Using Template Matching to Infer Parallel Design Patterns

ZIA UL HUDA, ALI JANNESARI, and FELIX WOLF, German Research School for Simulation Sciences, RWTH Aachen University, Aachen Germany

The triumphant spread of multicore processors over the past decade increases the pressure on software developers to exploit the growing amount of parallelism available in the hardware. However, writing parallel programs is generally challenging. For sequential programs, the formulation of design patterns marked a turning point in software development, boosting programmer productivity and leading to more reusable and maintainable code. While the literature is now also reporting a rising number of parallel design patterns, programmers confronted with the task of parallelizing an existing sequential program still struggle with the question of which parallel pattern to apply where in their code. In this article, we show how template matching, a technique traditionally used in the discovery of sequential design patterns, can also be used to support parallelization decisions. After looking for matches in a previously extracted dynamic dependence graph, we classify code blocks of the input program according to the structure of the parallel patterns we find. Based on this information, the programmer can easily implement the detected pattern and create a parallel version of his or her program. We tested our approach with six programs, in which we successfully detected pipeline and do-all patterns.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms: Design, Performance

Additional Key Words and Phrases: Parallel pattern detection, pipeline detection, do-all detection, parallelism

ACM Reference Format:
Zia Ul Huda, Ali Jannesari, and Felix Wolf. 2014. Using template matching to infer parallel design patterns. ACM Trans. Architec. Code Optim. 11, 4, Article 64 (December 2014), 21 pages.
DOI: http://dx.doi.org/10.1145/2688905

1. INTRODUCTION

The success story of multicore processors means that the market for single-core processors is shrinking continuously. During the past decade, the degree of parallelism available in the hardware has grown quickly and decisively. Unfortunately, turning an existing sequential program into a parallel one is not an easy task. Although modern compilers are able to detect the parallelism available in simple loops [Kennedy and Allen 2002; Vachharajani et al. 2007], a wide range of sequential programs cannot profit from these strategies because such compilers may miss coarser-grained but sometimes more scalable parallelism.

To strengthen programmer productivity, over the years software engineers have collected a comprehensive list of sequential design patterns [Gamma et al. 1995]. These patterns improve the quality, reusability, and maintainability of the software. To

Authors' address: Z. Ul Huda, A. Jannesari, and F. Wolf, German Research School for Simulation Sciences, RWTH Aachen University, 52062 Aachen, Germany; email: {z.ul-huda, a.jannesari, f.wolf}@grs-sim.de.

DOI: http://dx.doi.org/10.1145/2688905

simplify the development of parallel programs, many parallel design patterns have been introduced [Mattson et al. 2004]. These patterns provide reusable solutions for common problems and help avoid concurrency bugs like deadlocks and data races, which are very difficult to locate. These patterns are usually implemented via standard parallel programming interfaces such as OpenMP or Intel TBB. Jahr et al. [2013] presented a systematic pattern-based approach to convert an existing sequential program into a parallel one. However, their method requires intimate familiarity with the code to be parallelized as well as comprehensive knowledge of parallel constructs and patterns.

Many useful techniques have been proposed to infer the design patterns underlying already existing sequential applications [Dong et al. 2008; Hayashi et al. 2008; Gupta et al. 2011]. One of them, *template matching* [Dong et al. 2008], identifies design patterns by mapping a UML diagram of the pattern onto the UML diagram of the target software. In this article, we show how template matching can also be used to find parallel patterns in sequential programs as a guideline for their later parallelization. Following the observation that most parallel patterns are based on assumptions about data dependences, we replace UML diagrams with data-dependence graphs to characterize both the pattern and the software. Our approach, which is implemented as an extension of the data dependence profiler DiscoPoP [Li et al. 2013; Li et al. 2015], automatically infers potential parallel design patterns from the dependence graph of the program and specifies the division of code blocks according to the pattern structure. Based on this information, the programmer can then easily parallelize the code by moving suggested code blocks into appropriate structures of the pattern. We currently support pipeline and do-all patterns, which frequently appear in applications from a wide variety of domains. Further patterns will be addressed in future work.

We applied our approach to five applications from the PARSEC 3.0 benchmark suite [Bienia 2011] and the open-source audio encoder LibVorbis [Overview 2014]. In PARSEC, which is shipped with parallel versions already included, our method successfully found all implemented pattern instances of the pipeline and do-all. For LibVorbis, which came without a preexisting parallel version, we implemented the detected patterns ourselves.

The remainder of the article is organized as follows. After introducing our dependence-profiling method and the structure of the resulting dependence graph in Section 2, we discuss our specific interpretation of template matching in Section 3. In Section 4, we present evaluation results, followed by a review of related work in Section 5. Section 6 presents our conclusions and sketches future work.

2. BACKGROUND

The techniques employed for the detection of sequential design patterns are based on UML specifications of patterns and programs. In our approach, which targets parallel design patterns, the role of UML diagrams is played by data dependence graphs. The nodes of a dependence graph represent pieces of code and its edges represent data dependences. The dependences determine whether or not two nodes can be executed concurrently.

To obtain the dependence graph of a sequential program, we use an enhanced version of the dependence profiler DiscoPoP [Li et al. 2013; Li et al. 2015]. The version used in this study is based on LLVM [Lattner and Adve 2004] and follows a hybrid (i.e., combined static and dynamic) dependence analysis approach. An overview of the workflow of DiscoPoP is shown in Figure 1. The source code is converted into LLVM-IR. The first phase includes static and dynamic analysis of the input program and produces the program execution tree. This tree includes information about the control regions and the dependence graph. Although this output is still input sensitive, it is

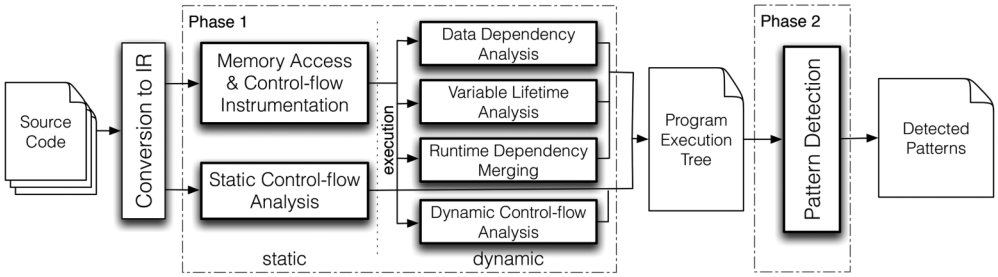

Fig. 1. Workflow of DiscoPoP.

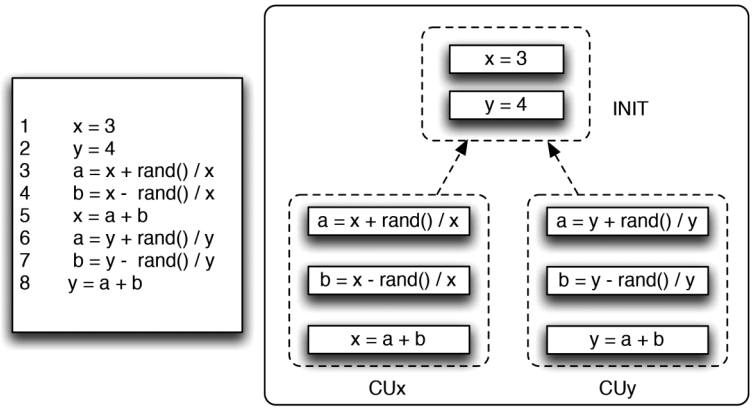

Fig. 2. A simple example of a CU graph.

much more efficient than purely dynamic methods. To mitigate the effects of input sensitivity, we profile each of our test programs with five different inputs and consider all dependences that occur. DiscoPoP's dependence profiling has an average slowdown of 137x and memory consumption of 540MB. The last phase of DiscoPoP is the pattern detection running on the program execution tree.

The dependence graph, generated dynamically by DiscoPoP, considers the program as a collection of *computational units* (CUs) that are connected via true data dependences, which are the main obstacles to parallelization. This type of dependence graph is called a *CU graph*. A CU is a piece of code that follows the read-compute-write pattern. Figure 2 shows a simple CU graph consisting of two CUs built from a set of instructions (i.e., source lines). Lines 1 and 2 initialize variables x and y. They are merged into one *INIT* node, a special type of CU, since they write to these variables for the first time without any computation. Lines 3 and 4 read x and perform some computation involving intermediate variables a and b, and eventually line 5 writes the result back to x. These lines together form CU_x. CU_y is created using the same rules, covering the code from lines 6 to 8. Both CU_x and CU_y use the data from *INIT*. This creates a true dependence between *INIT* and these CUs. The next section gives detailed insight into how we can infer parallel design patterns from CU graphs.

3. APPROACH

The input of our pattern detection algorithm is a CU graph mapped onto the dynamic execution tree of the program. Both are produced by DiscoPoP. Figure 3(a) shows an execution tree with dependences between nodes. To build the tree, we currently consider

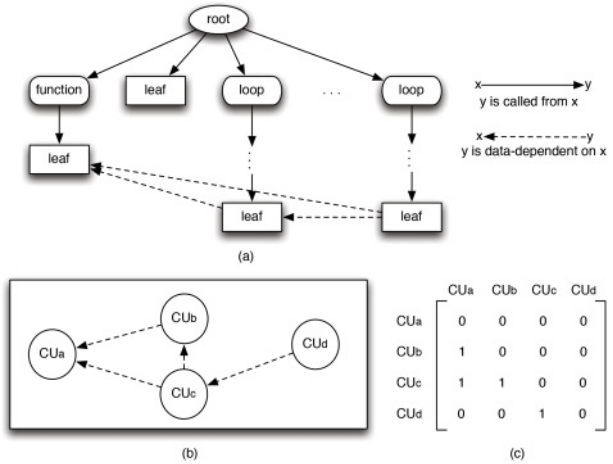

Fig. 3. (a) An execution tree with dependences. (b) A sample CU graph inside a leaf node. (c) Adjacency matrix of the sample CU graph.

only function calls and loops. Each function call or start of a loop forms a new child node. Different iterations of a loop are merged together into one loop node. The leaf nodes contain the code of the parent node divided into CUs. Data dependences may exist both within the same and between different leaf nodes. Data dependences between the CUs of a leaf node are shown in Figure 3(b). This CU graph can be easily converted into an adjacency matrix as in Figure 3(c), where each row and column represent a CU and the corresponding entry in the matrix indicates whether a data dependence exists between them.

We build upon the template-matching technique introduced by Dong et al. [2008]. There, both the template and target program are represented by vectors. Cross-correlation between two vectors is used to determine how similar they are. We adapt this concept for the detection of parallel patterns in CU graphs. The templates used by Dong et al. are well defined. They are derived from UML diagrams of sequential patterns, which expose a fixed structure with a fixed number of components. On the other hand, the structure of parallel design patterns is also well defined, but the number of components in a pattern may vary significantly. For example, the number of stages in a pipeline may differ, depending on the nature of the application. Hence, our parallel pattern templates cannot be of fixed length as in the case of sequential patterns. A parallel pattern matches if certain conditions are met. For example, for a pipeline to exist, there should be some separable stages inside a repeating code section such as a loop. Each stage should depend on the data produced by the previous stage. Obviously, such conditions are less rigid than those derived from fixed UML diagrams, which is why the size of our pattern templates must be adjustable.

Algorithm 1 shows the overall workflow of our approach. We first look for hotspots in the input program—sections such as loops or functions that have to shoulder most of the workload. For each hotspot and pattern, we then create a pattern vector $\vec{p}$, whose length is equal to the number n of CUs in the hotspot. The pattern vector plays the role of the template to be matched to the program. After that, we create the pattern-specific graph vector $\vec{g}$ of the hotspot's CU subgraph, which represents the part of the program to which the pattern vector is matched. Vectors $\vec{p}$ and $\vec{g}$ are derived from adjacency matrices reflecting dependences in the pattern and in the CU graph, respectively. As a next step, we compute the correlation coefficient of the two vectors using the following

ALGORITHM 1: Parallel pattern detection

```
tree = getExecutionTree(serialProgram)
Hotspots = findHotspots(tree)
for each h in Hotspots do
    CUGraph = getCUGraph(h)
    n = getNumberOfCUs(CUGraph)
    for each p in ParallelPatterns do
        p⃗ = getPatternVector(p, n)
        g⃗ = getGraphVector(p, CUGraph)
        CorrCoef[h, p] = CorrCoef(p⃗, g⃗)
    end
end
return CorrCoef
```

formula:

$$CorrCoef = \frac{\vec{p} \cdot \vec{g}}{\|\vec{p}\| \, \|\vec{g}\|}. \tag{1}$$

The correlation coefficient of the pattern vector and the graph vector of the selected section tells us whether the pattern exists in the selected section or not. The value of the coefficient is always in the range of [0, 1.0]. A 1.0 indicates that the pattern exists fully, whereas a 0 indicates that it does not exist at all. A value in between shows the pattern can exist but with some limitations, which we need to work around. Our tool points out the dependences that cause the value of the correlation coefficient of a pattern to be less than 1.0. This helps the programmer to resolve these specific dependences if he or she wants to implement that pattern.

Our tool not only indicates whether a parallel design pattern has been found in some section of the program but also shows how the code must be divided to fit the structure of the pattern. This helps the programmer implement the pattern and create a parallel version of the sequential input program. One should keep in mind that dynamic dependence profiling is never completely accurate and some dependences may not be reported due to the input sensitivity. Though we mitigate the input sensitivity by using different inputs, the programmer still needs to verify the correctness of the transformed parallel code. So far, we support the detection of *pipeline* and *do-all* patterns, which are discussed in the following two subsections.

3.1. Pipeline

Implementing a pipeline only makes sense if its stages are executed many times. For this reason, we restrict our search for pipelines to loops, functions with multiple loops, and recursions. In order to find a pipeline, we first let DiscoPoP deliver the CU graphs of all hotspots. Because DiscoPoP counts the number of read and write instructions executed in each loop or function, we currently use this readily available metric as an approximation of the workload when searching for hotspots. A more comprehensive criterion, including execution times and workload, will be implemented in the future. We then compute an adjacency matrix for each hotspot graph, which we call the *graph matrix*. For each graph matrix, we create a corresponding pipeline pattern matrix of the same size, which we call the *pipeline matrix*. Pipeline matrices encode a very specific arrangement of dependences expected between CUs. For example, there must be a dependence chain running through all CUs in the graph because a pipeline consists of a chain of dependent stages. This specific property helps us to derive the pipeline pattern vector from the matrix.

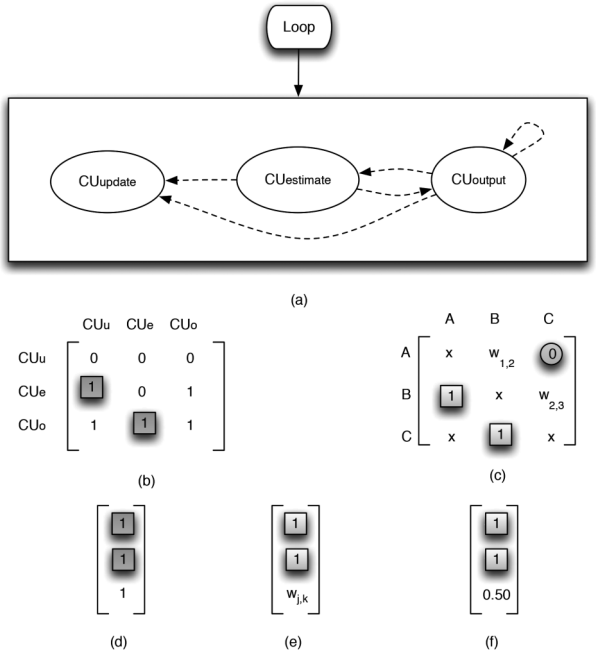

Fig. 4. (a) CU graph for Listing 1. (b) Corresponding graph matrix. (c) A 3x3 pipeline matrix. (d) Graph vector of graph matrix. (e) A three-stage pipeline vector with weight variable $w_{j,k}$. (f) The resulting pipeline vector with $w_{j,k}$ value 0.50.

To make our approach easier to understand, we explain it using an example: the bodytrack application from the PARSEC 3.0 benchmark suite [Bienia 2011]. Bodytrack tracks a 3D pose of a markerless human body with multiple cameras through an image sequence. Listing 1 shows the main loop of the bodytrack. Inside the loop body, line 2 retrieves an input frame, line 5 processes it, and line 6 writes the output to a file. Optionally, line 8 writes a bmp image.

For this loop, DiscoPoP returns the CU graph shown in Figure 4(a). The loop body consists of three CUs: CU_{update} covering lines 2 and 3, $CU_{estimate}$ covering line 5, and CU_{output} covering lines 6 through 8. The corresponding graph matrix of size 3x3 is shown in Figure 4(b).

```
1    for(i=0; i<num_of_frames; ++i} {
2        if{!pf.Update(i)) {
3            return 0;
4        }
5            pf.Estimate(estimate);
6            WritePose(output, estimate);
7            if(outputBMP){
8                outputBMP(estimate);
9            }
10   }
```

Listing 1. Main loop of bodytrack.

According to the dimensions of the graph matrix, we create a pipeline pattern matrix of size 3x3, as depicted in Figure 4(c). We call it the *pipeline matrix*. Each row or column

of this matrix represents a stage in the pipeline and the entries of the matrix represent dependences between them. Note that a pipeline matrix of any arbitrary number of stages can be created using the same pattern as shown in Figure 4(c). The entries of the pipeline matrix have the following specific meaning:

—An "x" means *don't care*, that either 1 or 0 can be in its place. These dependences do not affect pipeline creation.
—A 1 indicates a mandatory dependence. The 1 entries together represent the chain of dependences along the stages of the pipeline. We call them the *chain dependences*.
—$w_{j,k}$ indicates forward dependences in the pipeline. Each individual $w_{j,k}$ quantifies the weight of this forward dependence. A *forward dependence* exists if a stage S_j of pipeline iteration i depends on the result of a stage S_k of the previous iteration $i-1$, with $j < k$. Hence, a forward dependence adversely affects the execution of the pipeline because an earlier stage of an iteration has to wait for the results of a later stage of the previous iteration. We calculate the value of each weight $w_{j,k}$ using the following formula:

$$w_{j,k} = 1 - \frac{(k-j)}{(\#stages - 1)}. \tag{2}$$

#stages represents the total number of stages in the pipeline. The weight decreases as the distance between two stages with forward dependences increases. Increasing distance between two stages with a forward dependence implies lower pipeline performance because the next iteration has to wait longer for the result of a later stage of the previous iteration.
—The 0 in the last column of the first row (encircled) of the pipeline matrix ensures that the first stage does not depend on the last stage because such a dependence would effectively serialize the pipeline. To rule out a serial pipeline, we always first verify that the graph matrix also contains the mandatory 0 in the last column of the first row.

The next step is to create the *graph vector* and the *pipeline vector* from the graph matrix and the pipeline matrix, respectively, and to calculate their correlation coefficient. To create these vectors, we use the chain and forward dependences. Figure 4(d) shows the graph vector and Figure 4(e) shows the pipeline vector. These vectors are created as follows.

We fill the vectors from top to bottom with the exception of the last element using the entries in chain dependence locations of their corresponding matrices. In both cases, the last vector element is reserved for forward dependences. If there is no forward dependence in the graph matrix, the last element of both vectors becomes 0. If there is at least one forward dependence present in the graph matrix, the last element of the graph vector becomes 1 and the last element of the pipeline vector becomes the minimum across all forward-dependence weights in the pipeline matrix whose corresponding entry in the graph matrix is nonzero.

Figure 4(f) shows the pipeline vector after calculating all weights and choosing the minimum or zero. In general, the smaller the weight value in the pipeline vector, the smaller is the value of the correlation coefficient. Finally, we calculate the correlation coefficient between the two vectors according to Equation (1), which tells us whether a pipeline exists in Listing 1 or not. We detected a three-stage pipeline pattern in the loop. The three stages correspond to the three CUs in the CU graph in Figure 4(a).

In our example, the correlation coefficient is 0.96. The value of the correlation coefficient is less than 1.0, showing that the detected pipeline has a forward dependence that needs to be resolved. Therefore, some additional code restructuring is needed to implement the pipeline. One way to resolve a forward dependence in a pipeline is to

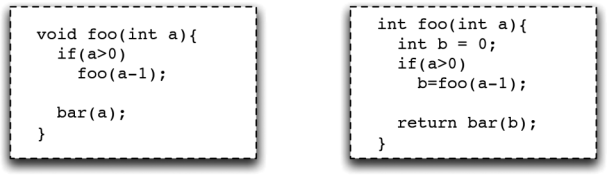

Fig. 5. Recursive functions: (a) pipeline can be implemented; (b) pipeline cannot be implemented.

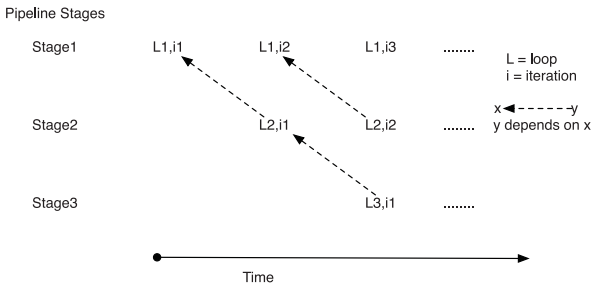

Fig. 6. A pipeline with iterations from different loops in each stage.

merge all the stages in the loop created by the forward dependence. Another solution is to add synchronization between the stages that are affected by the forward dependence.

The diagonal entries of a graph matrix represent self-dependences of CUs. A self-dependence of a CU in a loop is a loop-carried dependence. If a pipeline stage does not have any loop-carried dependence, then multiple instances of this stage can run in parallel. Such a stage is called a parallel stage. Our graph matrix of the hotspot, for which a pipeline is detected, readily shows which stages are parallel stages by taking their diagonal entries into account.

3.1.1. Recursive Function Pipeline. A pipeline may also be created from a recursive function. The only precondition is that the current recursive invocation does not depend on the results generated by subsequent recursive invocations, as done in the function shown in Figure 5(a). The function in Figure 5(b) calls itself and then uses the results generated by those recursive calls in further computations. The usage of recursion results prevents a pipeline because we need the next recursive invocation to complete before the current invocation can continue. To implement a detected pipeline for a recursive function, the code has to be restructured into a loop, which requires additional code refactoring. In the evaluation section, we show how we detect a pipeline in a recursive function of one of our test programs.

3.1.2. Multiloop Pipeline. Another special case is a pipeline that is hidden because it stretches across more than one loop. In such a scenario, one iteration of a loop depends on one or more iterations of a preceding loop. This can be easily transformed into a pipeline by mapping iterations of different loops onto different stages of the pipeline. Figure 6 shows a pipeline covering multiple loops. To detect such multiloop pipelines, we examine the dependence relation between the iterations of these loops. If it is a one-to-one relation (i.e., the ith iteration of the second loop depends only on the jth iteration of the first loop) or a linear relation, then we can easily transform such loops into a pipeline. Currently, relations between the iterations of different loops are determined manually. We aim to derive this information automatically in future versions.

```
1    for{i=0; i<num; ++i}{
2        InitializeDensitiesAndForces(i);
3    }
4    for{i=0; i<num; ++i}{
5        ComputeDensities(i);
6    }
7    for{i=0; i<num; ++i}{
8        ComputeDensities2(i);
9    }
10   for{i=0; i<num; ++i}{
11       ComputeForces(i);
12   }
```

Listing 2. Loops in function `ComputeForce()` of fluidanimate.

For the sake of simplicity, we currently consider only loops spread across the same function. Listing 2 contains the loops in function `ComputeForce()` of the PARSEC benchmark fluidanimate, for which such a relation can be shown. Each loop in the function is treated as a CU and a graph matrix reflecting the dependences between these loops is created. After this point, we proceed as usual.

3.2. Do-All

A loop can be parallelized according to the do-all pattern if there are no loop-carried or interiteration dependences. A forward or self-dependence is always loop carried, as the control flow within a loop iteration moves in a forward direction, which is why dependences within the same iteration must point backward. Note that inner loops in loop nests, which may reverse the control-flow direction whenever a new inner iteration starts, are treated separately. The absence of forward or self-dependences is easy to verify based on the graph matrix, whose upper triangle shows all forward and self-edges (encircled in Figure 7).

Of course, there may also exist loop-carried dependences in backward edges of the graph matrix. However, to reliably distinguish them from intraiteration dependences, our dependence profiler would have to record the iteration number along with each memory access, substantially increasing its memory overhead. On the other hand, loop-carried dependences in a backward direction that are not accompanied by dependences in a forward direction in the same loop are very rare. Basically, the absence of forward and self-dependences in a loop is a good indicator of the absence of loop-carried dependences. For this reason, we decided to refrain from the costly classification of backward dependences into loop carried or not loop carried.

The pattern vector $\vec{p}$ for a do-all pattern is of length equal to the number of elements of the upper triangle of the CU matrix. All elements in $\vec{p}$ are initialized with 1. All elements of the upper triangle of the graph matrix are taken as elements of the graph vector $\vec{g}$, with their values being reversed from 0 to 1 and vice versa. Figure 7 shows some loop graphs along with their graph vectors $\vec{g}$.

Once we have determined the graph vector of our target loop, we calculate the correlation coefficient according to Equation (1). If the correlation coefficient is 1.0, the loop is a do-all loop. A correlation coefficient of less than 1.0 means that the loop cannot be parallelized with the do-all pattern in its current form due to some loop-carried dependences. The closer the value of the correlation coefficient is to 1.0, the fewer the number of loop-carried dependences. These loop-carried dependences are highlighted in the output of the tool and the programmer can try to resolve them manually, if the parallel implementation using the do-all pattern seems profitable.

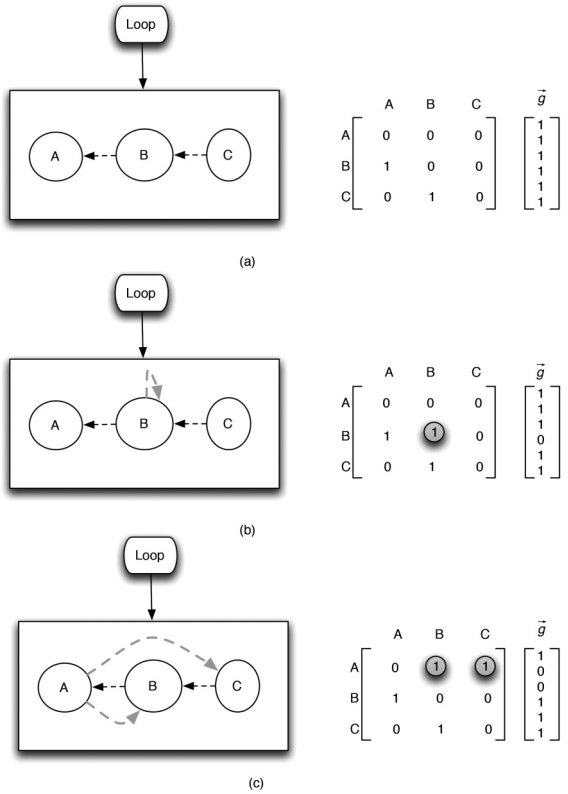

Fig. 7. Example of loops for do-all detection: (a) do-all loop; (b) no do-all loop due to self-edge; (c) no do-all loop due to forward edge.

4. EVALUATION

We tested our approach using five shared-memory applications from the PARSEC 3.0 benchmark suite [Bienia 2011] and the open-source audio encoder LibVorbis [Overview 2014]. Each of the five PARSEC programs is shipped with a sequential and a parallel version. We chose bodytrack, dedup, and ferret because their parallel versions contain pipelines. We chose blackscholes because it is parallelized using the do-all pattern. Fluidanimate was included as a demonstrator of multiloop pipelines. The time and memory consumption of the pattern detection heavily depends on the depth of the program execution tree. On average, the profiling caused a slowdown of 137x, while the pattern detection caused a slowdown of 62x. The pattern detection consumed 152.95MB of memory on average for all six applications compared to 540MB consumed by the profiler. The pattern detection was applied to all hotspots in these six applications. Table I summarizes the hotspots found in our test cases.

4.1. Pipeline

We tested all of the six candidate programs for the presence of pipeline patterns. We implemented the detected pipelines in all the programs. We also compared the results obtained for bodytrack, dedup, and ferret with their pipeline versions, which are already available. The speedups of the detected pipeline versions were measured on an Intel Xeon X5670 CPU with six cores and hyperthreading enabled. There were no pipelines detected in blackscholes. Table II gives an overview of the detected pipelines

Table I. Hotspots in Test Cases

Application	LOC	Hotspots	Execinst%*
bodytrack	7,698	7	88.62%
blackscholes	914	2	99.75%
dedup	3,347	2	99.88%
fluidanimate	3,987	4	99.54%
ferret	11,263	2	99.96%
LibVorbis	54,743	2	99.80%

*Percentage of executed read and write instructions covered by the identified hotspots.

Table II. Detected Pipelines with Correlation Coefficients

Application	Pipelines Detected	Corr Coeff	Pipelines Implemented	Pipelines in PARSEC
bodytrack	1	p1: 0.96	1	1
blackscholes	0	NA	0	0
dedup	2	p1: 1.00 p2: 1.00	2	1
fluidanimate	1	p1: 0.94	1	0
ferret	2	p1: 1.00 p2: 1.00	2	1
LibVorbis	2	p1: 1.00 p2: 1.00	1	NA

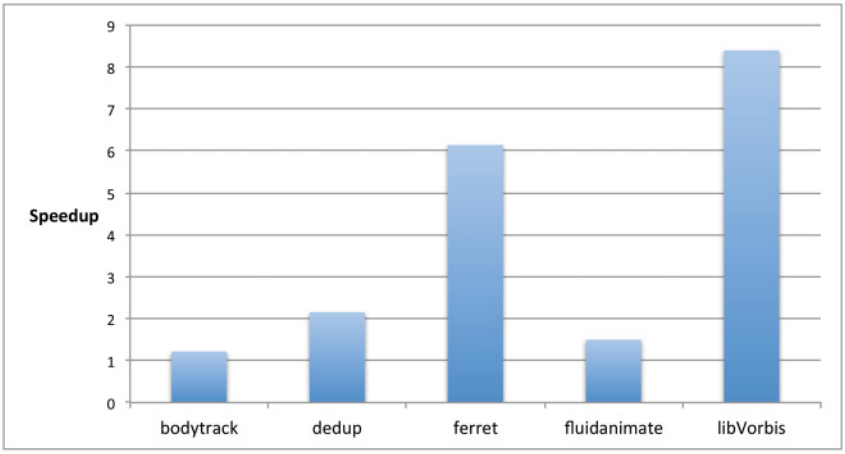

Fig. 8. Average speedups of pipeline implementations in all applications.

along with their correlation coefficients, while Figure 8 shows the average speedups achieved after implementing the detected pipelines. All the speedups were measured using 12 threads.

4.1.1. Bodytrack. In one of bodytrack's hotspots, we detected a pipeline pattern with a correlation coefficient of 0.96. Additional restructuring of the code was needed. It is the loop shown in Listing 1, which iterates over all input frames to track the body of a person. It covers 88.62% of the executed read and write instructions. As shown in Figure 4, the pipeline consists of three stages. CU_{update} updates a frame, $CU_{estimate}$ calculates estimates, and CU_{output} writes the data to a file. Due to the forward dependence

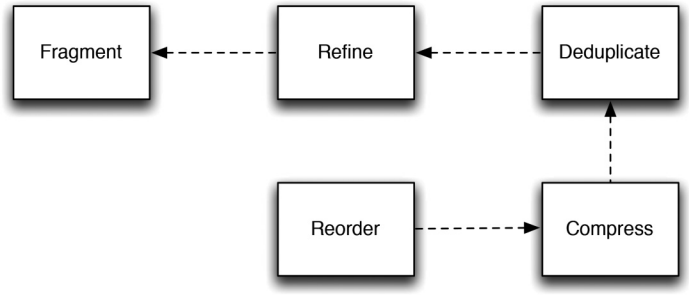

Fig. 9. Five-stage pipeline in the parallel version of dedup.

between $CU_{estimate}$ and CU_{output}, we analyzed the code and decided to merge these two CUs into a single stage, forming a two-stage pipeline. We achieved an average speedup of 1.2 for this implementation because both stages of the pipeline were serial stages.

The pipeline implementation of bodytrack has indeed been implemented with two stages: an update stage, which reads the data, and an estimate and output stage. This example demonstrates that a forward dependence does not necessarily present an insurmountable obstacle to the formation of a pipeline. Sometimes, a combination of merging stages and shifting the responsible instructions into other stages does the trick.

4.1.2. Dedup. Pipeline patterns were reported in two hotspots of dedup. These hotspots are the two nested loops shown in Listing 3. They cover almost all of the executed read and write instructions. In the outer loop, we found a three-stage pipeline with `readInput()` and `Fragment()` as the first two stages and the entire inner loop as the last stage. The correlation coefficient of this pipeline is 1.0. We detected another pipeline, this time with four stages, in the inner loop: `Refine()`, `Deduplicate()`, `Compress()`, and `Reorder()`. The correlation coefficient of the second pipeline is also 1.0. We implemented both pipelines nested into one another. An average speedup of 2.3 was achieved. All of the stages detected were serial stages.

```
1    while{!EOF}{
2        readInput(Buffer);
3        Fragment(Buffer);
4
5        while{Blocks = Split(Buffer)}{
6            Refine(Block);
7            Deduplicate(Block);
8            Compress(Block);
9            Reorder(Block);
10       }
11   }
```

Listing 3. Main hotspot loop of dedup.

The existing parallel version of dedup in PARSEC runs a flat five-stage pipeline, which is shown in Figure 9. It covers the entire loop nest from Listing 3 except for reading the input at the entry of the outer loop. Instead of reading it periodically, the parallel version reads the input in one piece at program start. The remaining stages we detected in dedup match those of the benchmark's parallel version exactly.

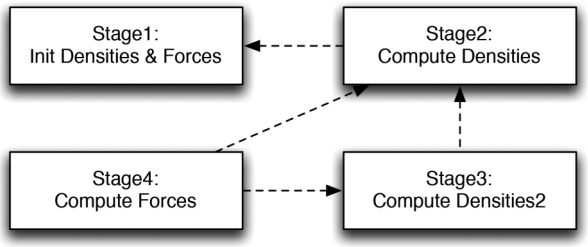

Fig. 10. Self-implemented pipeline in fluidanimate.

The speedups of our own nested-pipeline version and of the flat-pipeline version from the benchmark were almost the same. The reason is that most of the work is done by the inner loop with the loop-carried dependence, which makes it a bottleneck. So the decision of creating a nested versus a flat pipeline does not affect the speedup.

4.1.3. Fluidanimate. In fluidanimate, the reported pipeline consists of multiple loops in the function `ComputeForce()`, consuming almost 90% of the read and write instructions. Listing 2 shows the pseudo-code of `ComputeForce()`. Its correlation coefficient of 0.94 is slightly less than 1.0 because of a forward dependence found between the loops at lines 4 and 10 of Listing 2. As discussed in the previous section, we need to examine possible cross-loop dependences to decide whether a multiloop pipeline exists. We analyzed the code for cross-loop dependences and discovered that each iteration of both the second and the fourth loop in Listing 2 changes the densities of the current cell and its neighbors, which causes the forward dependence. However, it was found that the cell update affects only those neighbors in the grid whose index is less than the index of the current cell. So running a loop from the highest to the lowest index would solve this problem.

Based on this workaround concept, we designed our pipeline for `ComputeForce()`. The only problem with this implementation was that stage 4 had to wait until all the neighboring cells of the current cell had passed through stage 2, decreasing the efficiency of the pipeline. In Figure 10, this is illustrated by the dependence between the second and the fourth stage. This drawback had already been anticipated by the previously mentioned forward dependence in the sequential version of `ComputeForce()` between lines 4 and 10 of Listing 2. We achieved an average speedup of about 1.5. Note that merging the stages 2, 3, and 4 due to the forward dependence would not improve the speedup because merging these three stages would create a highly imbalanced pipeline. The parallelization of fluidanimate available in PARSEC is done using data parallelism. Perhaps for this reason, there is no pipeline in that parallel version.

4.1.4. Ferret. Ferret traverses an image database and reports all images that appear similar to an input image. The serial version of ferret searches recursively through all subdirectories of an input directory. Listing 4 shows the workflow of ferret. Functions `scan_dir()` and `dir_helper()` traverse the directory tree, while `do_query()` compares images and writes its findings to a file.

We found two hotspots in ferret. `scan_dir()`, the first one, checks the entries of a directory and calls `do_query()` if an image was found or calls `dir_helper()` to proceed in child directories. We found a two-stage pipeline in this function, which is depicted in Figure 11(a). The correlation coefficient of this pipeline is 1.0. After carefully examining

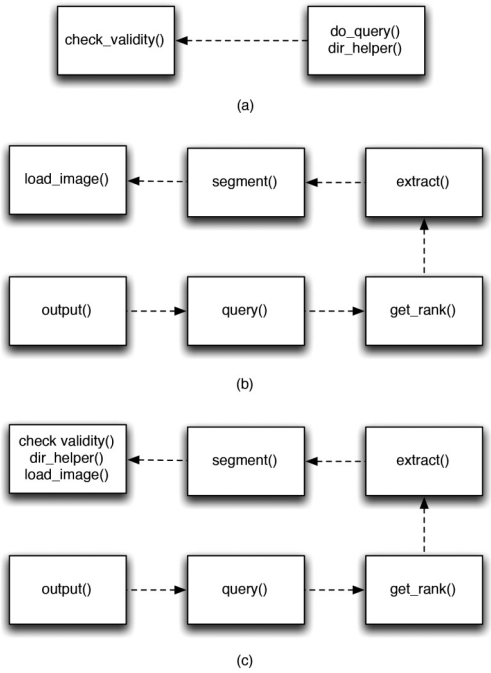

Fig. 11. Pipeline in ferret: (a) pipeline detected in `scan_dir()`; (b) pipeline detected in `do_query()`; (c) pipeline implemented in PARSEC.

the code, we decided that `dir_helper()` should be moved to stage 1. This is because it calls `scan_dir()` again, which marks the start of the next pipeline iteration.

`do_query()`, the second hotspot, is nested inside the first one and covers 96% of the total read and write instructions. There, we found a six-stage pipeline with correlation coefficient of 1.0, which is illustrated in Figure 11(b). We implemented the nested pipelines by refactoring the recursion into iterations and achieved a speedup of 6.14.

We compared the reported pipelines with the parallel implementation in PARSEC. The PARSEC version features only one pipeline, which is shown in Figure 11(c). It has six stages and is essentially a combination of the two we detected, similar to our findings for dedup. The first stage of the benchmark's implemented pipeline merges the entry stages of the two detected pipelines. This is advisable because their workload is so small. The remaining implemented stages correspond to the last five stages of the second pipeline we found.

The difference between the speedups of the two implementations with single flat and nested pipelines, respectively, was negligible. The reason was the same as in the case of dedup. This comparison demonstrates that our division into CUs is not always optimal. To improve the load balance between pipeline stages, future versions of our tool will merge the CUs of adjacent pipeline stages if appropriate.

4.1.5. LibVorbis. LibVorbis [Overview 2014] is an audio encoding and decoding library, which converts wave files into compressed Ogg files. We used version 1.3.3 for our study. We analyzed a client application distributed alongside the library, which uses the library for conversion purposes. We found two hotspots in the form of two loops with one of them nested inside the other—very similar to the code in Listing 3. In the outer loop, we detected a pipeline with two stages: the first one for data input and the second

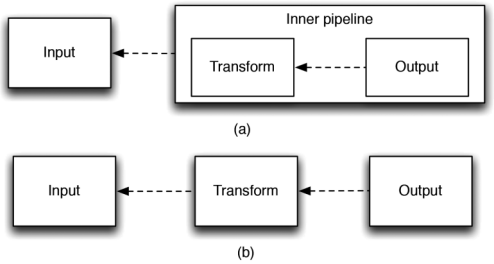

Fig. 12. Implemented pipelines of LibVorbis: (a) nested and (b) merged.

```
1    scan_dir( ent, path){
2        check_validity(ent, path);
3        if(IS_FILE(name)){
4            do_query(name);
5        }
6        else if(IS_DIR(name)){
7            dir_helper(ent, path);
8        }
9    }
10
11   dir_helper( ent, path){
12       dir = get_list(path+ent);
13       while (1) {
14           file = dir.get_next();
15           if (file) {
16               scan_dir(file.name, dir);
17           }
18           else{
19               break;
20           }
21       }
22   }
23
24   do_query(name){
25       image = load_image(name);
26       mask = segment(image);
27       dataset = extract(segment);
28       result = query(dataset);
29       rank = get_rank(result);
30       output(file, name, result, rank)
31   }
```

Listing 4. Pseudo-code of ferret.

one covering the entire inner loop. In the inner loop, we detected another pipeline with two stages, the first one for the transformation of the data and the second one for their output. The correlation coefficient of both pipelines was 1.0.

We decided to implement two parallel versions: one with nested pipelines and one with the two pipelines merged into a single flat three-stage pipeline, as shown in Figure 12(a) and 12(b), respectively. The Transform stage in both cases was detected as a parallel stage. Again, the comparison of the speedups of the two implementations followed the results for dedup and ferret. The speedups of the two implementations were almost identical. The reason is that the Transform stage consumes most of the

Table III. Detected Do-All Loops with a Correlation Coefficient of 1.0

Application	Do-All Detected	Do-All Implemented	Detected by Intel ICC
bodytrack	6	6	0
blackscholes	2*	1	2
dedup	0	0	0
fluidanimate	0	0	0
ferret	0	0	0
LibVorbis	0	0	0

*One do-all loop is not an intrinsic part of blackscholes.

Table IV. Detected Hotspot Loops with a Do-All Correlation Coefficient of Less Than 1.0

Application	Hotspot Loops	Corr Coeff
bodytrack	1	d1: 0.98
blackscholes	0	NA
dedup	2	d1: 0.99 d2: 0.94
fluidanimate	2	d1: 0.98 d2: 0.98
ferret	0	NA
LibVorbis	2	d1: 0.96 d2: 0.70

runtime in both cases. With three different input files, we achieved an average speedup of 8.4.

4.2. Do-All

Table III summarizes the the results of do-all pattern detection, where the correlation coefficient was 1.0. We compared the identified do-all patterns with the parallel implementations in PARSEC and with the do-all loops detected statically by the Intel C compiler (ICC) version 14.0.3. We did not find any profile-guided tool available to compare our results with. We could not find any do-all loops in hotspots of dedup, fluidanimate, ferret, and LibVorbis with a correlation coefficient of 1.0. ICC found do-all loops only in blackscholes. Table IV shows the hotspot loops whose correlation coefficients for do-all were less than 1.0. The value of the do-all correlation coefficient depends on the number of loop-carried dependences and the total number of CUs in a loop. The detailed results are presented next.

4.2.1. Bodytrack. We classified six loops in bodytrack as do-all loops, each with a correlation coefficient of 1.0. All of these loops exist in functions called from the loop shown in Listing 1. They are specified in the functions `FlexFilterRowV()`, `FlexFilterColumnV()`, `GradientMagThreshold()`, `CalcWeights()`, `GenerateNewParticles()`, and `GetObservation()`. All of these six loops are parallelized in PARSEC according to the do-all pattern. No loop other than these six is parallelized. This means we reported neither false negatives nor false positives. ICC did not find any do-all loops in bodytrack.

Our approach can detect parallel patterns at different levels of the hierarchy in the program execution tree. Detection of a combination of more than one parallel pattern at the same level would need specific templates for each combination. That new templates need to be defined for each combination of patterns is a limitation of the current approach, which will be addressed in future work.

The correlation coefficient of do-all detection for the loop in Listing 1 was 0.98. It was due to two loop-carried dependences, as shown in Figure 4(a). The first dependence was because $CU_{estimate}$ uses the values generated by CU_{output} in the previous stage and the second was a self-dependence of CU_{output}. We could not resolve these dependences to make this loop a do-all loop. This shows that loop-carried dependences can sometimes not be resolved, even if their number is low. A correlation coefficient close to 1.0 does not necessarily mean that the loop can be easily converted into a do-all loop.

4.2.2. Blackscholes. In blackscholes, we found two nested loops with a correlation coefficient of 1.0, together covering 99.75% of the total read and write instructions. A closer investigation of the code revealed that the outer loop is not an intrinsic part of blackscholes, but rather a wrapper in PARSEC that runs the blackscholes algorithm a specified number of times on the same input. This is why it was not parallelized in PARSEC, although it could have been in principle. The inner loop is shown in Listing 5. It is the main blackscholes loop, which calculates the value of an option. The parallel implementation in PARSEC follows the do-all pattern. ICC also found the same loops to be do-all loops. We did not find any loop in blackscholes with a correlation coefficient of less than 1.0.

```
1    for (i=start; i<end; i++) {
2        price = BlkSchlsEqEuroNoDiv( i, 0);
3        prices[i] = price;
4    }
```

Listing 5. Do-all loop in blackscholes.

4.2.3. Dedup. We could not find any hotspot loops in dedup with a correlation coefficient of 1.0. ICC also failed to find any do-all loops in dedup. The two nested hotspot loops for which we implemented pipelines received a correlation coefficient of 0.99 and 0.94, respectively.

The outer loop could not be converted into a do-all loop because of the serial input and one variable causing loop-carried dependences. This dependence could not be resolved. Similarly, the inner loop could not be converted to do-all due to the splitting of input buffer in each iteration depending on the split from the previous iteration. Also, the output of results in this loop was serial.

4.2.4. Fluidanimate. Two hotspot loops in the `ComputeForce()` function had a do-all correlation coefficient of 0.98. Because both of these loops update the densities of neighboring cells in multiple iterations, these loops could not be parallelized using the do-all pattern.

4.2.5. LibVorbis. The do-all search in LibVorbis produced results similar to the one for dedup. The two nested hotspot loops that were identified as pipelines were also checked for do-all patterns and correlation coefficients of 0.96 and 0.70 were computed. The outer loop had loop-carried dependences due to the serial file input and some internal flags of the LibVorbis library. These dependences were not resolvable, as the LibVorbis library uses these flags for maintaining its internal state. The inner loop also depended on the internal flags of the LibVorbis library. Moreover, the file output was sequential. These dependences could not be resolved to realize a do-all pattern in the loop.

The evaluation demonstrates that our approach can successfully detect parallel patterns. Patterns other than pipeline and do-all can also be detected using template matching. For instance, in the case of the map/reduce pattern, we could construct a

template vector that looks for the absence of forward dependences in a CU matrix and allows self-dependences for the reduction operations in the loops. Similarly, in the case of the master/worker pattern, where each worker task is usually independent of other tasks, we could construct a template vector that looks for independence between CUs of a region and classifies them as worker tasks. The detection of these patterns and others will be added to the tool in future.

5. RELATED WORK

In the field of software engineering, the detection of design patterns is being widely pursued. Major attention is paid to the detection of sequential design patterns in UML diagrams of existing programs. For this purpose, Qiu et al. [2010] use subgraph discovery via graph isomorphism. First, they convert UML diagrams of both the application under study and the pattern into abstract syntax graphs. Then, they use a finite state machine over these graphs to detect the pattern in the application.

While Hayashi et al. [2008] use meta-patterns to infer the sequential patterns underlying the design of a program, Tsantalis et al. [2006] use matrix similarity algorithms on adjacency matrices. However, Dong et al. proved that the approach of Tsantalis et al. is unable to distinguish between multiple design-pattern instances in the program's design. To eliminate this drawback, they proposed an improvement based on template matching, a method well known from the area of digital image processing. To the best of our knowledge, there is no previous work on parallel pattern detection using template matching techniques.

During the last few years, the automatic discovery of parallelism in sequential programs has also been the subject of research projects. Raman et al. [2012] extract the parallelism from sequential code directly at compile time. A drawback of their approach is that they require programmers to manually analyze the code and include hints for possible parallelism into the program prior to compilation.Vandierendonck et al. [2010] go one step further and give the hints to the programmer on where to put the hints in sequential code. Again, the programmer has to approve the hints proposed by their compiler. Intel Advisor [Labs 2014] points out the hotspots in the serial code by profiling and helps the programmer to annotate and see the expected speedups without changing the code. Campanoni et al. [2012] extract parallelism by running iterations of a loop on separate threads, fulfilling loop-carried dependences using signals between threads. They also propose an architectural improvement to make their approach more feasible [Campanoni et al. 2014]. Our approach tries to detect parallel patterns in sequential code and suggests a solution without any special architectural requirements other than those commonly satisfied.

Rul et al. [2010] have developed a tool that follows an approach very close to ours. Their profiler can identify loop-carried dependences. Their profiling usually lasts overnight, which is very high compared to the slowdown caused by DiscoPoP. A post analysis of the dependence graphs of the loops is done to detect pipelines. Only loops that contain function calls are analyzed, whereas our approach searches all kinds of loops and functions for parallel patterns. Similarly, Kremlin [Garcia et al. 2011] can detect do-all loops with high precision, but it lacks the ability to detect any other pattern.

DOMORE [Huang et al. 2013] can detect dependences across loop iterations dynamically and synchronize them when needed. Tournavitis et al. introduced a compiler that can identify parallelism using dependence analysis [Tournavitis et al. 2009; Wang et al. 2014]. Initially, they only sought do-all parallelism in loops. Later, they included pipeline detection in the compiler [Tournavitis and Franke 2010]. Compared to our approach, theirs is much less generic when it comes to the addition of new design patterns. For each new design pattern, they need to integrate a separate

detection technique into their compiler. Our method, in contrast, can handle the inclusion of new patterns with minimal overhead, as only the new pattern's template needs to be added. The template-matching algorithm remains the same for all the patterns.

Ketterlin and Claus [2012] developed a tool named Parwiz, which profiles the dependences of the input program and then points the programmer to potential parallelism. At the moment, Parwiz supports only do-all parallelism. Zhang et al. [2009] developed Alchemist, which profiles dependences and identifies code regions that can be run in parallel. Their approach lacks the detection of parallel patterns. Molitorisz [2013] tracks control dependences to discover master/worker patterns, mainly focusing on dependences between function calls.

Raman et al. [2008] proposed a technique called *parallel stage decoupled software pipelining*, which converts programs with pointer-based loops into pipelines. Huang et al. [2010] showed that the approach of Raman et al. can have more parallelism at each stage of the detected pipeline based on the dependence analysis of each stage. The pipeline detected by our approach can also be enhanced using the same techniques as Huang et al. Rul et al. [2007] uses the approach of Raman et al. on the level of functions. Lee et al. [2013] transform user-annotated code into pipelines using Cilk. Given that our approach can identify the code locations of pipeline stages automatically, it could help users of the approach of Lee et al. to quickly find the places to annotate. Similarly, the work of Kambadur et al. [2012] can also be applied to the parallelized version developed after applying our approach to find further hotspots and parallelizing these code regions.

6. CONCLUSIONS AND FUTURE WORK

In this article, we described how to infer potential parallelization patterns from sequential programs. To this end, we adapted the idea of template matching, which is already employed to detect sequential design patterns in UML diagrams. A specific challenge we addressed is the—in comparison to UML diagrams—less rigid structure of parallel design patterns. Our approach identifies those parts of the code that can be parallelized and shows how to map them onto the pattern structure, enabling a quick implementation of the pattern. We believe that this will provide significant help for organizations facing the challenge of parallelizing large numbers of legacy programs.

In the future, we plan to add further patterns such as master/worker, map/reduce, or workpile to our tool's pattern repository. We also intend to estimate speedups for each detected pattern. In this way, the programmer may become able to choose among several pattern alternatives for the same piece of code, taking their correlation coefficients and expected speedups into account. Ultimately, however, our vision is the pattern-guided semiautomatic transformation of sequential code into a parallel version.

ACKNOWLEDGMENTS

We would like to thank the graduate students of our multicore lab and our master's student Bo Zhao for their programming support.

REFERENCES

Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *Proceedings of the ACM / IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. 217–228. DOI: http://dx.doi.org/10.1109/ISCA.2014.6853215

Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings*

of the 10th International Symposium on Code Generation and Optimization (CGO'12). ACM, New York, NY, 84–93. DOI:http://dx.doi.org/10.1145/2259016.2259028

Jing Dong, Yongtao Sun, and Yajing Zhao. 2008. Design pattern detection by template matching. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC'08)*. ACM, New York, NY, USA, 765–769. DOI:http://dx.doi.org/10.1145/1363686.1363864

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Boston, MA, USA.

Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, USA, 458–469. DOI:http://dx.doi.org/10.1145/1993498.1993553

Manjari Gupta, Akshara Pande, and A. K. Tripathi. 2011. Design patterns detection using SOP expressions for graphs. *SIGSOFT Softw. Eng. Notes* 36, 1 (Jan. 2011), 1–5. DOI:http://dx.doi.org/10.1145/1921532.1921541

Shinpei Hayashi, Junya Katada, Ryota Sakamoto, Takashi Kobayashi, and Motoshi Saeki. 2008. Design pattern detection by using meta patterns. *IEICE - Trans. Inf. Syst.* E91-D, 4 (April 2008), 933–944. DOI:http://dx.doi.org/10.1093/ietisy/e91-d.4.933

Jialu Huang, Thomas B. Jablin, Stephen R. Beard, Nick P. Johnson, and David I. August. 2013. Automatically exploiting cross-invocation parallelism using runtime information. In *Proceedings of the 11th International Symposium on Code Generation and Optimization*. IEEE Computer Society, Shenzhen, China, 1–11.

Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. 2010. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, New York, NY, USA, 121–130. DOI:http://dx.doi.org/10.1145/1772954.1772973

Ralf Jahr, Mike Gerdes, and Theo Ungerer. 2013. A pattern-supported parallelization approach. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'13)*. ACM, New York, NY, USA, 53–62. DOI:http://dx.doi.org/10.1145/2442992.2442998

Melanie Kambadur, Kui Tang, and Martha A. Kim. 2012. Harmony: Collection and analysis of parallel block vectors. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 452–463. DOI:http://dx.doi.org/10.1145/2366231.2337211

Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Alain Ketterlin and Philippe Clauss. 2012. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 437–448. DOI:http://dx.doi.org/10.1109/MICRO.2012.47

Intel Labs. 2014. Intel Advisor XE. (2014). https://software.intel.com/en-us/intel-advisor-xe.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, USA, 75.

I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. 2013. On-the-fly pipeline parallelism. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'13)*. ACM, New York, NY, 140–151. DOI:http://dx.doi.org/10.1145/2486159.2486174

Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. 2015. DiscoPoP: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*. Springer International Publishing, Springer Berlin Heidelberg, 1–10 (to appear).

Zhen Li, Ali Jannesari, and Felix Wolf. 2013. Discovery of potential parallelism in sequential programs. In *Proceedings of the 42nd International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*.

Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*. Addison-Wesley Professional.

Korbinian Molitorisz. 2013. Pattern-based refactoring process of sequential source code. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*. Retrieved from http://ps.ipd.kit.edu/backend/tl_files/ipd/Dateien/cat_veroeffentlichunge n/Molitorisz_CSMR2013.pdf.

Libvorbis API Overview. 2014. LibVorbis. Retrieved from http://xiph.org/vorbis/doc/libvorbis/overview.html.

Ming Qiu, Qingshan Jiang, An Gao, Ergan Chen, Di Qiu, and Shang Chai. 2010. Detecting design pattern using subgraph discovery. In *Proceedings of the 2nd International Conference on Intelligent Information and Database Systems: Part I (ACIIDS'10)*. Springer-Verlag, Berlin, 350–359. http://dl.acm.org/citation.cfm?id=1894753.1894795.

Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: A system for flexible parallel execution. *SIGPLAN Not.* 47, 6 (June 2012), 133–144. DOI:http://dx.doi.org/10.1145/2345156.2254082

Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. ACM, New York, NY, 114–123. DOI:http://dx.doi.org/10.1145/1356058.1356074

Sean Rul, Hans Vandierendonck, and Koen De Bosschere. 2010. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Comput.* 36, 9 (2010), 531–551. DOI:http://dx.doi.org/10.1016/j.parco.2010.05.006

Sean Rul, Hans Vandierendonck, and Koen De Bosschere. 2007. Function level parallelism driven by data dependencies. *SIGARCH Comput. Archit. News* 35, 1 (March 2007), 55–62. DOI:http://dx.doi.org/10.1145/1241601.1241612

Georgios Tournavitis and Björn Franke. 2010. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, 377–388. DOI:http://dx.doi.org/10.1145/1854273.1854321

Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F. P. O'Boyle. 2009. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. *SIGPLAN Not.* 44, 6 (June 2009), 177–187. DOI:http://dx.doi.org/10.1145/1543135.1542496

Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. 2006. Design pattern detection using similarity scoring. *IEEE Trans. Software Eng.* 32, 11 (2006), 896–909.

Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. IEEE Computer Society, Washington, DC, 49–59. DOI:http://dx.doi.org/10.1109/PACT.2007.66

Hans Vandierendonck, Sean Rul, and Koen De Bosschere. 2010. The parallax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, 389–400. DOI:http://dx.doi.org/10.1145/1854273.1854322

Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'Boyle. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Trans. Archit. Code Optim.* 11, 1 (Feb. 2014), Article 2, 26 pages. DOI:http://dx.doi.org/10.1145/2579561

Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. 2009. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'09)*. IEEE Computer Society, Washington, DC, 47–58. DOI:http://dx.doi.org/10.1109/CGO.2009.15

Received June 2014; revised October 2014; accepted November 2014