Making the Extraction of Thread-Level Parallelism Mainstream

Simone Campanoni Timothy Jones Glenn Holloway
Gu-Yeon Wei David Brooks

Abstract

Improving system performance increasingly depends on exploiting microprocessor parallelism, yet mainstream compilers still do not parallelize code automatically. Promising parallelization approaches have either required manual programmer assistance, depended on special hardware features, or risked slowing down programs they should have speeded up. HELIX is one such approach that automatically parallelizes general-purpose programs without requiring any special hardware. In this article we show that in practice HELIX always avoids slowing down compiled programs, making it a suitable candidate for mainstream compilers. We also show experimentally that HELIX outperforms the most similar historical technique that has been implemented in production compilers [5].

1 Introduction

Although chip multiprocessors are commonplace, compilers rarely exploit the cores they make available. There has been exciting research on parallelization of programs, but the results have not found their way into mainstream compilers. Developers need an automatic way of transforming sequentially-designed source code into multi-threaded object code. What will it take to enable compilers to extract thread-level parallelism as routinely as they now exploit instruction-level parallelism?

Our work on HELIX [2] suggests an answer. HELIX extracts thread-level parallelism automatically from sequential programs by transforming select loops into parallel form.

To qualify for routine use in a general-purpose compiler, an optimization technique needs at least these properties: (i) It must be fully automatic, not dependent on programmer guidance or intervention. (ii) It must nearly always improve the quality of the object code, and almost never make it worse. (iii) It must rely only on hardware features that are widely available in commercial processors. HELIX has these properties. It is fully automatic, not dependent on source code annotations or modifications by the user. It never produces code that slows down execution, and it speeds up regular and irregular workloads significantly on a real multicore commodity processor. For thirteen C language benchmarks from the SPEC CPU2000 suite, it yields overall speedups averaging 2.25× on a six-core CPU, with a maximum of 4.12×.

Historically, languages for parallel programming have allowed programmers to distribute loop iterations over processing elements [5], which is what HELIX does automatically. We compare HELIX to that historical technique and show that our automatic approach liberates more parallelism and achieves better speedup.

The following sections describe HELIX and explain why the emergence of chip multiprocessors has made it practical. Through measurements of our working system, we show that even for a target architecture with unfavorable characteristics, HELIX avoids slowing programs down, and we analyze the sources of its speedups for a typical modern target processor.

2 Motivation

Like a physician, a production compiler should “first, do no harm”. If a code optimization algorithm risks slowing down the compiled program, it needs to come with an effective, efficient test for deciding whether to apply the transformation.

Recently, a number of research projects have succeeded in extracting threads from sequential code [7, 8, 10, 12, 13, 14]. Although such approaches are promising, they lack the applicability test that can guarantee attempted optimizations will do no harm. The complexity of their sophisticated transformation algorithms makes it hard to model their effects efficiently. On the other hand, HELIX includes a heuristic that is inexpensive to compute and

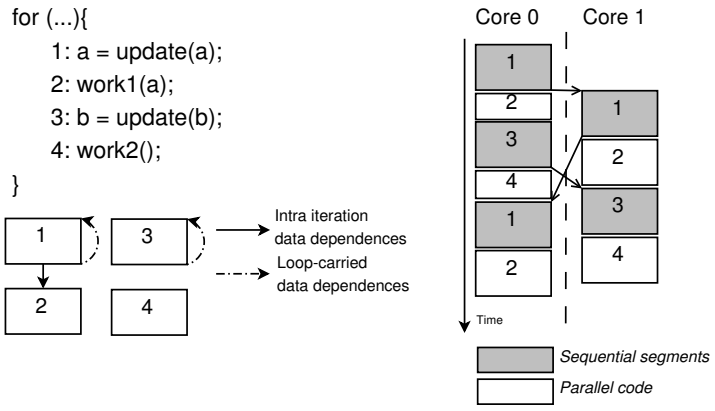

Figure 1: Execution of code produced by HELIX for a dual-core processor. Note that code blocks 1 and 3 must each be executed sequentially, but since they are independent, HELIX overlaps them in time.

effective at predicting when its code transformations will improve performance. Section 4 shows that by applying this heuristic to profiles obtained from training runs of the program being compiled, HELIX always avoids slowdowns and often produces significant speedups.

HELIX is based on a simple idea: To parallelize a loop, distribute its iterations among several hardware threads running on separate cores of a single processor. Spreading loop iterations over separate processing elements is not new; it has been done since multiprocessing computers first appeared [5]. This approach is extremely sensitive to the cost of communication between processing elements, because the execution of one iteration may depend on data produced by an earlier iteration. This sensitivity drives optimizer design towards solutions that minimize communication overhead, usually by giving up some thread-level parallelism (TLP) in the resulting compiled code. Every data dependence that crosses a loop iteration boundary gives rise to some code that must be executed in loop-iteration order, even when the iterations are running in separate hardware threads. We call this code a *sequential segment* because for a given data dependence, the sequential segment cannot be run in parallel. Its execution by separate threads must be synchronized to maintain the correct evaluation order. Sequential segments arising from different data dependences can in principle be executed in parallel. But treating the sequential segments of a loop as independent in this way requires more synchronization between threads. Historically, when loop iterations were run in parallel, their sequential segments were clumped together to minimize communication overhead. That constraint reduces parallelism, both because sequential segments never run concurrently and because intra-iteration dependences trap some code within the sequential clump that could otherwise remain unsynchronized, outside all sequential segments.

With the emergence of multicore microprocessors, the cost of inter-processor communication has dropped dramatically, since independent processing elements on a single chip can communicate through a shared memory cache. With current commodity processors, HELIX is able to greatly reduce communication overhead by exploiting both the memory consistency model and simultaneous multi-threading (SMT). Since HELIX is less sensitive to communication overhead, it does not have to compromise TLP, so that more code from a parallelized loop is free to be run in parallel.

3 HELIX

HELIX chooses the most profitable loops to parallelize by relying on a profile obtained using representative input (e.g., SPEC benchmark training input). Parallelized loops run one at a time. The iterations of each parallelized loop run in round-robin order on the cores of a single processor. HELIX applies code transformations to minimize the inefficiencies of sequential segments, data transfer, synchronization, and thread management. Our paper describing HELIX [2] contains much more detail.

Sequential segments. HELIX inserts code to ensure that data dependences across loop boundaries are implemented correctly. That creates the sequential segments. Data produced by iteration i and consumed by iteration $i + 1$ is forwarded through memory from the thread for i to that for $i + 1$. If the distance between producer and consumer is

greater than one iteration, data passes through successive threads to its destination.

Although the sequential segments of a given dependence must run in loop-iteration order, those of different dependences may run in parallel. HELIX executes distinct sequential segments concurrently whenever possible, as shown in Figure 1, where sequential segments 1 and 3 overlap.

Data transfer. Transferring data between threads could be a significant source of overhead. However, we have shown [2] that careful selection of loops to parallelize keeps the amount of data forwarded between threads small, compared with the amount consumed within each iteration.

Thread synchronization. Threads synchronize by sending signals. When a sequential segment ends, for example, a signal to the successor thread notifies it that the corresponding sequential segment can start. HELIX minimizes the number of signals sent by exploiting redundancy among them. We have shown [2] that only 10% of signals remain. Moreover, HELIX reduces the perceived signal latency by exploiting the SMT technology of the underlying platform. It couples each thread running an iteration with a *helper thread* that runs on the same core and ensures that intercore transmission of each signal begins as soon as it is sent. (Exploiting SMT to help critical threads was introduced in [3] and adapted for different domains later [9, 11].)

3.1 Choosing Loops to Parallelize

HELIX 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 (loops that, if parallelized, best speed up the program) is critical for achieving significant speedups.

Speedup model. Our heuristic chooses loops by using a simple model based on Amdahl's law, which 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 into Amdahl's law to produce the following model for the speedup of a parallelized program:

$$\frac{T_{\text{orig}}}{T_{\text{seq}} + \frac{P}{N} + O}$$

Here T_{orig} is the time consumed by the unparallelized program, T_{seq} is the time spent running sequential code in the parallelized program, and $P = \sum_{i=1}^{Loops} P_i$, where $Loops$ is the number of parallelized loops and P_i is the time spent running the code of loop i outside its sequential segments. Overhead $O = \sum_{i=1}^{Loops} O_i$, where O_i is the added overhead for loop i, which is:

$$O_i = Conf_i + Sig_i \times S + Words_i \times M \quad (1)$$

Here $Conf_i$ is the time spent configuring loop i, Sig_i is the overall number of signals sent during loop i (computable from the static number of sequential segments and the iteration and invocation counts of i), and S is the time spent per signal, which is assumed to be constant. Finally, $Words_i$ is the number of CPU words transferred between loop iterations and M is the time required to transfer a CPU word between cores.

The fixed model parameters S and M are determined by our target platform (see Section 4.2). To obtain parameters for loop i (P_i, $Conf_i$, Sig_i, and $Words_i$) HELIX parallelizes and profiles each loop individually.

Loop selection is the only facet of HELIX that uses profiles. While some elements of the speedup model might be accurately predictable through static analysis alone, it would be difficult to estimate the data traffic between threads without profiling ($Words_i$). This is lightweight, embarrassingly parallel profiling, which measures only loop invocation and trip counts, data traffic, and the time spent in sequential segments. Interprocedural static analysis typically takes much longer than parallelizing and profiling loops.

Loop selection. The HELIX loop selection algorithm builds a nesting graph for the whole program, in which each loop is represented by a node, and directed edges connect each node to those for the immediately enclosing loops. The algorithm labels each node with two values, `T` and `maxT`, representing time savings due to parallelization. The `T` value for a loop is the total amount of time (over all loop invocations) that would be saved by parallelizing that loop. The

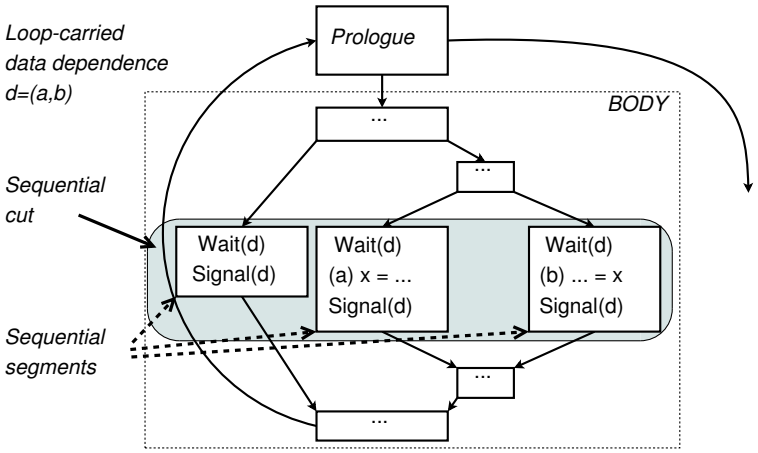

Figure 2: Insertion of $Wait(d)$ and $Signal(d)$ due to a RAW data dependence $d = (a, b)$. Sequential segments that only contain $Wait$ and $Signal$ operations handle dependences that span multiple iterations.

`maxT` value is either the same as `T` or else the total time saved by parallelizing a set of subloops, if that total is greater than `T`. Initially, using profiles obtained by parallelizing each loop individually, HELIX sets both `T` and `maxT` for each node to the time saved by executing the corresponding loop in parallel form. Then, leaving `T` unchanged, it propagates new `maxT` values through the graph by replacing each `maxT` with the sum of the `maxT` values of immediate subnodes whenever that sum exceeds the current `maxT` value. Propagation continues until a fixed point is reached.

HELIX then collects the set of loops to parallelize by scanning the labeled nesting graph, starting from the outermost loops. Whenever `maxT` exceeds `T`, it searches deeper, because parallelizing a collection of more deeply nested loops will save more time than parallelizing the current loop. The loops chosen are those for which $\texttt{T} = \texttt{maxT} > 0$.

3.2 Algorithm for Parallelizing One Loop

Our algorithm for parallelizing a given loop transforms it into the form shown in Figure 2, where the *prologue* is the smallest subgraph of a loop's control flow graph that is needed to determine whether the next iteration's prologue will be executed, and the *body* is the rest of the loop.

Iterations execute without speculation, each starting in a parallel thread once the prologue of the preceding iteration has completed. As the body of iteration i begins, the prologue of iteration $i + 1$ is triggered. In the steady state, the bodies of multiple loops execute in parallel.

Data dependences that cross loop boundaries are identified by applying interprocedural pointer analysis to the whole program [6]. (We call this set of dependences D_{Data}.) HELIX associates a sequential segment with each such dependence.

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 execute in the correct order. The operation $Wait(d)$ blocks execution of a thread until the predecessor thread sends a *data signal* by executing $Signal(d)$. As a result, the code between $Wait(d)$ and $Signal(d)$ executes in loop iteration order, satisfying the data dependence d. For example, in Figure 1 blocks *1* and *3* run in loop iteration order, while *2* and *4* execute in parallel.

HELIX ensures that every path through the loop body reaches a sequential segment of each dependence once, making the number of data signals sent during one invocation of a parallelized loop equal to the number of iterations times the number of dependences.

Minimizing signal count. HELIX avoids inserting redundant $Wait$ and $Signal$ operations. Where possible, it uses a single signal to synchronize multiple data dependences.

The $Wait$ and $Signal$ operations are implemented with simple loads and stores, using a dedicated per-thread memory area. Depending on the memory consistency model of the underlying platform, memory barriers may need

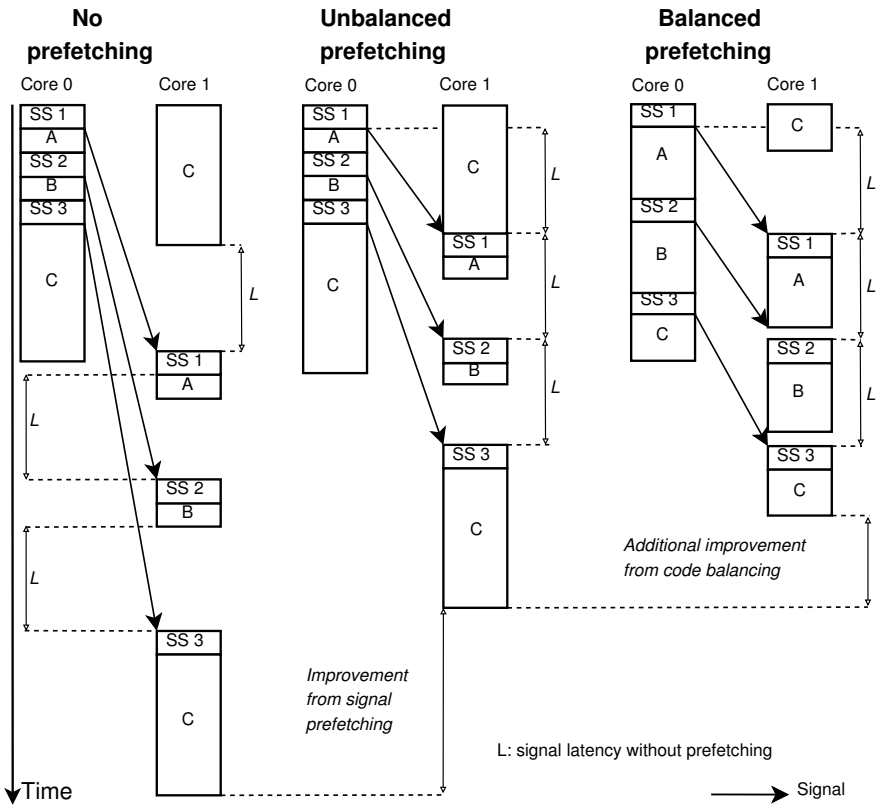

Figure 3: Importance of balanced prefetching.

to be added before the loads and after the stores. However, since this memory has only one reader (the current thread) and one writer (the predecessor thread), this is not required on our Intel-based evaluation system. Dependences that span more than one iteration are synchronized by signals sent through the intervening iterations.

Helper threads. When core c_i sends a signal to core c_j, it writes a value to a designated memory location, which puts it in c_i's first private cache. The value is not forwarded to c_j's private cache until c_j issues the corresponding $Wait$ operation (i.e., a load instruction). It then takes several clock cycles for c_j to receive the value (110 cycles in our testbed). The caches act as a pull system. However, when cores have SMT capabilities, HELIX adds a helper thread to each core to pull signals from the destination side. By issuing a sequence of $Wait$ operations, one per dependence, the helper thread on c_j pulls signals from c_i as soon as they are produced. (No synchronization is needed between a helper thread and its associated iteration thread. Each remains in lock step with signals from the predecessor core.)

Prefetching works best when signals occur at regular intervals, so HELIX schedules code to space sequential segments evenly within a loop's body. Consider Figure 3, which shows the run-time execution of a parallelized loop with three sequential segments, SS1, SS2, and SS3. The first case, "No prefetching", represents the execution of the code produced without using helper threads. In this case, every signal takes L clock cycles to pass between cores, because signal forwarding starts only when the receiver tries to enter the corresponding sequential segment. The second case, "Unbalanced prefetching", shows execution when helper threads are used without the scheduling algorithm. In this case, only the code labeled C, between SS3 and SS1, is long enough to cover the signal latency, so only signals coming from SS1 are fully prefetched; the others are only slightly prefetched. In the last case, "Balanced prefetching", HELIX has moved code from segment C into segments A and B so that the longest paths in each are closer in length. This code balancing further reduces the time wasted waiting for signals to pass between cores.

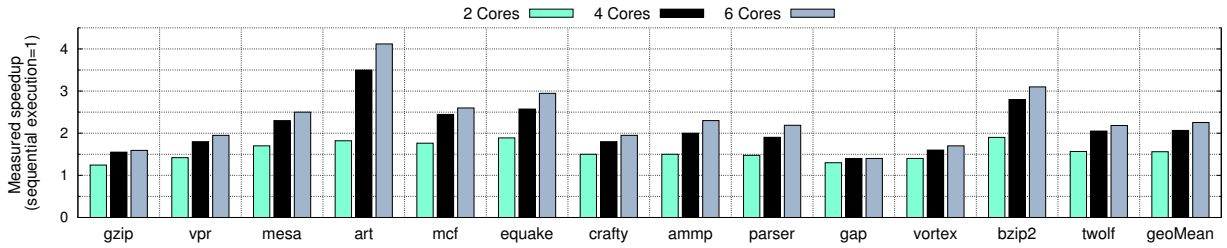

Figure 4: Speedups achieved by HELIX on a real system

4 Evaluation

After describing our experimental setup, we show the speedups achieved by applying the HELIX transformation to SPEC benchmarks. To make the case for including HELIX in production compilers, we show that HELIX never slows down programs. Finally, we analyze how HELIX outperforms comparable historical approaches to loop parallelization.

4.1 Experimental Setting

HELIX extends the ILDJIT compilation framework [1], which generates native machine code from CIL bytecode, so we used GCC4CLI [4] to translate benchmarks written in C to CIL.

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. The data dependence analysis that we rely on [6] requires too much memory to handle either 176.gcc or 253.perlbmk. Using reference inputs, we compute speedups resulting from parallelization by running entire benchmarks to completion on a real system.

Hardware platform. Our experiments use an Intel® Core™ i7-980X with six cores, 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 in size. 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.

4.2 Achievable Speedups

Figure 4 shows the measured speedups of whole application runs with reference input data on real hardware after compilation by HELIX. Baseline runs are fully optimized for single-threaded execution. The geometric mean of the resulting speedups on our six-core CPU is 2.25×, with a maximum of 4.12×.

In the speedup model, Equation (1) is the basis for selecting loops to parallelize, where signal latency S is assumed to be four cycles (the cost of a fully prefetched signal), and the memory transfer delay M is 110 cycles (experimentally measured using microbenchmarks). (Although the level of speedup achievable by the HELIX method is very sensitive to the latency of intercore signals, we have not found the quality of loop selection to be particularly sensitive to the value of S in the speedup model.) We used the training inputs of the benchmarks when collecting profiles for loop selection. For all evaluations, we used the reference inputs.

While the speedup model uses a target number of cores N, the resulting code runs well without alteration when more or fewer cores are available.

4.3 Avoiding Slowdown

To show that HELIX never makes performance worse, we performed two sets of experiments. The first shows that every loop chosen for parallelization contributes positively to the speedup shown in Figure 4. The second shows that HELIX tolerates variation in communication overhead well. For the latter experiment, we artificially varied communication overhead by selectively disabling HELIX's algorithms for minimizing it.

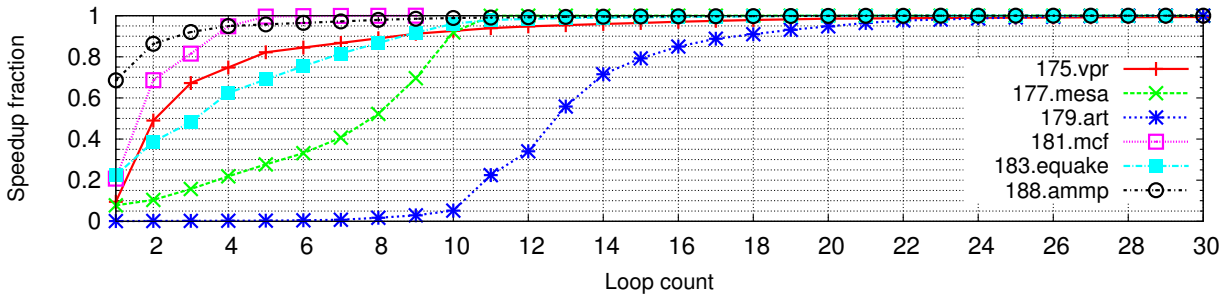

Figure 5: Percentage of speedups shown in Figure 4 achievable when the number of parallelized loops is constrained. The fact that the speedup fraction increases monotonically shows that no loop slowed down because HELIX parallelized it. Even though the speedup fraction increases monotonically with loop count, the curvature can change sign (as with art and mesa) depending on the distribution of the number of clock cycles saved.

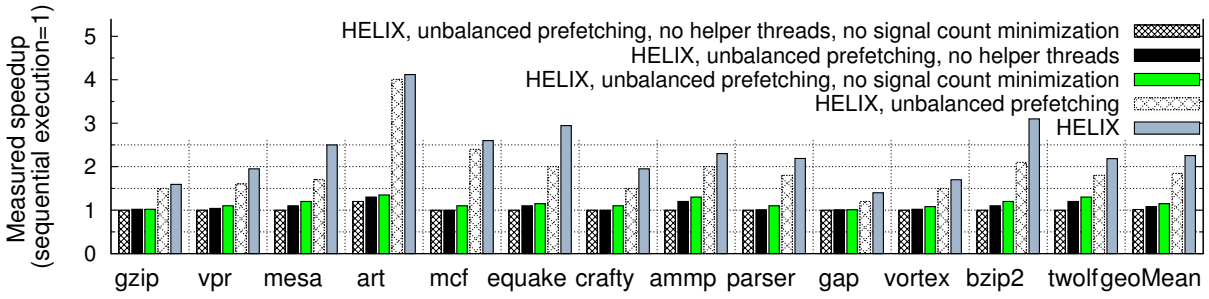

Figure 6: Speedups achieved when signal count minimization and helper threads are disabled, either separately or together. Balanced prefetching is disabled for these measurements.

HELIX does not parallelize loops that could slow down execution because the speedup model described in Section 3.1 conservatively overestimates the run-time overhead of a loop that has been parallelized. In fact, since different sequential segments run in parallel (as shown in Figure 1), only a subset of data signals slow down execution, because data signals also execute in parallel.

This overestimation has two consequences for selection of loops to parallelize: (i) every parallelized loop speeds up program execution; and (ii) some loops that could speed up the execution if parallelized are nevertheless not chosen. We evaluated loops of the second kind for every benchmark we considered. All together, they cover a negligible fraction ($< 1\%$) of the total execution time of the original program.

Chosen loops. Figure 5 shows the fraction of speedups shown in Figure 4 achieved by constraining the number of parallelized loops chosen in Section 4.2. Loops are sorted in descending order according to the number of clock cycles saved by running them in parallel mode.

Figure 5 shows that (i) the speedup fraction increases monotonically with the number of parallelized loops (parallelizing each successive loop always speeds up the program); and (ii) the speedup fraction quickly approaches 100% (i.e., a few loops yield most of the achievable speedup), but the rate of convergence depends on the benchmark.

Handling overhead. To show the effectiveness of HELIX at tolerating different levels of communication overhead, we configured it to produce code with different run-time intercore communication overheads. Specifically, we turned off either the signal count minimization optimization and/or helper threads. Those are the components of HELIX that have the most impact on both communication overhead and program execution time. For each configuration, we obtained profiles for loop selection by instrumenting code produced for that configuration.

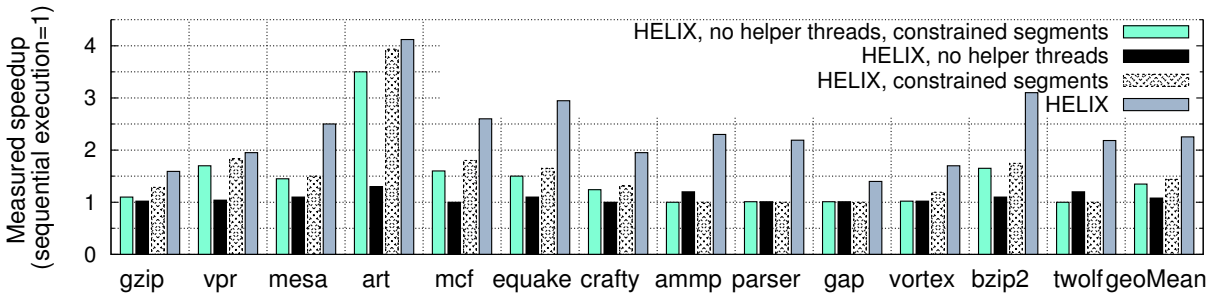

Figure 7: Comparison between grouping sequential segments together and leaving them independent. For both cases, loops were selected as described in Section 3.1.

Speedups for six cores are shown in Figure 6. The first bar for each benchmark depicts its speedup with every optimization targeting communication overhead disabled. HELIX avoids slowing down execution in this case by not selecting the same loops as in the experiments for Figure 4. Even if the latter are very hot loops, the unoptimized communication overhead outweighs the benefit of parallelizing them.

4.4 Comparison with Historical Approaches

As mentioned in Section 2, sequential segments were historically confined to a single code region per loop because of high communication overhead between processors. A single sequential segment requires just one synchronization per iteration. Reducing thread synchronization overhead allows sequential code to be broken into multiple segments and optimized, liberating code that would otherwise be constrained by intra-iteration data dependences. This allows the sequential segments for different data dependences to execute concurrently.

Minimizing synchronization. To evaluate the importance of reducing thread synchronization overhead, we measured speedups with either signal count minimization or signal prefetching disabled. Figure 6 shows the speedups achieved for six cores when these optimizations from the HELIX transformation were turned off.

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

The fourth bar of Figure 6 shows the effect of using both signal optimizations, but without benefit of code balancing for signal prefetching. The difference between this fourth bar and the second and third bars shows that only when both signal optimizations are used together can significant speedups be obtained. Finally, the difference between the fourth bar and the last one shows that spacing sequential segments to help the signal prefetching mechanism improves speedups significantly.

Exploiting parallelism among sequential segments. DOACROSS loop parallelization [5] groups sequential segments together to reduce communication overhead, so we compared HELIX with DOACROSS, i.e., with the case where sequential segments are clumped in a single code region. Since the original DOACROSS technique does not include an approach for loop selection, we used the HELIX algorithm, tuned for this special case, to select its most profitable loops. Moreover, we implemented the DOACROSS technique using the HELIX approach to synchronizing threads and to forwarding data.

Even though the DOACROSS technique does not include helper-thread support, we measured its speedups with helper threads enabled and disabled (first and third bar of Figure 7 respectively). Note that, even when signaling overhead is minimized, the speedup achieved for DOACROSS (third bar) is lower than that obtained by HELIX (fourth bar). This difference is only due to the additional parallelism liberated by HELIX, which executes sequential segments in parallel whenever possible.

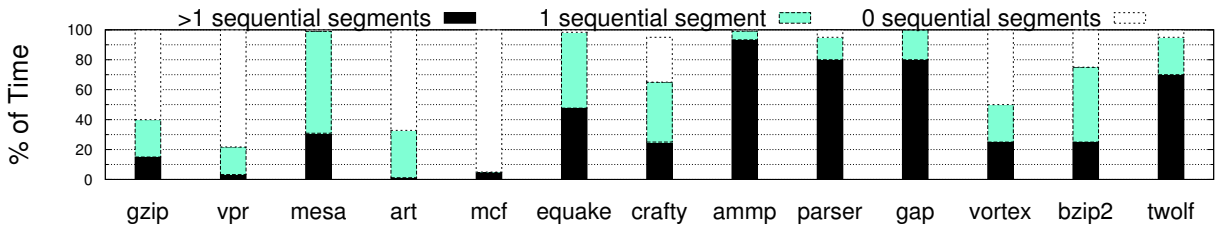

Figure 8: Breakdown of time spent in the parallelized loops used to achieve the speedups shown in Figure 4 based on their numbers of sequential segments.

Figure 7 also illustrates sensitivity to the signaling overhead reduction performed by helper threads. The difference between the second and fourth bars of the figure shows that code produced by HELIX is very sensitive to prefetching. On the other hand, code produced by DOACROSS is almost insensitive to this effect (note the difference between the first and third bars). The reason is that DOACROSS is designed to compromise TLP to minimize communication between threads, leaving almost nothing for helper threads to optimize. HELIX tries to extract as much parallelism as possible, leaving a big gap for helper threads to close.

To better understand why the difference between DOACROSS with helper threads and HELIX is smaller for some benchmarks, we analyzed the loops parallelized by HELIX. Figure 8 groups these loops into three categories: (i) 0 SS (i.e., no sequential segments); (ii) 1 SS (i.e., only one sequential segment); and (iii) $>$ 1 SS (more than one sequential segment). The bar shows the execution time spent by the original program in loops of these categories.

Since there is no parallelism that HELIX can exploit beyond that of DOACROSS when a loop belongs to either the first or second categories, the difference between HELIX and DOACROSS is more marked for benchmarks that spend more time in loops belonging to the last group. Consider benchmarks ammp, parser, gap and twolf. In these cases, the original program spends between 70% and 90% of its time in loops with several sequential segments. By exploiting parallelism among those segments, HELIX achieves speedups where DOACROSS cannot.

5 Conclusion

We have shown that HELIX avoids slowing down programs. Taken together with the facts that it parallelizes code automatically and it does not rely on special hardware, this shows HELIX to be a suitable candidate for inclusion in mainstream compilers. While a related manual technique has appeared in production compilers, we have shown that HELIX liberates more parallelism, achieving better speedup, and does so automatically.

The dominant technique for automatic loop parallelization in sequential programs is called *decoupled software pipelining* [12, 8]. DSWP transforms a loop to exploit multiple hardware threads (the number of which depends on the loop's original structure) while tolerating communication latency among them. In contrast, HELIX applies a simpler transformation, so the effects of parallelization are easier to predict with accuracy, and the resulting code adapts well to varying numbers of available cores. If the trend toward faster intercore communication in microprocessors continues, the benefits of HELIX's direct approach should be increasingly attractive.

Acknowledgements

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. We thank the anonymous reviewers for many suggestions that helped improve this article.

References

[1] S. Campanoni et al. A highly flexible, parallel virtual machine: Design and experience of ILDJIT. *Softw. Pract. Exper.*, 2010.

[2] S. Campanoni et al. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. *CGO*, 2012.

[3] R. Chappell et al. Simultaneous subordinate microthreading (SSMT). *ISCA*, 1999.

[4] R. Costa et al. GCC4CLI. http://gcc.gnu.org/projects/cli.html.

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

[6] B. Guo et al. Practical and Accurate Low-Level Pointer Analysis. *CGO*, 2005.

[7] B. Hertzberg and K. Olukotun. Runtime automatic speculative parallelization. *CGO*, 2011.

[8] J. Huang et al. Decoupled software pipelining creates parallelization opportunities. *CGO*, 2010.

[9] D. Kim et al. Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. *CGO*, 2004.

[10] A. Kotha et al. Automatic parallelization in a binary rewriter. *MICRO*, 2010.

[11] C-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. *SIGARCH Comp. Arch. News*, 2001.

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

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

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