Detecting the Existence of Coarse-Grain Parallelism in General-Purpose Programs

Sean Rul, Hans Vandierendonck, and Koen De Bosschere

Ghent University, Department of Electronics and Information Systems/HiPEAC, B-9000 Gent, Belgium, {srul,hvdieren,kdb}@elis.ugent.be

Abstract. With the rise of chip-multiprocessors, the problem of parallelizing general-purpose programs has once again been placed on the research agenda. In the 1980s and early 1990s, great successes were obtained to extract parallelism from the inner loops of scientific computations. General-purpose programs, however, stayed out-of-reach due to the complexity of their control flow and data dependences.
More recently, thread-level speculation (TLS) has been tauted as the definitive solution for general-purpose programs. TLS again targets inner loops. The program complexity issue is handled by checking and resolving dependences at runtime using complex hardware support. However, results so far have been disappointing and limit studies predict very low potential speedups, in one study just 18%.
In this paper we advocate a completely different approach. We show that signficant amounts of coarse-grain parallelism exists in the outer program loops, even in general-purpose programs. This coarse-grain parallelism can be exploited efficiently on CMPs without additional hardware support.
This paper presents a technique to extract coarse-grain parallelism from the outer program loops. Application of this technique to the MiBench and SPEC CPU2000 benchmarks shows that significant amounts of outer-loop parallelism exist. This leads to a speedup of 5.18 for bzip2 compression and 11.8 for an MPEG2 encoder on a Sun UltraSPARC T1 CMP. The parallelization effort was limited to 10 to 20 person-hours per benchmark while we had no prior knowledge of the programs.

1 Introduction

The landscape of computer architecture research has fundamentally changed with the introduction of chip-multiprocessors (CMPs). Instead of achieving high single-thread performance using complex processor cores, the focus is now on the efficient computation using multiple cores and, more importantly, the difficult task of extracting multiple threads of execution from sequential programs. To understand the complexity and importance of this task, one needs to be aware that the number of cores on a processor is expected to grow exponentially in the future [1], implying that the demand for thread-level parallelism will increase significantly over the coming years. We claim that the outer program loops contain significant amounts of parallelism that is easy to exploit on CMPs without the need for specialized hardware support.

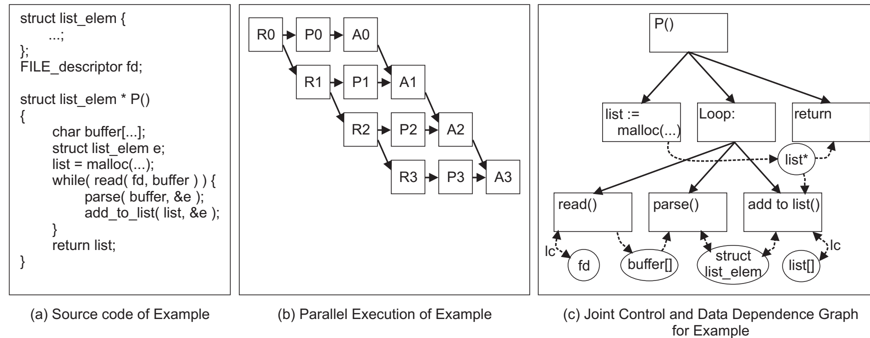


Fig. 1. Motivating example with coarse-grain parallelism.

1.1 Motivating Example

Figure 1(a) shows an example of a parallelizable outer loop. The example concerns a function `P()` that reads records from a globally defined file descriptor `fd` and places each record in a list. First, a list is dynamically allocated. Then, each read line is parsed and its contents are placed in a structure `list_elem`. This structure is then placed in the list. Finally, the address of the list is returned by `P()`. Any of the functions `read()`, `parse()` and `add_to_list()` may in turn contain other loops, but these are not discussed here, nor is the presence of nested loops considered by our techniques when parallelizing the loop in `P()`.

Note that it is not essential for the example that the information is actually read from a file. The data can also be read from another data structure. What is essential here is that (i) each loop iteration operates as an independent pipeline moving data from one stage to the next and (ii) each loop iteration operates on largely distinct data. Indeed, data flows through the functions `read()`, `parse()` and `add_to_table()` as if they were the stages of a pipeline. Furthermore, the data operated on by a single loop iteration is quite unique. E.g., the buffer is filled with data from a file during each iteration, but that data is not used again in a later iteration. The same observation can be made for the `list_elem` structure. There are, however, a few variables that are re-used in each loop iteration which imposes a dependence on the order of the loop iterations. These variables are the file descriptor and the list of structures.

The presence of cross-iteration dependencies implies that the loop is a DO-ACROSS loop [4] rather than a DO-ALL loop with huge amounts of parallelism [12]. Yet, this DO-ACROSS loop still has a fair amount of parallelism as illustrated in Figure 1(b). Each box represents the execution of a piece of code in the loop body, with R indicating the `read()` call, P indicating the `parse()` call and A indicating the call to `add_to_table()`. The number in the box represents the loop iteration number. Dependencies between the parts of the loop body are shown by edges. This graph shows that the parts of code executed in a single loop iteration are dependent on each other. It also shows that code part R_{i+1} has a cross-iteration dependence on part R_i from the previous loop iteration (because of the file descriptor). The same holds for part A due to the

list of structures. Part P has no cross-iteration dependencies as it reads only from the `buffer` variable. There is a cross-iteration output-dependence on the buffer, but such dependencies do not limit parallelism as they can be removed by privatizing the buffer, i.e., each thread operates on a different buffer [13].

The graph of Figure 1(b) shows the dependencies between the code parts in the loop and hence it can be used to transform the sequential code into explicitly parallel code. Indeed, the dependence edges show where synchronization points should be inserted in the code in order to obtain correct parallel code. Each code part can thus be viewed as a critical section that requires access to a particular set of data structures. Furthermore, these critical sections are optimally defined in the sense that they protect the largest amount of code that still allows maximum parallelism between loop iterations.

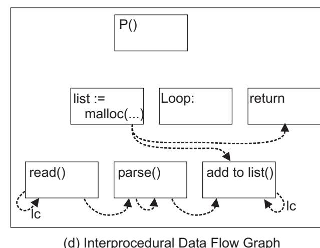


Fig. 1. Motivating example (ctd.).

1.2 Paper Overview

In this paper, we present techniques to detect coarse-grain DO-ACROSS loops in a partially automatic way and to aid the programmer to parallelize these loops. Our approach is described by the following main points:

1. We focus on coarse-grain parallelism located in the outer program loops, instead of the inner loop parallelism detected by other approaches. Parallelism in loops is thereby typically concerned with the general flow of the algorithm. Furthermore, we extract DO-ACROSS parallelism [4], which corresponds closely to pipeline-like operation on a data set. These properties facilitate understanding the code and recognizing the correctness of parallelism.
2. Instead of striving for exact (but conservative) static analysis of programs, we employ dynamic control and data dependence analysis. This dynamic analysis turns out to correctly identify coarse-grain parallelism in practice. Yet, verification is still required.
3. The programmer is presented with a semantically rich description of the parallelizable loops, including function names, code line references, data structure names, etc. Furthermore, the analysis indicates which data structures should remain shared, and which data structures should be privatized.

In the remainder of this paper, we first discuss related work (Section 2). Then, Section 3 provides an overview of our techniques to extract coarse-grain parallelism from sequential programs. Section 4 explains the dynamic control flow and data dependence analysis and Section 5 discusses the extraction of DO-ACROSS parallelism from the dependence information by matching to a template. In Section 6, we measure the presence of coarse-grain parallelism in several benchmarks and measure the speedup achieved on Sun UltraSPARC T1. Finally, Section 7 concludes this paper.

2 Related Work

It is now well understood how to parallelize perfectly nested loops that access arrays (see, e.g. [17]), or even non-perfectly nested loops (e.g. [8]). Furthermore, it is possible to work around procedure boundaries when parallelizing loops (see, e.g. [5]). Research on automatic parallelization has produced several high-performance parallelizing compilers yielding high-performance parallel code [2]. All of this work deals, however, with inner loops accessing array elements. To exploit the potential of upcoming CMPs, it will be necessary to extend the scope of automatic parallelization to a broader class of programs. Recent work of Bridges et al. [3] has looked into this direction, however, still requiring programmers insight.

Thread-level speculation systems (TLS) can avoid the problem of program analysis by speculating the non-existance of dependences [6]. This requires, however, a large amount of hardware support for handling thread contexts, but also extensions to the memory system to detect dependence violations [6,9,15]. In some cases, it is assumed that specialized high-bandwidth interconnections exists between processor cores. These interconnections should operate as queues [11]. Thus, TLS has a high hardware cost which should be considered carefully.

While thread-level speculation seems a promising avenue to pursue, results so far have been disappointing. Steffan and Mowry [15] manually apply TLS optimizations to SPEC92 and SPEC95 benchmarks. Although they speedup speculative regions by a factor of 4 on 4 processors, overall program speedups are in the range of 1.03 to 3.87 with results varying strongly between benchmarks. This is due to a low coverage of speculatively parallelized code. Warg and Stenström [16] find a limit on parallelism of 3.5 when considering modules. Prabhu and Olukotun [14] again apply TLS manually on SPECint codes. They achieve speedups larger than 2 only when they can restructure the program such that traditional loop-level parallelism can be exploited.

When carefully factoring out non-speculative parallelism, one comes to the surprising conclusions that: (i) the potential of TLS is limited to 18% speedup on SPECint benchmarks and (ii) non-speculative thread-level parallelism has much larger potential than TLS, even in SPECint programs [7].

3 Method Overview

3.1 Dynamic Control Flow and Data Dependence Analysis

The first step in parallelization is the analysis of control flow (i.e. detecting loops and the code that can be called from those loops), and the analysis of data dependencies (i.e. what regions of code are data dependent). Two complementary approaches have been developed: static analysis and dynamic analysis [10]. Static analysis is performed on the program's source code. Hence, all possible execution paths can be considered which places static analysis at the sound end of the spectrum. Dynamic analysis is performed on a particular execution of the program, which makes dynamic analysis more precise (i.e. it reaches a higher level

of detail) as it works with real values. This detailed analysis result is, however, only applicable to one particular execution of the program.

While static analysis is generally preferable over dynamic analysis, static alias analysis is hard and must make conservative assumptions, whereby many false data dependencies are reported. These false dependencies obscure the parallelism in a program. Consequently we use dynamic analysis in this work: control flow and data dependencies are measured at the assembly code level during a profiling run of the program. We have observed that the coarse-grain dependencies between large code regions (e.g. functions) is very consistent across program inputs: a function typically uses a data structure or it does not use it. In contrast, fine-grain data dependencies between individual statements are much more sensitive to branch behavior and data values.

It is important to note that any inaccuracies in the analysis and parallelization scheme stem from inaccuracies in the dependence graphs. Therefore, these inaccuracies can be removed by using static analysis instead of dynamic analysis.

3.2 The JCDDG and IDFG: Different Views on Data Dependencies

The results of the control flow and data dependence analysis is presented in the Joint Control and Data Dependence Graph or JCDDG (Figure 1(c)) and in the Interprocedural Data Flow Graph or IDFG (Figure 1(d)).

The JCDDG vaguely resembles a syntax tree as constructed by a compiler. A major difference is that nodes in the JCDDG are code regions, containing multiple instructions, while syntax trees represent all operations separately. Code regions are represented as boxes in the JCDDG and may be function bodies, loops or basic blocks. The JCDDG also contains the data structures, represented by ovals. The data flow is tracked by showing which code regions produce and consume data in or from each data structure. Note that data consumer edges may be labeled “lc” when the data consumption implies a loop carried dependence, i.e. the data was produced in a different loop iteration than when it was consumed.

The IDFG presents a view on data dependencies that is complementary to the JCDDG. Indeed, the JCDDG is data structure-centric as it shows the data structure where a code region produces or consumes data. Thus the JCDDG may be ambiguous about the producing code region. E.g., in Figure 1(c), the JCDDG suggests that the `parse()` function may be consuming data from the `add_to_list()` function, while in practice this is not true. The IDFG directly links the producing and consuming code regions without indicating the data structures that mediate this dependence (Figure 1(d)). Thus, it presents a complementary view. Both graph representations are necessary for this work.

3.3 The Role of the Programmer

The techniques presented in this paper already perform much work such that the role of the programmer is limited. While we can automatically propose a parallelization of a program, this parallelization may be incorrect due to the dynamic control flow and data dependence analysis. The first task of the programmer is

thus to verify that the proposed parallelization is indeed correct. We argue that this task is fairly straightforward due to the nature of the coarse-grain nature of the parallelism (it becomes obvious to understand why the parallelism exists) and due to the presentation of the analysis results (which clearly points out the code and data structures that are involved). Note that this task may disappear when static alias analysis becomes sufficiently precise for our purposes.

A second task of the programmer is to notify our tools that certain dependencies need not be enforced in practice. E.g., verbose output of a program need not remain in the original sequential program order. Also, the order of system calls may be interchanged in some situations, e.g. when operating on totally distinct files. Compilers cannot make such decisions in general as they cannot judge the external requirements on the program. The programmer can indicate such situations to our tools by removing dependencies from the JCDDG and IDFG and use the tools interactively to try to extract even more parallelism.

The third task of the programmer is to parallelize the program. To facilitate this task, it is important to provide clear and user-friendly output of the analysis. We strive for clarity by reporting the parallel loops in a program in a concise manner (the code regions and data structures involved), but also on how to handle data structures in order to implement this parallelism (e.g. when data structures should be privatized). Descriptive names are given to data structures such that it is easy to track their origin.

4 Dynamic Control Flow and Data Dependence Analysis

Dynamic program analysis is performed on a particular execution of a program. Hereto, the program is run on one or more representative inputs and its execution is followed at the assembly level. The effects of the individual instructions are analyzed and used to incrementally build up the Joint Control and Data Dependence Graph (JCDDG) and Interprocedural Data Flow Graph (IDFG).

Joint Control and Data Dependence Graph The JCDDG presents a data structure-centric view (Figure 1(c)) as it shows the data structures that code regions produce and consume. It contains coarse-grain code regions rather than individual assembly instructions in order to limit the size of the JCDDG. These code regions are determined by static analysis of the compiled program. There are three types of code regions in the JCDDG: function bodies, loop bodies and snippets. Snippets are code fragments that are executed in between function calls. They serve to order the execution of loads and stores in a function with respect to loads and stores that are executed in loops or in callee functions.

Control flow is identified by organizing code regions in a tree. The children of a node in the tree correspond to the sub-regions of a code region, i.e. loops and snippets that are contained in functions, and functions that are called from within a loop or function. The children of a node are implicitly ordered, i.e. control flow within a code region is sequential. If a code region contains multiple call sites to a function, then multiple edges connect the code region to the function, with all edges sorted in sequential code order.

The actual control flow in the program corresponds to tracing a path through the tree, where control flow can move down in the tree (enter a sub-region of the current region) or it can move up in the tree (exit a region).

Data structures are created in one of three different ways: (i) they are statically initialized when the program is loaded, (ii) they are dynamically allocated by calls to one of the `malloc()` family of functions or (iii) they are created on the stack. Each data structure in the program is identified and is given a descriptive name that helps the programmer to locate it in the source code.

Data flow edges (dotted lines) indicate which code regions access a data structure. The data flow edges point from a code region to a data structure when the code region writes (any field of) the data structure. The code region is said to be a *producer* of the data (structure). The dataflow edge points from the data structure to a code region if that code region reads (any field of) the data structure. This code region is called a *consumer* of the data (structure).

The Interprocedural Data Flow Graph The IDFG gives a code region-centric view (Figure 1(d)): what code region is actually producing the data that another code region consumes. As such, the IDFG contains only nodes to represent code regions and no data structure nodes. Also, control flow edges are not represented in the IDFG as they can be read from the JCDDG. Edges in the IDFG link two code regions and point from the producing code region to the consuming code region. The IDFG edges correspond to conventional data dependence edges.

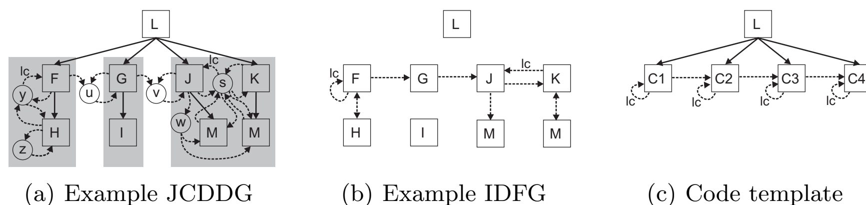
Control Flow Analysis Dynamic control flow analysis operates by detecting code region entry and exit. This is maintained in a code region stack. When a new code region is entered, the code region is pushed on the stack. The code region is popped off the stack when it is exited. The code region stack identifies the code region node in the JCDDG or IDFG where the current point of control it is.

Data Dependence Analysis The goal of data dependence analysis is to identify the data structures that a code region produces or consumes and the code regions that are actually dependent on each other. This information is collected using two data structures: the last producer table and the memory dependence matrix.

The *last producer table* tracks the code region that last produced a value at a particular memory address. We maintain the table as a set of regions of memory, where all bytes in a region have the same producer. Regions are merged and split as needed to track the correct producer.

A *memory dependence matrix* tracks the producing and consuming code regions for a particular data structure. The entry at row f and column g in the dependence matrix records the number of times that code region g consumed a value produced by code region f in this data structure.

It is clear to see that the memory dependence matrix for a data structure describes all incoming and outgoing edges for this data structure's node in the JCDDG. The data dependence edges in the IDFG can be constructed by combining the memory dependence matrices for all data structures.



(a) Example JCDDG (b) Example IDFG (c) Code template

Fig. 3. An example joint control and data dependence graph (JCDDG), a corresponding interprocedural data flow graph (IDFG) and the code template searched for during parallelization.

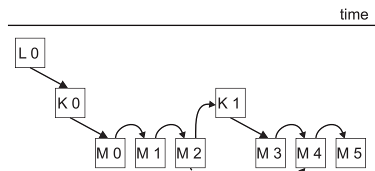


Fig. 2. Execution of 3 nested loops. Edges indicate control flowing from loop L over loop K to several iterations of loop M. When loop M finishes, a new iteration of loop K starts. The numbers in the boxes indicate the loop iteration counts.

Loop-Carried Dependencies The presence of loop-level parallelism depends strongly on the presence of loop-carried dependencies, i.e. a code region in one loop iteration is data dependent on a code region in a previous loop iteration. Loop-carried dependencies can be detected by using *annotated paths.*

The annotated path of a code region annotates the loops with their loop iteration counts. We update the last producer with the annotated path, but the memory dependence matrix uses non-annotated paths. Moreover, each cell in the memory dependence matrix now has two fields: the dependence count as before and a bit mask indicating the loops for which loop-carried dependencies occur.

An example is depicted in Figure 2 where the 3 loop bodies are executed iteratively and the loop counters increase continuously as time progresses from left to right. The path of code region M is $L.K.M$. In this path, loop L is the outermost loop and loop M is the innermost loop. The loops are executing iterations i, j and k, respectively, yielding the annotated path $L_i.K_j.M_k$.

A data dependence is detected from loop M with annotated path $L_0.K_0.M_2$ to loop M with annotated path $L_0.K_1.M_4$. The memory dependence matrix records the self-dependence from $L.K.M$ upon itself as well as the fact that loop K is the outermost loop in this dependence that has a distinct loop count.

When analyzing the parallelizability of loop K we note that the dependence in question is loop carried. When analyzing the parallelizability of loop L, then we treat the same dependence as not loop-carried, as the dependence manifests itself inside the same iteration of loop L. When analyzing the parallelizability of loop M, then the dependence points back to a piece of code that was executed *before* the loop we attempt to parallelize. Thus, the dependence need not be considered in the parallelization of loop M.

5 Parallelization

Parallelism is identified through analysis of the profiling information. The analysis is performed using two complementary graph representations: the joint control and data dependence graph or JCDDG (Figure 3(a)) and the interprocedural data flow graph or IDFG (Figure 3(b)). Each graph identifies a different key aspect of program behavior and is used in a different step of the parallelization algorithm.

The format of parallelism that we are looking for is depicted in (Figure 3(c)): A loop L contains multiple code regions that are data dependent on each other. A code region may have loop-carried dependencies only on itself which restricts parallelism to the DO-ACROSS style.

Our parallelization algorithm treats the loop structure of Figure 3(c) as a template against which loops are matched. Loops that match are easily parallelized in DO-ACROSS style. Other loops are not considered further.

Loop-carried dependencies are annotated with the label "lc" in the JCDDG and IDFG. These labels depend on the loop that is analyzed for parallelism. The loop-carried dependence labels are computed by inspecting the bit-mask stored in the memory dependence matrices (see Section 4). The dependence is labeled as loop-carried only if the dependence points from one loop iteration to a later iteration that is executed before the loop is exited.

To match a loop against the template, code regions are merged into larger code *clusters* such that each remaining cluster corresponds to one of the code regions C_i of the template.

The initial set of clusters is determined by control flow information. When matching loop L to the template, every code region that is called directly from the loop L is the seed for an initial cluster. All code regions that are called (in-)directly from these seeds are added to the appropriate clusters.

These initial clusters are further merged based on data dependencies. Backward data dependencies are taken care of by performing topological sorting of the clusters and by detecting connected components. We perform these algorithms in a single pass to simultaneously sort the clusters by data dependencies and merge mutually dependent clusters. As a result, loop-carried dependencies now depart from and arrive at the same cluster, as depicted in Figure 3(c).

When parallelizing a loop, shared data structures must be handled appropriately. In practice, some data structures may remain shared, others must be privatized [13], i.e. a separate copy is created for every parallel executing code region that access the data structure. Every code cluster is a critical region that has exclusive write access to a particular set of data structures.

The rule to determine whether a data structure must be privatized is simple: it is necessary *only* when a parallel execution scheme allows for the simultaneous execution of a code cluster that modifies the data structure and a code cluster that reads or writes the same data structure. Note that the code clusters are constructed in such a way that privatizing the appropriate data structures delivers correct parallel code.

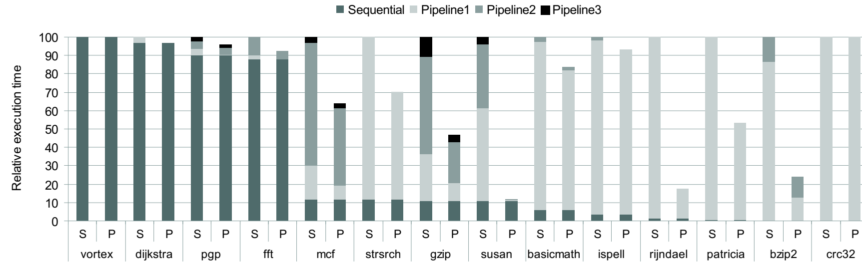


Fig. 4. Relative execution time of detected pipelines in sequential execution (S) and in parallel execution (P).

6 Evaluation

We evaluate the techniques proposed in this paper using benchmarks from the MiBench suite and the SPEC2000 integer suite. We use the small input sets (MiBench) or the training input sets (SPEC) during profiling. We profile the programs with a modified version of Dynamic SimpleScalar. Parallelized program speedups are measured using large or reference input sets on a 32-thread Sun UltraSPARC T1. All the benchmarks are compiled using the `gcc` compiler version 4.1.0 with optimization level `-O3`.

Execution Time Reduction During the analysis our tools give an estimation of the possible reduction in execution time. It gives an idea of the potential gains of the suggested parallelization.

For DO-ALL parallellism we assume that the execution time can be reduced to the time of one iteration: $t_{par} = t_{1iter}$. For DO-ACROSS loops $t_{par} = t_{1iter} + (n-1)t_{lcmax}$, where n is the number of iterations and t_{lcmax} the execution time of the most time-consuming cluster with loop-carried dependencies.

Figure 4 shows for each parallelizable outer loop its original execution time (S: sequential version) and its estimated reduced execution time (P: parallel version). We show a representative selection of the benchmarks.

In some benchmarks, such as `vortex`, it is not possible to find useful DO-ACROSS parallelism. A frequent cause is that the analysis leads to only one dominating pipeline stage that is loop carried. This reduces the speedup to zero, enforcing a sequential execution.

Other benchmarks contain pipelines that represent only a minor fraction of the execution time ($< 10\%$). Consequently the potential speedup of such programs is limited. This occurs in the benchmarks `dijkstra`, `pgp` and `fft`.

Some benchmarks contain more parallelism but still have a sequential part. The reduction in execution time can vary widely. In case of `mcf`, `stringsearch` and `gzip` the reduction is 30% or more. While for `basicmath` and `ispell` the reduction is limited due to an unbalanced pipeline. The pipelines found in `susan` are DO-ALL loops, hence the large speedup.

Finally, in a number of benchmarks, the sequential part is almost non-existent. On average these programs tend to give a fair upto decent reduction in execution time. This property can be found in `rijndael`, `patricia` and `bzip2`.

Scalability Speedup We study the scalability of `bzip2` and an `mpeg2-encoder` to a high number of threads. Synchronization is implemented using POSIX primitives. The execution time of the sequential and parallel versions of the benchmark are run on real hardware: a Sun UltraSPARC T1.

The analysis reveals two parallelizable outer loops for bzip2 (Figure 4). The first loop is a 4-stage pipeline in the compression part of the program and takes about 86% of the execution time. The second loop is a 2-stage pipeline in the decompression part taking about 14% of total execution time. The MPEG2 encoder is taken from the mediabench suite. The detected coarse-grain parallellism in this case comes down to the fact that groups of pictures (GOPs) can be encoded as separate entities. We used the composite parallellization scheme where one thread handles the preparation of the GOP, one or more threads process the GOP and a last thread writes the encoded GOP.

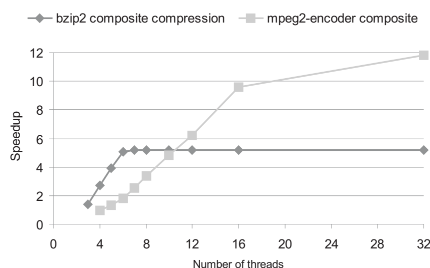


Fig. 5. Scalability of speedup for *bzip2* and *mpeg2-encoder*.

Figure 5 shows for both the `mpeg2-encoder` and `bzip2` the speedup of the composite parallellization in function of the number of threads. For `bzip2` we see that the program scales up to 7 threads (speedup of 5.18), thereafter no further significant increase in speedup is measured. The `mpeg2-encoder` on the other hand scales up to 32 threads with a speedup of 11.82.

7 Conclusion and future work

This paper presents a framework for extracting coarse-grain parallelism from sequential programs. We use dynamic analysis techniques to obtain precise control and data dependence information. We consider only dependencies between large regions of code in order to focus on coarse-grain parallelism.

Coarse-grain parallelism is detected using the control and data dependence information. We use two graph representations of the dependence information: the Joint Control and Data Dependence Graph (JCDDG) and the Interprocedural Data Flow Graph (IDFG). Coarse-grain DO-ACROSS parallel loops are detected by matching the JCDDG and IDFG to a preset template. We currently detect only coarse-grain DO-ACROSS parallel loops. Extracting other types of parallelism from the dependence information will be investigated in future work.

We applied our framework to the Mibench benchmarks, several SPECint2000 benchmarks and an MPEG-2 encoder from Mediabench. The proposed methods

successfully identifies coarse-grain parallelism in most of the benchmarks. Implementations of this parallelism result in speedups of 5.18 for bzip2 compression and 11.8 for the MPEG2-encoder on a Sun UltraSPARC T1.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. Sean Rul is supported by a grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Hans Vandierendonck is a post-doctoral research fellow with the Fund for Scientific Research-Flanders (FWO). This research is also funded by Ghent University and HiPEAC.

References

1. K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Dept., University of California, Berkeley, December 18 2006.
2. W. Blume et al. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Par and Distr Tech*, 2(3):37–47, 1994.
3. M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for multi-core. In *MICRO '07*, 2007.
4. R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *ICPP*, 1986.
5. M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Supercomputing*, 1991.
6. L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, 1998.
7. A. Kejariwal, X. Tian, W. Li, et al. On the performance potential of different types of speculative thread-level parallelism. In *ICS '06*, 2006.
8. A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97*, 1997.
9. W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP '06*, 2006.
10. N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, 2004.
11. G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO-38*, 2005.
12. D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans on Comp*, 29(9):763–776, Sept. 1980.
13. D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.
14. M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP '05*, 2005.
15. J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *HPCA '98*, 1998.
16. F. Warg and P. Stenstrom. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *PACT*, Sept. 2001.
17. M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.