Factoring Out Ordered Sections to Expose Thread-Level Parallelism

Hans Vandierendonck, Sean Rul and Koen De Bosschere
Ghent University,
Dept. of Electronics and Information Systems,
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{hvdieren,srul,kdb}@elis.ugent.be

Abstract

With the rise of multi-core processors, researchers are taking a new look at extending the applicability auto-parallelization techniques. In this paper, we identify a dependence pattern on which auto-parallelization currently fails. This dependence pattern occurs for ordered sections, i.e. code fragments in a loop that must be executed atomically and in original program order. We discuss why these ordered sections prohibit current auto-parallelizers from working and we present a technique to deal with them. We experimentally demonstrate the efficacy of the technique, yielding significant overall program speedups.

1. Introduction

Chip manufacturars have made a shift towards building multi-core processors. This puts a large demand on techniques for extracting thread-level parallelism (TLP) from applications. As these multi-core processors are shipped in all kinds of systems (server, desktop and laptop), it has become necessary to exploit TLP also in applications that have traditionally not been considered as good candidates for multi-threaded execution: programs with complex control flow and hard-to-predict memory access patterns.

We have found that, in these applications, TLP is often limited due to small details. In particular, a loop nest can be highly parallel, except for a few statements in the loop that introduce a dependency between loop iterations. Yet, these statements are so few (or they are perhaps never executed in practice), that parallelisation should be possible, at least to some extent. Some of these dependence patterns are well known, e.g. reductions and critical sections, and solutions have been in use for a long time.

The contribution of this paper is to provide a solution to a more complicated case: the case where operations must remain in program order (more stringent than critical sections), but the parallelized code region does not depend on values computed by these operations. These ordered sections may be very benign, such as the generation of debugging output when a flag is set. On the other hand, they may also concern updating critical data structures. We present a method to allow parallelization of this construct.

Moreover, we demonstrate that our method also helps to deal with respecting the order of system calls and dealing with multiple call sites to exit() by callee functions. These issues are very important for dealing with program side-effects and for respecting sequential program semantics.

We demonstrate the effectiveness of our method on several benchmarks (bzip2 from SPECint2000, mcf from SPECint2006, clustalw from BioPerf). The intent of this work is to present methods than can be implemented in auto-parallelizing compilers. While we are working on such a compiler, actual compiler algorithms and implementation are out of the scope of this work.

This paper is structured as follows. In Section 2 we discuss the researched problem and we present our solution in Section 3. We experimentally evaluate our solution in Section 4. We discuss related work in Section 5 and summarize conclusions in Section 6.

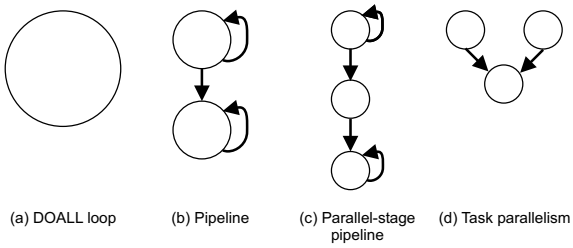


Figure 1. Types of parallelism frequently investigated in the current literature. For the loop cases, only the instructions in the loop are shown.

2. The Problem

It is recognized that programs with complex control-flow and/or complex memory access patterns require specific types of parallelism. Rather than DOALL loops, one should search for pipelines [9, 12], parallel-stage pipelines [10] and task parallelism. Furthermore, some researchers advocate using speculation to reduce the code structure to one of these types of parallelism and thus expose parallelism [4, 13].

In any of the cases above, we demand that dependencies between statements respect a particular pattern. This is easily represented using the program dependence graph (PDG) [6], where statements are represented as nodes and dependencies between statements are represented as directed edges. Edges are inserted for control and data dependencies, and also for memory dependencies. Parallelism is detected in the PDG by computing strongly connected components (SCC) in the graph [9]. The cited types of parallelism can be graphically represented by the dependencies between the strongly connected components (Figure 1).

However, more often than not, the parallel code regions may contain code fragments that introduce dependencies and inhibit the exploitation of parallelism. These code fragments may be very benign, such as the generation of debugging output when a flag is set, or they may be updates to data structures that must be executed in the original program order. Hence, we call these code fragments *ordered sections*, in correspondence to the same term in OpenMP [8]. In this paper, we focus on particular ordered sections, namely those that do not produce values consumed by the remainder of the loop or task. As such, there is a certain amount of slack in executing these ordered sections and an opportunity for exploiting parallelism arises.

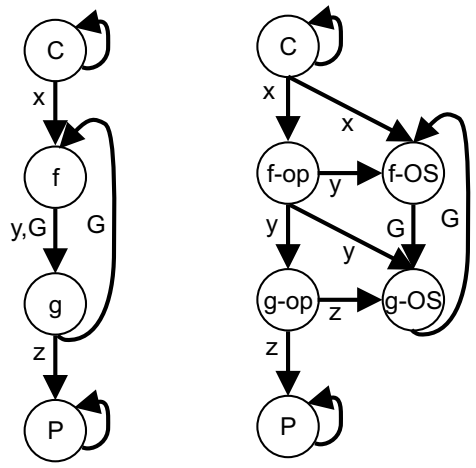
```
loop:
  while(condition) {
    x = consume();
    y = f(x);
    z = g(y);
    produce(z);
  }

f(x) {
  y = operate on x;
  G = ordered\_section\_1(G,x,y);
  return y;
}

g(y) {
  z = operate on y;
  G = ordered\_section\_2(G,y,z);
  return z;
}
```

Figure 2. Pseudo-code for a parallelizable loop with ordered sections.

To illustrate the problem, we draw upon an example. The pseudo-code in Figure 2 describes a parallelizable loop with ordered sections. The loop takes input data x and transforms it into output data z in two steps. We assume that a good parallel partitioning of the loop places functions $f(\cdot)$ and $g(\cdot)$ in separate pipeline stages.

This pipeline, however, is obscured by the presence of ordered sections in these functions. Each of the ordered sections draws upon some values produced by the main code and they update a shared global variable G. Function calls are represented as a single node in the program dependence graph. Hereby, they are treated as an undivisable entity. It is no longer possible to separate the ordered sections from the remainder of the code. The corresponding PDG is depicted in Figure 3(a). We observe that nodes $f(\cdot)$ and $g(\cdot)$ are cyclically dependent through variable G.

Note that the cyclic dependence is there only because the ordered sections are located in functions called from the loop. Hence, the ordered sections are necessarily merged together with the remainder of the



(a) Loop PDG (b) Loop PDG with inlining

Figure 3. Program dependence graph for the example loop (left) and for the example loop where functions f and g are inlined.

functions $f(\cdot)$ and $g(\cdot)$. The problem disappears when, for instance, we inline the bodies of the functions into the loop. The corresponding PDG (Figure 3(a)) contains two nodes for each function: the operate step (op) and the ordered section (OS). The ordered sections still reduce to a single SCC. As the remainder of the code is no longer dependent on the ordered sections, the pipeline can be readily seen.

Note that the situation of Figure 3(b) corresponds to loops where ordered sections do not occur in callee functions, a situation that is parallelizable by prior approaches [9, 5, 13]. In this work, we propose a scalable solution to the general problem of ordered sections in callee functions. Function inlining is not a suitable solution to this problem; we used it only for didactical reasons. Function inlining is not suitable because (i) function inlining is an optimization in its own right with sensitive cost/performance models, and (ii) it is not sufficiently generic in the face of deep call trees and (mutually) recursive functions.

3. The Solution

Conceptually, the proposed solution is to remove ordered sections from callee functions and to move them to the loop body. Hereby the necessary dependences between function calls remain while the ordered sections reduce to an SCC without outgoing edges. The method to achieve this is to create a queue of ordered sections remaining to be processed. Ordered sections are enqueued when a callee function would have entered one. They are dequeued only when program dependences allow so, typically in the last stage of the pipeline.

3.1. Dissecting Ordered Sections

Three properties of ordered section are important to describe how ordered sections are factored out.

The *update set* of an ordered section is the set of all program variables that are modified by the ordered section.

The *copy set* of an ordered section is the set of all program variables that the ordered section reads, excluding program variables in the update set and excluding loop-constant program variables.

The *factored code* of an ordered section is a fragment of the control flow graph of the containing function, i.e. basic blocks and instructions. The factored code contains all the instructions operating on the update set and all of their dependents. Control flow instructions between basic blocks must be properly duplicated. Separating factored code from the remainder of the code is actually quite similar to splitting a loop body in pipeline stages and can be handled in the same way, e.g. [9].

The factored code may not include function return statements as this would violate the proposition that the majority of the code is not dependent on the ordered section. Furthermore, ordered sections do not cross function boundaries for reasons of simplicity.

3.2. Operations on the PDG

When factoring out an ordered section from a callee function, we must update the program dependence graph to reflect a reduction of dependencies for the function call node.

Hereto, we add a *consuming call node* to the PDG, besides the original function call node. The original call node represents the non-ordered section part of the function, plus the code to enqueue ordered sections. The consuming call node represents taking ordered sections from the queue and executing them.

Dependencies in the PDG are updated in the following way:

1. All outgoing dependencies from the original call node that indicate updates to a variable in the update set of the ordered section are redirected to start from the consuming call node.

2. All incoming dependencies to the original call node that indicate dependence on a variable in the update set of the ordered section are redirected to point to the consuming call node.

3. A dependence is added from the original call node to the consuming call node to indicate the causality between producing ordered sections in the queue and consuming them. All incoming dependencies of the ordered section that are not part of its update set are implicitly captured by this queue dependence.

There can be only one consuming call node for every original call node, even when multiple ordered sections are factored out. In the case of multiple ordered sections, the PDG is updated by steps 1 and 2.

3.3. Code Transformation

Hopefully, the modified PDG will expose additional parallelism. When parallelism is found and parallel code is generated, additional code must be included to factor out ordered sections and to consume the ordered sections.

To enqueue a message for an ordered section, a message is constructed containing an ordered section ID (each ordered section is assigned a unique ID to identify it) and a copy of all the variables in its copy set.

In the functions, instructions belonging to ordered sections are removed. Instead, code is inserted to queue the ordered section. The queued information consists of an ordered section ID and a copy of every program variable in its copy set.

Finally, additional code is added to the main loop to dequeue ordered sections. This code consists of a loop that takes every ordered section from the queue and executes the corresponding code using program variables from the copy set where necessary. When the number of enqueued ordered sections is expected to be small, then it is possible to add the consuming loop to the last pipeline stage of the loop. In this case, the queue grows to its maximum size during the execution

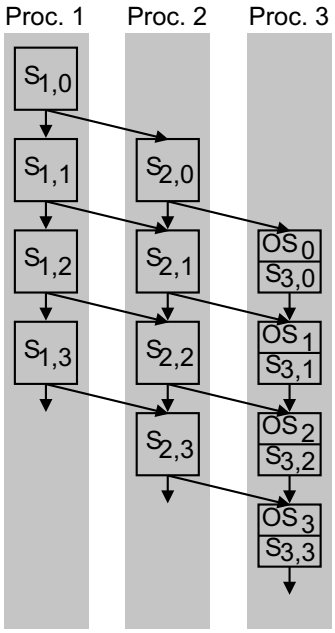


Figure 4. Execution chart of loop with pipeline parallelism and factored ordered sections.

of each pipeline stage. The queue is emptied only after the last pipeline stage has executed.

The general solution is to create an additional thread that executes ordered sections as they are inserted in the queue. The thread starts when the first pipeline stage starts executing. It stops by a special message that is sent when the last pipeline stage finishes. It is likely that this thread is idle very often. It is needed only to limit the size of the queue.

When the size of the ordered section queue is expected to be bounded, and because it is often the case that ordered sections are responsible for only a fraction of the code executed in the loop, the first approach works quite well. It is used in all the examples in the evaluation section.

However, when a queue grows beyond acceptable size, it is possible to limit memory consumption by blocking a thread when it tries to add additional elements to the queue. When all previous loop iterations have finished, the ordered sections may be dequeued and executed and the thread may continue execution.

An execution chart of the resulting code transformation is depicted in Figure 4 for a 3-stage pipeline. Code block $S_{p,i}$ executes pipeline stage p for loop iter-

ation i. Code blocks $S_{1,i}$ and $S_{2,i}$ may insert ordered sections in queue Q_i, which is specific to loop iteration i. Ordered sections are consumed in step OS_i, before executing step $S_{3,i}$. The latter pipeline stage directly executes ordered sections (this is an optimization to avoid unnecessary queueing overheads). Note that the queue for iteration $i+1$ is not consulted before iteration i has completely executed. This timing is necessary to respect the semantics of ordered sections.

3.4. Handling System Calls

Ordered sections help dealing with system calls. Again, these remarks apply to system calls embedded in functions called from a parallelized loop.

Compilers generally treat system calls with great conservatism, causing all system calls embedded in functions called from a loop to be mutually dependent. Such a situation kills parallelism. Building on ordered sections, it is possible to ameliorate this situation. One starts by trying to factor out every system call as an ordered section. This operation fails when the system call produces input data for the loop, e.g. a *read()* call.[1] If, however, all system calls can be factored out, then conservatively correct thread-level parallelism may be exposed.

Furthermore, when functions called from a parallelized loop contain multiple calls to *exit()* or *abort()*, then ordered sections help to provide a correct implementation. Without recognizing program exits, it is possible that the wrong exit is taken and perhaps the wrong error message is printed. This can happen, e.g., when pipeline stage $S_{1,n}$ executing iteration i exits the program before pipeline stage $S_{2,1}$ is executed and has the opportunity to exit.

Handling multiple exits is straightforward using ordered sections: every function call that may not return is an ordered section and is factored out. Furthermore, at the point of executing the non-returning function call, the thread sets a flag that the current iteration i is finished, nullifying all code in pipeline stages $S_{p,i}$. Furthermore, all threads executing pipeline stages $q \leq p$, where p is the current pipeline stage, are blocked. The last pipeline stage continues execution and executes the first exit it encounters, respecting sequential semantics.

[1] Conservatism requires that calls to *read()* are treated as aliased to other system calls which may interfere, even *write()* on a different file descriptor. E.g. the program may be in communication with another program over a UNIX pipe, causing reads to block on writes from the other program. Interchanging reads and writes on such a program may lead to deadlock.

4. Experimental Evaluation

We evaluate the proposed code transformation for exposing thread-level parallelism using 3 benchmarks: bzip2 (taken from SPEC CPU2000[2]), mcf (SPEC CPU2006) and clustalw (BioPerf [1]).

We test these benchmarks on an Intel I7 quad-core processor (8 threads in total) and a Sun Niagara T1 8-core processor (32 threads in total). We use gcc 4.1.2 on the Niagara and gcc 4.3.2 on the Intel I7. Parallelism is expressed using POSIX or OpenMP. When using OpenMP, we compile with gcc 4.4 to have OpenMP 3.0 support.

4.1. Bzip2

The main compression loop in bzip2 (SPECint2000) is a 4-stage pipeline, where stages 2 and 3 are parallel stages [11]. Many functions in bzip2, however, may print debugging information depending on the value of a verbosity flag. To reconcile the goals of parallelizing the code and guaranteeing the correct ordering of print statements, we isolate these print statements in separate tasks using the method of this paper. Hereby, the pipeline becomes valid. The same transformation is applied to the 2-stage pipeline in the decompression code.

We implemented the parallel pipelines using the POSIX threads library [11]. The timing measurements (Figure 5) reveal that the compression stage benefits from up to 6 threads on the Niagara processor and up to 4 threads on the I7. Decompression can benefit from at most 2 threads due to the structure of the pipeline. Overall, a speedup of 2.97 and 2.00 is obtained on the Niagara and I7 processors, respectively.

4.2. Mcf

An almost DO-ALL loop occurs in the `primal_bea_mpp()` function of the mcf benchmark. This loop scans over the edges in a graph and

[2] `http://www.spec.org/`.

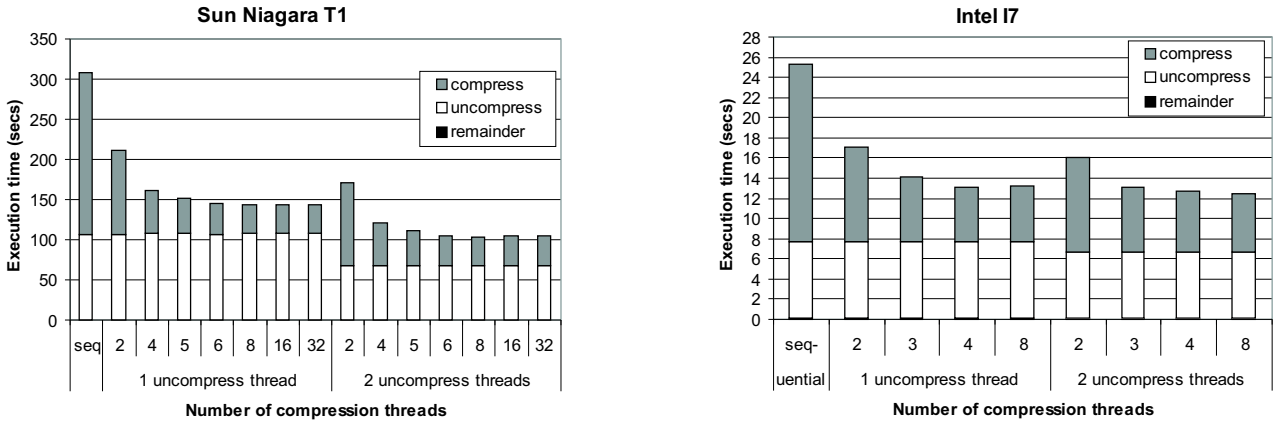


Figure 5. Execution time of bzip2 on two multi-core processors.

collects edges meating a specific criterium, which are added to the back of a list. The scanning code does not modify memory and can be freely parallelized, but the order in which elements are added to the list is important. Thus, we factor out the updates to the list and let the scanning code run in parallel. Figure 6 shows that performance scales to a 1.52 speedup with 4 threads on the I7 and to 2.18 with 16 threads on the Niagara T1.

4.3. Clustalw

Clustalw spends almost all its execution time in two stages: pairwise alignment and progressive alignment [14].

Pairwise alignment of sequences using the Smith-Waterman algorithm is trivially parallel as the alignment of every pair of sequences is independent. However, the code prints the score of each pair of sequences as it progresses, hence current compiler techniques do not recognize this parallelism. We use the same code transformation as in bzip2 to enable the parallelism. Figure 7 shows the performance scaling of this highly parallel loop, which shows near-perfect scaling.

The progressive alignment stage contains much less parallelism. The bulk of the computation is in a function `pdiff()` where two loop nests can be executed in parallel. Figure 7 shows that this parallelism reduces progressive alignment execution time from 17.0 seconds to 10.0 seconds, using 2 threads.

Additional parallelism is present between the doubly-recursive calls in this function, as each call is almost independent of the other calls. The term "al-

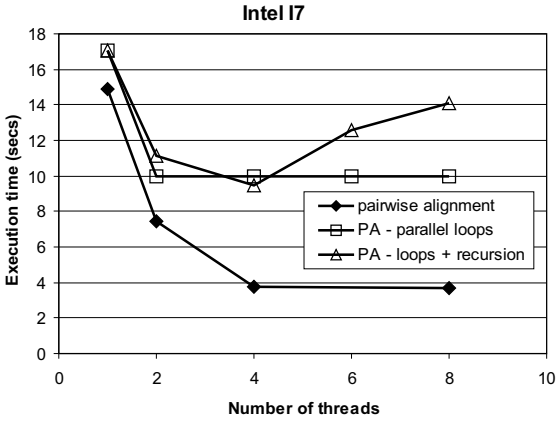


Figure 7. Execution time of clustalw. PA stands for "progressive alignment."

most" refers to a serialization that occurs due to updates to a shared array (`displ[]`) that occurs mostly in the leaf calls of the recursion and sometimes in between the two recursive calls. With the technique of this paper and using 4 threads, performance improves by 5.4% (Figure 7). Although this speedup is not very high, the aim of the exercise is to show that the parallelism can be correctly extracted.

Performance can probably be improved with more implementation work. We used the OpenMP task construct to express the parallelism, but OpenMP schedules tasks in a sub-optimal order, especially since tasks are also used to express the parallelism between the loop nests. [3] Performance is significantly improved if processors are allocated in pairs, one for each loop

[3] Nesting OpenMP sections in tasks turned out to perform worse due to repetitive creation and destruction of threads.

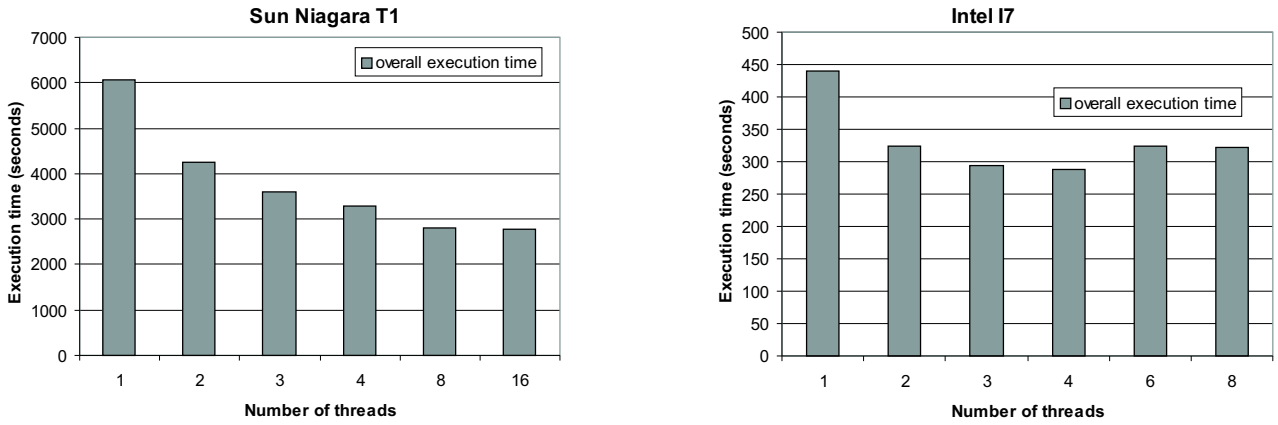


Figure 6. Execution time of mcf on two multi-core processors.

nest, a trick we applied in our implementation for the Cell B.E. [14].

5. Related Work

The context of this work is the search for ways to automatically parallelize control-flow intensive applications. Decoupled software pipelining is a compilation technique to recognize pipelines and to distribute the pipeline stages across threads [9]. Parallel-stage pipelines are pipelines where some stages are not dependent on themselves and allow additional parallelism [10, 11] Program demultiplexing attempts to extract threads by viewing a sequential program as a set of interleaved threads [2]. All of these approaches succeed in discovering TLP to some extent. In our experience, however, they get stuck on particular dependence patterns, one of which is discussed in this paper.

Thread-level speculation (TLS) aims to expose TLP that is not provably correct [7, 3]. By adding hardware and/or software checkpointing and restore mechanisms for memory, it is possible to undo the effects of misspeculation. Many proposals of TLS mechanisms can, however, not expose the same coarse-grain parallelism as the proposed technique can. For instance, speculative threads may not perform side-effects that cannot be undone, e.g. I/O. Also, TLS executes ordered sections speculatively, making these an important source of thread squashes. These thread squashes may be avoided with the technique proposed in this paper.

Copy-or-Discard [13] is a software TLS technique that exploits parallel-stage pipeline parallelism speculatively. The code transformation proposed by the authors performs a similar transformation as the one proposed in this paper: Instructions that are likely part of a cross-iteration dependence are moved to a sequentially executed epilogue for the loop. In their case, however, they base themselves on profile information to determine what code to move. Consequently, if a dependence is not caught by profiling information, then the dependent instructions remain in the parallel loop body and speculation will fail. This work, in contrast, yields parallel speedups whether the dependence is execised or not.

The IPOT programming model [15] proviades annotations for identifying transactions in programs, which are executed by an underlying transactional substrate. IPOT provides several annotations that allow the programmer to identify cross-transaction dependences, which can reduce dependence violations on ordered sections. These include reduction patterns, which are a particular type of ordered section, and race conditions that do not impact the program outcome (e.g. updates to a cut-off limit in branch and bound algorithms).

The term *ordered section* is based on the OpenMP construct that indicates that a critical section must execute in the original program order. In OpenMP, however, ordered sections are limited to loops and at most one ordered section may exist per loop [8]. With our technique, we allow multiple ordered sections per loop that are strung together in the correct program order. Furthermore, our discussion of ordered sections also applies to non-DOALL loops and to task parallelism.

6. Conclusion

Ordered sections are code fragments that must be executed in original program order. Within an otherwise parallel loop, they can strongly inhibit the efficient exploitation of parallelism.

This paper presents a method for efficiently executing such ordered sections in the case where the remainder of the loop is not dependent on the values computed in the ordered sections. We extract the ordered sections into tasks and we copy part of the data environment, if necessary. When executing an ordered section, a task is generated for it and placed in queues. Finally, the tasks are taken from the queues in sequential program order and are executed.

We demonstrate the efficacy of this technique on several benchmarks, allowing the parallelization of loops that are otherwise not parallelizable. Scalability of the parallelization is discussed on two multicore processors: a quad-core Intel I7 and a 32-thread Sun Niagara processors. We are currently working on implementing the proposed code transformation in a compiler.

References

[1] D. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications. In *The IEEE International Symposium on Workload Characterization*, pages 163 – 173, Oct. 2005.

[2] S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 302–313, 2006.

[3] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, 2006.

[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 223–234, 2007.

[5] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 71–81, 2004.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[7] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, 1998.

[8] OpenMP application program interface, version 3.0. http://www.openmp.org/, may 2008.

[9] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, 2005.

[10] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, 2008.

[11] S. Rul, H. Vandierendonck, and K. De Bosschere. Extracting coarse-grain parallelism from sequential programs. In *International Conference on Principles and Practices of Parallel Programming*, pages 281–282, Feb. 2008.

[12] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.

[13] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 330–341, 2008.

[14] H. Vandierendonck, S. Rul, M. Questier, and K. De Bosschere. Experiences with parallelizing a bio-informatics program on the Cell BE. In *3rd HiPEAC Conference*, pages 161–175, Jan. 2008.

[15] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–89, 2007.