Architecture-Dependent Partitioning of Dependence Graphs

M. Beck and E. Zehendner

Dept. Mathematics & Computer Science
Friedrich Schiller University
D-07740 Jena, Germany

Th. Ungerer

Dept. Computer Design & Fault Tolerance
University of Karlsruhe
D-76128 Karlsruhe, Germany

Abstract

Performance tuning of non-blocking threads is based on graph partitioning algorithms that create serial code blocks from dependence graphs. Previously existing algorithms are directed toward deadlock-avoidance and maximisation of run-length. The latter criterion often generates a high synchronisation overhead. This paper presents a partitioning algorithm for dependence graphs that uses a heuristic to determine a cost-efficient solution based on an architecture-dependent cost function. We present empirical results based on benchmark programs that were compiled with MIT's Id compiler, extended by our architecture-dependent partitioning algorithm. The results demonstrate a reduction in software overhead with our architecture-dependent partitioning algorithm, compared with previously existing partitioning methods. The execution of the sample programs on an emulator for the Monsoon dataflow architecture shows a reduced number of processor cycles.

1 Introduction

Programs that were designed to execute on von Neumann architectures consist of serial code. Each instruction designates a single successor instruction that depends on the program order and the machine status. Instruction execution on the superscalar microprocessors happens out-of-order, due to the application of a local dataflow principle within an instruction window. However, the processor-external view of the instruction execution must follow the serial control flow due to the serial program order. This requirement results in a complex microprocessor organisation using register renaming, reorder buffering, and a completion or retirement phase during pipeline execution, that slows down execution speed.

However, data and control flow of an algorithm already define a partial order on the set of instructions in the code block. Dataflow or dependence graphs are a suitable medium to describe these dependencies. When using the *dataflow scheme*, programs are compiled into dataflow graphs that represent the data dependencies among instructions. Scheduling is data-driven: an instruction is ready to execute as soon as all required operands are available. The availability of operands is signalled by tokens that conceptually are propagated on the arcs of the dataflow graph. Dataflow architectures can be viewed as hardware interpreters of dataflow graphs. They use token matching prior to instruction execution. This synchronisation scheme is able to exploit all possible parallelism at instruction level but, unfortunately, leads to superfluous control overhead when executing sequences of instructions.

Arvind et al. [1] analysed the computational scheme of dataflow architectures and compared them to von Neumann architectures. As regards the cost of program execution, a program code can be divided into the so-called *basic work* that must be executed on each target architecture and into an architecture-dependent part called *overhead*. The sources of overhead in dataflow architectures are the additional code for unfolding of parallelism (several outbound arcs of a node in the dataflow graph) and for synchronisation (several inbound arcs of a node). A conceptual source of potential speed-up is the clipping of parallelism during the unfolding phase, automatically resulting in less synchronisation overhead. A trade-off must be found between the cost of unfolding parallelism and the benefits from utilising parallelism. We are thoroughly convinced that the trade-off should strongly depend on an architectural cost function.

To solve the overhead problem of fine-grained dataflow, dataflow graphs can be partitioned into subgraphs each with its own synchronisation interface and parallel unfolding interface to the remainder graph. Each subgraph that exhibits a low degree of parallelism can be identified within a dataflow graph and transformed into a sequential thread.

A *thread* in this sense is a subset of the instructions within a procedure body such that a compile-time ordering can be determined which is valid for all contexts in which the procedure can be invoked. Second, the thread should be *non-blocking*, i.e., once the first

instruction in a thread is executed, it is always possible to execute each of the remaining instructions in the compile-time ordering, without interruption or execution of instructions from other threads [12].

This proceeding is supported by the architectural proposals of dataflow computers using the *hybrid dataflow model* [3] where a thread of instructions is executed consecutively without matching further tokens except for the first instruction of the thread. Values passed between instructions from the same thread are stored in registers instead of writing them back to memory. These registers may be referenced by any succeeding instruction in the thread.

The next section describes and analyses the different strategies for dependence graph partitioning due to Iannucci [7], Hoch et al. [6], and Schauser [12, 13], which are predecessors and presuppositions of our own architecture-dependent partitioning method, presented in section 3. The architecture-dependent partitioning algorithm is a heuristic for determining a cost-efficient solution that is based on an architecture-dependent cost function. It can be proven that the algorithmic solution does not deteriorate during the proceeding of the algorithm [2]. Moreover, our architecture-dependent partitioning criterion can be replaced by a very simple rule in case of a hybrid dataflow architecture as target architecture. We present empirical results in section 4 before the conclusions.

2   Partitioning algorithms

Non-strict dataflow languages like Id create static and dynamic dependences between instructions, that must be observed during the compile-time partitioning of dataflow graphs. *Static dependences* are the "true" data dependences, while *dynamic dependences* are caused by control dependences or by split-phase transactions [1]. Partitioning methods may generate additional static and dynamic dependences that are not present in the original dataflow graph and that may cause deadlock. *Safe partitioning algorithms* perform only deadlock-free transformations, thereby generating deadlock-free (safe) partitions from deadlock-free programs.

Iannucci's partitioning algorithm [7]—called *method of dependence sets*—generates a safe partition of the dataflow graph. The dependence set of a node i is the set of all names of annotated nodes from which node i is reachable traveling along static arcs, only. Nodes with identical dependence sets are assembled into a thread. Conceptually, this is a depth-first traversal of the graph, where each end node of a dynamic arc serves as a starting point for a separate thread. A node is added to a thread if no dynamic arcs and no static arcs stemming from nodes with a different anno-

tation end in that node. If a node is not added, the subgraph starting at this node is cut off, and the node itself becomes a starting point for a new traversal, generating a new thread. The algorithm terminates when all instructions are assigned to threads.

Hoch et al. [6] enhanced Iannucci's algorithm by a further criterion for thread fusion. The goals of their partitioning algorithm are the maximisation of the thread length and the minimisation of the synchronisation between threads. In addition to Iannucci's annotations, all starting nodes of dynamic arcs are marked by Hoch et al.'s partitioning. Iannucci's dependence sets are called *entry sets*, and the analogous sets which are based on the starting nodes of dynamic arcs are called *exit sets*. Nodes are assembled to a thread if either their entry sets or their exit sets are the same. Hoch et al.'s algorithm is also safe.

Schauser [12, 13] extended the ideas of Iannucci and Hoch et al. by two proposals of partitioning algorithms: iterated partitioning and separation constraint partitioning. *Iterated partitioning* is an extension of Hoch et al.'s algorithm. The dependence sets (Hoch et al.'s entry sets) and demand sets (Hoch et al.'s exit sets) are computed. Then dependence-set partitioning and demand-set partitioning are applied alternately in the iterated partitioning scheme. During *dependence-set partitioning*, nodes with the same dependence sets are assembled to threads, while in the case of *demand-set partitioning* nodes with the same demand sets are assembled. Dependence-set partitioning and demand-set partitioning are greedy algorithms: they both seek to group together nodes into maximal subsets, where the sole criterion for grouping nodes together is whether they depend on the same set of inlet or outlet annotations [12]. To create a safe partition, an intermediate step called *subpartitioning* is introduced that splits threads with internal dynamic dependences. Thereby dependence-set partitioning as well as demand-set partitioning are proven to be safe.

It can easily be seen that, in general, nodes with the same dependence set may have different demand sets—and vice versa. This observation is the basis for the *iterated partitioning*: A partition of the dataflow graph is generated starting with one of the two methods described above. Then a reduced graph is constructed that consists of threads as nodes and dependences between threads as arcs. Multiple arcs joining the same nodes are reduced to a single arc, that becomes dynamic whenever any of the omitted arcs was dynamic. This process is repeated with the resulting graph until a stationary partition is reached. Each step is a safe transformation.

Although the iterated partitioning algorithm is more

powerful than dependence-set partitioning or demand-set partitioning alone, in some cases it may still fail to group nodes which can safely be merged into a single thread. The second method of Schauser, *separation constraint partitioning*, does not exhibit this limitation. It stems from a dual approach. The previous methods place two instructions in a thread if specific criteria based on the reachability of the nodes are fulfilled.

Schauser's separation constraint partitioning computes separation constraints which tell for any two nodes whether they can be merged or not. Two nodes are not assembled in the same thread if they are joined by an indirect dependence. Such a dependence constitutes a *separation constraint* that arises due to non-strictness and long latency communication.

The separation constraint partitioning computes separation constraints from a dataflow graph. Nodes without a separation constraint are assembled into a thread. This yields a reduced graph, and the process is repeated until the partition consists only of threads with mutual separation constraints.

The resulting partition is not unique, in contrast to a partition generated by one of the previously stated methods. The thread length is maximised in the sense that it is not possible to lengthen any longest thread by adding further nodes. Only the result partition is safe. Actually Schauser uses a mixture of separation constraint partitioning and of iterated partitioning for the implementation of partitioning, due to the complexity of the algorithm.

The primary goal of the partitioning methods stated above is the creation of a safe partition. Quantitative measures of the target architecture are not considered by these algorithms. The methods tend to create long threads with reduced interthread communication. In dataflow architectures, however, a context switch is cheap. The main goal of partitioning should be a reduction in synchronisation cost. The cost function for synchronisation is architecture-dependent, and is not linear in the number of arcs to synchronise. Since execution of coarse-grained threads causes additional cost, an analysis of the total cost is necessary.

All partitioning algorithms described above are based on Iannucci's method of dependence sets with a safe partition as single goal, partly enhancing Iannucci's method by the second goal of maximising the run length. All algorithms are provably safe. The lack of an appropriate cost function implies that runtime efficiency deterioration cannot be excluded by the proceeding of the algorithm. Our own architecture-dependent partitioning algorithm states a heuristic for determining a cost-efficient solution. It can be proven that there is no deteriorating by the proceeding to-wards the algorithmic solution. Moreover, in case of a hybrid dataflow architecture as target architecture, the architecture-dependent partitioning criterion can be replaced by a very simple rule.

3   Architecture-dependent partitioning

We now present a simple analytic cost model that describes the execution of threads on a dataflow architecture. We assume a dynamic dataflow architecture with explicit token-store (ETS) [9], where a token is passed, in succession, through a token queue, a matching unit, an instruction fetch unit, an ALU, and a form token unit. The architecture provides a set of internal registers that are used to store intermediate results. The matching unit accesses the frame memory, using the direct-matching scheme. Each instruction in the instruction set can appear as a synchronisation point according to the direct-matching scheme, aside from some special instructions where the matching unit synchronises a set of tokens without passing any values. The form token unit generates up to two result tokens for each processed instruction. Processing of an instruction needs a complete execution cycle, even in case of a mismatch in the matching unit.

How a code block is executed on such an architecture? The code block consists of a set of threads. Each thread is composed of a synchronisation interface, a thread body, and a parallel unfolding interface. The leading instructions of a thread form the, usually tree-like, synchronisation interface. Thread body and parallel unfolding interface may be interleaved; together they are organised as a totally ordered set of instructions. The instructions for the thread body as well as for the synchronisation interface are chosen such that, after all inbound arcs have been synchronised, the instructions in the thread body and parallel unfolding interface can be sequentially executed without interruption. The succession of instructions in the synchronisation interface as well as in the thread body follows the dataflow principle. The form token unit generates, for each processed instruction in the thread body, a result token destined to the next instruction in the sequential thread; this result token is directly passed to the matching unit in the following cycle. Thus, for each instruction in the thread body, at most one result token can be destined to an instruction in another thread, except for the last instruction of a thread, that can send up to two result tokens to different threads. Values can be passed in registers between instructions of the same thread body; each of these instructions can read at most one further value from the frame. We distinguish the following approaches to pass values to threads:

1. Values are transferred between threads on tokens. In this case, all instructions in the thread body are ready for execution as soon as all tokens on inbound arcs have been synchronised and the values passed on them have been made available to the instructions in the thread body.

2. Values are passed between threads via an ETS frame. Then, all instructions in the thread body are ready for execution as soon as a precalculated number of tokens have been synchronised and the values in the frame have been made available to the instructions in the thread body.

The cost of the synchronisation interface and the parallel unfolding interface, measured in the number of processor cycles, is different for these approaches. In this paper we only present results for the case that values are passed on tokens; the other case is treated in [4]. In the sequel we use the following notation:

I the set of threads of the code block (i.e., a description of the partition)

I the number of threads in the code block, $I = |\mathbf{I}|$

N_i the number of instructions in the body of thread i

S_i the number of tokens carrying values that are synchronised via the synchronisation interface of thread i

U_i the number of tokens carrying no values that are synchronised via the synchronisation interface of thread i

$\sigma(S, U)$ the cost (in processor cycles) of the synchronisation by the synchronisation interface for S tokens carrying values and U tokens carrying no values (if values are passed via the frame, then S denotes the number of values needed from other threads, covering the expenses of store resp. load operations)

F_i the number of result tokens generated by the parallel unfolding interface of thread i

$\varphi(F)$ the cost (in processor cycles) of generating F result tokens by the parallel unfolding interface

σ and φ are architecture-dependent cost functions for an optimum coding of the synchronisation interface and the parallel unfolding interface, relative to the chosen approach to transfer values between threads. In our basic model, that covers the mandatory features of dynamic dataflow architectures, φ is defined by $\varphi(F) = \max(F - 2, 0)$; function σ is depicted in

Fig. 1. Under these conditions, the number of processor cycles needed to execute a single thread i is given by $T_i = \sigma(S_i, U_i) + N_i + \varphi(F_i)$. The cost for executing the complete code block is $T(\mathbf{I}) = \sum_{i=1}^{I} T_i$.

Now we study the effects of generating a new partition of a code block by merging some threads; we confine ourselves to the merging of only two threads. Some formulae derived in [4] imply that the synchronisation interface constitutes the main source of additional overhead introduced by coarsen the partition. In consequence, the synchronisation cost function provides us with an architecture-dependent criterion to decide whether merging of threads would be advantageous. Such a criterion can—and should—be used in every partitioning algorithm that iteratively determines the final partition by merging some threads.

As an example of our proposed proceeding, we show how to modify the iterated dependence-demand-set partitioning algorithm following Schauser [12]. Assume the code block has been decomposed into disjoint basic blocks; the latter represent our program as directed acyclic subgraphs that are connected through their interfaces. We annotate these graphs with inlets and outlets, as described in [12].

The algorithm annotates all end nodes of dynamic dependencies with unique names. As mentioned above, the *dependence set* of a node i is the set of all names of annotated nodes from which node i is reachable travelling along static arcs, only. If node i is itself an endpoint of a dynamic dependence, its own name is added to its dependence set. The analogous sets which are based on the starting nodes of dynamic arcs are called *demand sets* [12].
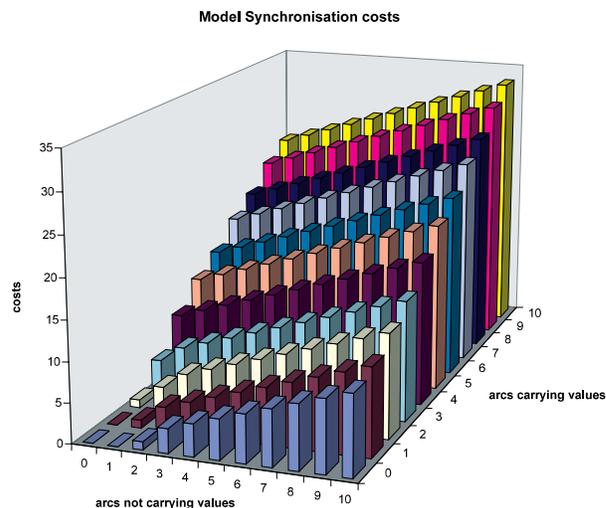
Figure 1: Synchronisation cost σ in the basic model

Whereas Schauser in one partitioning step merges all nodes with identical dependence set resp. demand set, we only merge two threads if the Schauser criterion holds and the synchronisation cost function indicates that this merging will be useful. Such merging does not change the actual dependence set resp. demand set, thus the newly generated thread can immediately participate in further merging transformations. Also, two nodes are not assembled in the same thread if they are joined by an indirect dependence (*separation constraint* [13]).

Architecture-dependent threads (partitioning algorithm)

1. Count the number of inbound arcs of each node in the basic block, separately for arcs carrying values resp. arcs carrying no values; then calculate the value of function σ for each node.

2. Determine the dependence sets of all nodes.

3. Choose an arbitrary pair of nodes i and j with identical dependence set, and merge them if the following conditions hold:

   - there is no separation constraint between node i and node j

   - merging nodes i and j results in a node k with

   $$\sigma(S_k, U_k) \leq \sigma(S_i, U_i) + \sigma(S_j, U_j) \qquad (*)$$

   Repeat step 3 until the partition becomes stationary.

4. Determine the demand sets of all nodes.

5. Choose an arbitrary pair of nodes i and j with identical demand set, and merge them if the following conditions hold:

   - there is no separation constraint between node i and node j

   - merging nodes i and j results in a node k with

   $$\sigma(S_k, U_k) \leq \sigma(S_i, U_i) + \sigma(S_j, U_j) \qquad (*)$$

   Repeat step 5 until the partition becomes stationary.

6. Repeat steps 1–5 until the partition eventually becomes stationary.

The proposed merging criterion prevents additional synchronisation cost. However, we have the impression that this criterion might be hardly practicable, in particular if few mergings would be refused. Thus we also derived a more handy test, that can be applied under certain conditions.

The course of the function σ has a jump whose position ($S + U = 3$) and height (Fig. 1) depend on the capabilities of the matching unit in the target architecture. The height of this jump prevents a merging of threads that have more inbound arcs carrying a value than can be coded within a single instruction to the matching unit. We can take advantage of this fact and simplify our merging criterion: Threads should not be merged if the synchronisation interface of the new thread generated from them could not be coded as a single instruction. If we substitute this simplified merging criterion into the partitioning algorithm described above (formulae marked by an asterisk), we always end up with a final partition that uses not more cycles than the initial nonpartitioned code block.

In [4] we studied the impact of several architectures on the partitioning algorithm; in the sequel we direct our discussion toward the Monsoon architecture [9]. Function φ there is identical to the one of our proposed basic model; function σ is depicted in Fig. 2. In the following section we analyse some sample partitions and show experimental results.

4 Experimental work

In this section we compare the performance of our partitioning algorithm to the algorithms of Schauser [12], i.e., iterated partitioning and separation constraint partitioning. The algorithms given by Schauser
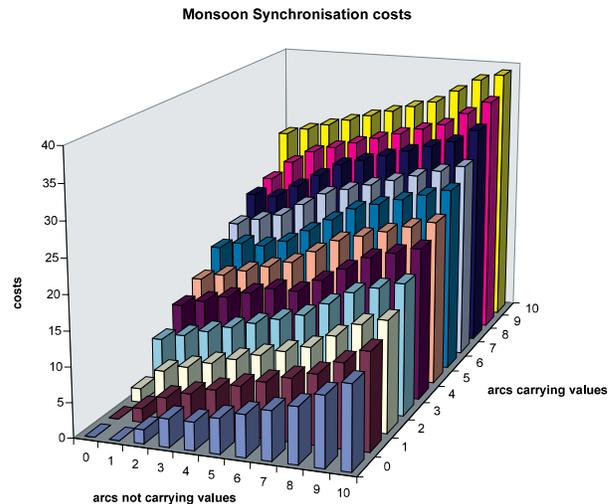
Figure 2: Synchronisation cost function σ of Monsoon

aimed at a partition of a code block into threads that should be as long as possible and thus reduce interthread communication.

We compiled and executed a program (see [15]) for the solution of the heat diffusion problem with the Id compiler provided by the programming environment ID World [8]. The compiler generates a dataflow graph and produces code for the Monsoon dataflow machine [9]. We studied in detail the partitioning of one cycle-free subgraph from the generated dataflow graph that corresponds to the following fragment of the program:

```
|[i,u2] = A[i,u2] ||i <- l1+1 to u1-1
```

For evaluation of the different partitions we generated code that could be executed on the Monsoon. Relevant parameters of this code are the number and the length of the threads, the number of arcs connecting the threads, their synchronisation cost, and the dynamic length (measured in processor cycles) of the partitions excluding latencies. The results of our analysis are given in Tab. 1. Schauser's partitioning algorithms generate long threads but at the expense of a high synchronisation cost. The threads generated by our algorithm have shorter average length than with Schauser's algorithms, and thus are more suited to hide the latency of split-phase transactions.

So far we compared partitions destined to the same architecture. In the sequel we relate the partition proposed by us for the Monsoon with partitions destined to other hybrid dataflow architectures. To study this problem, we constructed a testbed based on the ID World environment. Since we were interested in comparing different hybrid dataflow architectures, we developed a stand-alone emulator whose behavior can be adapted to several architectural properties through parameters. This emulator is based on the instruction set of the Monsoon. The specification of the Monsoon architecture has been preserved as far as possible. However, our emulator supports 256 registers. SVC-instructions do not call handler functions but implement their functionality, blocking the pipeline during a prescribed number of cycles. We emulate 1 to 16 processors and structure memory elements, but network conflicts are disregarded.

The instruction set and the specification of the various units in the processing element have been adapted to the needs of different architectures. Here we focus on provisions that are directed towards a cheaper synchronisation. Besides Monsoon, we distinguish three further modes of operation:

1. The 2.5-address machine. This mode adopts the separation of match offset and operand offset as in the Epsilon project [5]. The matching unit can address two operands in the frame independently. Results are passed in registers.

2. The 3-address machine. The matching unit is as in the 2.5-address machine. Results can be written back to the frame.

3. Load-store-architecture. From the EM project [11] we adopted the RISC-like execution of the instructions. The matching unit serves two purposes: implementing the direct-matching scheme as well as reading/writing values from/to the frame (the emulator differs from the EM project in this respect).

In addition to the emulator we developed a code generator. We partition the Monsoon assembly code generated by the ID World environment. A compiler backend is used to generate threaded code for the various modes of operation of the emulator.

Fig. 3 shows the results of the emulation for the program *speech* [10]; for a detailed review of further benchmark programs see [4]. The architecture dependent partitioning method (directed towards Monsoon code) is compared with non-partitioned Monsoon code and with Schauser's iterated dependence-demand-set partitioning on idealised competing architectures (Monsoon_th, 2.5 Addr, 3 Addr, and Load_store in Fig. 3). The results show that iterated partitioning can not diminish the overhead of fine-grain dataflow without additional cost. Architecture-dependent partitioning generates code with a cost of about 80% of that of fine-grained code. Thus we reached our goal to reduce the overhead without additional cost. The reduction in the degree of parallelism, caused by the partitioning, apparently had no negative effect on the utilisation of the pipeline during our experiments.
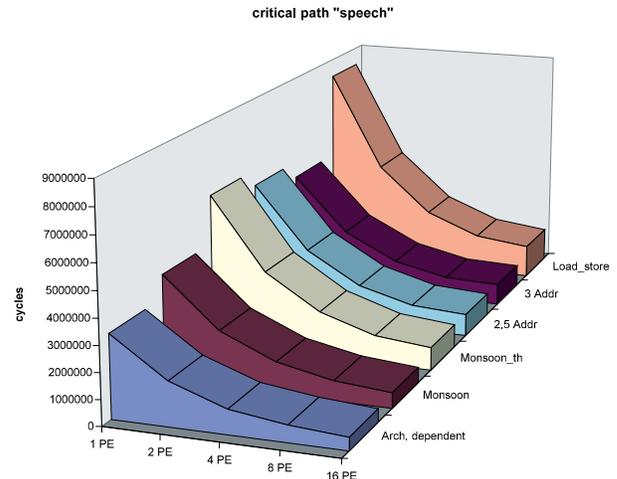
Figure 3: Execution time of benchmark *speech*

Table 1: Analysis of partitioning the sample graph

	dataflow	iterated partitioning	separation constraint partitioning	method of dependence sets	architecture-dependent partitioning
# of threads	26	13	11	15	12
maximal length	1	6	7	3	5
average length	1.0	2.0	2.36	1.73	2.16
# of arcs connecting threads	47	26	24	30	26
average synchronisation cost	1.81	1.53	2.09	1.06	1.25
# of processor cycles	58	49	47	46	45

5 Conclusions

We presented a new architecture-dependent partitioning algorithm to create non-blocking threads from dependence graphs. Previously published partitioning algorithms are directed toward deadlock-avoidance and maximisation of run-length, and often generate a high synchronisation overhead. In contrast, our partitioning algorithm uses a heuristic to determine a cost-efficient solution based on an architecture-dependent cost function. It can be proven that the algorithmic solution does not deteriorate during the proceeding of the algorithm. Moreover, the architecture-dependent partitioning criterion can be replaced by a very simple rule in case of a hybrid dataflow architecture as target architecture. We presented empirical results based on benchmark programs compiled with an extension of MIT's Id compiler. The results demonstrate a reduction in software overhead with our architecture-dependent partitioning algorithm, compared with previously existing partitioning methods. The execution of the sample programs on an emulator for Monsoon shows a reduced number of processor cycles.

The reduction of software overhead due to architecture-dependent partitioning may also be applicable outside the scope of non-strict dataflow languages and hybrid dataflow architectures. The synchronisation overhead we encountered in non-blocking threads generated from dependence graphs is inherent to fine-grained parallel multithreaded execution. Therefore optimising compilers for superscalar or multithreaded processors [14] may profit from our partitioning method.

References

[1] Arvind, D. Culler, and K. Ekanadham. The price of asynchronous parallelism: an analysis of dataflow architectures. *CONPAR88*, pages 541–555, September 1988.

[2] M. Beck. *Architekturabhängige Partitionierung von Datenflußgraphen, Dissertation*. Friedrich-Schiller-Universität Jena, 1997.

[3] M. Beck, T. Ungerer, and E. Zehendner. Classification and performance evaluation of hybrid dataflow techniques with respect to matrix multiplication. *Workshop PARS*, pages 118 –126, April 1993.

[4] M. Beck, T. Ungerer, and E. Zehendner. Architecture-dependent partitioning of dependence graphs. *Berichte zur Rechnerarchitektur 3, 23, Friedrich-Schiller-Universität Jena*, 1997. *ftp://ftp2.informatik.uni-jena.de/pub/AG/OPC/Be-97-BR-3:23.*

[5] V. Grafe and J. Hoch. The Epsilon-2 multiprocessor system. *J. Parallel and Distributed Computing*, 10:309–318, 1990.

[6] J. E. Hoch et. al. Compile-time partitioning of a non-strict language into seqential threads. *In Proc. 3rd Symp. on Parallel and Distributed Processing*, 1993.

[7] R. Iannucci. Toward a dataflow / von Neumann hybrid architecture. *15th Ann. Int. Symp. Comp. Arch., Honolulu*, pages 131–140, 1988.

[8] R. P. Johnson. Monsoon id world user's guide (draft). *CSG Memo 334, MIT LCS, 545 Tech. Square, Cambridge, MA*, 1992.

[9] G. Papadopoulos and D. Culler. Monsoon: an explicit token-store architecture. *17th Ann. Int. Symp. Comp. Arch., Seattle*, pages 82–91, 1990.

[10] A. Sah. Parallel language support for shared memory multiprocessors. *Masters thesis, Computer Science Div., University of California at Berkeley*, 1991.

[11] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. *16th Int. Symp. on Comp.Arch.*, pages 46–53, 1989.

[12] K. Schauser. Compiling lenient languages for parallel asynchronous execution. *PhD thesis, Computer Science Div., University of California at Berkeley*, 1994.

[13] K. Schauser, D. Culler, and Goldstein. Separation constraint partitioning - a new algorithm for partitioning non-strict programs into sequential threads. *Proc. Principles of Programming Languages*, 1995.

[14] J. Silc, B. Robic, and T. Ungerer. Asynchronity in parallel computing: From dataflow to multithreading. In *Journal of Parallel and Distributed Computing Practice*, January/February 1998.

[15] K. Traub. relax.id. id-world example suite. *MIT LCS, 545 Tech. Square, Cambridge, MA*, 1991.