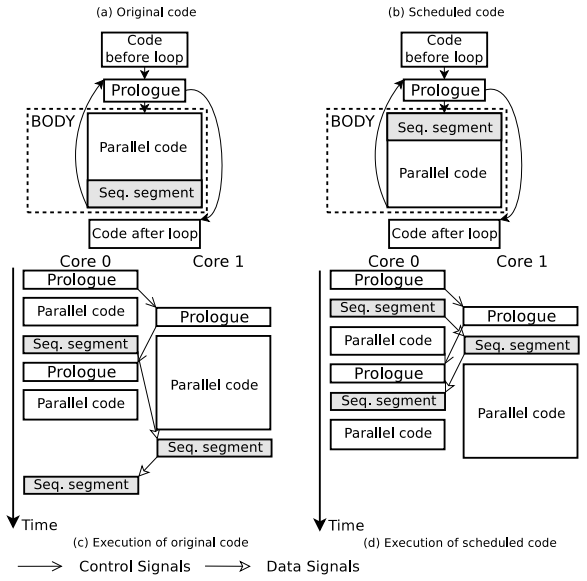

Figure 5: Reducing execution stalls by percolating sequential segments upwards.

moved up the control flow graph as much as possible.

Step 6: Minimizing signals. HELIX applies code optimizations that remove the redundancies introduced by naïve insertion of $Wait$ and $Signal$ operations. It eliminates redundant $Wait$s through data flow analysis. If every control path that leads to $Wait_i$ contains another, $Wait_j$, from the same dependence, then $Wait_i$ is redundant. Moreover, since sequential segments are percolated upwards by Step 5, it may be possible to merge some of them to reduce the overall number of signals. HELIX merges sequential segments if they can be scheduled without parallel code between them.

Finally, HELIX uses the synchronization of one dependence, say d_j, to synchronize another, say d_i, provided d_i is redundant due to d_j. A data dependence d_i is redundant due to d_j if $Wait(d_j)$ is available (in the data flow sense) at every occurrence of $Wait(d_i)$. This redundance relation can be viewed as a graph, called the *data dependence redundance graph*, whose nodes are the data dependences of a loop. This graph contains an edge $d_j \rightarrow d_i$ if and only if d_i is redundant due to d_j. The following theorem specifies which subset of dependences need to be synchronized.

THEOREM 1. *Let $G = (N, E)$ be a data dependence redundance graph and let $N_{to\text{-}synch} \subseteq N$ be the set of dependences that includes every node without incoming edges and one node per cycle of G. Synchronizing the set $N_{to\text{-}synch}$ synchronizes the entire set of dependences N.*

Thanks to the above theorem, HELIX can keep only the $Wait$s that are related to $N_{\text{to-synch}}$.

Step 7: Inserting inter-thread communication. The synchronization required for loop-carried dependences is implemented using a per-thread memory area called the *thread memory buffer*, which resides in the system's shared memory. Threads are bound to specific cores and they execute parallel loop iterations in round-robin order. A thread uses its thread memory buffer to receive signals from the thread that is executing the preceding iteration. For signalling purposes, a thread can only read from its own memory buffer and can only write to the memory buffer of the successor thread (the one that will execute the next loop iteration). The successor/predecessor relation of parallel threads is statically fixed and the last thread writes to the first thread's buffer, thereby creating a cycle of reads and writes through the buffers. The buffer of a given thread is initialized by its predecessor before that thread is unblocked by a *control signal* (i.e., a store to a per-thread memory location called `IterationFlag`).

Loads and stores are inserted into the code to forward data produced and stored in loop boundary live variables. The $Wait$ and $Signal$ operations are implemented using simple loads and stores. Depending on the memory consistency model of the underlying platform, memory barriers may need to be added before the loads and after the stores. However, this is not required on our Intel-based evaluation system (see Section 2.3).

Step 8: Coupling with helper threads. When a core, say c_i, sends a signal to another, say c_j, it writes a value to a designated memory location in the thread memory buffer, which places it into c_i's first private cache. The value is not forwarded to c_j's private cache until c_j issues the corresponding $Wait$ instruction (i.e., a load operation). It then takes several cycles for c_j to receive the value (110 clock cycles in our testbed). The caches act as a pull system. When cores have SMT capabilities, HELIX transforms the caches into a push system for signaling, where c_i pushes the signal to c_j as soon as it is produced. This leads to a significant reduction in c_j's waiting time. HELIX mimics a push memory system on top of a pull one by coupling an additional *helper thread* to each iteration thread running on the same core. Helper threads are in charge of prefetching signals for their coupled iteration threads.

In Step 4, HELIX inserts $Wait$s into the code such that every path through the CFG contains all $Wait$ instructions. An example is shown in Figure 2. The code executed by the helper threads is a straight line of $Wait$s, one per sequential segment, where a $Wait(d_i)$ is executed after $Wait(d_j)$ if $Wait(d_j)$ is available just before $Wait(d_i)$.

Since a given helper thread can prefetch one signal at a time, HELIX schedules the code executed by iteration threads to space sequential segments evenly throughout the loop, until enough code is inserted to achieve even signal prefetching. The algorithm for this scheduling is described in Figure 6. To understand its impact, consider Figure 7 which shows the run-time execution of a parallelized loop that includes three sequential segments, SS1, SS2 and SS3. The first case, "No prefetching", represents the execution of the code produced by Step 7. In this case, every signal takes S clock cycles to be transferred between cores, because the signal forwarding starts only when the receiver tries to enter the corresponding sequential segment. The second case, "Prefetching without balancing", shows execution when helper threads are used without the scheduling algorithm of Figure 6. In this case, only the interval between SS3 and SS1 is long enough to prefetch the signal, so only signals coming from SS1 are fully prefetched; the others are just slightly prefetched. The last case, "Prefetching with balancing", shows how prefetching signals evenly leads to a better use of idle core clock cycles. Note that when helper threads are used (the second and third cases), prefetching a signal starts as soon as the previous one has been prefetched or the previous iteration leaves the corresponding sequential segment. Finally, note that the overall time spent executing parallel code is constant over these three cases ($A + B + C$).

As input, the code scheduling algorithm of Figure 6 takes the loop to schedule and the platform-dependent latencies for both an unprefetched and a fully prefetched signal. Initially, the code belonging to loop `L` that can run in parallel is untagged (line 2). The algorithm runs until there is no more untagged parallel code (check

```
1  helper_thread_schedule(L,max_latency,min_latency){
2   untag_parallel_code(L);
3   delta = max_latency - min_latency;
4
5   while (untagged_parallel_code(L) > 0){
6     DISTS = compute_sequential_segments_distances(L);
7     D_j_j+1 = max_distance_below(DISTS, delta);
8     if (D_j_j+1 == 0) break ;
9     DISTS = remove_distance(DISTS, D_j_j+1);
10    D_k_k+1 = max_distance_below(DISTS, delta);
11    j = delta - D_j_j+1;
12    k = delta - D_k_k+1;
13    add = min(1, j - k);
14    SSs = get_sequential_segments(D_j_j+1);
15    add_code_between_sequential_segments(L,SSs,add);
16    tag_code_between_sequential_segments(L,SSs);
17  }
18 }
```

Figure 6: Code scheduling applied to iteration threads to balance signal prefetching by helper threads.

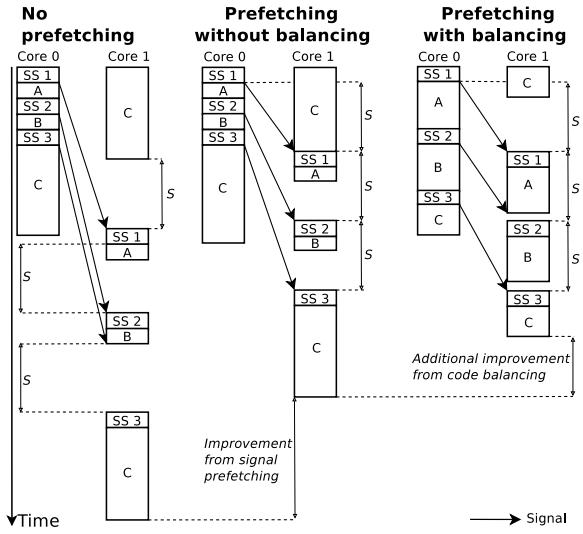

Figure 7: Importance of balanced prefetching.

performed in line 5) or all pairs of sequential segments have enough space between them (check performed in line 8). The distance between sequential segments is computed by estimating the number of clock cycles needed to execute the longest path between them (line 6). Ideally, the algorithm separates every pair of sequential segments enough to allow full prefetching of signals. A signal can be fully prefetched if the code between sequential segments executes for more cycles than the difference between the two input latencies (i.e., `delta`). Since we want to prefetch signals evenly, the algorithm considers the two closest sequential segments (i.e., `j,j+1`) (line 7). In order to prevent greedy placement of parallel code, the algorithm considers the next two closest sequential segments as well (i.e., `k,k+1`) (lines 9-10). Untagged parallel code is moved between the closest sequential segments to increase their run-time spacing by a minimum of `1`, and a maximum of the difference between the two pairs of sequential segments (lines 11-15). The parallel code moved is then tagged to avoid moving it again later (line 16).

Step 9: Merging parallel loops inside the program. The function that contains a parallel loop might or might not be called within a loop that is running in parallel. Since only one loop can run in parallel at a time, HELIX keeps the sequential version of each parallel loop to use in case another one is already running in parallel. A conditional branch is inserted in a pre-header placed just before the two versions. Execution is directed to the parallel loop if no other parallel loop is executing (determined by a global variable). Otherwise execution proceeds with the sequential version of the loop.

When a loop has multiple successors, HELIX associates a unique value with each exit path of its prologue. The thread that executes the final iteration sets an exit variable to the value for the exit path taken. When all parallel threads have finished, the main thread checks this variable and jumps to the correct successor.

2.2 Choosing Loops to Parallelize

Transforming a loop into parallel threads can speed up loop execution, provided the benefits of concurrency outweigh the costs of inter-thread communication. The selection of loops to parallelize must be guided by the nature of the loops, the profile of the program given realistic inputs, and the properties of the transformation. Our approach devotes all cores of a processor to one parallelized loop at a time, so multiple independent loops cannot run concurrently, and loops nested within a parallel loop cannot be selected for parallelization. Therefore, selecting the most profitable loops to transform (i.e., loops that, if parallelized, best speed up the program) is critical for achieving significant speedups.

Different sets of loops correspond to separate solutions to the problem. To choose the best set of loops, we need to compare these solutions, by estimating and comparing their speedups. Since the number of possible solutions grows exponentially with the number of loops in the program, our approach constrains the number of sets considered in order to find a solution within reasonable time.

Our heuristic identifies the most promising loops to parallelize by using a graph we call the *dynamic loop nesting graph*. A simple model based on Amdahl's law drives the search. Since loops are analyzed and executed one at a time, without merging or splitting, we use a model that computes the speedups of single loops.

Speedup model. Amdahl's law [5, 18] describes the effect of applying N cores in parallel to a program that executes sequentially in unit time. However, parallelization of a loop can add significant overhead. Therefore, in choosing loops, we incorporate overhead in Amdahl's law to produce the following speedup model for parallelization of a program: $\text{Speedup}(P,N,O) = \frac{1}{(1-P)+\frac{P}{N}+O}$, where O is the added overhead (e.g., due to thread synchronization). Since by hypothesis different parallelized loops cannot run simultaneously, we have $P = \sum_{i=1}^{Loops} P_i$ and $O = \sum_{i=1}^{Loops} O_i$, where $Loops$ is the number of parallelized loops, P_i is the relative time spent running the code of loop i outside its sequential segments (code regions executed sequentially to preserve program semantics) and O_i is the added overhead for that loop. Terms P_i and O_i depend on the specific code transformation technique used to parallelize the loops. We later describe how we obtain those factors for the HELIX parallelizer.

Loop selection algorithm. Our algorithm for choosing the best loops to parallelize uses the speedup model described above, together with a dynamic loop nesting graph. The scope of our analysis is the whole program (hence multiple functions), and we consider a loop within a function called within a loop to be a subloop of the latter [6]. We extend the well-known loop nesting tree to have program-wide scope, and we call it the *static loop nesting graph*. It is not a tree because a function can have multiple callers. The *dynamic loop nesting graph* is a subgraph of the static graph that omits edges not traversed during profiling execution. For example, Figure 8 shows a fragment of the dynamic loop nesting graph from the SPEC CPU2000 benchmark 179.art. The graph is not a tree because function *reset_nodes*, which includes two loops, is called

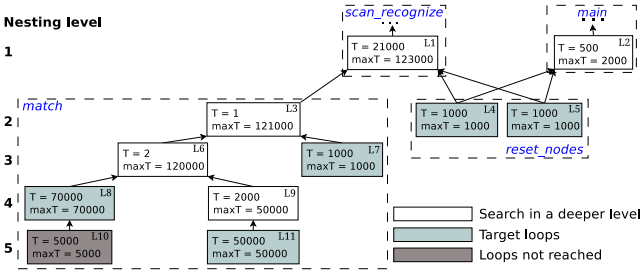

Figure 8: Loop selection algorithm applied to benchmark 179.art. The graph is the result of applying the HELIX speedup model, assuming one clock cycle per signal.

both by a loop from *main* and by a loop from *scan_recognize*.

The loop selection algorithm adds two attributes to each node (loop) in the dynamic loop nesting graph, which are set just before running the algorithm: the *saved time* (`T`) and the *maximum saved time* (`maxT`). `T` is computed by exploiting the speedup model; it is set to the difference between the time spent in the sequential version of the loop and that in the parallel version. If no speedup is achieved, then `T` is 0. This attribute does not change during the execution of the algorithm. The `maxT` attribute provides information about the time that could be saved by parallelizing either the current loop or the best combination of its subloops. Initially, `maxT` is set equal to `T`. The loop selection algorithm has two phases. The first propagates `maxT` through the dynamic loop nesting graph. The second uses `maxT` to identify loops to parallelize.

The first phase of the algorithm propagates `maxT` through the graph from inner to outer loops. If the sum of the `maxT` attributes of a loop's subloops exceeds its current `maxT` value, then that sum becomes its new `maxT` attribute. This step repeats throughout the dynamic loop nesting graph until a fixed point is reached.

The second phase of the algorithm starts from the outermost loop nodes and searches downward until it reaches a node with `maxT` is equal to `T`. The algorithm works top-down because by going deeper in the nesting graph, we lose code to parallelize. Ideally we would choose outermost loops. However, time saved by parallelization can increase by going deeper because the fraction of time spent running in parallel might increase as sequential segments become smaller. Every time there is a split in the path (when a loop has multiple subloops), the search continues in all directions. Nodes that terminate the search with `maxT` greater than zero represent the loops selected for parallelization. The rationale behind this phase is that there is no point in going below a node for which the time saved by parallelization equals or exceeds that for any combination of subloops.

Figure 8 shows the dynamic loop nesting graph after both phases of the algorithm have been applied. Note that parallelized loops belong to different nesting levels; in general, choosing loops at a fixed level (e.g., outermost loops) is not the best solution. Consider loops L8 and L9, which belong to the fourth loop nesting level. L8 is a better choice than its child L10 because the latter has a smaller body (i.e., less code to parallelize). On the other hand, even though L9 embraces more code than its child L11, the size of its sequential segments are larger than those of L11, so L11 has a larger `T` than L9. Since L11 has more code to run in parallel than L9, it is the more profitable loop to choose.

HELIX speedup model. By using generic parameters P_i and O_i, the speedup model described above abstracts away details of the specific parallelization technique applied. After profiling the original program, we can compute those parameters specifically for HELIX. P_i is the time spent in the body of the i-th parallelized loop outside its sequential segments. O_i, the overhead of that loop, is

$$O_i = Conf_i + Sig_i \times S + \left\lceil \frac{Bytes_i}{CPU_{word}} \right\rceil \times M \qquad (1)$$

where $Conf_i$ is the time spent to configure loop i (e.g., initialize thread memory buffers), Sig_i is the overall number of signals sent inside loop i, which is equal to $C\text{-}Sig_i + D\text{-}Sig_i + ((N-1) \times 2 \times Invoc_i)$ where $C\text{-}Sig_i$ and $D\text{-}Sig_i$ are the overall number of signals sent to satisfy control and data dependences within loop i respectively, and $Invoc_i$ is the number of times loop i is executed. For every invocation of loop i, HELIX sends $N-1$ signals to start or stop parallel threads. These occur just after the loop is configured and just after its last iteration is executed. Hence, $((N-1) \times 2 \times Invoc_i)$ is the overall number of signals sent to start and stop parallel threads for loop i. Factor S is the time spent per signal, which is assumed to be constant. Finally, $Bytes_i$ is the number of bytes transferred between loop iterations, CPU_{word} is the number of bytes of a CPU word and M is the time spent to transfer a CPU word between cores.

$Invoc_i$ is profiled from the original program and $Conf_i$ is profiled by adding the extra instructions needed to configure loop i to the original program just before loop i. Finally, $C\text{-}Sig_i$ and $D\text{-}Sig_i$ are computed by profiling the overall number of iterations of loop i: $C\text{-}Sig_i$ is equal to this number and $D\text{-}Sig_i$ is equal to the same number times the number of sequential segments per iteration in that loop, which is statically known.

2.3 HELIX and Memory Consistency

HELIX depends on memory consistency [2] and communication order. Some multicore processors guarantee that stores to memory performed by a single core are seen in the same executing order by others. If this memory consistency is not enforced by hardware, then HELIX adds memory barriers to enforce it. Since memory loads and stores are used for synchronization, the communication order discipline requires that stores to shared data locations come before the store of the synchronizing flag in the thread producing the shared values, and on the consuming side, the synchronizing flag is loaded and tested before any of the shared data. For control dependences, shared data is the contents of the thread memory buffer, and the synchronizing flag is `IterationFlag` (Step 7 from the HELIX algorithm). Note that this synchronization scheme works because the communication model of HELIX ensures that there can be only one reader and one writer per memory location at the same time. In this paper, we assume a single-chip Intel machine that provides total store ordering (see Section 8 of the Intel Developer's Manual [1]), which is enough for our purposes. Therefore, in our testbed, HELIX satisfies both control and data dependences by using loads and stores. On the other hand, if this memory consistency did not exist, HELIX would need to insert memory barriers after synchronizing stores and before synchronizing loads.

3. EVALUATION

This section evaluates HELIX using the C language benchmarks from the SPEC CPU2000 suite, showing the speedups achieved on a commodity multicore single-die system. After describing the characteristics of the loops automatically chosen for parallelization by HELIX, we show that only a small fraction of the overall data consumed by loop iterations needs to be forwarded between cores, and we perform a limit study of signal prefetching to demonstrate the effectiveness of helper threads and to highlight the opportunity for further improvements. Moreover, we analyze the loop selection algorithm to explain why choosing the right loops to parallelize is

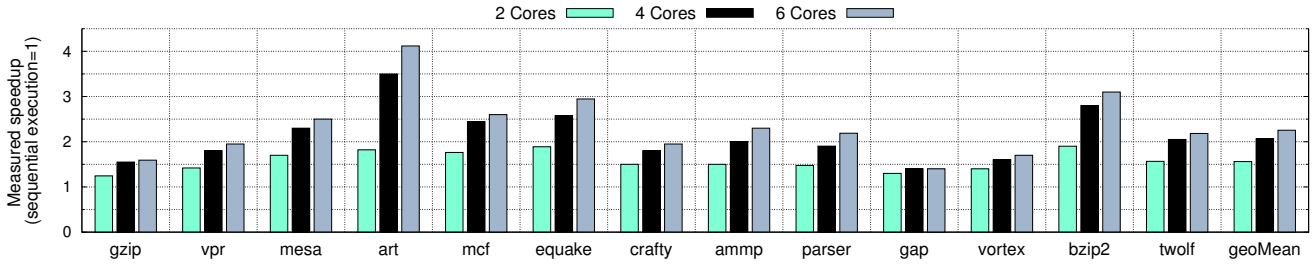

Figure 9: Speedups achieved by HELIX on a real system

critical for achieving consistent speedups, and we assess the impact on this algorithm of over- or underestimating signal latency. We also demonstrate that blindly choosing loops at a single specific nesting level (e.g., outermost loops) can hide otherwise exploitable parallel code.

3.1 Experimental Setting

HELIX extends the ILDJIT compilation framework [8], version 0.7. ILDJIT generates native machine code from CIL bytecode, so we used GCC4CLI [14] to translate benchmarks written in C to CIL. ILDJIT has several advantages for this work. It has a rich intermediate representation (IR), including metadata descriptions of source data types. It supports code instrumentation at the IR level, and its built-in optimization suite is easy to extend with new code analysis and transformation passes.

Benchmarks. To evaluate our scheme we use 13 out of the 15 C language benchmarks from the SPEC CPU2000 suite, because GCC4CLI only supports C. We use the training inputs to select loops for parallelization, and the reference inputs for validation. The DDG analysis that we rely on [17] requires too much memory to handle either 176.gcc or 253.perlbmk.

Measuring Performance. We compute speedups resulting from parallelization by running entire benchmarks to completion on a real system using the reference inputs. For each benchmark, in order to choose loops for parallelization, we first build a static loop nesting graph and a data dependence graph. Then we profile the graphs (using training inputs) to generate the dynamic loop nesting graph. Subsequent profiling runs for the HELIX-optimized form of each loop generate data on execution times by providing a breakdown of time spent configuring its parallel threads, executing its prologue, and executing its sequential segments. From this data, the algorithm described in Section 2.2 chooses a set of loops to parallelize.

Given the multi-threaded and, hence, non-deterministic, nature of the parallelized programs on a real system, we repeat each experiment a minimum of 10 times until the width of the 95% confidence interval is less than 5% of the sample mean.

Hardware Platform. Our experiments use an Intel® Core™ i7-980X with six cores, each operating at 3.33 GHz, with Turbo Boost disabled. The processor has three cache levels. The first two are private to each core and are 32KB and 256KB each. All cores share the last level 12MB cache, which is used to forward data values across cores of the same processor through the MESIF cache coherence protocol.

3.2 HELIX Evaluation

Figure 9 shows the measured speedups of whole application runs on real hardware after compilation by HELIX. Loops have been chosen using Equation (1) with four clock cycles for the signal latency (S), which corresponds to a fully prefetched signal (a hit in the first level cache), and 110 cycles for memory transfers (M), which is experimentally measured using microbenchmarks, together with profiling data. The geometric mean of the resulting speedups on a six core CPU is 2.25×, with a maximum of 4.12×.

Benchmarks	gzip	vpr	mesa	art	mcf	equake	crafty	ammp	parser	gap	vortex	bzip2	twolf
Parallelized loops	29	32	13	22	13	15	21	17	21	14	12	24	31
Loop candidates	178	409	257	150	141	171	536	193	572	822	273	210	731
Loop-carried dependences(%)	38	31	26	20	54	27	12	18	20	31	30	12	22
Signals removed(%)	96.1	91	88	80	95	90	95	90	91	98	92	87	91
Data transfers(%)	10	6	3	0.1	9	1	2	4	3	3	4	12	5
Maximum code (KB)	40	30	50	30	50	60	100	60	35	30	30	30	50

Table 1: Characteristics of parallelized loops.

Table 1 shows the number of loops chosen by HELIX to parallelize and the total number of candidate loops. The fraction of loop-carried dependences inside the parallelized loops is low, highlighting the additional degree of freedom that HELIX is able to exploit, in contrast to techniques such as DSWP that must also handle intra-iteration data dependences, Moreover, Table 1 shows the drastic reduction in signals used for synchronization that Step 6 of the HELIX algorithm is able to achieve. Further, this table shows that only a small fraction of data has to be forwarded between cores. Finally, the maximum code size for each iteration thread that has to be loaded into the instruction cache is negligible, so the instruction cache miss rate is very low.

Importance of Reducing Signaling Overhead. Signaling overhead is the main source of inefficiency in code produced by the HELIX algorithm. Both steps 6 and 8 (Section 2.1) constrain this overhead. Step 6 minimizes the number of overall signals sent per loop iteration. Step 8 minimizes the overhead per signal.

Figure 10 shows the speedups achieved for six cores when some steps of the HELIX transformation are disabled to illustrate their contribution. In each case, loops are chosen for parallelization using profile data obtained by instrumenting code produced for the specific case. As a result, HELIX avoids slowdown even when both steps 6 and 8 are disabled (first bar for each benchmark), so that signaling overhead is high. In this case, loops where most of the time is spent are not chosen by the HELIX transformation algorithm described in Section 2.2, because that would lead to a significant signaling overhead.

The second and third bars show what happens when step 8 or step 6 is disabled, respectively. Only a small speedup is achievable in either case. When step 8 is disabled, fewer signals are sent, but each one stalls execution for too many cycles (110 on our platform). When step 6 is disabled, even if step 8 reduces the overhead per signal, too many signals are sent overall.

The last bar of Figure 10 shows the effect of using both steps 6 and 8, but without benefit of the code scheduling algorithm described in Figure 6. The differences between this last bar and the second and third bars show that only when steps 6 and 8 are used together

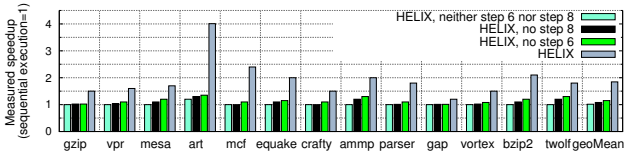

Figure 10: Speedups achieved when steps 6 and 8 of the HELIX code transformation algorithm are disabled, either separately or together. The code scheduling algorithm described in Figure 6 is disabled for these measurements.

can significant speedups be obtained. Finally, the difference between the last bar and the speedups shown in Figure 9 shows that spacing sequential segments to help the signal prefetching mechanism improves speedups significantly, especially for benchmarks like equake, mesa, and vpr that require negligible amounts of data to be transferred between iterations.

3.3 Signal Prefetching

In order to evaluate the effectiveness of helper threads in improving TLP, we compare the speedups achieved by HELIX with two additional schemes. The first has each helper thread execute $Wait$ instructions in the same order as its corresponding iteration thread. We call this *matched prefetching*. The second scheme represents an ideal scenario in which all signals are completely prefetched and available in the first level cache, even if this is unrealistic given the current code schedule. We call this *ideal prefetching*.

To compute speedups for matched and ideal prefetching we rely on the model introduced in Section 2.2. One parameter of this model is the signal latency S. Note that when there is no prefetching, this latency does not depend on the executed code. On the other hand, with the prefetching mechanism used by HELIX, this latency varies with the workload. We first compute the average number of clock cycles between the current sequential segment and the next one, then take the difference between that and the unprefetched signal latency (which represents the possible benefit of signal prefetching). We then compute the minimum of this benefit and the latency for a fully prefetched signal (which is four clock cycles in our system, the latency of a hit in the first level data cache). The signal latency to use in the model is computed on a per-loop basis by averaging this minimum value across sequential segments.

To measure the unprefetched signal latency on our system, we developed a microbenchmark that experimentally computes the number of clock cycles needed to send a signal between cores. We used it to test several implementations of $Wait$ and $Signal$ (including spin locking, snoop locking, collision avoidance locking, and tournament locking [35]), and found that our load-store implementation performs best on our platform. Its measured signal latency is 110 cycles, double the latency of single accesses to the last level cache (L3). This is because the receiving core must continuously check the memory location used to convey the signal and, therefore, needs to access the shared cache (L3), which takes 55 cycles. The sending core only writes to its L1 cache, but the value needs to be forwarded to the receiving core. This is similar to an access to the last level cache [1], which accounts for the other 55 cycles.

The difference between the geometric mean of speedups for HELIX and matched prefetching is 0.1, which shows the accuracy of Step 8 of the HELIX algorithm (see Section 2.1) in computing the sequence of $Wait$ instructions executed by helper threads.

The geometric means of speedups for matched prefetching and ideal prefetching differ by 0.4, which shows the room that a code scheduling algorithm might have to reorganize code executed by iteration threads to help the prefetching mechanism. However, since ideal prefetching does not check the feasibility of fully prefetching signals, it would be difficult for static code scheduling to always close this gap.

3.4 Model validation

Despite abstracting away details, the HELIX speedup model is quite accurate. To measure its accuracy and demonstrate its robustness, we compare the model's speedup estimates, which are based on profile data, against HELIX's actual speedups on the real system. Space limitations prevent our showing the data. However, the observed error between model-based and experiment-based speedups is below 4% for every benchmark considered. VTune [21] shows that this error can be attributed to false cache sharing within the first two private cache levels.

3.5 Loop Selection

To highlight the importance of choosing the most profitable loops for parallelization, Figure 11 plots the breakdown of time that each benchmark spends on different types of code when the code produced by HELIX is constrained to run on a single core. For each benchmark, the first collection of bars corresponds to seven different fixed loop nesting levels, from the outermost to the innermost. The last bar of each group, set apart from the others, corresponds to the time breakdown obtained when using variable nesting levels for loop selection, as described in Section 2.2. We assume an optimistic 0-cycle communication latency for this analysis.

HELIX enables a significant fraction of the code to run in parallel ("Parallel" in the figure, which is the P factor from Equation (1)). This is code from a loop's body that does not belong to any sequential segment. For example, consider *art*, where almost 100% of the time is in parallel code. Note that no single fixed nesting level can maximize the fraction of parallel code across all of the benchmarks. In contrast, our loop selection algorithm consistently maximizes parallel code for all benchmarks. Furthermore, since the algorithm aims to maximize the fraction of parallel code (i.e., P) and loop prologues are not part of it, the chosen loops have small prologues. This is highly desirable, as prologues run sequentially. Note that even though these benchmarks were not designed for parallelization, they possess significant amounts of parallelism.

Low signal latency and smart selection of loops to parallelize are both essential for HELIX. The two are linked, because loop selection is very sensitive to signal latency. Loops chosen by our algorithm belong to different nesting levels, which motivates the previous finding that loop nesting levels cannot be set *a priori*. To further study the importance of loop selection, we explore the effects of different signal latencies.

Impact of signal latency. Poor estimates of signal latency during loop selection can severely degrade speedups. Figure 12 plots speedups for two corner cases on a real system. In one case, signal latency is underestimated (as 0 clock cycles). For most of the benchmarks, an aggressive assumption of 0-cycle latency during loop selection actually leads to slowdown (speedup < 1). Loops at deeper nesting levels have smaller sequential segments, but require more inter-core communication. Therefore, assuming low-cost communication leads the algorithm to reduce sequential segments, but the large penalty inhibits speedup. In contrast, an overestimate of the signal latency (110 cycles) deters the algorithm from choosing loops at deeper levels, and the system fails to exploit the low latency during execution.

Impact on loop nesting level. Figure 13 reinforces this point, using a six-core processor. For each of the benchmarks, each bar

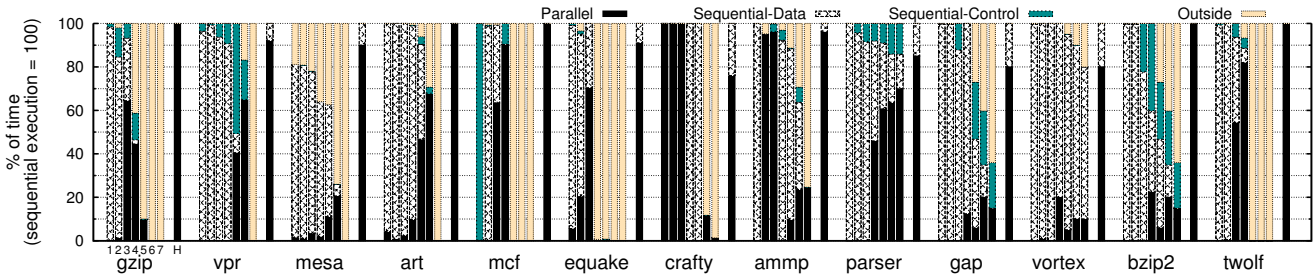

Figure 11: Time breakdown of benchmark execution varying the loop nesting level from 1 to 7 and including the loops selected by HELIX (H). Components are: parallel code (Parallel), sequential segments (Sequential-data), prologues (Sequential-control), and code outside chosen loops (Outside).

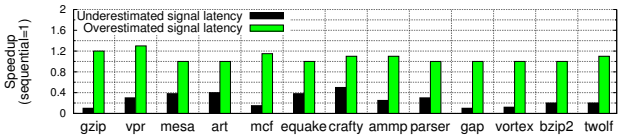

Figure 12: Impact of a poor choice of loops to parallelize. Loop selection assuming 0-cycle signal latency gives the left bars. Assuming 110-cycle latency gives the right bars.

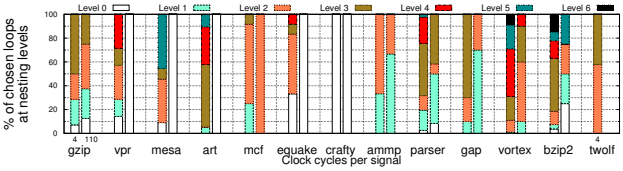

Figure 13: Nesting level distribution of parallel loops for various signal latencies on a six-core processor.

presents a percentage breakdown of the loops' nesting levels chosen by our loop selection algorithm for different signal latencies. As signal latency grows from 4 to 110 cycles, the algorithm prefers outermost loops as soon as it does not produce slowdown. For example, in twolf, for latencies greater than or equal to 110 clock cycles, there is no benefit in parallelizing loops. Other analyses, not shown for brevity, show that loop selection is insensitive to the number of cores.

4. RELATED WORK

The closest approach to HELIX is the DOACROSS technique [15, 20], which has been studied in depth for regular workloads [4, 24, 36]. DOACROSS executes sequential segments without exploiting TLP between them [15]. Moreover, it does not permit either irregular control flow or irregular memory accesses within the loop [15]. Since HELIX has no such constraints and it considers a broader set of options during loop transformation, it can be seen as a generalization of the DOACROSS scheme that can be applied both to regular and irregular code. Aiken and Nicolau [3] presented perfect pipelining, a scheme for unrolling and compacting loops for parallelization, and showed that DOACROSS is a restriction of their scheme. Xu and Chaudhary [40] developed a time stamp algorithm for run-time parallelization of DOACROSS loops with indirect access patterns. Chen and Yew [10] found that most cross-iteration dependences in these types of loops are quite simple. Tournavitis et al. [38] proposed an approach to loop parallelization that uses profile data to help the compiler eliminate conservative data dependences across loop iterations. This technique is not fully automatic, however, relying on users to indicate whether loops can be parallelized, since the compiler cannot ensure correctness.

Recent work on DOALL parallelism has used code transformations and thread-level speculation techniques to expose hidden parallelism in general purpose programs [41]. In contrast, our work is non-speculative.

Decoupled software pipelining (DSWP) [30] addresses the limitations of the DOACROSS approach by extracting fine-grained pipeline parallelism. DSWP automatically breaks the CFG of each loop into strongly connected components (SCC) chosen to minimize the transfer of variables between threads and balance the computation load among threads. The technique can produce significant speedups when this kind of parallelism is available in the program. However, it is limited by the number of SCCs in each loop, which is usually much smaller than the loop iteration count [34].

Bridges et al. [7] extended DSWP with thread-level speculation and manual insertion of source code extensions. This builds on the manual parallelization work of Prabhu and Olukotun [31]. Bridges' approach parallelizes loop iterations speculatively and allows the compiler to choose among several valid program outcomes. Thies et al. [37] also use annotations that indicate pipeline boundaries for DSWP. However, these approaches rely on programmer assistance to explicitly add the language extensions.

Raman et al. [32] studied speculative parallelization using software transactional memory. They introduced a multi-threaded software transaction using memory versioning to separate speculative and non-speculative state. Speculative decoupled software pipelining [39] is used to parallelize loops with the speculative portion of the code wrapped in a transaction. In order to remove the constraint on the number of threads extracted, DSWP has been mixed with the DOALL technique [19, 33]. In [33], simulation is used to investigate a fast inter-thread communication hardware mechanism. In [19], the proposed approach is manually applied to the benchmarks considered. Finally, in all these works [7, 19, 30, 32, 33, 34, 37, 39] loops are selected manually.

Elimination of redundant synchronizations has been studied for regular workloads [12, 13, 25]. Our work has the same goal, but for irregular workloads.

Code scheduling algorithms to improve TLP across loop iterations in already parallelized programs have been proposed [28, 29]. They target coarser-grained code sections than the ones HELIX operates on, and they do not consider the signaling problem, which we address by merging sequential segments with no loss of TLP. Finally, a code scheduling algorithm that targets the DOACROSS technique has been proposed [11]. Since it is specific to DOACROSS, it cannot provide broader options available with HELIX, such as execution of independent sequential segments in parallel.

Dynamic schemes to execute loop iterations in parallel when they are detected at run time to be independent have been proposed [16, 42]. In [16] loop iterations are speculatively executed in parallel

with the possibility of fixing the execution if they are misspeculated. In [42], loop iterations are executed in parallel after they are recognized to be independent at run time; in this work the focus is on minimizing the overhead due to the computation of the dynamic data dependences by performing an approximated and conservative analysis while the original code is in execution.

Exploiting SMT to help critical threads was introduced in [9] and adapted in different domains later on [22, 27]. We have shown how to adapt this idea to mitigate our specific problem of sending signals by automatically generating helper threads.

5. CONCLUSION

HELIX is a new, fully automatic parallelization technique that can speed up execution of irregular single-threaded programs on chip multiprocessors. Our experimental results show that by reducing communication costs and increasing thread level parallelism, HELIX achieves a mean speedup of 2.25× (with a maximum of 4.12×) over a broad range of applications on a real six-core system. Choosing the right loops to parallelize is one key to the success of our implementation, which combines an analytical model of loop speedup with profiling data to choose the most profitable loop sets. Our results show that synchronization, not data transfer, is the main bottleneck, which can be significantly reduced by proper use of SMT technology. In future work, we expect our implementation to exploit fast hardware implementations of signaling to obtain better speedup.

Acknowledgements

Authors thank the anonymous reviewers for their hard work that allowed us to improve the paper significantly. This work was possible thanks to the sponsorship of Microsoft Research, HiPEAC, the Royal Academy of Engineering, EPSRC and the National Science Foundation (award number IIS-0926148). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

6. REFERENCES

[1] Intel 64 and IA-32 Architectures Software Developer's Manual. Specification, 2010.
[2] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 1995.
[3] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. *ESOP*, 1988.
[4] J. Allen and K. Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann, 2002.
[5] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proc. Spring Joint Computer Conference*, 1967.
[6] A. Appel. *Modern Compiler Implementation in Java, 2nd edition*. 2002.
[7] M. Bridges et al. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, 2008.
[8] S. Campanoni et al. A highly flexible, parallel virtual machine: Design and experience of ILDJIT. *Softw. Pract. Exper.*, 2010.
[9] R. Chappell et al. Simultaneous subordinate microthreading (SSMT). *ISCA*, 1999.
[10] D-K. Chen and P-C. Yew. An empirical study on DOACROSS loops. 1991.
[11] D-K. Chen and P-C. Yew. Statement re-ordering for DOACROSS loops. *ICPP*, 1994.
[12] D-K. Chen and P-C. Yew. On effective execution of nonuniform DOACROSS loops. *IEEE Transactions on Parallel and Distributed Systems*, 1996.
[13] D-K. Chen and P-C. Yew. Redundant synchronization elimination for DOACROSS loops. *Parallel and Distributed Systems, IEEE Transactions on*, 10(5), May 1999.
[14] R. Costa et al. Gcc4cli. http://gcc.gnu.org/projects/cli.html.
[15] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. *ICPP*, 1986.
[16] F. Dang and L. Rauchwerger. Speculative parallelization of partially parallel loops. In *5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 285–299, 2000.
[17] B. Guo et al. Practical and accurate low-level pointer analysis. 2005.
[18] J. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31, May 1988.
[19] Jialu H. et al. Decoupled software pipelining creates parallelization opportunities. *CGO*, 2010.
[20] A. Hurson et al. Parallelization of DOALL and DOACROSS loops - a survey. *Advances in Computers*, 1997.
[21] VTune. http://software.intel.com/en-us/intel-vtune.
[22] D. Kim et al. Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. *CGO*, 2004.
[23] H. Kim et al. Scalable speculative parallelization on commodity clusters. *MICRO*, 2010.
[24] J. Lim et al. A loop allocation policy for DOACROSS loops. *SPDP*, 1996.
[25] D. Liu et al. Optimal loop parallelization for maximizing iteration-level parallelism. *CASES*, 2009.
[26] J. Lo et al. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *TCS*, 1997.
[27] C-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. *SIGARCH Comp. Arch. News*, 2001.
[28] A. Nicolau et al. Synchronization optimizations for efficient execution on multi-cores. *ICS*, 2009.
[29] A. Nicolau et al. Techniques for efficient placement of synchronization primitives. *PPoPP*, 2009.
[30] G. Ottoni et al. Automatic thread extraction with decoupled software pipelining. *MICRO*, 2005.
[31] M. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. *PPoPP*, 2000.
[32] A. Raman et al. Speculative parallelization using software multi-threaded transactions. *ASPLOS*, 2010.
[33] E. Raman et al. Parallel-Stage decoupled software pipelining. *CGO*, 2008.
[34] R. Rangan et al. Performance scalability of decoupled software pipelining. *TACO*, 2008.
[35] J. Seung-Ju and K. Gil-Yong. Spin-block synchronization algorithm in the shared memory multiprocessor system. *SIGOPS Oper. Syst. Rev.*, 1994.
[36] H-M. Su and P-C. Yew. Efficient DOACROSS execution on distributed shared-memory multiprocessors. *ACM/IEEE conference on Supercomputing*, 1991.
[37] W. Thies et al. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. *MICRO*, 2007.
[38] G. Tournavitis et al. Towards a holistic approach to auto-parallelization. *PLDI*, 2009.
[39] N. Vachharajani et al. Speculative decoupled software pipelining. *PACT*, 2007.
[40] C.-Z. Xu and V. Chaudhary. Time stamp algorithms for runtime parallelization of DOACROSS loops with dynamic dependences. *TPDS*, 2001.
[41] H. Zhong et al. Uncovering hidden loop level parallelism in sequential applications. *HPCA*, 2008.
[42] X. Zhuang et al. Exploiting parallelism with dependence-aware scheduling. In *PACT*, pages 193–202, 2009.