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 maximization of run-length. The latter criterion often generates a high synchronization 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 most advanced von Neumann architectures—the superscalar microprocessors [22] like the PentiumPro, the HP8000, or the MIPS R10000—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 organization using register renaming, reorder buffering, and a completion or retirement phase during pipeline execution, that slows down execution speed.

Although the program order is total, the execution ordering must not be such restricted. Data and control flow of the program 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 signaled 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 synchronization 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] analyze the computational scheme of dataflow architectures and compare 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 synchronization (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 synchronization overhead. A trade-off must be found between the cost of unfolding parallelism and the benefits from utilizing parallelism. We are thoroughly convinced that the trade-off should strongly depend on an architectural cost function.

To solve the overhead problem of fine-grain dataflow, dataflow graphs can be partitioned into subgraphs each with its own synchronization interface and parallel unfolding interface to the remainder graph. Each subgraph that exhibits a low degree of paral-

lelism can be identified within a dataflow graph and transformed into a sequential thread. By serializing subgraphs to threads the goal of overhead reduction can be reached.

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 pause, interruption, or execution of instructions from other threads [19].

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, Hoch et al., and Schauser, 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 algorithm [2]. Moreover, the architecture-dependent partitioning criterion can be reduced to 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

Iannucci identifies the following, partly contradicting goals for partitioning algorithms [10]:

- *Maximization of exploitable parallelism*: A degree of parallelism in the application program, that is higher than the parallelism that can directly be utilized by the machine, is used for the hiding of memory and network latencies in dataflow architectures.

- *Maximization of run length*: Longer threads lead to longer intervals between context switches (run length). Run lengths which are long compared to pipeline depth have a positive effect on shortening critical path time. Short run lengths tend to create pipeline bubbles. Locality might be increased by longer run lengths but parallelism is restricted.

- *Minimization of explicit synchronization*: Explicit synchronization operations and instructions necessary for the unfolding of parallelism are main sources of overhead in dataflow architectures. Restriction of such overheads is a main source of higher efficiency. However, also the degree of parallelism shrinks.

- *Deadlock avoidance*: Deadlocks must be avoided. Two instructions whose execution order is dynamically determined cannot be statically scheduled in a single thread, otherwise potentially leading to a deadlock.

- *Maximization of machine utilization*: Partitions can be compared on the basis of how well they keep the dataflow processor pipeline full. In contrast to von Neumann architectures a pipeline in a (hybrid) dataflow machine executes instructions of different contexts in consecutive pipeline stages. The dataflow pipeline starves, if there are not enough contexts available, resulting in a bad machine utilization. This metric is architecture dependent. It is less general than the previously described but no less important.

Non-strict dataflow languages like Id [13] 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 partitions from deadlock-free programs.

The creation of a graph partition that is optimal relative to a realistic weight function is NP-complete [24]. In general, only approximate solutions are determined using iterative methods. The advantages of such algorithms are their simplicity, run-time efficiency, and prove of the safety of the solution by using only safe transformations in each iteration step.

Existing partitioning algorithms can be classified as depth-first or breadth-first. *Depth-first algorithms*

[5, 10] partition by choosing a path from an input to an output of a (sub)graph, assembling the visited instruction nodes into a thread, removing the corresponding nodes from the graph in the process. The algorithm is repeated until no instructions remain unpartitioned. Such partitionings tend to produce long threads while breadth-first algorithms allow to control partitioning by the degree of parallelism [12, 23].

Usually, cost functions that are easy to verify are used to control the partitioning. How far the generated partitions satisfy the goals stated above is described by heuristics that are empirically validated. In the following we describe several depth-first partitioning algorithms in more detail.

The goal of Iannucci's partitioning algorithm [10]—called *method of dependence sets*—is the generation of 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 to a thread. Conceptually, it is a depth-first traversal of each end node of dynamic arcs, forming its own thread. A node is added to a thread, if no dynamic arcs and no static arcs stemming from nodes with a different annotation end in that node. If a node is not added, the subgraph starting with this node is cut and the node itself is a starting point for a new traversal, generating a new thread. The algorithm terminates when all instructions are assigned to threads.

Hoch et al. [9] enhance Iannucci's algorithm by a further criterion for thread fusion. The goals of their partitioning algorithm are the maximization of the thread length and the minimization of the synchronization 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 [18, 19, 20] extends 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, and dependence-set 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 and demand-set partitioning are greedy partitioning algorithms: they both seek to group together nodes into maximal subsets, where the sole criteria for grouping nodes together is whether they depend on the same set of inlet or outlet annotations [19]. To create a safe partitioning, an intermediate step called *subpartitioning* is introduced, that splits threads with internal dynamic dependences. Thereby the dependence-set partitioning as well as the 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. That 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 between the same nodes are omitted. The remaining arc is dynamic whenever one of the omitted multiple arcs is a dynamic one. The 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 or demand-set partitioning, 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 nonstrictness 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. Thereby a reduced graph is yielded and the process repeated until the partitioning 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. Threads are maximized by their length in

the sense that it is not possible to lengthen the longest threads by adding further nodes. Only the result partition is safe. Actually Schauser uses a mixture of separation constraint 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 the 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 a partitioning should be the reduction of the synchronization cost. The cost function for synchronization is architecture-dependent, and is not linear in the number of arcs to synchronize. 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 maximization of run length. All algorithms are provable 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 towards the algorithmic solution. Moreover, in case of a hybrid dataflow architecture as target architecture, the architecture-dependent partitioning criterion is reduced to 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) [15], 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 synchronization point according to the direct-matching scheme, aside from some special instructions where the matching unit synchronizes 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 synchronization interface, a thread body, and a parallel unfolding interface. The leading instructions of a thread form the, usually tree-like, synchronization interface. Thread body and parallel unfolding interface may be interleaved; together they are organized as a totally ordered set of instructions. The instructions for the thread body as well as for the synchronization interface are chosen such that, after all inbound arcs have been synchronized, the instructions in the thread body and parallel unfolding interface can be sequentially executed without interruption. The succession of instructions in the synchronization 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 synchronized 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 synchronized and the values in the frame have been made available to the instructions in the thread body.

The cost of the synchronization 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 synchronized via the synchronization interface of thread i

U_i the number of tokens carrying no values that are synchronized via the synchronization interface of thread i

$\sigma(S, U)$ the cost (measured in processor cycles) of the synchronization by the synchronization 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 (measured 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 synchronization 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 synchronization interface constitutes the main source of additional overhead introduced by coarsen the partition. In consequence, the synchronization 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

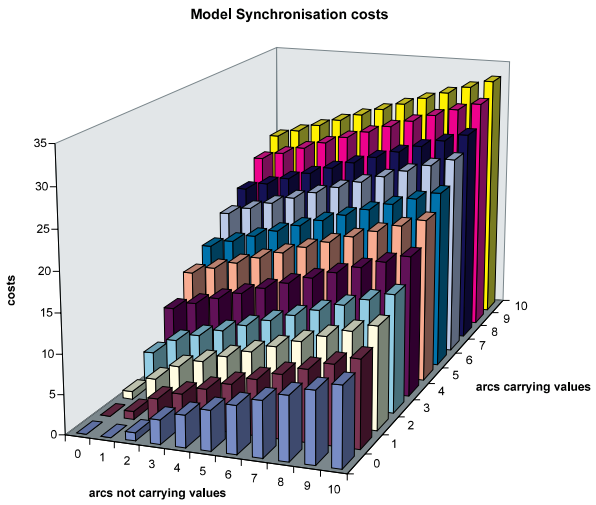

Figure 1: Synchronization cost function σ of the basic model

partitioning algorithm of Schauser et al. [18]; we follow Schauser [19] in our terminology. 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 [19].

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 traveling 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* [19].

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 synchronization 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* [20]).

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 synchronization 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 ($S + T = 3$) 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 synchronization 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 fewer 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 [15]. 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 analyze some sample partitions and show experimental results.

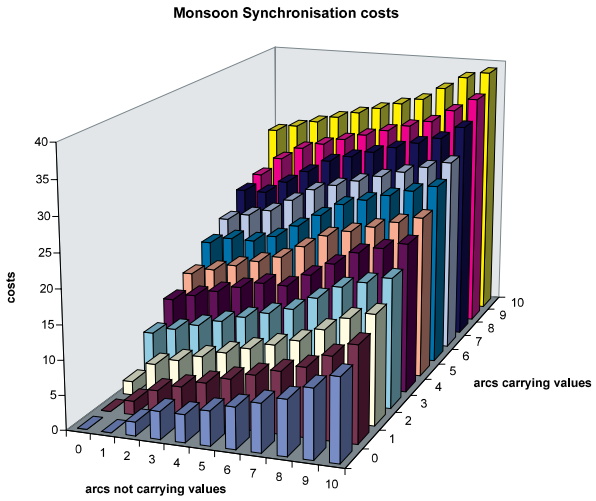

Figure 2: Synchronization cost function σ of Monsoon

4 Experimental work

In this section we compare the performance of our partitioning algorithm to the algorithms of Schauser [19], i.e., iterated partitioning and separation constraint partitioning. The algorithms given by Schauser aimed at a partition of a code block into threads that should be as long as possible (second Iannucci criterion) and thus reduce interthread communication (third Iannucci criterion).

As an example we used a program for the solution of the heat diffusion problem, written in the dataflow language Id [13]; the code of this program can be found in [25]. We compiled and executed this program within the programming environment ID World [11] of the MIT. The latter comprises an Id compiler that produces code for the Monsoon dataflow machine [15], an emulator of Monsoon, and a collection of diagnosis tools. The Id compiler generates a dataflow graph from the program. All partitioning algorithms we investigated use cycle-free directed graphs. Cycle-containing graphs are first decomposed into cycle-free subgraphs before any of these methods can be applied.

We studied in detail the partitioning of one cycle-free subgraph from the dataflow graph that corresponds to the following fragment of the program:

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

For 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, and their synchronization cost
- 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 synchronization 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.

The basis of our experimental environment is still the ID World environment [11]. This system generates monasm code (i.e., Monsoon machine code). The monasm code can be emulated in 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, with the following exceptions:

- 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 can emulate any number of processors and structure memory elements, as long as this number is a power of 2.
- We disregard network conflicts; network instructions can be penalized with latencies.

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 synchronization. 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 [7]. 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 [17] 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.

Our simulation results compare our method to partition Monsoon code with Schauser's iterated dependence-demand-set partitioning on idealized competing architectures and with non-partitioned Monsoon code. We used the benchmark programs *fib*, *sorts* [21], and *speech* [16] to compare the generated code on the basis of the following two criteria:

- The number of tokens in the system measures the quality of the generated code. (Good code introduces few tokens.)
- The shortest path is correlated to the resulting degree of parallelism and thus to the possibility of hiding latencies.

The three sample programs have been processed on the emulator for 1 to 16 processors (and 1 to 16 structure memory elements). Fig. 3 shows the results of the emulation for the program *speech*; for a detailed review of all results see [4]. The results prove that iterated partitioning can not diminish the overhead of fine-grain dataflow without additional cost. Only after optimizing the architecture parameters we achieve threaded code that executes at the same speed as fine-grained code. One way out would be to hide

the synchronization cost by decoupling the matching from the processing, as in *T [14] or the Decoupled Graph/Computation model [6]. 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 utilization of the pipeline during our experiments.

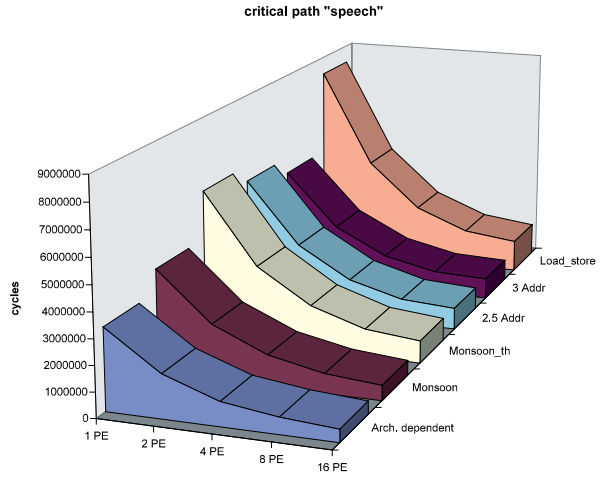

Figure 3: Speed-up of benchmark *speech*

5 Conclusions

In this paper, 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 maximization of run-length, and often generate a high synchronization 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 reduced to 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 technique of cost functions can not only be used to assess the merging of threads but is also applicable to analyze other known methods for the reduction of overhead in dataflow architectures. We plan to compare with hybrid methods some modifications of the fine-grain dataflow principle, like the TUP instruction [8]. The goal of these investigations is an evaluation of various performance-improving techniques for dataflow architectures.

Multithreading techniques and fine-grain parallelism are playing an increasingly important role in processor microarchitectures and in optimizing compilers. The synchronization overhead we encountered in non-blocking threads generated from dependence graphs is inherent to fine-grained parallel multithreaded execution. 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.

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. Available under ftp://ftp2.informatik.uni-jena.de/pub/AG/OPC/Be-97-BR-3:23*, 1997.

[5] L. Bic. A process-oriented model for efficient execution of dataflow programs. *J. Parallel and Distributed Computing*, 8(12):42–51, December 1990.

[6] P. Evripidou and J.-L. Gaudiot. The USC decoupled multilevel data-flow execution model. *In: Gaudiot, J.-L., Bic, L. (eds.): Advanced Topics in Data-Flow Computing, Prentice Hall, Englewood Cliffs*, pages 347–379, 1991.

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

[8] J. Gurd, C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

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

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

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

[12] B. Lee and K. Krishna. Program partitioning for multithreaded dataflow computers. *Proc. 26th Ann. Hawaii Intern. Conf. on System Sciences*, 1993.

[13] R. Nikhil. Id language reference manual, V. 90.1. *Technical Report CSG Memo 284-2, Laboratory for Computer Science, MIT*, July 1991.

[14] R. Nikhil, G. Papadopoulos, and Arvind. *T: A multithreaded massive parallel architecture. *19th Ann. Int. Symp. Comp. Arch., Gold Coast, Australia*, pages 156–167, 1992.

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

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

[17] 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.

[18] K. Schauser. Compiling dataflow into threads. *Master thesis, Computer Science Div., University of California at Berkeley*, 1991.

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

[20] 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.

[21] A. Shaw. sorts.id. ID-World example suite. *MIT LCS, 545 Tech. Square, Cambridge, MA*, 1991.

[22] M. Slater. The microprocessor today. *IEEE Micro*, pages 32 – 44, December 1996.

[23] J. Strohschneider and K. Waldschmidt. Adarc: A fine grain dataflow architecture with associative communication network. *Proc. EUROMICRO'94, Liverpool*, 1994.

[24] K. Traub. Sequential implementation of lenient programming languages. *TR-417, MIT LCS, 545 Tech. Square, Cambridge, MA*, 1988.

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

Table 1: Analysis of partitioning the sample dataflow graph

	dataflow	iterated partitioning	separation constraint partitioning	method of dependence sets	architecture-dependent partitioning
number 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
number of arcs connecting threads	47	26	24	30	26
average synchronization cost	1.81	1.53	2.09	1.06	1.25
number of processor cycles	58	49	47	46	45