Function Level Parallelism Driven by Data Dependencies

Sean Rul Hans Vandierendonck Koen De Bosschere

Department of Electronics and Information Systems (ELIS),
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
E-mail: {srul, hvdieren, kdbosche}@elis.ugent.be

Abstract

With the rise of Chip multiprocessors (CMPs), the amount of parallel computing power will increase significantly in the near future. However, most programs are sequential in nature and have not been explicitly parallelized, so they cannot exploit these parallel resources. Automatic parallelization of sequential, non-regular codes is very hard, as illustrated by the lack of solutions after more than 30 years of research on the topic. The question remains if there is parallelism in sequential programs that can be detected automatically and if so, how much parallelism there is.

In this paper, we propose a framework for extracting potential parallelism from programs. Applying this framework to sequential programs can teach us how much parallelism is present in a program, but also tells us what the most appropriate parallel construct for a program is, e.g. a pipeline, master/slave work distribution, etc.

Our framework is profile-based, implying that it is not safe. It builds two new graph representations of the profile-data: the interprocedural data flow graph and the data sharing graph. This graphs show the data-flow between functions and the data structures facilitating this data-flow, respectively.

We apply our framework on the SPECcpu2000 bzip2 benchmark, achieving a speedup of 3.74 of the compression part and a global speedup of 2.45 on a quad processor system.

1 Introduction

Today we are at the dawn of a new era in computer architecture. While during the past decade the computer scene was mainly dominated by complex uniprocessors with deep pipelines, we now see the rise of CMPs containing several slimmer cores. This transition was fueled by several incentives. Firstly, the instruction level parallelism (ILP) has been exploited to its full extent, such that extracting more ILP becomes overly complex. Secondly, power dissipation kept increasing alarmingly, caused by implementing a single large core with long wires and clocked at high frequencies.

While this transition was necessary, we are also confronted with new issues. A big question is how we can exploit all this parallel processing power in the new processor generation? Although there is a group of programs, such as scientific and media applications, that inherently have a lot of easily exploitable thread-level parallelism (TLP), another majority of programs are inherently sequential. These programs, exemplified by the SPECcpu integer benchmark suite, have remained out of scope of research on parallelization up to the last few years.

Automatically parallelizing inherently sequential programs is hard due to the difficulty of correct static analysis of both control flow and data flow. To extract large amounts of thread-level parallelism, it is necessary to look past these limiting control flow and data flow restrictions. In this paper, we develop a framework for extracting thread-level parallelism from sequential programs that assumes perfect knowledge of these dependencies. As such, it is able to discover large amounts of TLP.

Our goal is to discover non-speculative parallelism in the first place, without being restricted by the unpredictability of control flow and data flow. Hereto, we measure the control flow and data flow exhibited during a particular run of the program. Analysis of this control flow shows opportunities for parallelization. However, by measuring dependencies during one or more particular runs of a program, the parallelism extracted by our framework may be unsafe, i.e., particular dependencies may arise depending on the input data set of the program. This situation can be handled using, e.g. thread-level speculation (TLS) systems [9, 14] that allow dependencies to be violated, but detects and corrects them with the aid of hardware support. It is also possible to use our framework as an analysis tool for a parallel programmer, who can use it to detect parallelism, but still needs to validate their correctness.

Our analysis focusses on memory dependencies, since register dependencies can be predicted [15] or precom-

puted [1]. We form two graph representations that form an abstraction of the profiled dependencies. These will, together with the call graph, help in detecting the chunks of code that can be parallelized.

The field of applications of our framework is very broad. Our main objective is to utilize the parallel processing power in chip multiprocessors (CMPs) for sequential programs. The same request for thread-level parallelism exists when programming the Cell processor [2]. A Cell processor contains a control processor and eight synergistic processing elements (SPEs) for performing streaming operations. Each SPE has its own local storage, so the additional problem arises of how to partition the data in a program across the SPEs. Here too, we believe our framework can help: when identifying threads, it also identifies the data structures private to a thread and shared by threads. Multiprocessor embedded systems also pose the problem of partitioning code and data across multiple cores, but here power, communication and cost constraints become very important. IMEC has developed the SPRINT [13] tool to partition code and data for embedded systems taking these constraints into account.

This paper is organized as follows: in Section 2 we describe our approach to find parallelism in a sequential program, the next section gives some results for a real life example as a proof of concept. A comparison with related work is made in Section 4. In Section 5 we summarize our contributions and look ahead on future work.

2 Method

As mentioned before, memory dependencies impede parallelism. Therefore we want to locate these dependencies in the program. Since an exact analysis is very difficult due to the required alias-analysis, we choose for a profile based approach. While a profile based technique is less general than an exact parallelization, since the information may be input dependent, it can give valuable information for parallelization and be used as a hint to the programmer.

Another issue that needs taking care of is at which level the memory dependencies are measured. This choice mainly determines the granularity of our parallelism. Since the main application field targets CMPs or multiprocessors, the granularity of the parallelism should be large enough in order to justify the use of threads, which comes with a certain overhead. We attempt to parallelize large chunks of code, with an order of magnitude of at least millions of instructions. Moreover the chunks may not be data dependent on one another. Therefore an initial choice is to look at the function level, which keeps life simple. For our profiling technique we decided to measure the memory dependencies between different functions. This approach gives a sufficiently detailed overview of the program.

Profiling distinguishes all functions, but also all data structures used by the program, such that it is possible to detect exactly what data structures communicate data from one function to another and what data structures are private to a function. This information is essential to parallelize functions in the program. To facilitate detecting parallelism between functions, based on data dependencies, we introduce two new graph representations to visualize these dependencies. These graphs identify parallelizable code, as well as data structures that need synchronization. The main focus of this paper is on the construction of the new graph representations and the ability of detecting parallel code. As this work is in its infancy, some steps in this process are not yet completely formalized.

2.1 Analysis

To build up a call graph, we record all the function calls and returns. These caller/callee relations form a first restriction on program parallelism, since the caller passes arguments and the callee may give a return value back. We keep track of how many times the function is called, the number of different functions it calls and the fraction of execution time it consumes. The latter one is taken into account for balancing the work between different threads. In Figure 1 we give an example of a call graph.

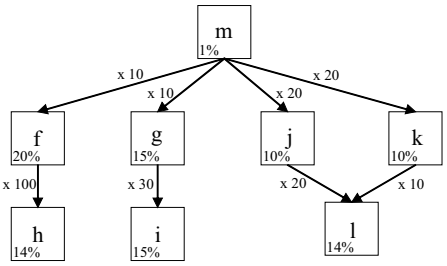


Figure 1. Example of call graph

Data dependencies are measured during a profiling phase to discover which function is reading what data from which functions. We record this information in a matrix-like data structure, one for each memory address. The matrix basically records data flow: cell (i, j) of the matrix shows how many times function i read a value that was produced by function j. Table 1 shows an example of this data structure for address `0x1000`. Each function that accesses this address has its own row. The column *Producer* shows the functions accessing this address. The column *Times produced* shows for each function how many times it has written a value to this address. The columns *Consumer* show for each row how many times it has read a value, produced by a function indicated in the subcolumn. All this information allows us to reconstruct the different producer/consumer relations between functions. To obtain this information all

load and store instructions update the data structure representing their involved memory address. A function that issues a store instruction increments the value in the column *Times produced* on the row representing the current function. Furthermore the variable *Current Producer* associated with this memory address, is altered to the current function. When a function performs a load instruction, the column in *Consumer* is selected by *Current Producer* and the value in the row of the current function is incremented. Figure 2 shows a small example of memory operations in a trace, which result in the values stored in Table 1.

Address `0x1000`				
Producer	Times produced	Consumer		
		`F1`	`F2`	`F3`
`F1`	1			
`F2`	2	1	2	
`F3`			2	

Table 1. Matrix representation of memory behavior of exemplary trace

For 32-bit applications addresses are aligned on 4 byte, so a byte-operation to address `0x1000FFF2` and a word-operation to address `0x1000FFF0` are both stored in a data structure labeled with address `0x1000FF0`. A quad-operation is regarded as two 4-byte operations.

```
Function   Operation   Current producer
   F1        store           ??
   F2        load            F1
   F2        store           F1
   F2        load            F2
   F3        load            F2
   F2        store           F2
   F2        load            F2
   F3        load            F2
```

Figure 2. Exemplary trace of memory operations to address `0x1000`

When there is a load operation for which there is no previous producer, we assume that it is constant (an extra *Consumer*-subcolumn in the matrix). This situation occurs for example when a program reads data from a constant data section, or when data is produced by a system call.

Interprocedural data Flow graph For each memory address that is touched during execution we build a matrix similar to the one presented in Table 1. With this information we can determine which function reads data from other functions. This can be represented in a directed graph where the nodes are the functions and the edges show the data streams (Figure 3). We use the notation $f \xrightarrow{x} g$ to indicate that a function g consumes x bytes of data produced by f. The next step is to search for strongly interconnected functions. That is, functions that share a large amount of data. This allows us to cluster functions sharing data structures (strongly connected functions). Clustering the functions divides the data streams in two categories: *intercluster data streams* and *intracluster data streams*. In Figure 3 the function clusters are shown with grey rectangles. To indicate that a function f belongs to cluster N we use the notation f_N. Consequently an intercluster data stream is noted as $f_N \rightarrow g_M$, while an intracluster data stream has the form $f_N \rightarrow g_N$.

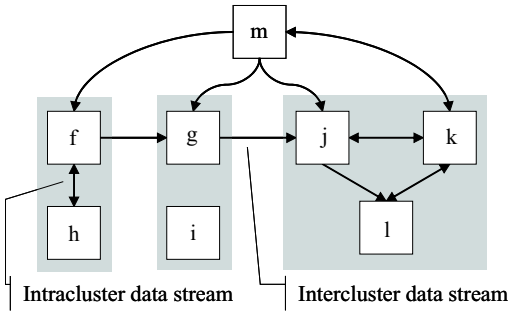


Figure 3. Example of interprocedural data flow graph

One of the aims of this clustering is to limit the intercluster data streams, in order to facilitate parallelization. This does not necessarily means that intercluster data streams should be non-existent or small in capacity. Such a requirement would be too restrictive, especially when we are dealing with sequential programs. A more reasonable goal is to impede the amount of bidirectional intercluster data streams. If data streams between two clusters of functions go in both directions, it is hard to parallelize them further on. Unidirectional intercluster data streams form less of a problem, since these are suitable for certain parallel constructs, as will be explained in Section 2.2. Bidirectional intracluster data streams cause less trouble, since the functions in that cluster are executed sequentially.

The clustering of functions is also guided by the call graph, since this graph imposes a certain hierarchy between different functions. For example in the call graph of Figure 1 we see that function h is only called by function f, so it may be a good choice to put these two functions in the same cluster. On the other hand if the fraction of execution time of these two functions is too large, they are not put together in consideration of finding a balanced solution. So the second role of the call graph is to find clusters that are

balanced in execution time.

Data sharing graph While the previous representation showed the existing data streams between functions, it does not show *how* the data is shared. In other words, in order to show which function uses what, we have to make another abstraction of the profiled information. The idea is to show both the data dependencies as well as the involved data structures. The resulting graph has two kinds of nodes: function nodes and data nodes. An edge from a function node to a data node indicates the number of write accesses made by the function. We use the notation $f \xrightarrow{w} ds_1$. The opposite, an edge from a data node to a function node, is seen as the number of read accesses from that function to the data structure, which is noted as $ds_1 \xrightarrow{r} f$.

Memory addresses that are used by the same functions with a similar read/write behavior are saved in the same data structure. This reduces the number of data nodes compared to the profiling phase, where the behavior of each memory address was stored in a different structure. It also forms a good heuristic for reconstructing the original data structures used in the program source.

If we examine one function and all the data structures it accesses, we can distinguish 4 types of data usage, which are represented in Figure 4. If a function reads data, that is written by another function, it is a *consumer*. On the other hand, if it writes the data, which is read by another function, it is a *producer*. Data that is both read and written by the same function, is seen as *private consumption*. Sometimes data is read, without a traceable origin, which is considered as *constant consumption*. If we do this classification for each function, we get a graph, the *data sharing graph* (Figure 5), which shows how the data is shared between different functions. Rectangular nodes represent functions, while elliptic nodes are data structures. Note that there can be function nodes that are not connected with any other functions (for example function i). These functions only communicate with other functions by passing arguments via registers and the stack.

If we map the clustering of functions, obtained with the interprocedural data flow graph, we detect which data structures become private within a cluster and which ones are shared between different clusters, respectively called *cluster private* and *cluster shared*. This will prove to be useful when the actual parallelization is performed, as described in section 2.3.

2.2 Parallel constructs

There are numerous techniques to make a program work in parallel. Each of them having different synchronization requirements and useful under different circumstances. In the software world there are three paradigms that are commonly used [5].

The first is the *Master-Slave*. In this paradigm the main master thread launches several slave threads and allocates to each slave a portion of the work to be done. Once the work of all the slaves is done, a new batch of them can be started. Synchronization can be achieved by using a *barrier*.

Another paradigm is the *Workpile*. In this case each thread fetches a portion of work from some form of a queue, called the *workpile*, until there are no more entries in the workpile. The worker threads can also push extra work on the workpile.

A third paradigm is called the *Pipeline* paradigm. It is based on the simple producer/consumer relation: one pipeline stage produces data for the next stage, which digests this data and on its turn passes it on to the next stage.

For our purpose we need to detect parallel constructs based on the information we can extract from our interprocedural data flow graph and data sharing graph. In this paragraph we look at the last paradigm, the pipeline, and show that this construct can easily be detected in our interprocedural data flow graph and this in a formalized fashion.

The first requirement is that between several clusters of functions the intercluster data streams are unidirectional. Also there are no dependencies from f_m to g_n with $m < n$. This last requirement is interpreted as a function from cluster m that in sequential executions comes before all the functions of cluster n, is not dependent from a previous execution of a function from cluster n. The shared data between the cluster represents the pipeline registers. Depending on the partitioning of work, we can consider two cases. The first is the *heterogeneous pipeline* in which each stage of the pipeline handles a different function clusters. The second is called the *homogeneous pipeline* where each stage executes the same code. This is similar to the master-slave paradigm, but the synchronization is different.

2.3 Parallelization

If the previous analysis succeeded in finding parallel constructs, the next step consists of the actual parallelization. We define the pieces of code that are involved in the parallelization. This can involve the whole program or sometimes only certain parts, depending on the findings of the previous analysis. Since we looked at data dependencies between functions, the marked parts are a single function or a cluster of functions.

The memory addresses are also mapped on the corresponding data structures from the original program. For static data structures this mapping is trivial. For mapping dynamic data structures we have to record the addresses during profiling when these data structures are allocated.

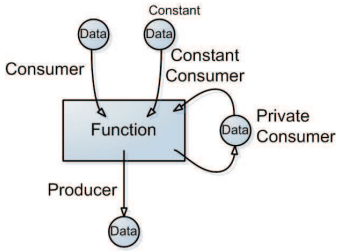


Figure 4. Classification of data dependencies.

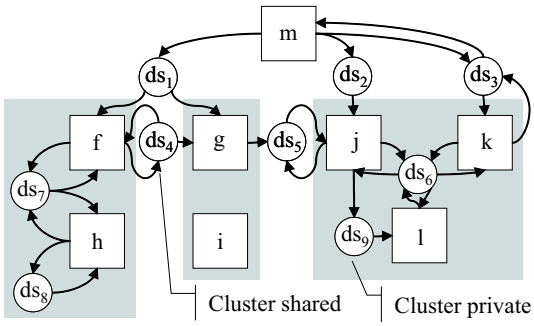


Figure 5. Example of data sharing graph.

Furthermore we detect the data structures that can be private for the different threads. Given the fact that sharing involves communication, which on its turn entails sequentiality, we want to minimize the amount of shared data. The information of which data is shared and which can be made private, can be obtained from the data sharing graph described in Section 2.1. It is clear that cluster private data of cluster N are placed in the thread that executes the functions of cluster N and need not be visible to other threads.

The necessary initialization and startup code for the threads is generated in order to preserve a correct execution of the program. Also, some code is required to complete the work after the threads are finished. New data structures are introduced to allow passing on shared data between different threads. Again the data sharing graph helps in this task. When there exists only one version of a shared data structure, this data has to be locked when it has to be altered. This mutual exclusion will prevent data races. Sometimes, a shared data structures is read and written multiple times before being passed on to the next pipeline stage. In this case, it is advantageous to create multiple instances of the shared data structure, so the data itself needs no locking. In case of a pipeline construct, the pipeline registers (the data that is shared between two pipeline stages) can be duplicated. Each pipeline stage works on its own pipeline register and passes it on to the next stage when ready. To enforce the correct execution the necessary synchronization needs to be added.

3 Results

3.1 Experimental setup

We profile the program with a modified version of Dynamic SimpleScalar [3]. The profiling code was incorporated in this simulator. The program we used as a test case was *bzip2* from the SPEC2000 benchmark suite. For our experiments during profiling, we used the reference input *program*. The parallelized version of the program was tested on a multiprocessor platform with four Itanium processors.

3.2 Compression part

Analysis In a first effort to parallelize *bzip2*, we let the profiler concentrate on the compression part of the program. In Figure 6 the part of the call graph that is responsible for more than 99% of the execution time during compression is depicted. To prevent the graph from being overloaded, we have left out the library functions that were called inside these functions. This call graph already gives a big hint in how we could cluster our functions, nevertheless without more information we can't know for sure that there are no intercluster data streams that restrain possible parallelism.

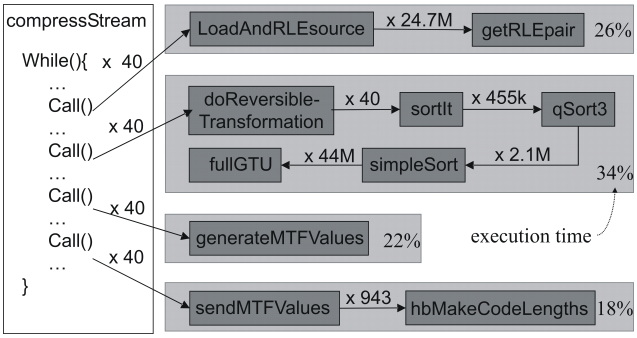


Figure 6. Partial call graph of compress procedure in *bzip2*

This is where the interprocedural data flow graph comes in. Figure 7 shows this information. Again in order to keep the overview, we discarded the library functions and their data streams. Based on the call graph and the interprocedural data flow graph of *bzip2*, we can safely identify four function clusters. The first cluster is dedicated to applying run length encoding on the input. The second block performs a reversible transformation. The third block is a move-to-front encoding. The last block is responsible for the final mapping and output. Based on this clustering we

see that the intercluster data streams are unidirectional (Figure 7), which provides opportunities for parallelization using the pipeline paradigm. Furthermore, the balancing of execution time between the different clusters (Figure 6) is acceptable.

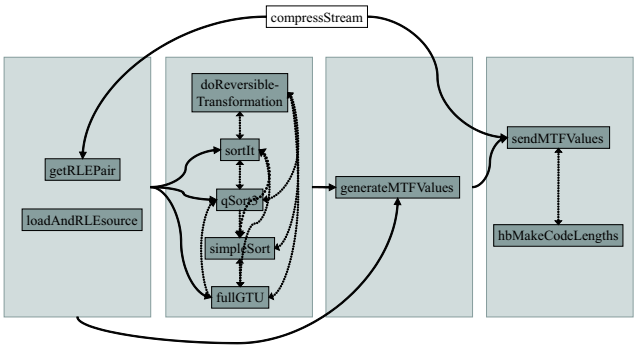


Figure 7. Simplified interprocedural data flow graph of compression procedure

A first parallelized version of the compression, *heterogeneous*, is a pipeline where each function cluster is assigned to a different thread (Figure 8(a)). Consecutive pipeline stages are connected with a synchronized pipeline register. A second parallel version, *homogeneous*, is depicted in Figure 8(b), in this case each thread runs the same code. This requires that there are no dependencies between two successive runs of the second and third cluster (dotted arrows). After closer code inspection it was verified that these dependencies were void: data structures that were shared between the two stages, were initialized (reset to zero) before use in the second cluster. This second approach has two benefits over the first: firstly the number of threads is not limited to the number of clusters and secondly it requires less synchronization (only for reading input and writing output).

Performance To compare the performance of our parallel versions of the compression, we compare the execution time of the original sequential version to the execution time of the code that is now parallelized. The results for the compression are shown in the left of Figure 9. As was expected, the homogeneous version is faster (speedup 3.60) than the heterogeneous (speedup 2.64), due to less synchronization overhead. We can also look at the speedup of the homogeneous version in function of the number of used threads (light bars in Figure 10). When we use only one thread, there is a slowdown compared to the base version, caused by the overhead of thread related code. If the number of threads is larger then the number of processor cores we first see a small increase in speedup (6 threads), but afterwards the speedup decreases (due to insufficient resources).

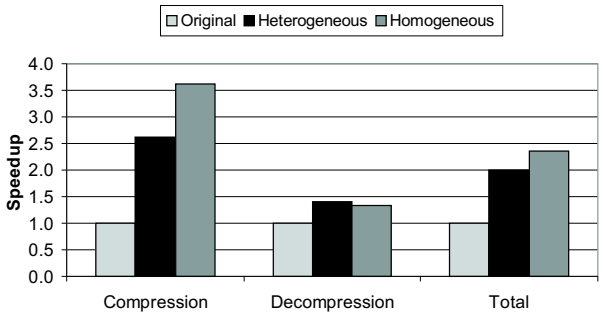


Figure 9. Speedup results of parallelized *bzip2* using 4 threads for compression and 2 for decompression

3.3 Decompression part

Analysis After the fruitful parallelization of the compression part, we also take a look at the decompression part. The analysis is completely analogous to the former. However, in this case the code is more restricted by memory dependencies for reading the input file. Therefore it was only possible to split the decompression procedure in two parts. Once again we make a homogeneous and a heterogeneous version.

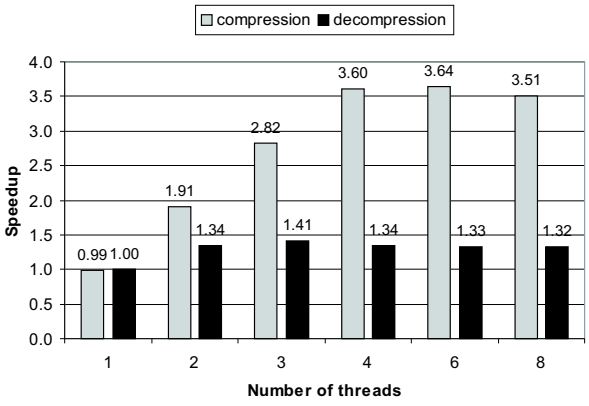
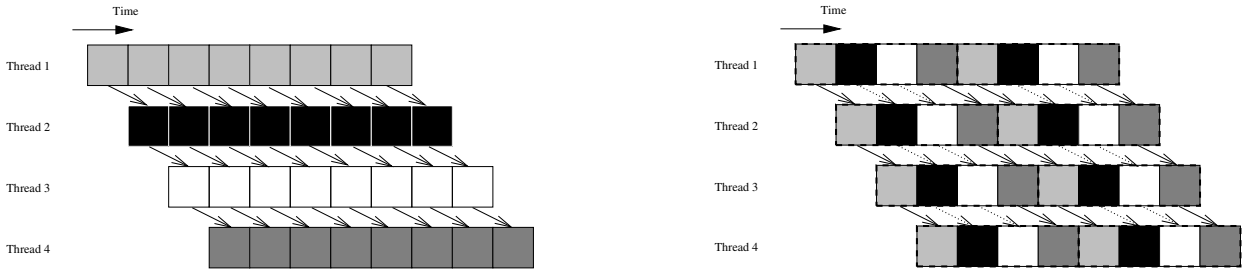


Figure 10. Speedup results for homogeneous parallelized version in function of used threads

Performance The speedup of the parallel version is illustrated in the middle of Figure 9. In this case the speedup is only about 1.41, partly due to less balanced threads. The



(a) Heterogeneous pipeline

(b) Homogeneous pipeline

Figure 8. Parallel versions of compression. Each color represents a cluster of functions, the arrows indicate synchronization.

dark bars in Figure 10 show the speedup of the homogeneous version in function of the number of used threads. The results show a similar behavior as in the case of compression.

3.4 Entire program

Finally we look at the results when we put both our parallel version for compression and decompression together. The speedup of the total execution time is represented on the right in Figure 9. The heterogeneous version has a speedup of 1.99, while the homogeneous version can run 2.45 times faster than the original version. Similar speedups are achieved when using other input files and different options such as the blocksize of the compression.

4 Related work

When it comes to parallelization a lot of research went into loop transformations. Wolf and Lam showed how parallelism in perfectly nested loops could be extracted and automatically transformed [16]. In case the loops are not perfectly nested other techniques can be applied that allow transformations with a minimum of synchronization [6]. However, most of this work is focused on numerical programs and works on the level of loop iterations, while our work is more concerned with sequential programs and looks at the function level.

In order to parallelize programs there have been several proposals using speculative techniques. For example, *thread-level speculation* (TLS) [14] determines unlikely dependencies and based on this information speculatively parallelizes a program. Roth and Sohi [12] introduced *speculative data-driven multithreading*, in which critical sections are pre-executed in parallel in order to speedup the original program. In most cases these methods are based on heuristics. There also have been proposed profiling techniques [7, 11] that use profile information to determine the interesting spawning points for speculative threads.

The performance achieved with TLS is, however, disappointing. E.g. it is reported [4] that at most 31% of the execution of bzip2 can be parallelized using TLS, implying that the maximum speedup of TLS is 1.44 assuming perfect speculation and no threading overhead. In contrast, we obtain a realistic speed-up of 2.45. Prabhu and Olukotun [10] manually select regions to apply TLS and manually transform source code to extend the speedup obtainable with TLS. Nonetheless, they achieve speedups larger than 2 only when they can restructure the program such that traditional loop-level parallelism can be exploited.

The graphs introduced in this paper are new types of data flow graphs. Data flow graphs are categorized as either static data flow graphs or dynamic data flow graphs. Static data flow graphs are used by a compiler to make decisions in register allocation and optimizations. Zhao introduced the so called *Multithreaded Dependence Graph* [17] for program slicing. The dynamic version measures the data dependencies during a particular program run. *Redux* [8] is such a profiling tool, introduced by Nethercore *et al*. It measures the data dependencies at the level of assembler instructions and uses this information for debugging and program slicing.

5 Conclusion and future work

In this paper we presented a framework for extracting potential thread-level parallelism from sequential programs. We assume perfect knowledge of control flow and data flow by measuring all data dependencies occuring in a profiled execution of a program. Using the profiled information, we construct the function call graph and an interprocedural data flow graph to detect function-level parallelism. The

functions in a program are clustered such that strongly communicating (dependent) functions are members of the same clusters. Thread-level parallelism between function clusters is detected by analyzing inter-cluster data flow.

Explicit synchronization between threads is required to guide the correct communication of data between parallelized functions. The data structures facilitating this communication are identified by means of the data sharing graph.

We applied our framework to the *bzip2* benchmark from the SPECcpu2000 suite. For the compression part, a 4-stage pipeline was detected. Pipelining compression yielded a speedup of 3.74. Decompression was split in a 2-stage pipeline, yielding a 1.41 speedup. Overall, the parallel version of bzip2 is 2.45 times faster then the base version.

This work is still in its initial phase, so several aspects in our approach need to be worked out further and be formalized in the near future. Once the steps are more formalized, the process can also become more automated. A point of special interest goes into finding more parallel constructs. In this paper we mainly focused on pipeline constructs, but it is clear that other constructs can be extracted from our graphs that may uncover more parallelism. More parallel constructs will also enable us to produce results for more programs and make the presented technique more versatile. Furthermore bidirectional data streams need to be investigated more accurately, due to their frequent occurrence. This could be achieved by collecting some timing information during the profiling phase. Depending on the distribution in time these data streams could also prove to be useful for certain parallel constructs.

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] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, 2001.

[2] B. Flachs, S. Asano, H. Dhong, et al. The microarchitecture of the synergistic processor for a cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):63–70, Jan. 2006.

[3] X. Huang, J. E. B. Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Feb. 2003.

[4] A. Kejariwal, X. Tian, W. Li, et al. On the performance potential of different types of speculative thread-level parallelism. In *ICS06: Proceedings of the 2006 International Conference on Supercomputing*, pages 24–35, 2006.

[5] S. Kleiman, D. Shah, and B. Smaalders. *Programming with threads*. SunSoft Press, Mountain View, CA, USA, 1996.

[6] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214, 1997.

[7] P. Marcuello and A. González. Thread-Spawning Schemes for Speculative Multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 55–64, 2002.

[8] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2):1–22, Oct. 2003.

[9] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, 1999.

[10] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 142–152, 2005.

[11] C. G. Quinones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on precomputation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, June 2005.

[12] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 37, 2001.

[13] IMEC's new SPRINT design methodology transforms sequential code to multi-threaded code. http://www.imec.be/wwwinter/mediacenter/en/Sprint 2004.shtml.

[14] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[15] D. M. Tullsen and J. S. Seng. Storageless Value Prediction Using Prior Register Values. In *ISCA'99: Proceedings of the 26th International Symposium on Computer Architecture*,, pages 270–279, 1999.

[16] 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.

[17] J. Zhao. Multithreaded dependence graphs for concurrent java programs. In *Proceedings of 1999 International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 13–23, May 1999.