HELIX: MAKING THE EXTRACTION OF THREAD-LEVEL PARALLELISM MAINSTREAM

IMPROVING SYSTEM PERFORMANCE INCREASINGLY DEPENDS ON EXPLOITING MICROPROCESSOR PARALLELISM, YET MAINSTREAM COMPILERS STILL DON'T PARALLELIZE CODE AUTOMATICALLY. HELIX AUTOMATICALLY PARALLELIZES GENERAL-PURPOSE PROGRAMS WITHOUT REQUIRING ANY SPECIAL HARDWARE; AVOIDS SLOWING DOWN COMPILED PROGRAMS, MAKING IT A SUITABLE CANDIDATE FOR MAINSTREAM COMPILERS; AND OUTPERFORMS THE MOST SIMILAR HISTORICAL TECHNIQUE THAT HAS BEEN IMPLEMENTED IN PRODUCTION COMPILERS.

Simone Campanoni
Harvard University

Timothy M. Jones
University of Cambridge

Glenn Holloway
Gu-Yeon Wei
David Brooks
Harvard University

Although chip multiprocessors are commonplace, compilers rarely exploit the cores they make available. There has been exciting research on automatic parallelization of programs, but the results have not found their way into mainstream compilers. Developers need a mechanical way of transforming sequentially designed source code into multithreaded 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 suggests an answer.[1] Helix is a prototype compiler that 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 three properties. First, it must be fully automatic, not dependent on programmer guidance or intervention. Second, it must nearly always improve the quality of the object code, and almost never make it worse. Third, it must rely only on hardware features that are widely available in commercial processors. Helix has all three of 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 13 C language benchmarks from the SPEC CPU2000 suite, it yielded overall speedups averaging 2.25× on a six-core CPU, with a maximum of 4.12×.

Historically, parallel-programming languages have allowed programmers to distribute loop iterations over processing elements,[2] which Helix does automatically. We compare Helix to this traditional implementation and show that our automatic approach liberates more parallelism and achieves better speedup.

Published by the IEEE Computer Society

In this article, we 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.

Motivation

Like a physician, a production compiler should "first, do no harm." If a code optimization algorithm risks slowing down the compiled program, it must be accompanied by an effective, efficient test for deciding whether to apply the transformation.

Recently, several research projects have succeeded in extracting threads from sequential code.[3-8] 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 efficiently model their effects. On the other hand, Helix includes a heuristic that's inexpensive to compute and effective at predicting when its code transformations will improve performance. 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.[2] This approach is extremely sensitive to the cost of communication between processing elements, because data consumed by one iteration might be produced by an earlier iteration. This sensitivity drives optimizer design toward solutions that minimize communication overhead, usually by giving up some thread-level parallelism 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 can't 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 a loop's sequential segments 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, interprocessor communication costs have dropped dramatically, because independent processing elements on a single chip can communicate through a shared memory cache. With current commodity processors, Helix can greatly reduce communication overhead by exploiting both the memory consistency model and simultaneous multithreading (SMT). Because Helix is less sensitive to communication overhead, it doesn't have to compromise thread-level parallelism, so more code from a parallelized loop can be run in parallel.

Code generation

Helix chooses the most profitable loops to parallelize by relying on a profile obtained using a representative input (for example, a SPEC benchmark training input). Parallelized loops run one at a time. The iterations of each parallelized loop run in round-robin order on a single processor's cores. Helix applies code transformations to minimize the inefficiencies of sequential segments, data transfer, synchronization, and thread management. Our paper describing Helix contains more details.[1]

Helix inserts code to ensure that data dependences across loop boundaries are implemented correctly. That creates the sequential segments. Data produced by

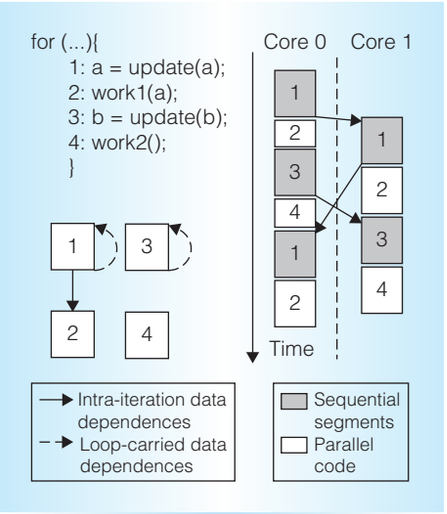

Figure 1. Execution of code produced by Helix for a dual-core processor. Code blocks 1 and 3 must each be executed sequentially, but because they're independent, Helix overlaps them in time.

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 the 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 can run in parallel, as in Figure 1, where sequential segments 1 and 3 overlap.

Transferring data between threads could be a significant source of overhead. However, we've shown that carefully selecting which loops to parallelize keeps the amount of data forwarded between threads small, compared with the amount consumed within each iteration.[1]

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 that only 10 percent of signals remain.[1] Moreover, Helix reduces the perceived signal latency by exploiting the underlying platform's SMT technology. 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 by Chappell et al.,[9] and it was later adapted to different domains.[10,11])

Choosing loops to parallelize

Helix devotes all processor cores to one parallelized loop at a time, so multiple independent loops can't run concurrently, and loops nested within a parallel loop can't 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:

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

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, the added overhead for loop i, is given by

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

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.

Our target platform determines the fixed model parameters S and M. 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. Although some elements of the speedup model might be accurately predictable through static analysis alone, it would be difficult to estimate $Words_i$, the data traffic between threads, without profiling. 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 $maxT$, each representing time savings due to parallelization. A loop's T value is the total amount of time (over all loop invocations) that would be saved by parallelizing that loop. The $maxT$ 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 more deeply, because parallelizing a collection of more deeply nested loops saves more time than parallelizing the current loop. The loops chosen are those for which $T = maxT > 0$.

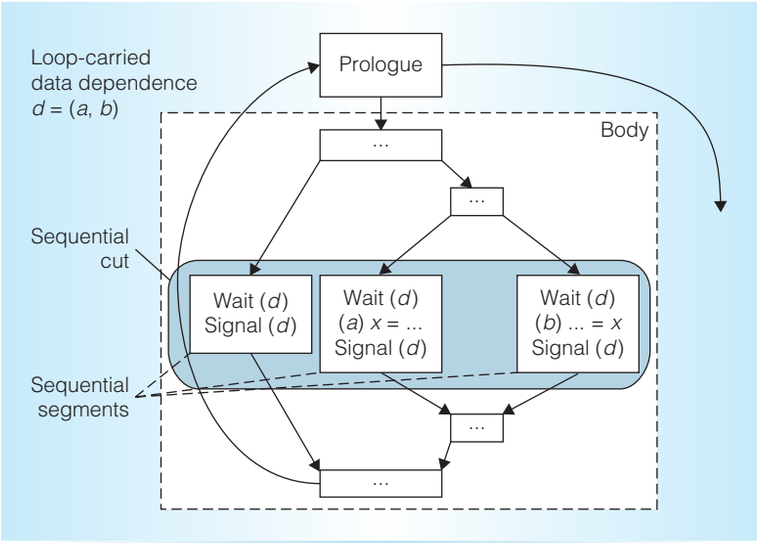

Figure 2. Insertion of *Wait*(*d*) and *Signal*(*d*) due to a RAW data dependence $d = (a, b)$. Sequential segments that contain only Wait and Signal operations handle dependences that span multiple iterations.

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 preceding iteration's prologue 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.

We call the set of data dependences that cross loop boundaries D_{Data}. By applying interprocedural pointer analysis to the whole program,[12] Helix identifies the dependences in D_{Data}, and it associates a sequential segment with each.

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 a thread's execution 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, whereas blocks 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 might need to be added before the loads and after the stores. However, because 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 the first private cache of c_i. The value is not forwarded to the private cache of c_j until the latter issues the corresponding Wait operation (that is, 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 prefetches 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 runtime execution of a parallelized loop with three sequential segments, SS 1, SS 2, and SS 3. 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.

Evaluation

To demonstrate that our approach is suitable for production compilers, we applied Helix to benchmarks from the SPEC CPU2000 suite, and we evaluated the resulting code on a real processor. Helix never slowed programs down, and it compared favorably to DOACROSS, the most similar historical approach to loop parallelization.

Experimental setting

Helix extends the Intermediate Language Distributed Just-In-Time (ILDJIT) compilation framework,[13] which generates native machine code from Common Intermediate Language (CIL) bytecode, so we used GCC4CLI (http://gcc.gnu.org/projects/cli.html) to translate benchmarks written in C to CIL.

Benchmarks. To evaluate our scheme, we used 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 requires too much memory to handle either 176.gcc or 253.perlbmk.[12] Using reference

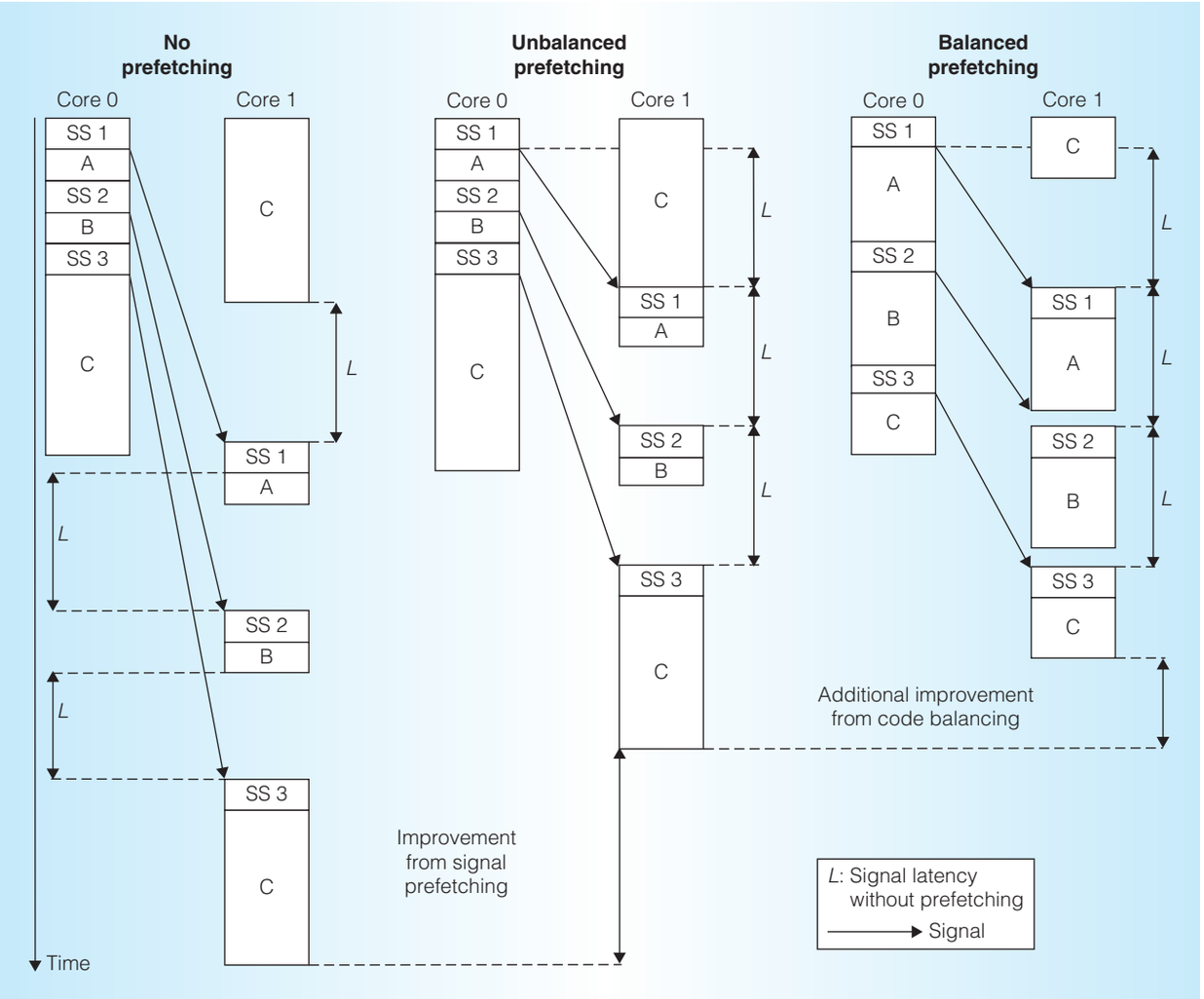

Figure 3. Importance of balanced prefetching. Without helper threads or balanced code scheduling, a parallelized loop with three sequential segments, running on a dual-core processor, might behave as shown on the left. With helper threads but no code balancing, signal prefetching reduces delay, as shown in the middle. Adding balanced code scheduling further reduces delay, as shown on the right.

inputs, we computed speedups resulting from parallelization by running entire benchmarks to completion on a real system.

Hardware platform. Our experiments used 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 32 Kbytes and 256 Kbytes in size. All cores share the last-level 12-Mbyte cache, which is used to forward data values between cores of the same processor through the MESIF cache coherence protocol.

Achievable speedups

Figure 4 shows the measured speedups of entire 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 was 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

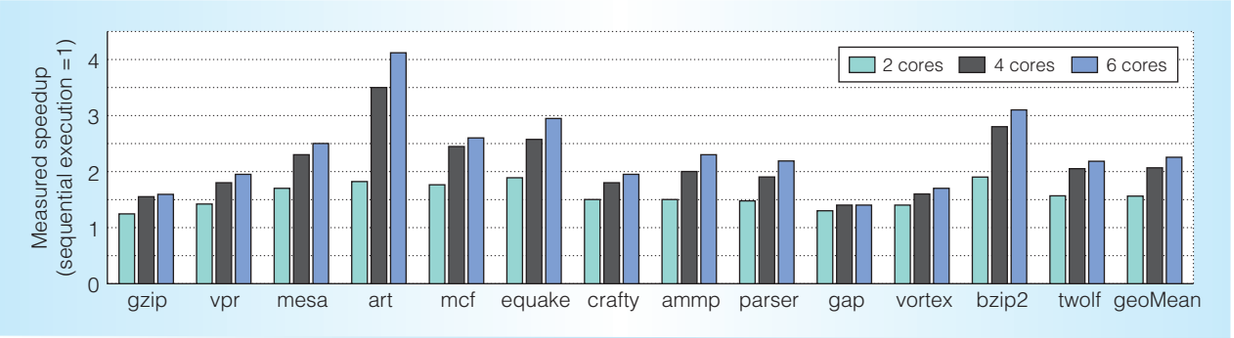

Figure 4. Whole-program speedups of benchmarks from SPEC CPU2000 achieved by Helix, using two, four, or six cores on a real six-core processor.

(experimentally measured using microbenchmarks). (Although the level of speedup achievable by the Helix method is sensitive to the latency of intercore signals, we haven't 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.

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

Avoiding slowdown

To confirm 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 speedups given 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.

Helix doesn't parallelize loops that could slow down execution, because the speedup model we described earlier conservatively overestimates the runtime overhead of a parallelized loop. In fact, because different sequential segments run in parallel (as Figure 1 shows), only a subset of data signals slow down execution, since data signals also execute in parallel.

This overestimation has two consequences for selecting loops to parallelize: every parallelized loop speeds up program execution, but 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 covered a negligible fraction (less than 1 percent) of the original program's total execution time.

Chosen loops. Figure 5 shows the cumulative contributions of successive loops to the speedups given in Figure 4. Loops are sorted in descending order according to the number of clock cycles saved by running them in parallel mode.

The speedup fraction increases monotonically with the number of parallelized loops (parallelizing each successive loop always speeds up the program), and the speedup fraction quickly approaches 100 percent (that is, a few loops yield most of the achievable speedup), though the rate of convergence depends on the benchmark.

Handling overhead. To show Helix's effectiveness at tolerating different levels of communication overhead, we configured it to produce code with different runtime intercore communication overheads. We selectively turned off helper threads, or signal count minimization, or both. Those are the components of Helix that have the greatest 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 6 shows speedups for six cores. The first bar for each benchmark depicts its

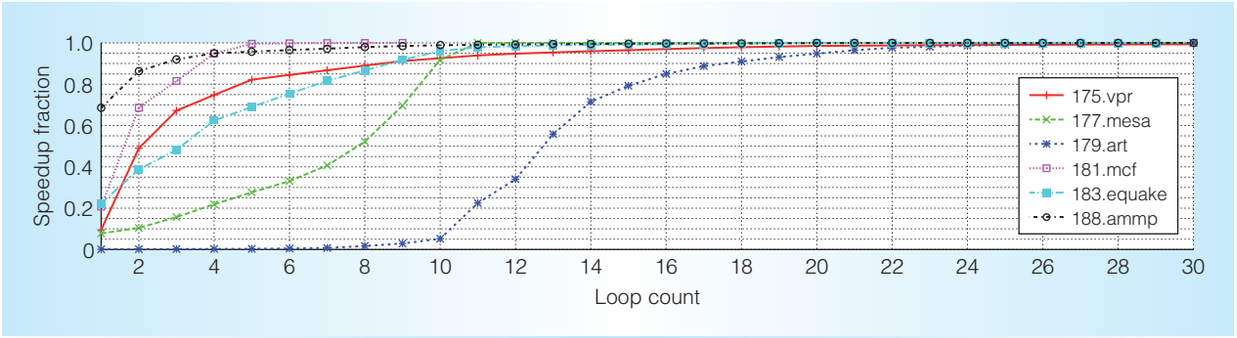

Figure 5. Percentage of speedups given in Figure 4 achievable using a subset of the loops selected for parallelization. The monotonic increase in speedup fraction shows that no loop slowed down because Helix parallelized it. Sorting the loops in decreasing order of cycles saved gives some curves a snakelike shape because of the inverse relationship between cycles saved and speedup.

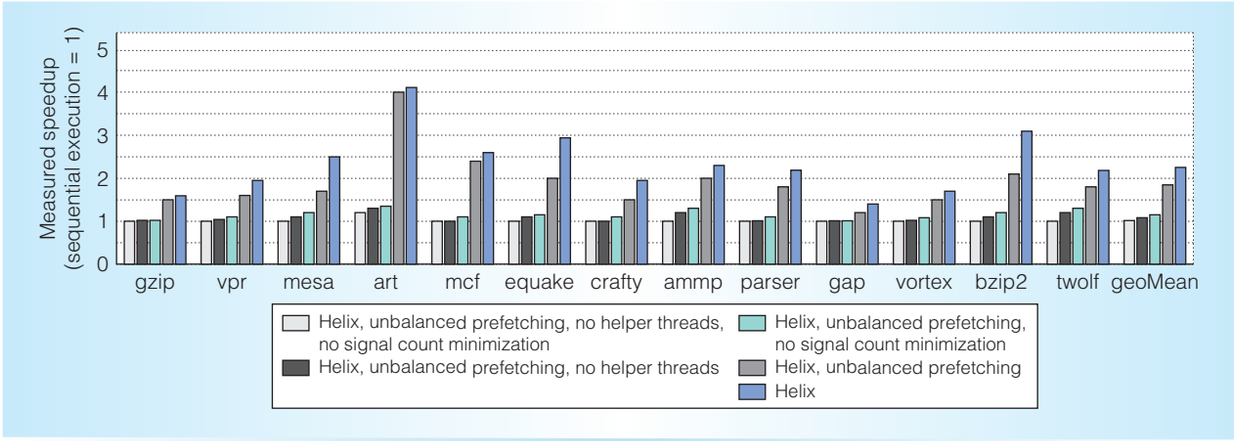

Figure 6. Speedups achieved when signal count minimization and helper threads were disabled, either separately or together. Balanced prefetching was disabled for these measurements.

speedup with every optimization that targets 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.

Comparison with historical approaches

As we mentioned earlier, 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 lets Helix break sequential code into multiple segments and optimize it, liberating code that would otherwise be constrained by intra-iteration data dependences. As a result, the sequential segments for different data dependences can 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 in Figure 6 show what happens when signal prefetching

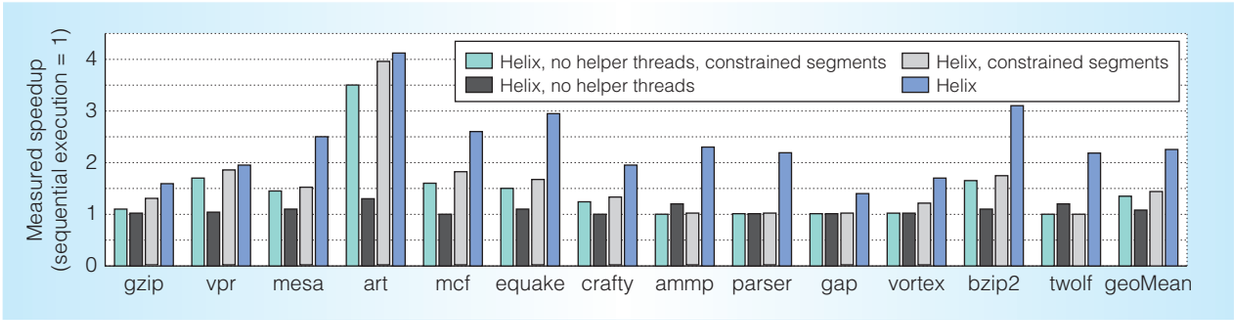

Figure 7. Comparison between grouping sequential segments together and leaving them independent. The first and second bars compare DOACROSS with Helix, without helper threads for either. The third and fourth bars compare them again, with helper threads for both. Combining prefetching with concurrency of sequential segments improves performance significantly.

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 in Figure 6 shows the effect of using both signal optimizations, but without the 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 and fifth bars shows that spacing sequential segments to help the signal-prefetching mechanism improves speedups significantly.

Exploiting parallelism among sequential segments. DOACROSS loop parallelization groups sequential segments together to reduce communication overhead, so we compared Helix with DOACROSS—that is, with the case in which sequential segments are clumped in a single code region.[2] Because the original DOACROSS technique does not include an approach for loop selection, we used the Helix algorithm, tuned for this special case, to select the most profitable loops for DOACROSS. Moreover, we implemented the DOACROSS technique using the Helix approach to synchronizing threads and forwarding data.

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

Figure 7 also illustrates sensitivity to the signaling-overhead reduction performed by helper threads. The difference between the second and fourth bars shows that Helix produces code that's very sensitive to prefetching. On the other hand, code produced by DOACROSS is almost insensitive to this effect (compare the first and third bars). The reason is that DOACROSS is designed to compromise thread-level parallelism in order 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: no sequential segments, only one sequential segment, and more than one sequential segment. The bar shows the execution time spent by the original program in loops of these categories.

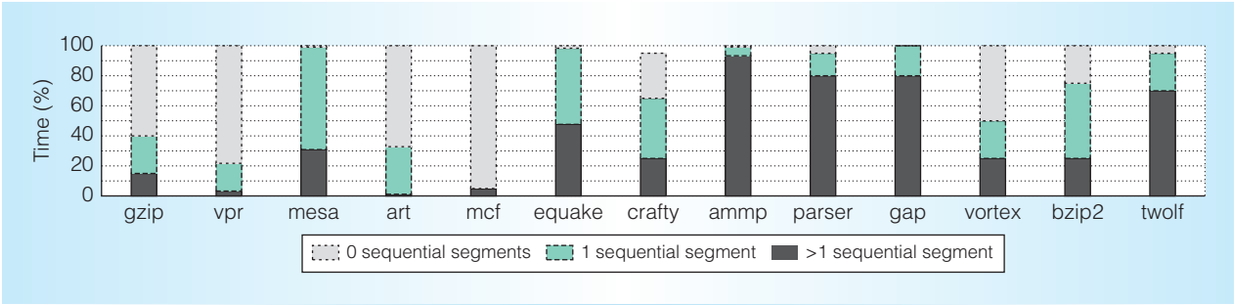

Figure 8. Breakdown of time spent in the parallelized loops used to achieve the speedups shown in Figure 4 based on their numbers of sequential segments. The difference in runtime behavior between DOACROSS and Helix increases with the number of loops having multiple sequential segments.

Because there's 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 spent between 70 and 90 percent of its time in loops with several sequential segments. By exploiting parallelism among those segments, Helix achieved speedups where DOACROSS could not.

Helix avoids slowing down programs, parallelizes code automatically, and doesn't rely on special hardware, so its approach is suitable for inclusion in mainstream compilers. Although a related manual technique has appeared in production compilers, 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 (DSWP).[4,6] 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. MICRO

Acknowledgments

This work was sponsored by Microsoft Research, the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC), the Royal Academy of Engineering, the Engineering and Physical Sciences Research Council, and the National Science Foundation (award number IIS-0926148). Any opinions, findings, conclusions, or recommendations expressed in this article 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., ''HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing,'' *Proc. 10th Ann. IEEE/ACM Int'l Symp. Code Generation and Optimization* (CGO 12), ACM, 2012, pp. 84-93.
2. R. Cytron, ''DOACROSS: Beyond Vectorization for Multiprocessors,'' *Proc. Int'l Conf. Parallel Processing* (ICPP 86), IEEE CS, 1986, pp. 836-844.
3. B. Hertzberg and K. Olukotun, ''Runtime Automatic Speculative Parallelization,'' *Proc. 9th Ann. IEEE/ACM Int'l Symp. Code Generation and Optimization* (CGO 11), IEEE CS, 2011, pp. 64-73.
4. J. Huang et al., ''Decoupled Software Pipelining Creates Parallelization Opportunities,'' *Proc. 8th Ann. IEEE/ACM Int'l Symp. Code Generation and Optimization* (CGO 10), ACM, 2010, pp. 121-130.

5. A. Kotha et al., ''Automatic Parallelization in a Binary Rewriter,'' *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2010, pp. 547-557.
6. G. Ottoni et al., ''Automatic Thread Extraction with Decoupled Software Pipelining,'' *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2005, pp. 105-118.
7. A. Raman et al., ''Speculative Parallelization Using Software Multithreaded Transactions,'' *Proc. 15th Architectural Support for Programming Languages and Operating Systems* (ASPLOS 10), ACM, 2010, pp. 65-76.
8. H. Zhong et al., ''Uncovering Hidden Loop Level Parallelism in Sequential Applications,'' *Proc. IEEE 14th Int'l Symp. High Performance Computer Architecture* (HPCA 2008), IEEE CS, 2008, pp. 290-301.
9. R. Chappell et al., ''Simultaneous Subordinate Microthreading (SSMT),'' *Proc. 26th Ann. Int'l Symp. Computer Architecture* (ISCA 99), IEEE CS, 1999, pp. 186-195.
10. D. Kim et al., ''Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors,'' *Proc. Int'l Symp. Code Generation and Optimization* (CGO 04), IEEE CS, 2004, pp. 27-38.
11. C-K. Luk, ''Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,'' *ACM SIGARCH Computer Architecture News,* vol. 29, no. 2, 2001, pp. 40-51.
12. B. Guo et al., ''Practical and Accurate Low-Level Pointer Analysis,'' *Proc. Int'l Symp. Code Generation and Optimization* (CGO 04), IEEE CS, 2005, pp. 291-302.
13. S. Campanoni et al., ''A Highly Flexible, Parallel Virtual Machine: Design and Experience of ILDJIT,'' *Software—Practice & Experience,* vol. 40, no. 2, 2010, pp. 177-207.

Simone Campanoni is a postdoctoral fellow in the School of Engineering and Applied Sciences at Harvard University. His research interests include code compilation challenges, static and dynamic compilation, runtime optimization, and advanced code analysis to extract coarse-grained parallelism for many-core architectures from general-purpose sequential code. Campanoni has a PhD in information engineering from Politecnico di Milano University.

Timothy M. Jones is a postdoctoral researcher in the Computer Laboratory at the University of Cambridge, where he holds a research fellowship from the Engineering and Physical Sciences Research Council and the Royal Academy of Engineering. His research interests include computer architecture and compiler optimization, with an emphasis on automatic parallelization, power reduction, and the application of machine learning to compiler and microarchitectural design. Jones has a PhD in informatics from the University of Edinburgh. He is a member of IEEE and the ACM.

Glenn Holloway is a systems programmer in the School of Engineering and Applied Sciences at Harvard University. His research interests include static and dynamic code optimization, register allocation, and automatic parallelization. Holloway has an AM in physics from Harvard University.

Gu-Yeon Wei is a Gordon McKay Professor of Electrical Engineering and Computer Science in the School of Engineering and Applied Sciences (SEAS) at Harvard University. He also serves as the associate dean for academic programs at SEAS. His research interests span multiple layers of a computing system, including mixed-signal integrated circuits, computer architecture, and runtime software for automatic code parallelization. Wei has a PhD in electrical engineering from Stanford University.

David Brooks is a Gordon McKay Professor of Computer Science in the School of Engineering and Applied Sciences at Harvard University. His research interests include power-efficient computer system design, variation-tolerant computer architectures, and embedded-system design. Brooks has a PhD in electrical engineering from Princeton University. He is a member of IEEE and the ACM.

Direct questions and comments about this article to Simone Campanoni, Harvard School of Engineering and Applied Sciences, Harvard University, 33 Oxford St., Maxwell Dworkin 308, Cambridge, MA 02138; xan@eecs.harvard.edu.