Estimating and Exploiting Potential Parallelism by Source-Level Dependence Profiling

Jonathan Mak[1], Karl-Filip Faxén[2], Sverker Janson[2], and Alan Mycroft[1]

[1] Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
[2] Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden

Abstract. Manual parallelization of programs is known to be difficult and error-prone, and there are currently few ways to measure the amount of potential parallelism in the original sequential code.

We present an extension of Embla, a Valgrind-based dependence profiler that links dynamic dependences back to source code. This new tool estimates potential task-level parallelism in a sequential program and helps programmers exploit it at the source level. Using the popular fork-join model, our tool provides a realistic estimate of potential speed-up for parallelization with frameworks like Cilk, TBB or OpenMP 3.0. Estimates can be given for several different parallelization models, varying in programmer effort and capabilities required of the underlying implementation. Our tool also outputs source-level dependence information to aid the parallelization of programs with lots of inherent parallelism, as well as critical paths to suggest algorithmic rewrites of programs with little of it.

We validate our claims by running our tool over *serial elisions* of sample Cilk programs, finding additional inherent parallelism not exploited by the Cilk code, as well as over serial C benchmarks where the profiling results suggest parallelism-enhancing algorithmic rewrites.

1 Introduction

Parallel programming is no longer optional. In order to enjoy continued performance gains with future generation multi-core processors, application developers must parallelize all software, old and new. While automatic parallelization based on static analysis is sometimes feasible, currently most software requires manual parallelization. Since this is a difficult task, there is urgent need for efficient tool support—in particular, tools that assist the programmer in understanding the potential for parallelization in the code, parallelizing code with high potential, and validating that the resulting parallel code is correct.

In this paper we address the first two issues, by presenting Embla 2, a tool which, when given a sequential program and an input, can estimate the amount of parallelism that would be available if the program were to be parallelized using language constructs such as those in OpenMP [4] or Cilk [3]. This parallelism measurement tool has been built by extending Embla [5], a Valgrind-based [17] profiler which captures dependence information, with functionality for simulating the effects of various parallel programming models based on independent fork-join task parallelism, a framework used in many parallel programming environments [3,12,21,13]. We validate our results against timing measurements of explicitly parallel Cilk programs.

Unlike traditional studies of parallelism limits [25,26] which focus on hardware parameters, we focus on the parallelization of program source code, and so use a different set of parameters to vary and different constraints for parallelization. The aim is to address questions such as, "Will thread-level speculation or parallel for-loops make much difference to the possible speed-up when parallelizing a program?" Section 2 discusses our parallelization models in depth.

We have used Embla 2 to investigate the potential for parallelization of three collections of programs: the SPEC CPU 2000 programs [7], MiBench [6] and the example programs in the Cilk 5.4.6 distribution. We contrast programs from these different sources and show the behaviour of the different parallelization models. Section 3 reports the results of these experiments.

The contributions can be summarized as follows:

– Dependences output by Embla 2 assist the programmer in parallelization by identifying appropriate source-level synchronization points. We show how synchronization points identified in serial elisions of example Cilk programs match those in the original hand-parallelized programs (Section 3.1).
– By mapping dynamic dependences back to program source, Embla 2 gives a realistic estimate of potential speed-up for parallelization using frameworks such as Cilk and OpenMP. Previous studies that we know of [10,26,19] have only considered *speculative* task-level parallelism—we give potential speed-ups for programs both with and without the use of thread-level speculation. Using Embla 2 we present estimates of inherent parallelism in various example programs (Sections 3.2 and 3.4).
– Critical paths output by Embla 2 can be used to identify at source-level bottlenecks that restrict the level of inherent parallelism. We illustrate this using examples from the MiBench benchmark suite, and suggest refactorings or algorithmic changes to increase potential parallelism (Section 3.5).

2 Task Models

Reflecting the task models of popular parallel programming environments such as Cilk [3], Java [12], TBB [21] and TPL [13], we use function calls/returns and (optionally) loop bodies to delineate tasks. The reasons for using function calls/returns are that functions represent a reasonable amount of usually self-contained work (finer-grain parallelism is largely exploited by superscalar processors), require minimal syntactic restructuring to exploit and have single entry/single exit control flow. Loop bodies are also considered as tasks in one of our variant models below. Naturally other legitimate coarse-grain parallelism will be left out, the extent of which we estimate with another variant model.

In all our models each function call is spawned as a task at its call site. The calling thread can then continue to execute statements that follow the function call without waiting for the call to return, until control reaches a statement that has a dependence on the call. At this point the calling thread must *synchronize* on the task, i.e. wait for the task to complete before going any further.

Our tool extends Embla [5], a Valgrind-based profiler that is based on these models and outputs data dependences between source lines in the same function. Embla works by observing instruction-level dependences that arise whenever two instructions access the same memory location and at least one of them is a write operation. The source and target instructions of each dependence are mapped to source lines within their *Nearest Common Ancestor* function.

In Embla 2, we calculate inherent parallelism by constructing a *dynamic dependence graph* for each function. This is a directed acyclic graph $G = (V, E)$ where each node $v \in V$ corresponds to an *instantiation*, or execution, of a *line*[1] of the function, and an edge $(u, v) \in E$ means that u must be completed before v can begin. Edges $u \to v$ are inserted for:

1. Data dependences between an instruction in the dynamic call tree of u to one in that of v (as observed by the original Embla infrastructure described above). Write-after-read and write-after-write dependences on the stack are ignored (these tend to be on easily privatizable variables), as well as all dependences between known commutative library functions, e.g. `malloc`.
2. Control dependences between u and v.
3. Dependences related to the parallelization model, that is, the restructuring of the code that we allow. E.g. to enforce the requirement that only function calls can be spawned as tasks, there is an incoming edge to each node from the last executed node that does not contain a function call. This effectively linearizes the graph except at function call spawns, the only points where forks are allowed.

The *cost* of a node v, $cost(v)$, is the minimum time required to execute the instantiation. In our model this is essentially the number of instructions executed on that line, with the additional requirement that if one of the instructions is a function call, then the length of the *critical path* of that call is included in the *cost* of the node. The *critical path CP* is the path in G with the largest total cost. The amount of inherent parallelism for a function call is then the cost of serial execution divided by the length of the critical path, i.e. $\sum_{v \in V} cost(v) / \sum_{v \in CP} cost(v)$. The inherent parallelism of a program is simply that of its `main` function. This figure is the speed-up of the parallelized program given unlimited processors and zero task creation overheads.

Potential optimizations are explored under variants of the baseline model:

Exact data dependences. By default, data dependences are *aggregated* per source line over a program's execution and then applied to all instantiations of the line.

[1] We would like to have a node corresponding to each execution of a *statement*, but the gcc debug tables that Embla uses are per-line not per-statement.

This models the parallelism that is possible by inserting synchronization points in source code, necessitating the same synchronization point for all instantiations. A variant is to use *exact* dependences by allowing each instantiation of the same line to have its own set of dependences as observed during execution[2], modelling the effects of dynamic techniques such as thread-level speculation (e.g. [23]), in the ideal case where the continuation of each task instance runs in parallel with the task right up until the point where a conflict would have been detected.

Loop iterations as tasks. As mentioned, there is an option to parallelize loops in Embla 2, where an established algorithm [1] is used to identify natural loops, iterations of which are spawned as tasks just like function calls. Updates to the loop index, which are outside the task boundaries, are still serialized.

Reduction operations. Reductions are accumulation operations on variables such as `acc += f(i);`, where the order in which instances of the associative reduction operation (`+`) are executed makes no difference to the final value of the accumulator. Embla 2 includes a helper program that statically identifies and annotates reduction operations in loops, dependences between which can be safely ignored by Embla to enable greater parallelism.

Spawn hoisting. Another way of enabling further parallelism is by spawning tasks earlier (a form of code motion optimization). In this variant, function calls are spawned as early as dependences allow, rather than only when control in the calling thread reaches the call site. This is modelled by relaxing constraint 3 above so that the incoming edge is only inserted to each node that does not contain a function call.

Line-level parallelism. By only considering function calls and potentially loop iterations as tasks, other forms of task-level parallelism are naturally excluded. To see how much more parallelism there is, we introduce a variant model where all lines can be spawned, regardless of whether they contain function calls. This is modelled by not applying constraint 3 at all. Note however that while this allows us to see more parallelism, some of the extra parallelism discovered will be too fine-grain to be exploitable.

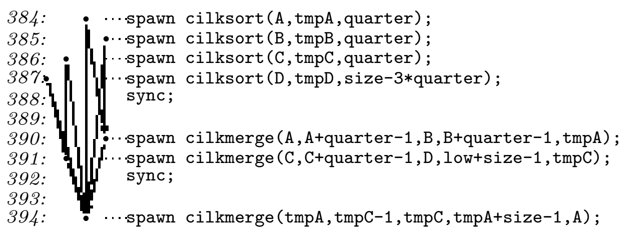
3 Evaluation and Test Suites

For models using aggregated data dependences, Embla 2 must be run twice—first to collect all observed dependences, and then to calculate a parallelism figure given those dependences. When run on an x86 desktop, the first run causes an average of 1500x slowdown of the profiled program, while the second causes around 700x. Valgrind itself, on which Embla 2 is based, causes an average slowdown of 8x.

We demonstrate the usefulness of Embla 2 using example Cilk programs packaged with the 5.4.6 release of Cilk[3]—programs known to have lots of task-level

[2] Cf. the difference between context-sensitive and context-insensitive static analysis.

[3] We have omitted programs that use the `inlet`, `abort` and `SYNCHED` keywords, as their translation into ordinary C is not straightforward.



```
spawn cilksort(A,tmpA,quarter);
spawn cilksort(B,tmpB,quarter);
spawn cilksort(C,tmpC,quarter);
spawn cilksort(D,tmpD,size-3*quarter);
sync;

spawn cilkmerge(A,A+quarter-1,B,B+quarter-1,tmpA);
spawn cilkmerge(C,C+quarter-1,D,low+size-1,tmpC);
sync;

spawn cilkmerge(tmpA,tmpC-1,tmpC,tmpA+size-1,A);
```

Fig. 1. An extract from cilksort and its corresponding relevant dependences

parallelism. Furthermore, the nature of Cilk means that these programs can be translated into semantically equivalent programs in ordinary C (known as *serial elisions*) simply by stripping the Cilk keywords[4]. We thus have a set of C programs and their manually-parallelized counterparts to compare with the parallelism discovered by Embla 2.

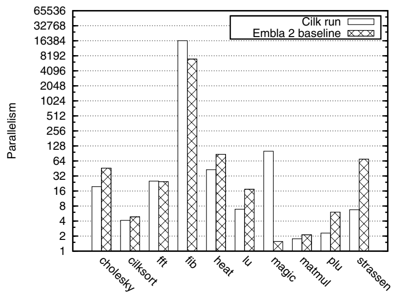
3.1 Finding Optimal Task Synchronization Points

Programmers can use aggregated dependences discovered by Embla 2 to realize the parallelism discovered using a language like Cilk, where function calls are spawned as tasks and later joined (or *synchronized*), by synchronizing on previously spawned tasks just before the first line dependent on them. As an example, Figure 1 shows an extract from the cilksort program, along with aggregated dependences discovered by Embla 2 on the program's serial elision. By spawning function calls and synchronizing before the first line that depends on previously spawned tasks, we arrive at the Cilk program with optimal parallelism, which in this case is the same as the original program. However, the programmer should be aware that as with all dynamic analyses, these dependences are based on running the program on sample inputs only and so the parallelization may not be safe for all possible inputs. Nevertheless, previous work [5] has shown a correlation between dependence coverage and code coverage.

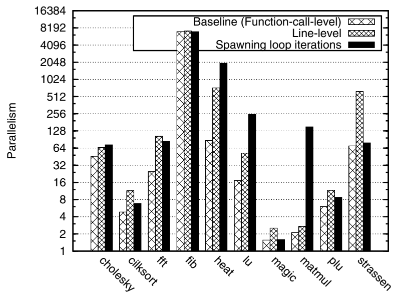
3.2 Amount of Parallelism Discovered

Figure 2(a) compares (on a logarithmic scale) the amount of parallelism found by Cilk's timing infrastructure (averaged over 60 runs) to that found by Embla 2 on their serial elisions. Our baseline model for this comparison uses aggregated data dependences, and considers only function-call-level parallelism without loop parallelization or spawn hoisting—the closest model we have to Cilk's. The graph shows that Embla 2 is able to find all of the task-level parallelism in most of the original Cilk programs. One notable exception is magic,

[4] Namely, `cilk`, `spawn` and `sync`.



(a) As calculated by Cilk and Embla 2



(b) Using variant models

Fig. 2. Parallelism of Cilk programs (note logarithmic scale)

and this is because the Cilk program uses an implicit *inlet*[5] at the statement `count += spawn execute(...);`, which is not a feature of our model. Later we show how, by extending our model, Embla 2 also discovers the parallelism here.

For a few examples Embla 2 can discover more parallelism than explicitly specified in the original Cilk program. We found several functions called synchronously that could have been spawned, as well as C library function calls that can be spawned with the addition of simple wrappers.

3.3 Effects of Optimizations

We now look more closely at the variant optimization models to see whether they affect potential parallelism in these programs.

Exact data dependences. In most programs in the Cilk test suite the difference between this model and the baseline is negligible. This suggests that for such programs most of the potential parallelism is statically achievable—runtime techniques such as thread-level speculation (TLS) would give few performance benefits. One exception is lu, which gives an almost-fourfold speed-up over the baseline. This is because in one nested loop the dependence for a certain task on itself exists only between iterations of the outer loop but not of the inner loop. With aggregated dependences this results in all instances of the task being serialized, while with exact dependences instances from the same iteration of the outer loop can still be parallelized. While TLS would address this issue, a cheaper method is to parallelize the inner loop—part of the speed-up seen in the third columns of Figure 2(b) can be attributed to this task.

Line-level parallelism. The amount of line-level parallelism in these programs is compared to the amount of function-call-level parallelism in Figure 2(b). We can see that for most of these programs the amount of line-level parallelism is around

[5] In Cilk an *inlet* is a block of code that is executed atomically after a task completes. The potential thread contention for such an atomic block might explain the large variance in actual parallelism in the Cilk run.

twice or more that of function-call-level parallelism. This is mostly due to simple statements performing arithmetic operations inside a function call or loop that can run in parallel. Each of these operations takes a small number of cycles, which means that it is not viable for each of these to be spawned. As mentioned, some of this fine-grain (mostly instruction-level) parallelism is realized already in existing superscalar processors. Nevertheless, operations may be grouped and extracted into tasks that are sufficiently large to see performance gains.

Loops. Looking at the first and third columns of Figure 2(b), which compares the potential parallelism of Cilk programs without and with spawning loop iterations, we can see that the use of parallel for-loops benefits most of the programs considered here, especially matmul. This confirms the view that the use of parallel for-loops is an excellent way to express task-level parallelism, in addition to function calls. Furthermore, some parallel programming environments (e.g. Cilk++ [14]) optimize parallel for-loops by deriving loop induction variables by divide-and-conquer instead of linearly, meaning that actual parallelism may be even higher than these figures. Embla 2 not only finds the amount of loop-level parallelism in a program, but can be used to easily identify candidate loops for parallelization. This can simply be done by searching through the dependences output by Embla 2 for dependences between iterations of the loop concerned. If there are no such dependences, then the loop can be parallelized.

Reduction operations. When dependences between reduction operations are discounted, the parallelism of magic vastly increases from under 2 to 376. This means that a parallel reduction mechanism, such as Cilk's implicit inlets, *hyperobjects* in Cilk++ and similar operations in OpenMP, is essential for the program's parallelism potential to be realized.

Spawn hoisting. With these Cilk programs, we find that spawn hoisting has a negligible effect on potential parallelism, as parallelism measurements with spawn hoisting is almost the same as those for the baseline in Figure 2(a). This suggests that, perhaps unsurprisingly, most function calls in these programs are already spawned at the earliest possible point, and little further hoisting is possible.

3.4 Parallelism in Other Benchmarks

We also ran the same analysis using Embla 2 on some of the benchmark programs in the SPEC CPU 2000 [7] (with the MinneSPEC reduced data input set [9]) and MiBench [6] suites, the results of which are displayed in Figure 3. As before, the baseline model uses aggregated data dependences, and considers only function-call-level parallelism without loop parallelization or spawn hoisting. The second model includes loop-level parallelism, while the third, Optimized model considers line-level parallelism and exact data dependences, hoisting spawns and ignoring control dependences. In general, we see that few benchmarks exhibit the level of parallelism seen in the Cilk examples. In fact, none of these benchmarks exhibit

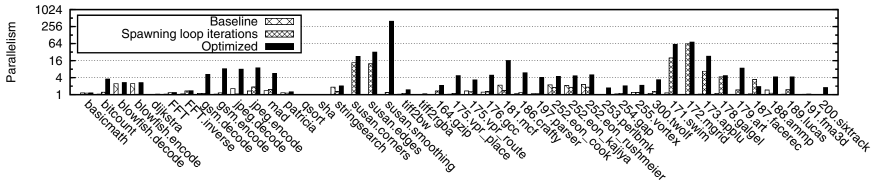


Fig. 3. Parallelism of benchmark programs

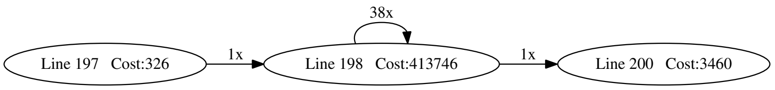


Fig. 4. Critical path of `sha_stream` function in `sha`

parallelism of over 3 under the baseline model, suggesting that spawning existing function calls alone is insufficient to effectively parallelize them.

There are, however, some benchmarks with a significant amount of loop-level parallelism. Some of them are the susan benchmarks from MiBench, which perform image smoothing, corner detection and edge detection on an image, and are data-parallel—the same computation is performed on each pixel in the image and the results for each pixel are independent of each other. This is reflected in our results, which show that using the small inputs, both susan.corners and susan.edges have potential parallelism of over 12 when loop iterations are spawned. This is not the case for susan.smoothing, however, as we will explain later. Note that for some programs, e.g. 252.eon, spawning loop iterations actually results in a lower level of parallelism than the baseline. This is because the baseline allows for partial overlap between loop iterations (a.k.a. software pipelining) whereas spawning loop iterations is an all or nothing proposition; the iterations are either completely independent or completely serial.

3.5 Increasing Inherent Parallelism by Critical Path Analysis

Even though Embla 2 finds lots of inherent task-level parallelism in example programs from the Cilk test suite, little is found in most general-purpose programs, meaning they cannot be transformed into highly concurrent programs simply by spawning existing function calls and loops. However, here Embla 2 can output the critical path of each function call, allowing us to examine the bottlenecks that prevent greater parallelism from being realized. This critical path can then be collapsed into a graph with one node per line, as shown in Figure 4. We now demonstrate this in some of the simpler examples and suggest refactorings or algorithmic changes that would increase inherent parallelism.

susan.smoothing. As mentioned, susan.smoothing is a data-parallel image smoothing application. Unlike susan.corners and susan.edges, little inherent parallelism is found here (1.19 under the Spawning loop iterations model). By examining the critical path, we found the main bottleneck dependence to be between instantiations of a certain loop within function `susan_smoothing`. Further examination of the loop reveals that the dependence is due to a variable that is incremented exactly once per iteration, and is therefore an induction variable. However, as the increment is in the loop body rather than the loop header, each execution of the loop body is dependent on the last, and as a result iterations of the loops are serialized. A simple solution is to move the increment operation to the loop header (manual code motion), which causes inherent parallelism to be vastly increased to around 1,090.

sha. The Secure Hash Algorithm program computes a 160-bit hash value from the contents of an input file. Under the Line-level parallelism variant model, Embla 2 reports a parallelism of 1.36. Most of the critical path can be attributed to dependences between calls to `sha_update` (line 198 in Figure 4). This function takes the existing hash value (the *digest*) and derives a new hash value which replaces it. Consequently, each call to this function must depend on the last as it requires the digest computed by the last call. This suggests that in order to increase the amount of inherent parallelism, the underlying algorithm must be modified, e.g. by dividing the file into blocks and computing independent digests for each block, which are then combined into one final digest.

Other examples. Applying the same analysis to other examples, we find that input/output forms a large part of several benchmarks. Calls to `scanf`/`printf` take up a tenth of the sequential execution time of dijkstra and 80% of that of FFT (MiBench). As input/output is unparallelizable without significant re-implementation, Amdahl's law means that the maximum speed-up, even if we were able to parallelize the rest of the program perfectly, would still be low. This suggests that a parallel implementation of input/output would be very useful.

4 Related Work

Various languages and libraries have been created to allow easy expression of task-level parallelism. Cilk(++)'s [3,14] model of fork/join parallelism matches our function-spawning model most closely, although a Cilk `sync` always joins with all tasks spawned in the current procedure activation rather than being able to pick a specific task to join. While OpenMP [4] originally focused almost exclusively on loops, version 3.0 adds support for task parallelism. Other notable examples offering similar functionality include Java's concurrency library [12], Intel's TBB [21] and Microsoft's TPL [13]. SMPSs [20] uses a related model where the programmer specifies dependences rather than synchronisation points.

Several studies have been done on limits of instruction-level parallelism (e.g. [25]), but fewer have tried to separate out task-level parallelism from instruction-level parallelism. Kreaseck et al. [10] explored limits of speculative task-level parallelism by executing function calls early, similar to the way we hoist spawns. They have however imposed the restriction that spawned function calls must be joined at their original call sites, which is a restriction we have felt to be unnecessary, and thus have not imposed in our analysis. In our model function calls can be joined as late as dependences allow (but always before the parent call returns). Other studies [26,19] have shown that data-value prediction, especially in regard to return values, is effective at increasing task-level parallelism. This is something we wish to consider further in the future.

Embla's source-level profiling algorithm [5] has been used elsewhere to discover dependences and assist programmers with parallelization. Most systems [27,24,11] are concerned with the parallelization of loops only. The Alchemist tool [28] uses it to select good task candidates for thread-level speculation. Nguyen et al. [18] use it to detect function-level parallelism, but uses a more restrictive parallelization model that is based on Scheme. We believe our tool is the first that can assist in parallelizing function calls as well as loops, while providing a realistic estimate of potential speed-up.

There are parallels between Embla 2 and on-the-fly data race detection [16,22]. Both use instrumentation to infer properties of the program through dynamic analysis. The main difference is that while race detection seeks to find unsafe parallelism (i.e. bugs) in an explicitly multi-threaded program, Embla 2 seeks to find potential safe parallelism in a sequentially written program.

There has been much research into automatic parallelization [8,2], most of which use only static analysis. However, the lack of precision when statically analyzing dependences remains a barrier, something which dynamic analysis tools such as Embla 2 can address. The downside is that the resulting parallelization may not be valid for runs not covered by the training set of input data. A hybrid approach would therefore be best.

Some of the speculative task-level parallelism uncovered by Embla 2 can be exploited with thread-level speculation (TLS) (e.g. [23]), although we have observed this is not needed for the Cilk test suite programs. One important factor affecting the performance of TLS is selecting good candidate threads for speculation, as frequent rollbacks would offset any gains in parallelism [15]. We believe that Embla 2 can be used to identify good candidates, as it allows us to look at the frequencies at which potential dependences materialize.

5 Conclusions

We have presented Embla 2, a tool designed to aid manual parallelization by estimating and locating inherent parallelism in programs as well as pinpointing bottlenecks. It works by profiling dependences and mapping them back to program source. The underlying model of parallelism treats each function call as a spawnable task, which is synchronized on as late as dependences allow. Variants of this model allow us to estimate the potential benefits of parallel for-loops and optimizations such as thread-level speculation.

Embla 2 can locate inherent parallelism in programs with lots of it, as shown by its reconstruction of parallel tasks in the Cilk test suite. For those without much readily available parallelism, such as many SPEC CPU 2000 and MiBench benchmarks, Embla 2 outputs critical paths to assist the programmer in pinpointing bottlenecks and finding solutions. Some programmer effort is still required to realize task-level parallelism in this case, and we suggest two extensions to provide further assistance: a source-to-source transformer that automatically parallelizes programs given dependences found by Embla 2; and a compiler that can perform code motion to move task-serializing operations, such as the induction variable update in susan.smoothing, outside task boundaries, thereby making the tasks independent and hence parallelizable. All in all, we believe Embla 2 would be a useful aid for programmers parallelizing legacy sequential applications using parallel programming frameworks such as Cilk.

Acknowledgements

We thank our reviewers for their invaluable input. The first author thanks St John's College, Cambridge for funding his studentship.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co. Inc, Boston (1986)
2. Blume, B., Eigenmann, R., Faigin, K., Grout, J., Hoe, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P., Weatherford, S.: Polaris: The next generation in parallelizing compilers. In: Proc. Workshop on Languages and Compilers for Parallel Computing. Springer, Heidelberg (1994)
3. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing 37(1), 55–69 (1996)
4. Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. IEEE Comput. Sci. Eng. 5(1), 46–55 (1998)
5. Faxén, K.F., Popov, K., Jansson, S., Albertsson, L.: Embla—data dependence profiling for parallel programming. In: CISIS 2008: Proc. 2nd Int'l Conf. on Complex, Intelligent and Software Intensive Systems. IEEE, Los Alamitos (2008)
6. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: WWC 2001: Proc. Int'l Workshop on Workload Characterization. IEEE, Los Alamitos (2001)
7. Henning, J.: SPEC CPU2000: measuring CPU performance in the new millennium. Computer 33(7), 28–35 (2000)
8. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco (2002)
9. KleinOsowski, A., Lilja, D.J.: MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. Comp. Arch. Letters 1 (2002)
10. Kreaseck, B., Tullsen, D., Calder, B.: Limits of task-based parallelism in irregular applications. In: Proc. Int'l Symp. on High Performance Computing (2000)
11. Larus, J.R.: Loop-level parallelism in numeric and symbolic programs. IEEE Trans. Parallel Distrib. Syst. 4(7), 812–826 (1993)
12. Lea, D.: A Java fork/join framework. In: Proc. ACM, Conf. on Java Grande (2000)

13. Leijen, D., Hall, J.: Parallel performance: Optimize managed code for multi-core machines. MSDN Magazine (October 2007)
14. Leiserson, C.E.: The Cilk++ concurrency platform. In: DAC 2009: Proc. 46th Annual Design Automation Conference. ACM, New York (2009)
15. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: a TLS compiler that exploits program structure. In: PPoPP 2006: Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (2006)
16. Mellor-Crummey, J.: On-the-fly detection of data races for programs with nested fork-join parallelism. In: Proc. of Supercomputing 1991, pp. 24–33. ACM Press, New York (1991)
17. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007)
18. Nguyen, H., Taura, K., Yonezawa, A.: Parallelizing programs using access traces. In: LCR 2002: Proc. 6th Int'l Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (2002)
19. Oplinger, J.T., Heine, D.L., Lam, M.S.: In search of speculative thread-level parallelism. In: Proc. 1999 Int'l Conf. on Parallel Architectures and Compilation Techniques. IEEE, Los Alamitos (1999)
20. Perez, J., Badia, R., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: 2008 IEEE Int'l Conf. on Cluster Computing (2008)
21. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc., Sebastopol (2007)
22. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems 15(4), 391–411 (1997)
23. Steffan, J.G., Colohan, C., Zhai, A., Mowry, T.C.: The STAMPede approach to thread-level speculation. ACM Trans. Comput. Syst. 23(3), 253–300 (2005)
24. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: PLDI 2009: Proc. 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation (2009)
25. Wall, D.W.: Limits of instruction-level parallelism. In: Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and Operating System. ACM, New York (1991)
26. Warg, F., Stenström, P.: Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In: PACT 2001: Proc. 2001 Int'l Conf. on Parallel Architectures and Compilation Techniques. IEEE, Los Alamitos (2001)
27. Wu, P., Kejariwal, A., Caşcaval, C.: Compiler-driven dependence profiling to guide program parallelization. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 232–248. Springer, Heidelberg (2008)
28. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: CGO 2009: Proc. 2009 Int'l Symp. on Code Generation and Optimization. IEEE, Los Alamitos (2009)