# Revisiting the Sequential Programming Model for the Multicore Era

Automatic parallelization has thus far not been successful at extracting scalable parallelism from general programs. An aggressive automatic thread extraction framework, coupled with natural extensions to the sequential programming model that allow for a range of legal outcomes rather than forcing programmers to define a single legal program outcome, will let programmers achieve the performance of parallel programming via the simpler sequential model.

**Matthew J. Bridges**
**Neil Vachharajani**
**Yun Zhang**
**Thomas Jablin**
**David I. August**
Princeton University

Processor manufacturers can no longer rely on increasing uniprocessor clock speed or microarchitectural improvements to provide performance improvements that continue past trends. Meanwhile, transistor count continues to grow exponentially, leading processor manufacturers to place multiple cores on a die. Machines with four or more cores are already shipping, and tomorrow's machines promise still more cores (see, for example, http://www.intel.com/pressroom/archive/releases/Intel_New_Processor_Generations.pdf). Unfortunately, most applications are not multithreaded—neither manually nor automatically—so adding cores leads to little if any performance improvement.

As the many publications dealing with decades-old issues of deadlock, livelock, and so on testify,[1–3] writing a parallel program that is correct and that outperforms its single-threaded counterpart is difficult. Such manual parallelizations are often hand-optimized to a specific number of cores or a specific kind of machine. This leads to performance that is tightly coupled to an underlying architecture, making it hard to port the application without significant investment. Because of this, a parallelization that is effective across a range of architectures approaches the impossible.

Researchers have proposed many new languages to reduce the difficulty of writing a parallel program.[4–6] These languages make parallel programming easier than in the past, allowing programmers to express high-level parallelism without the need for low-level primitives, such as locks. Unfortunately, because of the complexity of parallel programming, programmers often extract only the easy-to-obtain parallelism, leaving large regions of sequential code. Because the speedup of a parallel program is limited by the execution of the longest sequential



Published by the IEEE Computer Society



IEEE Micro micr-28-01-brid.3d 28/1/08 11:11:47 12 Cust # brid

region, additional parallelization is required for such programs to run efficiently, particularly as the number of cores grows.

Given these issues and the large body of single-threaded legacy applications, researchers have pursued techniques to extract threads from single-threaded programs without programmer intervention. Unfortunately, except in the scientific domain, automatic parallelization techniques have not proven successful enough for industry adoption.

This article shows how to bridge the gap between the simplicity of the sequential programming model and the performance potential of the parallel programming model. We propose an aggressive, full-featured compiler infrastructure for thread extraction that uses the sequential programming model. This infrastructure should bring together many techniques from compiler analysis and compiler optimization, including those that require hardware support, and apply them interprocedurally to any loop in the program, particularly outermost loops. To address the constraints that have inhibited previous automatic thread extraction techniques, we augmented the sequential programming model with natural, simple annotations that provide information to the compiler. This information lets the programmer specify that multiple, legal outcomes of a program exist, freeing the compiler from having to maintain the single correct output required by the sequential programming model. This article proposes two annotations, *Y-branch* and *Commutative,* which let the compiler extract scalable parallelism from several applications without resorting to the use of parallel concepts, making these annotations easy for programmers to use.

As a case study of the potential of the framework and annotations, we manually parallelized the SPEC CINT2000 benchmarks.[7] Compiler writers familiar with the technology performed the parallelization—acting, when possible, as a modern parallelizing compiler could be expected to perform. We present several applications from the SPEC CINT2000 benchmark suite to illustrate how an aggressive parallelization framework and annotations to the sequential programming model facilitate automatic thread extraction.

## A framework for automatic parallelization

An aggressive parallelization framework can produce scalable parallelism without changes to the programming model. This framework includes a compiler infrastructure to identify and extract parallelism in sequential code, plus hardware to efficiently execute the parallelized code.

To see how such a framework would extract parallelism, consider the 256.bzip2 application from the SPEC CINT2000 benchmark suite. This application compresses or decompresses a file using the Burrows-Wheeler transform and Huffman encoding. Here, we focus only on the benchmark's compression portion. Each iteration of the `compressStream` function, shown in Figure 1a, compresses an independent block of data. The data from the file passes through a run-length encoding filter and is stored in the global `block` data structure. Because of this, the `block` array contains a varying number of bytes per data block, indicated by `last`. Assuming data bytes exist to compress, the `doReversibleTransform` and `generateMTFValues` functions compress them before `sendMTFValues` appends them to the output stream.

Versions of the bzip2 algorithm that compress independent blocks in parallel have been implemented in parallel-programming paradigms (see http://compression.ca/pbzip2). The parallelization performed manually with mutexes and locks decomposes the original loop in `compressStream` into three separate loops, shown in Figure 1b. The first stage reads in blocks to compress, followed by a stage to compress, finishing with a stage to print the compressed blocks in order. Most of the parallelism extracted comes from executing multiple iterations of the second stage in parallel.

The same pipelined parallelism extracted manually can also be extracted by the decoupled software pipelining (DSWP) technique.[8,9] Unfortunately, in its base version, this technique cannot extract the DOALL, or iteration-level, parallelism present in the manual parallelization, because it

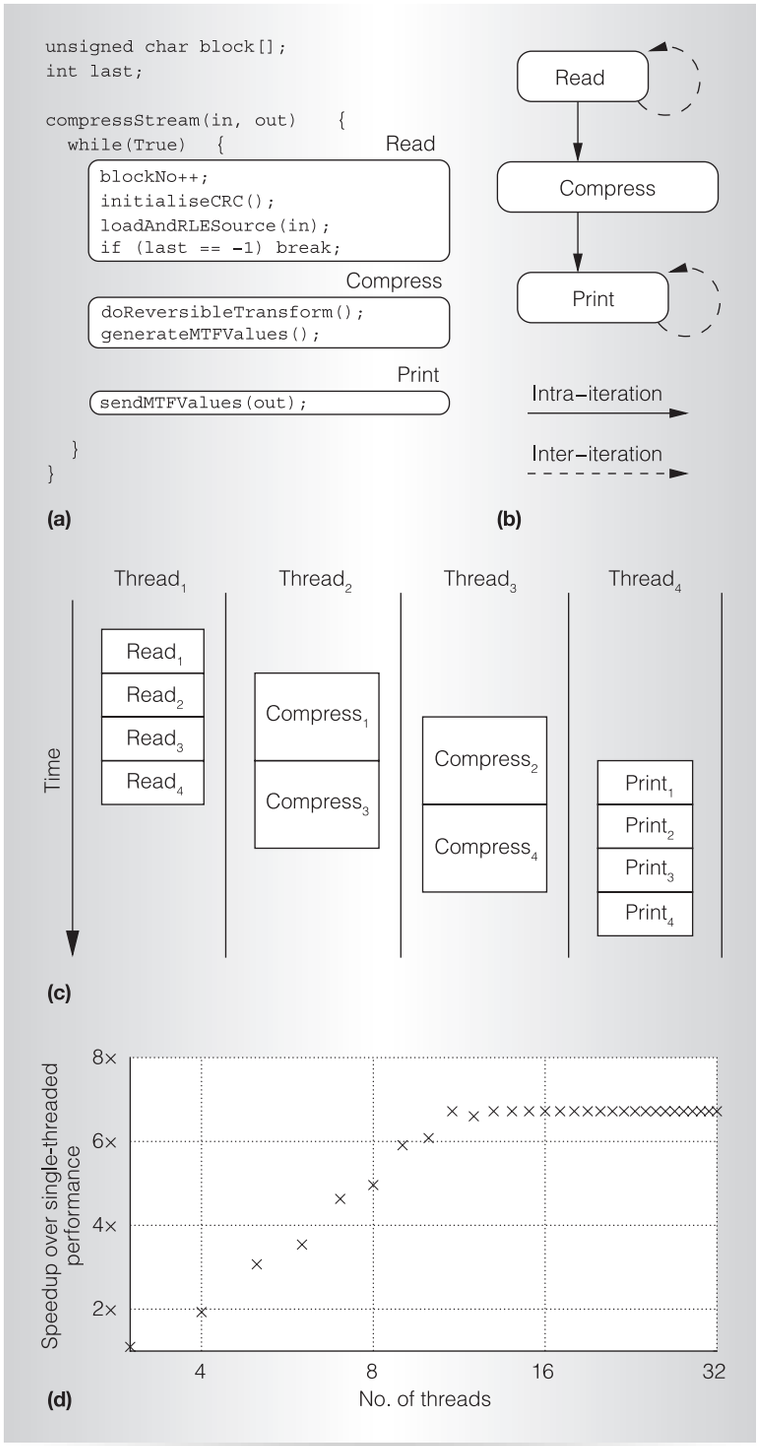



Figure 1. Parallelization diagram: pseudocode for **compressStream** from the SPEC CINT2000 256.bzip2 application (a), static dependencies (b), dynamic execution plan (c), and multithreaded speedup for 256.bzip2 (d).

only partitions the static body of a loop. In the case of 256.bzip2, DSWP could extract a four-stage pipeline, placing **doReversibleTransform** and **generateMTFValues** in their own pipeline stages. However, in practice this limits the speedup achievable to about 2× because **doReversibleTransform** takes about 50 percent of the loop's runtime. Instead, because no loop-carried dependencies exist in the Compress stage, it can be replicated several times, allowing multiple iterations to execute in parallel. This extension to DSWP, called parallel stage DSWP (PS-DSWP), allows the extraction of DOALL-like parallelism.[10]

Unfortunately, although the compress stage doesn't contain any actual loop-carried dependencies, the compiler is unable to prove this. In particular, the compiler can't prove that the accesses to **block** are limited by **last**, and must conservatively assume that writes from a previous iteration can feed-forward around the back edge to the next iteration. Adding speculation to DSWP lets the compiler break these dependencies. In speculative DSWP (SpecDSWP), the compiler speculates dependence recurrences in general, and loop-carried dependencies in particular.[11] To avoid excessive misspeculation, the framework uses a memory profiler to find memory dependencies that rarely—or, in the case of **block**, never—manifest themselves. Because breaking loop-carried dependencies is the key to PS-DSWP, the memory profiler profiles memory dependencies relative to loops, letting it determine if a dependence manifests itself within an iteration or between iterations.

Memory locations reused across iterations, such as the **block** array, present another problem because the same memory location is read and written during every iteration, leading to many loop-carried, false memory dependencies. To break these false dependencies, the framework places each iteration in a separate ordered memory transaction, similar to an epoch in STAMPede.[12]

Finally, the parallelization technique must parallelize not just a loop but also the functions called, directly or indirectly, from that loop. In 256.bzip2, the read and print phases are the **bsR** (bitstream read)



IEEE Micro micr-28-01-brid.3d 28/1/08 11:11:49 14 Cust # brid

and **bsW** (bitstream write) functions, respectively. These functions are found deep in the call tree, making it hard to expose the code inside for parallelization. Inlining until all calls to these functions are visible at the outermost loop level is not practical, because the loop quickly becomes too large to analyze. Because the base DSWP technique uses a program dependence graph (PDG) to facilitate parallelization, the system dependence graph (SDG)[13] can be used to facilitate interprocedural parallelization. Whole-program optimization techniques can also solve this problem.[14]

The framework can extract a parallelization that executes according to the dynamic execution plan shown in Figure 1c. As the number of compress-stage threads increases, performance increases, as Figure 1d shows. The only limitations on the speedup obtained are the input file's size and the compression level. Because the file size is only a few megabytes and the compression level is high, there are only a few independent blocks to compress in parallel. After 11 threads, 256.bzip2 contains no more work to perform in parallel.

## Extending the sequential programming model

Although an aggressive parallelization framework can automatically extract threads, the sequential programming model constrains the framework to extract threads while enforcing a single legal program outcome. This limits performance in many cases where the programmer wished to allow a range of legal outcomes, but had no means to communicate this to the framework.

### Nondeterministic branches

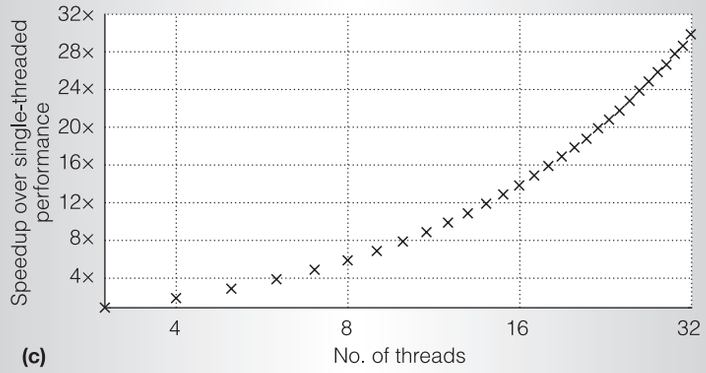
Many compression applications make a fundamental trade-off between the amount of compression achieved and the program's runtime performance. An example of this class of applications is 164.gzip, also from the SPEC CINT2000 benchmark suite. The 164.gzip application uses the Lempel-Ziv 1977 algorithm to compress and decompress files. As with 256.bzip2, we focus on the compression side. Unlike 256.bzip2, the choice in 164.gzip of when to end compression of the current block and begin a new block depends on various factors related to the compression achieved on the current block. This dependence makes it impossible to compress blocks in

```
1   #define CUTOFF 100000
2   dict = start_dictionary();
3   int count = 0;
4   while ((char = read(1)) != EOF) {
5     profitable = compress(char, dict)
6
7     if (!profitable) {
8       dict = restart_dictionary(dict);
9     } else if (count == CUTOFF) {
10      dict = restart_dictionary(dict);
11      count = 0;
12    }
13    count++;
14  }
15  finish_dictionary(dict);
```

(a)

```
1   dict = start_dictionary();
2   while ((char = read(1)) != EOF) {
3     profitable = compress(char, dict)
4
5     @YBRANCH(probability=.00001)
6     if (!profitable)
7       dict = restart_dictionary(dict);
8   }
9   finish_dictionary(dict);
```

(b)




(c)

Figure 2. Use of *Y-branch* in 164.gzip: pseudocode of manual parallelization of 164.gzip (a), pseudocode for **deflate** from 164.gzip (b), and multithreaded speedup for 164.gzip (c).



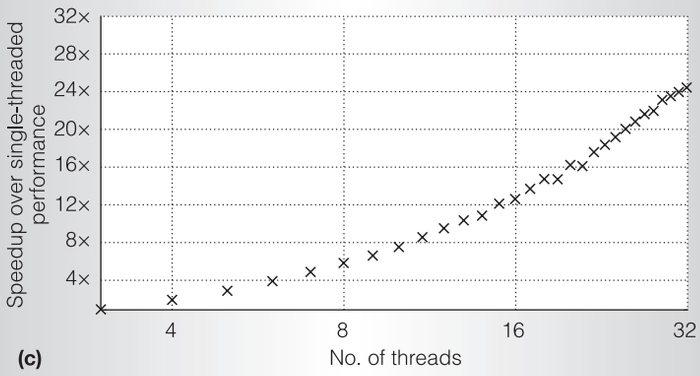
IEEE Micro micr-28-01-brid.3d 28/1/08 11:11:53 15 Cust # brid

```
1  void batch_process() {
2    errors = 0;
3    while (True) {
4      sentence = read();
5      if (!sentence) break;
6      printf(sentence);
7      err = parse(sentence);
8      errors += err;
9    }
10 }
```

(a)

```
1  char  *memory;
2  int current = 0;
3
4  @Commutative
5  void  *xalloc(size_tsize) {
6    void  *ptr = memory + current;
7    current += size;
8    return ptr;
9  }
```

(b)




(c)

Figure 3. Use of *Commutative* in the 197.parser application: pseudocode for **batch_process** (a) and memory allocation function **xalloc** (b) from 197.parser, and multithreaded speedup for 197.parser (c).

parallel, as in 256.bzip2, because the compiler can't predict the point at which a new block of compression will begin.

Manually parallelized versions of the gzip algorithm insert code to ensure that a new block starts at fixed intervals, leading to a parallelization similar to that extracted for 256.bzip2. When parallelizing this algorithm by hand (Figure 2a), the developer must make a choice that trades off parallel performance with output quality. Instead, this flexibility should be given to the compiler, which is often better at targeting the unique features of the machine for which it is compiling. Inspired by Wang et al.,[15] we propose the use of a Y-branch annotation in the source code. The semantics of a Y-branch allow the true path of any dynamic instance of the branch to be taken, regardless of the condition of the branch. The compiler is then free to generate code that pursues this path when it is profitable to do so. In particular, this lets the compiler balance the quality of the output with the parallel performance likely to be achieved.

Figure 2b illustrates how the Y-branch annotation can be used. Rather than inserting code to split the input up into multiple blocks, as in Figure 2a, the Y-branch communicates to the compiler that it can control when a new dictionary is started, allowing it to choose an appropriate block size. This gives the compiler the ability to break dependencies related to the dictionary, and to extract multiple threads. A probability argument informs the compiler of the relative importance of compression to performance. In the case of Figure 2b, a probability of 0.00001 was chosen to indicate that the dictionary should not be reset until 164.gzip has compressed at least 100,000 characters. Determination of the proper probability is left to a profiling pass or the programmer.

When we obtain this parallelization from 164.gzip, the only limitations to performance are the same as those in 256.bzip2. However, the block size is smaller than in **256.bzip**, causing the better performance scalability shown in Figure 2c.

## Commutative functions

Parallelization is also inhibited by strict enforcement of dependencies among functions whose call sites the programmer meant to be interchangeable, but which contain internal dependences. An example of this phenomenon occurs in the 197.parser application, which parses a series of sentences, analyzing them to see if they are grammatically correct. The loop in the **batch_process** function, shown in Figure 3a, is the outermost loop, of which



IEEE Micro micr-28-01-brid.3d 28/1/08 11:11:57 16 Cust # brid

the **parse** function call dominates the runtime of each iteration. Because each sentence is grammatically independent of every other sentence, all sentences can undergo parsing in parallel.

Unfortunately, dependence recurrences arising from the memory allocator prevent such a parallelization. Figure 3b shows the pseudocode of the memory allocator function **xalloc**. Upon startup, the memory subsystem allocates a 60-Mbyte chunk of memory, portions of which are returned by **xalloc**. The dependence recurrence on the **current** variable causes a large DSWP stage that PS-DSWP can't replicate. This, in turn, prevents the extraction of scalable parallelism.

Fortunately, **xalloc**, like many memory allocation functions, has the property that multiple calls to it are interchangeable even though it maintains internal state, so long as each call executes atomically. Just as we used the Y-branch annotation to give the compiler more information about legal execution orders, we now introduce another annotation, *Commutative,* which informs the compiler that the calls to **xalloc** can occur in any order.

In general, the Commutative annotation lets the developer leverage the notion of a commutative mathematical operator, even when, because of underlying dependences, a function is not actually commutative according to the traditional sequential programming model. By declaring a function as Commutative when it isn't, the programmer is allowing the program's execution order to change. In particular, Commutative allows an execution order that leads to potentially different values in different memory locations than the sequential version. By annotating the function with Commutative, the programmer indicates that such differences are legal. Also, the programmer annotates Commutative on the basis of the function's intended behavior, not its potentially many implementations.

The semantics of the Commutative annotation dictate that, outside the function, the function call's outputs depend only on that function call's inputs. Any internal dependence recurrences are ignored during the process of parallelization. This lets the compiler reorder calls to a Commutative function without the internal dependence getting in the way. The Commutative function itself executes atomically when called and, inside the function, dependencies local to the function are respected. This ensures that a well-defined sequence of calls to the Commutative function exists. For details about the use of Commutative among multiple functions that reference that same global variables or its use in speculative execution, see our original publication on this topic.[7]

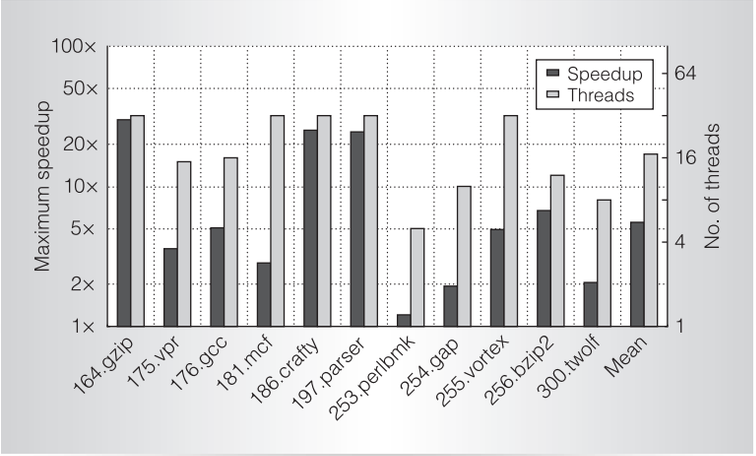



Figure 4. Maximum speedup achieved on up to 32 threads over single-threaded execution (black bars) and minimum number of threads at which the maximum speedup occurred (gray bars).

Once the framework uses Commutative to hide the dependence recurrence in **xalloc**, it can achieve a parallelization that scales well with the number of cores, as Figure 3c shows. The biggest limitation on the speedup is the variation among sentence processing times, which can be up to 100:1.

## Parallelizing SPEC CINT2000

To explore the potential for parallelism, we chose to parallelize the SPEC CINT2000 benchmark suite, not because it is considered amenable to automatic parallelization, but because it is not. Changing only 60 out of more than 500,000 lines of code, we achieved an average speedup of 454 percent targeting a 32-core system. Figure 4 shows the maximum speedup achieved for each benchmark, and the minimum number of threads



IEEE Micro micr-28-01-brid.3d 28/1/08 11:12:02 17 Cust # brid

**Table 1. Relating speedup to historic performance trends.**

| Benchmark | No. of threads | Speedup | Historic performance trend* | Ratio** |
|---|---|---|---|---|
| 164.gzip | 32 | 29.91 | 5.38 | 5.56 |
| 175.vpr | 15 | 3.59 | 3.71 | 0.97 |
| 176.gcc | 16 | 5.06 | 3.84 | 1.32 |
| 181.mcf | 32 | 2.84 | 5.38 | 0.53 |
| 186.crafty | 32 | 25.18 | 5.38 | 4.68 |
| 197.parser | 32 | 24.50 | 5.38 | 4.55 |
| 253.perlbmk | 5 | 1.21 | 2.18 | 0.55 |
| 254.gap | 10 | 1.94 | 3.05 | 0.64 |
| 255.vortex | 32 | 4.92 | 5.38 | 0.91 |
| 256.bzip2 | 12 | 6.72 | 3.34 | 2.01 |
| 300.twolf | 8 | 2.06 | 2.74 | 0.75 |
| Geometric mean | 17 | 5.54 | 3.97 | 1.39 |
| Arithmetic mean | 20 | 9.81 | 4.16 | 2.04 |

* Speedup needed to maintain existing performance trends, assuming 1.4× speedup per doubling of cores
** Ratio of actual speedup to expected historic speedup

at which the maximum speedup was achieved. The speedups are not an upper bound on the maximum speedup obtainable, because usually only one loop in the application was parallelized.

Table 1 presents the actual numbers for Figure 4. Additionally, Table 1 gives the historic performance trend, which details the expected performance increase for the number of transistors used. No statistics are available that directly relate the doubling of cores to performance improvement. However, historically, the number of transistors on a chip has doubled every 18 months, while performance has doubled every 36 months. Assuming that all new transistors are used to place new cores on a chip, each doubling of cores must yield approximately 1.4× speedup to maintain existing performance trends. Thus, the "historic performance trend" column represents the expected speedup for the number of threads, calculated as $(\text{no. threads})^{485}$. The final column gives the ratio of the actual performance improvement to that required to maintain the 1.4× speedup. The overall performance improvement indicates that the framework can extract sufficient parallelism to make good use of the resources of current and future many-core processors.

Automatic parallelization techniques are necessary to extract performance not only for the large body of sequential applications but also for the sequential portions of parallel threads. Without these techniques, performance on today's and tomorrow's processors will suffer. A proper framework allows the extraction of large amounts of parallelism; we are currently working toward making this framework a reality. For applications that this framework does not parallelize, simple additions to the standard sequential programming model will allow the framework to parallelize them. Ultimately, the framework and annotations will allow software programmers to develop within the simpler sequential programming model, while also obtaining the performance normally associated with a parallel programming model. MICRO

## Acknowledgments

We thank the entire Liberty Research Group for their feedback during this work. Additionally, we thank the anonymous reviewers for their insightful comments. The authors acknowledge the support of the GSRC Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work has also been supported by Intel. Opinions,

findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of our sponsors.

## References


1. J.C. Corbett, ''Evaluating Deadlock Detection Methods for Concurrent Software,'' *IEEE Trans. Software Engineering*, vol. 22, no. 3, Mar 1996, pp. 161-180.
2. P.A. Emrath and D.A. Padua, ''Automatic Detection of Nondeterminacy in Parallel Programs,'' *Proc. Workshop Parallel and Distributed Debugging*, ACM Press, 1988, pp. 89-99.
3. G.R. Luecke et al., ''Deadlock Detection in MPI Programs,'' *Concurrency and Computation: Practice and Experience*, vol. 14, no. 11, Aug. 2002, pp. 911-932.
4. B.D. Carlstrom et al., ''The Atomos Transactional Programming Language,'' *Proc. Conf. Programming Language Design and Implementation* (PLDI 06), ACM Press, 2006, pp. 1-13.
5. M. Frigo, C.E. Leiserson, and K.H. Randall, ''The Implementation of the Cilk-5 Multithreaded Language,'' *Proc. Conf. Programming Language Design and Implementation* (PLDI 98), ACM Press, 1998, pp. 212-223.
6. M.I. Gordon et al., ''A Stream Compiler for Communication-Exposed Architectures,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 291-303.
7. M.J. Bridges et al., ''Revisiting the Sequential Programming Model for Multi-Core,'' *Proc. Int'l Symp. Microarchitecture* (MICRO 07), IEEE CS Press, 2007, pp. 69-81.
8. G. Ottoni et al., ''Automatic Thread Extraction with Decoupled Software Pipelining,'' *Int'l Symp. Microarchitecture* (MICRO 05), IEEE CS Press, 2005, pp. 105-118.
9. R. Rangan et al., ''Decoupled Software Pipelining with the Synchronization Array,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 04), IEEE CS Press, 2004, pp. 177-188.
10. E. Raman et al., ''Parallel-Stage Decoupled Software Pipelining,'' to appear in *Proc. Int'l Symp. Code Generation and Optimization* (CGO 08), IEEE CS Press, 2008.
11. N. Vachharajani et al., ''Speculative Decoupled Software Pipelining,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 07), IEEE CS Press, 2007, pp. 49-59.
12. J.G. Steffan et al., ''The STAMPede Approach to Thread-Level Speculation,'' *ACM Trans. Computer Systems*, vol. 23, no. 3, Aug 2005, pp. 253-300.
13. S. Horwitz, T. Reps, and D. Binkley, ''Interprocedural Slicing Using Dependence Graphs,'' *ACM Trans. Programming Languages and Systems*, vol. 12, no. 1, Jan 1990, pp. 26-60.
14. S. Triantafyllis et al., ''A Framework for Unrestricted Whole-Program Optimization,'' *Proc. Conf. Programming Language Design and Implementation* (PLDI 06), ACM Press, 2006, pp. 61-71.
15. N. Wang, M. Fertig, and S.J. Patel, ''Y-Branches: When You Come to a Fork in the Road, Take It,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 03), IEEE CS Press, 2003, pp. 56-67.



**Matthew J. Bridges** is pursuing a PhD in the Department of Computer Science at Princeton University. His research interests include compilers, compiler architecture, and programming languages, with an emphasis on parallelism extraction. Bridges has a HBS in computer and information science from the University of Delaware and an MA in computer science from Princeton University. He is a student member of the ACM.

**Neil Vachharajani** is pursuing a PhD in the Department of Computer Science at Princeton University. His research interests include compilers, compiler architecture, and programming languages. Vachharajani has a BSE in electrical engineering and an MA in computer science from Princeton University. He is a student member of the ACM and a National Science Foundation Graduate Fellow.

**Yun Zhang** is pursuing a PhD in the Department of Computer Science at Princeton University. Her research interests include computer architecture, compiler optimization, distributed systems, and machine

learning. Yun has a BS from Peking University and an MS from the University of Toronto, both in computer science. She is a student member of the ACM.

**Thomas Jablin** is pursuing a PhD in the Department of Computer Science at Princeton University. His research interests include compilers and just-in-time profiling and optimization, with an emphasis on parallelism extraction. Thomas has a BA in computer science from Amherst College. He is a student member of the ACM.

**David I. August** is an associate professor in the Department of Computer Science at Princeton University, where he directs the Liberty Research Group (http://liberty.princeton.edu), which is studying next-generation architectures, code analyses, and code transformations to enhance performance, reliability, and security. August earned his PhD in electrical engineering from the University of Illinois at Urbana-Champaign, where he worked as a member of the IMPACT research compiler group.

Direct questions and comments about this article to David August, Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ, 08540; august@princeton.edu.







IEEE Micro micr-28-01-brid.3d 28/1/08 11:12:13 20 Cust # brid