A Distributed Memory Multiprocessor Based on Dataflow Synchronization


Institute for Computer Engineering and Communication Networks
Swiss Federal Institute of Technology, 8092 Zurich, Switzerland

Abstract

This paper describes the design of a multiprocessor with innovative fundamental concepts in the code executing hardware and distributed memory management. The coarse-grain dataflow architecture presented here is a fusion between dataflow and von Neumann concepts. A code-generator for the single-assignment language SISAL has been implemented and is used to benchmark the multiprocessor. A quick review of the project ADAM is followed by experimental results.

1. Introduction

The programmability, the efficient exploitation of parallelism and the ability to hide latency are the key elements of a scalable general-purpose multiprocessor. The research project ADAM expects to provide solutions for such multiprocessor environment by careful analysis of software and hardware restrictions [1].

Currently the project consists of the code-generator for the SISAL language, the multiprocessor-simulator, called the Metamachine and a new language for programming multiprocessors, called MFL [2]. The design of the language and the multiprocessor architecture have undergone multiple refinement steps aiming to bridge the traditional semantic gap between the instruction set architecture (ISA) and the high level languages.

It is our belief that the programmer need not be aware of the multiprocessor configuration and may nevertheless exploits the potential of a multiprocessor. This assumption requires the implementation of a dynamic load-distribution mechanism in the multiprocessor and languages which do not necessitate explicit programming of parallelism.

The dataflow approach combined with functional languages has been one of the most promising concepts so far. However, dataflow is inefficient for execution of sequential threads as is the von Neumann approach for execution of parallel code. The intuitive consequence is that a combination of the dataflow and the von Neumann principles will provide the best solution on a wide scale of applications. This combination of the two extremes in architecture design is called the coarse-grain dataflow principle.

2. Coarse-grain Dataflow Architecture

Another reason why the architecture being chosen in our project is dataflow is because a lot of experiences with this approach has been established internally (e.g. in the ETH Multiprocessor project Empress [3] and joint work with other groups [4]). Moreover, data-driven program execution is a very straightforward technique to exploit parallelism at maximum scale.

Thoughts on performance optimization and resource management aspects have led to a derivative of the fine-grain dataflow concept. The resulting approach, the coarse-grain dataflow technique differs from the original idea in as much as the scheduling quanta are no more single instructions but blocks of code (i.e. codeblocks).

Each codeblock is representing the smallest entity of software being individually scheduled and started in a codeblock processor. The term 'codeblock' is thus referring to a particular runtime instance of its static description (i.e. the 'codeblock code') as it is generated by the compiler. Such a runtime instance of a codeblock has its own 'context' and, as being shown later, assigned memory space for temporary data storage. From a granularity point-of-view a codeblock...
might be compared with a procedure or a function of a conventional high-level language. The main difference is the fact that a codeblock invoking a new codeblock will not automatically become inactive until the invoked codeblock terminates. Instead, as described in figure 1, both codeblocks will be executed concurrently. Codeblocks are sequential threads of program code which are woke up upon arrival of data and are suspended when data is missing. This efficient and simple synchronization mechanism, based on constructs similar to I-structures [5], provides efficient execution of sequential and parallel threads with low communication losses. The implemented fast context-switching between codeblocks can provide for a large latency hiding potential.

3. The Metamachine

The Metamachine is a simulator for the homogeneous multiprocessor based on the codeblock dataflow architecture. Because of its complete functionality the Metamachine can be used as an emulator, though the main reason for the construction of the Metamachine was to study the behavior of the multiprocessor, i.e. to simulate the codeblock multiprocessor and to detail the specifications for a hardware prototype. Figure 2 gives an idea of the extensive monitoring facilities of the Metamachine.

![Figure 2: The Metamachine's user interface](image)

The Metamachine simulates a distributed memory multiprocessor, consisting of many identical processing elements connected through a network. A processing element (PE) is split into four asynchronous functional units which are directly connected to each other: the sequencer, the codeblock manager, the object manager and the network manager (figure 3). The memory is distributed on every PE and can be accessed globally through unique object reference numbers. The static property of all codeblocks, namely the executable code itself is copied into the local memory of every PE at program-load-time.

---

2. The explanation of I-structures would be outside the scope of this paper. A simplified description is given at the end of chapter 3.
manager. All objects are referenced through an object number and are located in the object memory. The object manager has write access to the local memory of a codeblock. The objects may be records of any atomic types like boolean, real, integer or even objects, allowing thereby recursive data structures.

Every codeblock owns its local memory called frame. It is 32 words long and holds temporary variables, a local stack and any parameters received from the caller. Every frame word as shown in figure 5 contains two status bits, the present bit and the wait bit. These are needed for synchronization between codeblocks; a similar construct is presented in [8]. Any reference to a word with a cleared present-bit will cause a context switch for the processor of the active codeblock into wait state and start any of the ready codeblocks. The valid bit is needed for error propagation.

<table>
<thead>
<tr>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>32 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>present</td>
<td>valid</td>
<td>wait</td>
<td>value</td>
</tr>
</tbody>
</table>

**Figure 5: Format of a frame-word**

### 4. SISAL

The functional programming language SISAL [6] follows the single assignment convention [7], allowing the compiler to detect implicit parallelism. The programmer is thus not concerned with the detection and management of parallelism. Apart from the mentioned advantages the most important ones, as far as the codeblock project is concerned, include:

a) wide usage in parallel processing research centers;

b) architectural and/or implementation trade-off studies can be undertaken by comparing results with the ones of other research groups;

c) sharing and development of a pool of benchmark programs is feasible.

The front-end of the SISAL compiler being used in this project has been programmed at the Lawrence Livermore National Laboratories, USA. It is implemented on VAX computers and produces IF1 code, a text-file containing a list of dataflow graphs. The SISAL compiler and the IF1 interpreter have been ported by the author to the Macintosh II which is used for all software developments in the project ADAM.

### 5. IFFM, an interchange file format for the Metamachine

Most input information for the Metamachine is included in an IFFM file. IFFM is an easily readable and modifiable text-file format. A manual modification of code like break-point insertion, change of literal values or even programming in "assembler" is therefore easily done with the help of an ordinary text editor. An IFFM file consists of segments, e.g. the file can contain code segments, type-descriptor segments, codeblock-reference segments or any other encapsulated information structure.

### 6. ITI, the IF1 to IFFM compiler

The IF1 to IFFM compiler (ITI) is the back-end of a SISAL to IFFM compiler system, producing code for the Metamachine. In the following chapter some specific properties of the Metamachine code as well as the general attributes of the codeblock code generation and optimization will be discussed in detail.

A code unit is called a *function* at the SISAL level, a *graph* at the IF1 level and a *codeblock* at the Metamachine level. The mapping scheme is rather simple: a) every SISAL function is mapped to an IF1 top-scope graph and b) every such graph is sequenced to a codeblock.

The translation process can be augmented by some optimization passes concerning the parallel or sequential behavior of the program. Therefore a SISAL function can be partitioned into several codeblocks; a program is partitioned into logical entities called functions and the Metamachine code is partitioned into cost-balanced entities called codeblocks.

The code generator uses three different code representations. The first one is an acyclic dataflow graph with unidirectional connections, a derivative of the IF1 graph. The second representation is the sequential three-address intermediate code (TAC) being generated after sequentializing a DFG. The last one is the stack-oriented IFFM code as introduced in the previous chapter. The advantage of the TAC representation is the flexibility with respect to any changes of the target specifications and an easy implementation of the optimization algorithms.

### 7. Properties of codeblock-code

A *codeblock* is an execution unit with functional behavior. The *static description* of a codeblock, the executable code, is shared by multiple codeblocks. An expanded codeblock is composed of a *dynamic description* consisting of a program counter and a pointer to the allocated frame; this is the context of the codeblock. Local variables and received arguments are kept in the frame, which is described in chapter 3. The caller sends as argument either an atomic value or an object containing the parameters. Such an object is allocated in the caller-codeblock and deallocated in the called-codeblock. The called-codeblock also receives a globally unique address, called the *return-address* where the result should be written to. The *return-address* points to a frame-word of the caller’s frame with the present-bit cleared. The present-bit is set during the write access to the mentioned frame-word. If the caller-codeblock is waiting, it will be activated by the return of own result. A read-access to a word with cleared present-bit will switch the active codeblock to waiting-state.

The instruction set architecture (ISA) of the multiprocessor incorporates loads which behave as split-transactions. They initiate a request by clearing the present-bit and sending the request. The

3. the meaning of balance is described in the chapter 8.

4. expanded means that the codeblock has been initialized on a PE and is ready for execution, see also figure 3.
synchronization is done through testing the present-bit during a read-access. A cleared present-bit forces a fast context-switch to an alternate ready codeblock allowing thus to hide network or memory latency.

8. Code Optimization

Apart from the conventional optimization algorithms (e.g. peephole optimization, dead-code elimination, copy propagation [8]), the ITI optimizer is based on optimization techniques typical for the coarse-grain dataflow architecture: code reordering and graph partitioning.

To effectively hide the latency a PE must either have enough ready codeblocks to switch to, or the split-transactions should be used effectively to override a potential context-switch. The split-transaction consists of a request building instruction and a data consuming instruction. The first sends a request to the object manager, clearing the present bit. Code can be inserted between the request and consumption instructions through code reordering. With enough code available to fill the latency gap the possible context-switch can be avoided. As an example, figure 6 shows a 20 % shorter execution-time achieved through code reordering[5]. The idle-time in figure 6 is produced through network and memory latency.

The reordering generates somewhat longer machine code. This is due to the stack-oriented ISA of the Metamachine which favors the common, depth-first code order. The drawback of this kind of optimization is that excessive reordering may put too large demands on resources as frame-size or request-queues.

<table>
<thead>
<tr>
<th></th>
<th>execution</th>
<th>idle switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>reordered</td>
<td>8451 cyc</td>
<td>700 cyc</td>
</tr>
<tr>
<td>depth-first</td>
<td>10651 cyc</td>
<td>2630 cyc</td>
</tr>
<tr>
<td></td>
<td>25 %</td>
<td>449</td>
</tr>
<tr>
<td></td>
<td>8 %</td>
<td>244</td>
</tr>
</tbody>
</table>

Figure 6: Influence of code reordering[6]

A codeblock should have appropriate execution length, i.e. it should be balanced. A codeblock too small has excessive communication costs and should be inline expanded. A codeblock too large usually lessens the exploitable parallelism. A simple partitioning algorithm tries to split independent DFG parts into parallel executable graphs which are then mapped to codeblocks. The partitioning grain-size is determined with simple communication and sequencing cost analysis similar to [10]. With SISAL generated DFG's the partitioning becomes simple. As a result of SISAL's single-assignment rule no unresolvable cycles as described in [11] can occur. The partitioning algorithms of ITI tries to partition the graph into reasonable but not optimal parts. Therefore a heuristic method is used to balance the communication cost against the loss in parallelism.

5. The benchmark used is described shortly in chapter 11 and in detail in [9].

9. Exploitation of parallelism

The instruction level parallelism available in a DFG is used to hide network and memory latency; it is described in chapter 8. The forall-loop parallelism can be exploited in two ways: Through recursive loop-body calls or through a dedicated instruction called parallel-call (figure 7). The parallel-call overrides any other active codeblock forcing every processing element to work on a part of the parallel loop. Every parallel-called codeblock determines the bounds that it should compute using its PE number and signals the caller-codeblock when finished. The recursive loop-body calls consists of a call embedded at the beginning of a loop-body which calls the next iteration. Every caller waits on the synchronization signal from the called codeblock.

Figure 7: loop parallelism

10. Dynamic Load Distribution

The load distribution plays an important role in the performance of the coarse-grain dataflow multiprocessor. We assume that the load of a processing element is roughly proportional to the number of expanded codeblocks and the amount of data located on the same PE. A proportionate load distribution contributes to an optimal resource utilization, a very important performance component. The load-distribution scheme implemented in the Metamachine is based on a codeblock-token-ring. In the following this simple and efficient scheme is shortly described.

Every codeblock created by a call instruction exists first in the form of a token (figure 4). Such a codeblock token is floating on the token ring until a PE fulfilling the necessary expansion-criterion is found. In this case the codeblock is expanded from the token form into the executable codeblock by allocating frame and context. Codeblocks in token-form provide work-load distribution, when expanded they are fixed to a specific PE. A detailed discussion is given in [12]. On the other side, a PE should have multiple expanded codeblocks so that fast context-switch to a ready codeblock can occur, hiding this way the latency caused by memory or network access. A PE with many ready codeblocks will not idle.

The fulfillment of the expansion condition is regulated
by the MinReady parameter. This parameter represents on one hand the aim to execute as many codeblocks in parallel as possible and on the other hand the goal to increase locality. MinReady specifies the minimal number of ready codeblocks in one PE, a detailed discussion is given in (12). Figure 8 shows the influence of different MinReady parameters for the same benchmark. Based on the experimental results a value of 1 has been found to produce optimal results.

Figure 8: The influence of the MinReady parameter

11. Performance analysis

Usually speedup is used as the standard multiprocessor performance measure, and every multiprocessor attempts to realize a linear speedup. Nevertheless this measure may be very misleading because different benchmarks produce different speedups. The speedup chart in the figure 10 should therefore be viewed more as a comparison than as a computing-performance measurement.

The benchmark used in the following is the cellular automaton algorithm (9) programmed in SISAL. The algorithm fits well into the single-assignment world of SISAL and allows generation of parallel-loops. The figures 9 to 10 evaluate the techniques discussed in chapter 9 using the mentioned benchmark.

Figure 9: Execution time

The figure 9 does not show a significant advantage of a dedicated parallel-call instruction over the common call. This is due to the simple but very efficient dynamic load distribution scheme of the Metamachine.

As shown in figure 11, the codeblocks are distributed faster with the parallel-call but the latency hiding potential of the parallel-call construct is much lower than the one of the standard call construct. The reason is that with the latter the number of codeblock-tokens generated is usually larger then with the parallel-call. Using the dynamic load-distribution scheme and the MinReady parameter described in the previous chapter we observe that the common call is more flexible. It shows therefore, that the dynamic codeblock-distribution is not a performance bottleneck. It is more the effect of finite resources which tend to degrade the performance, especially on configurations with many processors.

Figure 11: Codeblock expansion

6. Specifies the minimal number of ready codeblocks on a PE.
7. The benchmark is described in chapter 11.
minimal effect on total execution time.
The current simulation is modeled with finite resources and speedup decreasing factors are arising with larger processor numbers.

The performance limiting components may be determined only by running benchmarks larger than presented here (e.g. the program SIMPLE[13]). The location of data has been identified as a potential bottleneck when running with many processor elements. If all working codeblocks depend on data located on only one PE, then the manager of such processor becomes overladen with requests. There are two possibilities to circumvent this bottleneck, either through distributed allocation of data (i.e. on all PE's) or through caching the read-only data (cache coherence). In the latter case, the single-assignment rule of SISAL identifies easily data with the read-only state. The figure 13 shows the improvement in terms of faster execution when data is distributed over the multiprocessor.

It is encouraging that similar research work is emerging from various places. Especially the dataflow group around Arvind is working on quite similar architectures ([14] and [11]). The coarse-grain dataflow architecture is a large experimenting ground with promising features but is still far from being fully understood. Flexible experimenting platforms like our Metamachine have therefore proven to be an absolute necessity for serious investigations.

The described IT1 compiler is running on an Apple Macintosh computer as an MPW-tool and is written in Modula 2. A graphical display program for the IT1 graphs has been implemented and is used for dataflow graph visualization. Together with the Metamachine the SISAL to IFFM compiler system will be the basic experimentation tool in the near future. The flexibility of the code generator allows easy adaptation to future changes of the codeblock machine. Using the Metamachine we hope to gain profound understanding and determine exact specifications of the proposed architecture making the realization of a hardware prototype possible. This will place us also in the favorable position of having software tools and applications ready when the hardware is realized.

13. Acknowledgments

This paper describes research undertaken within Fachgruppe Systemtechnik, Institut fuer Elektronik at the ETH Zuerich. The project ADAM emerged from the project EMPRESS [3] and was defined at the beginning by R. Buehrer and J. Wyttenbach. The simulator described has been realized by students and was subsequently improved by O. Maquelin.

The author wishes to thank all people who showed interest in his work, and especially O. Maquelin for his support and criticism. A first version of this paper has been co-authored by R. Buehrer.

Part of the project reported herein has been supported by grant no. 2.309-0.86 of the "Schweizer Nationalfonds".

Bibliography


