Limits of dependence analysis for automatic parallelization

Niall Murphy[1], Timothy Jones[1], Simone Campanoni[2], and Robert Mullins[1]

[1] University of Cambridge, Cambridge, UK,
`Niall.Murphy@cl.cam.ac.uk, Timothy.Jones@cl.cam.ac.uk, Robert.Mullins@cl.cam.ac.uk`

[2] Harvard University, Cambridge, MA 02138, United States,
`xan@eecs.harvard.edu`

Abstract. Automatic parallelization is an increasingly important technique for accelerating sequential applications on multicore processors. This approach relies on having an accurate compile-time dependence analysis to identify independent sections of code. Previously it has been assumed that improving this analysis would also improve the performance of parallelized code. In this paper we use novel profiling techniques to see how much room there is for improvement of the compile-time analysis. By feeding this knowledge back into the compiler we simulate a perfectly accurate dependence analysis. Although we find that the compiler does indeed overestimate the number of data dependences, this extra knowledge does not help the compiler to achieve better performance since the remaining dependences still prevent parallelization. However, study of the dynamic nature of these remaining dependences shows that fewer than 1% are realised on every dynamic instance of the instructions involved. We conclude that other avenues, such as speculation, must be explored to surpass current cyclic-multithreading style automatic parallelization efforts.

1 Introduction

Recent industrial trends point towards an increasing number of cores being placed on a single chip. However our ability to exploit parallelism in software is lagging behind. Writing correct and efficient multithreaded code is difficult and many legacy single-threaded applications exist which would be tedious to rewrite. One approach to improve the performance of such code is to automatically parallelize the code in the compiler and this technique has received much attention [1–4]. Automatic parallelization has been assumed to be limited by the accuracy of the dependence analysis it relies on. Therefore, the community has spent a significant amount of effort on improving this dependence analysis [5–7]. While these efforts were justified in the past, there is no evidence that this is still a limitation for today's compilers. What if dependence analysis is already good enough and further enhancements will result in only negligible performance improvements? In this paper we show that the upper limits of compile-time analysis for cyclic-multithreading style parallelization have been reached and that other avenues must be explored for further speedups.

To understand the potential for improving current dependence analysis we start with the HELIX parallelizing compiler [4, 8]. HELIX includes state-of-the-art dependence analysis [6] and has already demonstrated speedups for sequentially-designed programs traditionally considered challenging for automatic parallelization. Then we identify dependences claimed by this analysis which are never actually realized at runtime. These are the only dependences that a hypothetical improved compile-time dependence analysis could eliminate. These apparent dependences can be removed from the compile-time determined data dependence graph (DDG) to create an oracle DDG. This is the upper limit of how accurately the compiler can identify dependences given a specific input set.

By parallelizing the code again, replacing the compile-time DDG with the oracle DDG, we can determine an upper bound on potential speedups for purely static parallelization in HELIX. We found that, although the dependence profiling was able to remove dependences from the graph, for the considered benchmarks no further speedup was obtained. This demonstrates that improving the compile-time analysis beyond the current HELIX algorithm will not result in better performance. However, it should be noted that the oracle DDG includes any dependence which occurred even once during the running of the program. Further study of the remaining dependences in the oracle DDG showed that the majority were only realised occasionally at runtime. We conclude that there is still considerable scope for extracting further parallelism using runtime systems that take advantage of dynamic characteristics such as thread level speculation [9–13]. We are currently working towards determining an upper limit to the performance of thread level speculation using the HELIX infrastructure.

In this paper we will first look at the background of the automatic parallelization infrastructure we used in section 2. In section 3 we discuss the profiling methods we employed to collect a trace of the runtime behaviour of the loop and how we used this to generate the oracle DDG. The results obtained from running the compiler with the oracle DDG are presented in section 4. Section 5 discusses related work and finally we offer some conclusions from our study in section 6.

2 Background

HELIX [4] is a parallelizing compiler which has previously demonstrated speedups when parallelizing the SPEC2000 benchmark suite. HELIX assigns each iteration of a parallelized loop to a separate core and handles cross-iteration dependences by means of synchronization code inserted into the loop. HELIX uses a state-of-the-art interprocedural dependence analysis [6] which builds a conservative data dependence graph (DDG). Since each thread executes on a separate core and so has its own set of registers and its own stack, write-after-read and write-after-write dependences through registers and the stack can be omitted from the graph.

For all dependences which may require forwarding data from one iteration to the next, HELIX inserts synchronization code which forces the instructions involved in the dependence to execute in sequential order. Such instructions are grouped into relatively short sections of code called sequential segments. The parallelized loop body may contain several sequential segments with parallel code in between. When a particular core reaches the start of a sequential segment it issues a *Wait* instruction which halts execution until the corresponding sequential segment of the previous iteration has completed. Upon completion of a sequential segment a *Signal* instruction is executed which indicates to the next iteration that it is safe to execute the corresponding sequential segment. The performance obtained by exploiting this parallelism is measured in [14].

Due to the conservative nature of the compile-time dependence analysis, it is possible that some sequential segments will be generated even though no data is transfered between iterations. This will cause code to be executed sequentially which could in fact be executed in parallel. It is widely believed that an improved compile-time analysis could reduce the size and number of sequential segments, thereby increasing the amount of code which can execute in parallel and enhancing performance. HELIX has previously shown the results of parallelization using a state-of-the-art dependence analysis [8]. Is there value in further improving the analysis to increase DDG accuracy? In this paper we will do a study of the accuracy of the analysis and see what effects an improved analysis would have on performance.

3 Generating the Oracle Data Dependence Graph

Our first task was to build a data dependence graph (DDG) representing the most accurate graph the compiler could produce if it had perfect knowledge of the runtime behaviour of the program. We call this the oracle DDG. Initially we generate a compressed trace of the loop's control flow and memory accesses. This records the ordering of every memory access in the loop, which iteration of the loop it occurred in and what memory locations were read and written. Then the trace was analysed to find which pairs of instructions touched the same memory in different iterations. Each such pair was recorded as an edge in the DDG, thus creating the oracle DDG.

The remainder of this section discusses in more detail how the traces were generated and analysed, as well as the characteristics of the oracle DDG.

3.1 Generating the Traces

It was necessary to find a mechanism whereby the program could be run several times while preserving the identity of instructions from one execution to the next. To achieve this the code is first compiled to the compiler's intermediate representation (IR) and all instructions involved in the loop's control flow or in memory accesses are given a unique ID. The IR and dictionary of IDs are stored on disk and used as a starting point for all subsequent executions.

We run HELIX to produce parallelized code. Memory accesses and loop control flow instructions were instrumented to record two traces. The first trace records each memory location that was written or read by each instruction. The second trace records the instructions that were executed in each iteration of a loop and their order.

Memory Access Trace The memory access trace provides a complete record of every memory event which occurs during the execution of the loop. The trace is compressed using the SD3 scheme [15]. This takes advantage of the fact that memory instructions in a loop typically access memory in a stride fashion. For example, consider the following loop:

```
for ( int i = 0; i < 8; i++ )
    A[i] = 2*B[i];
```

The reads from array `B` will access memory in a sequence such as:

```
0x1100 0x1104 0x1108 0x110C 0x1110 0x1114 0x1118 0x111C
```

This can be represented as a base address (0x1100), a stride (4) and a number of repetitions (8), together referred to as a *memory set entry*. A complete trace for one instruction consists of a sequence of memory set entries. Some instructions, such as those involved in pointer chasing in dynamic data structures, do not compress well with this scheme. To compress these we take advantage of the fact that such instructions tend to be confined to a relatively small portion of the address space. Each memory set entry records the base address as the offset from the previous base address rather than the absolute address.

Control Flow Trace The memory access trace can be used to determine which instructions access the same memory locations but cannot distinguish between accesses which occur in the same iteration or which occur in different iterations. This information is necessary to find inter-iteration data dependences. Therefore a second trace is produced which records which dynamic instances of instructions occurred in which iterations of the loop.

This data is compressed in several ways. Firstly we use a novel compression scheme which takes advantage of the nested patterns which exist in the instruction trace. For example a loop with a number of levels of inner nested loops might produce a raw instruction trace which looks like the following, where each number is the ID of a static instruction:

```
1 2 3 4 5 4 5 4 5 2 3 4 5 4 5 4 5 2 3 4 5 4 5 4 5 6 7
```

To record the nested patterns we use a new grammar which is expressed here in Extended Backus-Naur Form (EBNF):

```
digit               = "0" | "1" | "2" | "3" | "4"
                    | "5" | "6" | "7" | "8" | "9"
instruction-id      = digit{digit}
num-repetitions     = digit{digit}
compression-pattern = "("compression-pattern","num-repetitions")"
                    | "("instruction-id")"
compressed-output   = {compression-pattern}
```

The example raw trace would be compressed to:

```
(1)((2)(3)((4)(5),3),3)(6)(7)
```

We record such a trace for each iteration of the loop. Since each iteration of a loop frequently executes the same instructions, the control flow pattern is stored along with a list of iteration numbers on which it occurs.

3.2 Finding Runtime Dependences

Once the traces have been collected, the dependences can be found by applying the SD3 algorithm for detecting dependences between stride representations of memory accesses. The algorithm is based on comparing pairs of instructions for conflicts. First the memory set entries are converted into intervals which represent the lowest and highest addresses touched by the entry. An interval tree is built for each instruction. This allows us to quickly find which memory set entries overlap between two instructions.

SD3 uses a dependence test known as *Dynamic-GCD*. This algorithm can return a list of all conflicts between two memory set entries in constant time. From the control flow trace we can determine which iterations the conflicting accesses occurred in. If a memory location is written in one iteration and read or written in any other iteration, this is inserted as an edge in the oracle DDG.

3.3 Characteristics of the Oracle DDG

The oracle DDG includes all dependences including read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) dependences. Another potential analysis would be to look at only RAW dependences since theoretically these are the only dependences which cannot be removed. For example, if a memory location is written at the start of each iteration and read later then there is no data flow between iterations through this location. In theory it would be possible to discount this inter-iteration dependence from the DDG, however, removing such dependences would require complex memory renaming techniques which HELIX does not currently provide. Therefore it is necessary to include all WAR and WAW dependences in the oracle DDG to ensure all dependences are satisfied.

The oracle DDG records a dependence edge between every two instructions that every touch the same memory location in the lifetime of the program.

Even if two instructions have many dynamic instances and only alias once, a dependence will still be recorded. Such a dependence would be a prime candidate for parallelization using speculative techniques where we could speculate that the dependence does not occur and perform runtime checks to guarantee correctness. We are also interested in using the same infrastructure to evaluate the potential for such techniques and a preliminary study has shown that a large proportion of dependences in the oracle DDG are not realised on every iteration of the loop.

4 Evaluation

In prior work it has been assumed that automatic parallelization could be improved by simply increasing the accuracy of the compiler's dependence analysis. In this section we will evaluate the validity of this assumption. Firstly, we describe the experimental setup and benchmarks used. The accuracy of the DDG produced by the current HELIX compile-time analysis will be quantified, showing that the compile-time DDG does significantly overstate the number of dependences. We show, however, that even using the oracle DDG which removes these extraneous dependences, the compiler is unable to produce code which performs better than that produced using the compile-time DDG. Further analysis of actual dependences reveals that most of them do not need to be satisfied all the time. This suggests that exploiting the dynamic nature of dependences is crucial and that there is significant scope for gaining performance with speculation.

4.1 Execution Model

To evaluate HELIX performance we run the generated code in an execution model which simulates how the code would run on a multicore system. The code is transformed using the ordinary HELIX optimizations but each iteration of the loop is forced to run sequentially on a single core. Callbacks to the execution model are inserted into the code which update counters with the latency of each basic block as it is executed. The starts and ends of sequential segments are also instrumented with callbacks. The execution model calculates how long each simulated core needs to wait at the start of each sequential segment. Parameters such as the latency to communicate signals between cores are configurable. The execution model produces deterministic results and this makes it easier to compare the performance of HELIX with the compile-time generated DDG versus the oracle DDG.

To evaluate the oracle DDG we modified HELIX to take the DDG as input. HELIX then performs its normal optimizations and transformations and generates sequential segments based on the new DDG. This code is instrumented with the execution model callbacks and executed.

4.2 Benchmarks

To test the performance of the improved DDG we used the *cbench* benchmark suite [16]. This is a set of sequential benchmarks based on the MiBench suite [17] but with the addition of multiple datasets. We intend to do further study of how the oracle DDG depends on the input dataset at a later stage although at the moment our results are based on single datasets. The suite includes applications from various domains such as security, telecommunications and office software.

4.3 Results

First we will look at the difference between the compile-time determined DDG and that produced by the runtime dependence analysis. We will take a sample application: *automotive_susan_e*, an image processing algorithm for detecting edges. Table 1 shows the number of DDG edges identified by both the compile-time dependence analysis and the oracle dependence analysis. Each edge represents a pair of instructions which are identified as being dependent including read-after-write, write-after-read and write-after-write dependences. Coverage indicates the percentage of overall execution time for the program that was spent in that loop. Accuracy indicates the percentage of compile-time identified dependences which were also identified by the oracle analysis.

		DDG edges		
Loop ID	Coverage	Compile-time	Oracle	Accuracy
A	3%	1	0	0%
B	3%	64	56	87%
C	16%	9610	2110	21%
D	16%	9610	1778	18%
E	18%	4	0	0%
F	18%	4	0	0%
G	56%	1	0	0%
H	56%	1	0	0%

Table 1: Accuracy of DDG for loops in automotive_susan_e

For this application we can see that the compile-time analysis has greatly overestimated the number of edges in the DDG. This means that the compile-time analysis identified many pairs of instructions which it determined could potentially have accessed the same memory locations but in reality did not alias even once during execution. Conventional wisdom would suggest that when we use our profile-determined oracle DDG to parallelize, we would be able to exploit more parallelism than previously.

Figure 2 shows the speedups HELIX achieves, relative to sequential execution for each of the loops in automotive_susan_e. These results were obtained running the HELIX execution model simulating a 16-core system. Contrary to

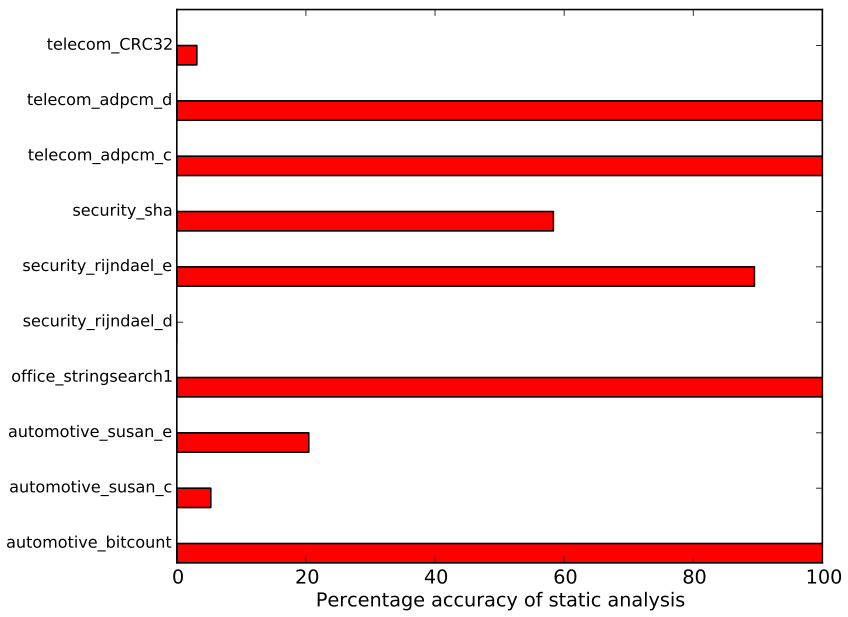

Fig. 1: Percentage of compile-time identified dependences which occurred at least once at runtime

expectation, no significant extra speedup was achieved despite the use of an oracle DDG (100% accuracy). Some of the loops were already achieving close to linear speedup with HELIX and as such there was little extra performance to gain anyway. However loops such as C and D which had no speedup or even a slowdown would have been expected to benefit from the improved DDG.

With a view to further understanding the nature of the remaining dependences in automotive_susan_e we ran another study which recorded the proportion of iterations in which data was passed to an adjacent iteration by each pair of dependent instructions. We found that fewer than 1% of the dependences in the oracle DDG were actually realised on every single iteration. This shows that the oracle DDG, which shows every pair of instructions which transfer data between iterations at least once, is a significant overestimation of the amount of inter-iteration data transfer which actually occurs dynamically. This suggests that there is extra parallelism which can be exploited using runtime execution systems like thread level speculation.

The results for automotive_susan_e are representative of the other applications we tested in cbench. The results can be seen in Figure 3. Although it was possible to improve the accuracy of the DDG for most loops, only one loop in

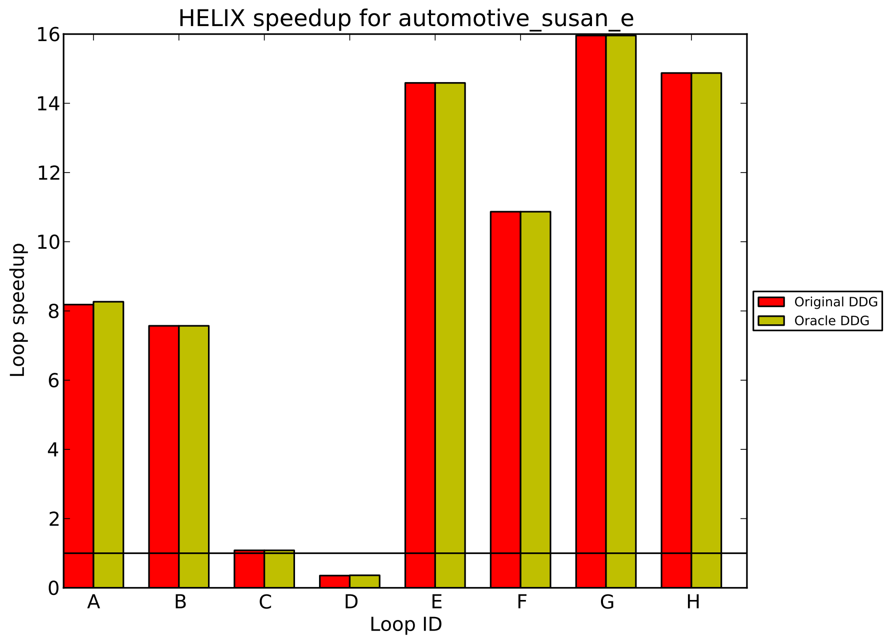

Fig. 2: Parallelization speedup for automotive_susan_e with original and oracle DDGs

security_rijndael_e experienced any speedup improvement when compiled with the oracle DDG.

5 Related Work

This work draws on previous research in various topics. Automatic parallelization has a rich history in the research community. Various authors have had success parallelizing regular code, particularly scientific applications in which dependence analysis is easier to perform accurately [18]. More recently the possibilities for parallelizing more general purpose code with more complex dependence behaviour have been explored. Zhong et al. [3] look at exploiting parallelism in loops by transforming the code to make it more amenable to thread level speculation. However they note that this technique is dependent on accurate dependence analysis and that if they used a more sophisticated analysis they may have achieved better results. Another approach to parallelizing irregular loops is Decoupled Software Pipelining (DSWP) [2]. In this work the authors show empirically that an improved dependence analysis would produce better results in some benchmarks. They suggest that using a dependence analysis such as that proposed by Guo et al. [6] would remove false dependences which harm performance. HELIX [4] uses an interprocedural pointer analysis based on that

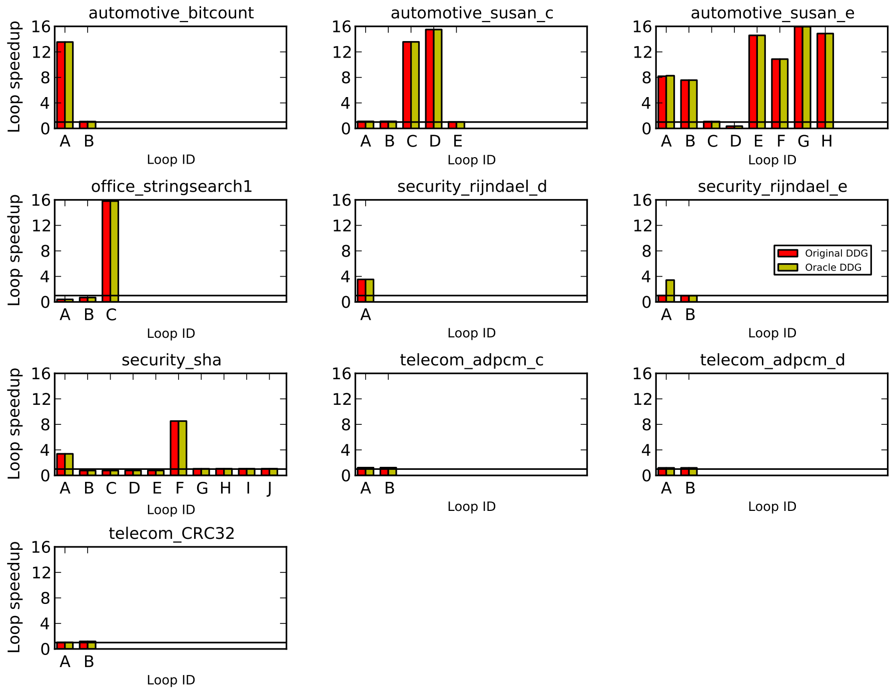

Fig. 3: Parallelization speedup for loops in cbench

proposed by Guo et al. to discover parallelism in loops. In this work we study the accuracy of the dependence analysis used in HELIX and show that that analysis is already sufficient to extract optimal statically parallelized code and that even an oracle analysis which has perfect knowledge of all actual dependences cannot produce better speedups.

We have also built on previous work in dependence profiling. Since static analysis often fails to eliminate false dependences, profiling is often used as a method of enhancing speculative parallelization schemes by detecting such dependences [19]. This paper uses dependence profiling as a means of finding the most accurate data dependence graph the compiler could theoretically produce. The scheme we use to detect dependences is based on that proposed by Kim et al. [15] whereby patterns in memory accesses are exploited to compress the overall amount of data that must be stored. We have expanded on this work by means of the addition of a compressed control flow trace which can be used to identify not only conflicting memory references but also the particular loop iterations in which they occurred.

There is a long history of studying the limits of parallelism in sequential code. Wall [20] examined the extent of instruction level parallelism that could be extracted by collecting a trace of the benchmark and scheduling instructions as early as possible. Austin and Sohi [21] used dynamic dependency graphs to show that more parallelism could be extracted than indicated by Wall but would need to be exploited with memory renaming. Mak and Mycroft [22] expanded on this work by also considering control dependences and proved that these hugely restricted the degree of available parallelism. Lilja [23] presents a survey of techniques for parallelizing loops and a technique for determining the maximum theoretical parallelism available in a loop although no results for actual benchmarks are shown. Vachharajani et al. [24] investigated the amount of parallelism which could be extracted using chip-multiprocessors (CMPs) taking into account constraints such as inter-core communication latency. This study showed that there was ample parallelism available for CMPs to exploit but that performance would not scale past 16 cores without using new techniques such as speculation. Our work focuses on loop level parallelism and also uses a practical execution model to see what speedups are possible when the maximum available parallelism is exploited.

6 Conclusions

In this paper we have evaluated the limits of compile-time analysis as a tool for improving the performance of automatic parallelization. We have simulated a perfect dependence analysis by collecting a complete trace of runtime memory events, building from this an oracle DDG and feeding the results back into the compiler to produce optimally parallelized code. We found that there were indeed many shortcomings in the dependence analysis resulting in numerous dependences being identified which were never realised during execution. However, the removal of these erroneous dependences did not result in improved performance in most cases. From this we conclude that improving compile-time analysis alone will not be enough to get better speedups with automatic parallelization. Our dependence profiling has demonstrated that there is a significant difference between the number of dependences which are realised at least once and the number of dependences which are realised all the time. We conclude that there is still considerable parallelism available to be exploited but that speculation would be required to take advantage of the dynamic behaviour of the code. We are currently working on an execution model which will simulate a system using thread level speculation to take advantage of such behaviour.

References

1. Alexander Aiken and Alexandru Nicolau, "Perfect Pipelining: A New Loop Parallelization Technique," pp. 221–235, 1988.
2. Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August, "Automatic Thread Extraction with Decoupled Software Pipelining," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2005, MICRO 38, pp. 105–118, IEEE Computer Society.
3. Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *In Proc. of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

4. Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay J. Reddi, Gu Y. Wei, and David Brooks, "HELIX: automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, New York, NY, USA, 2012, CGO '12, pp. 84–93, ACM.
5. Michael Hind, "Pointer Analysis: Haven'T We Solved This Problem Yet?," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, New York, NY, USA, 2001, PASTE '01, pp. 54–61, ACM.
6. Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August, "Practical and Accurate Low-Level Pointer Analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2005, CGO '05, pp. 291–302, IEEE Computer Society.
7. Nick P. Johnson, Taewook Oh, Ayal Zaks, and David I. August, "Fast Condensation of the Program Dependence Graph," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2013, PLDI '13, pp. 39–50, ACM.
8. Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu Y. Wei, and David Brooks, "HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 217–228, June 2014.
9. Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar, "Multiscalar processors," *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 414–425, May 1995.
10. Lance Hammond, Mark Willey, and Kunle Olukotun, "Data speculation support for a chip multiprocessor," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 58–69, Oct. 1998.
11. J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry, "A Scalable Approach to Thread-level Speculation," *SIGARCH Comput. Archit. News*, pp. 1–12, May 2000.
12. Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August, "Speculative Decoupled Software Pipelining," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Washington, DC, USA, 2007, PACT '07, pp. 49–59, IEEE Computer Society.
13. Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta, "Enhanced speculative parallelization via incremental recovery," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, New York, NY, USA, 2011, PPoPP '11, pp. 189–200, ACM.
14. S. Campanoni, T. M. Jones, G. Holloway, Gu-Yeon Wei, and D. Brooks, "Helix: Making the Extraction of Thread-Level Parallelism Mainstream," *Micro, IEEE*, vol. 32, no. 4, pp. 8–18, July 2012.
15. Minjang Kim, Hyesoon Kim, and Chi-Keung Luk, "SD3: A Scalable Approach to Dynamic Data-Dependence Profiling," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on.* Dec. 2010, pp. 535–546, IEEE.
16. "cbench: Collective benchmarks," http://www.ctuning.org/cbench, Accessed: 2014-12-04.
17. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001*

IEEE International Workshop, Washington, DC, USA, 2001, vol. 0 of *WWC '01*, pp. 3–14, IEEE Computer Society.

18. J. T. Lim, A. R. Hurson, K. Kavi, and B. Lee, "A loop allocation policy for DOACROSS loops," in *Parallel and Distributed Processing, 1996., Eighth IEEE Symposium on.* Oct. 1996, pp. 240–249, IEEE.
19. Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F. P. O'Boyle, "Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping," *SIGPLAN Not.*, vol. 44, no. 6, pp. 177–187, June 2009.
20. David W. Wall, "Limits of Instruction-level Parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, Apr. 1991, vol. 19 of *ASPLOS IV*, pp. 176–188, ACM.
21. Todd M. Austin and Gurindar S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 342–351, Apr. 1992.
22. Jonathan Mak and Alan Mycroft, "Limits of Parallelism Using Dynamic Dependency Graphs," in *Proceedings of the Seventh International Workshop on Dynamic Analysis*, New York, NY, USA, 2009, WODA '09, pp. 42–48, ACM.
23. D. J. Lilja, "Exploiting the parallelism available in loops," *Computer*, vol. 27, no. 2, pp. 13–26, Feb. 1994.
24. Neil Vachharajani, Matthew Iyer, Chinmay Ashok, Manish Vachharajani, David I. August, and Daniel Connors, "Chip Multi-processor Scalability for Single-threaded Applications," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 44–53, Nov. 2005.