COMBINING STATIC PARTITIONING WITH DYNAMIC DISTRIBUTION OF THREADS

Ronald Moore, Melanie Klang, Bernd Klauer and Klaus Waldschmidt

This paper presents a hybrid approach to automatic parallelization of computer programs which combines static extraction of threads (tasks) with dynamic scheduling for parallel and distributed execution. Fine-grain scheduling decisions are made at compile time, and coarse-grain scheduling decisions are made at run time. The approach consists of two components: compiler technology which performs the static analysis (thread extraction), and an architecture which takes over the responsibility for scheduling and distributing the threads. Each processor is augmented with a ***broker****, whose responsibility it is to* ***shop*** *for tasks for the processor to perform. This approach aims to provide an adaptive run-time distribution of computation for irregular problems such as the simulation of embedded systems. Finally, this approach is general enough to allow the seamless incorporation of heterogeneous hardware, in particular including dynamically reconfigurable hardware, e.g. FPGAs.*

1. Introduction

1.1 Motivation and Overview

Programming parallel computers is hard. In fact, the difficulty of designing parallel software is increasingly seen as the main barrier to wider acceptance and usage of parallel computers (e.g. Patterson 1997). Programming parallel computers is inherently harder than programming sequential computers, since computation as well as data must be *distributed* amongst the available resources.

This paper presents a hybrid approach to automatic parallelization of computer programs which combines static extraction of threads with dynamic scheduling. Fine-grain scheduling decisions are made at compile time, and coarse-grain scheduling decisions are made at run time. The approach consists of two components: compiler technology which performs the static thread extraction, and an architecture which takes over the responsibility for scheduling and distributing the threads amongst the available processors.

F. J. Rammig (ed.), *Distributed and Parallel Embedded Systems*
© Springer Science+Business Media New York 1999

This architectural component is itself distributed: Each processor is augmented with a *broker*, whose responsibility it is to *shop* for tasks for the processor to perform. See figure 1. Taken together, these brokers implement not only a Distributed Shared Memory (DSM), but also a Distributed Shared Scheduler. Each combination of a processor, a broker and a local memory is called an "attraction memory site", and functions like a giant, slow cache in a *Cache Only Memory Architecture* (COMA), compare (Hagersten et. al., 1992).

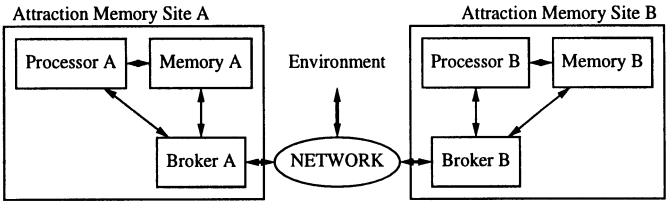

Figure 1: Abstract Schematic for a system with a Distributed Shared Memory (DSM) and a Distributed Shared Scheduler.

The architecture described in this paper has two goals. One goal is to provide an adaptive run time distribution of computation for irregular problems and for applications which will be used in dynamic, unpredictable environments.

However, the main goal is to attack the problem of *programmability*. The system presented in this paper takes over the complete task of distributing both data and computation, and as such has the potential to make programming parallel systems approximately as easy as programming sequential systems.

1.2. Background

The approach outlined in this paper draws together concepts from two previously independent fields of: Multithreaded Architectures on the one hand, and Cache Only Memory Architectures (COMAs) on the other: The concepts from the Multithreaded Architectures are used to define the basic scheduling terms (see Culler, 1993 or Nikhil, 1993); The concepts from COMAs are used to provide a DSM where data is free to migrate from one local memory to another (Hagersten et. al., 1992, Saulsbury et. al., 1996).

Previously, the questions of data distribution and computation distribution were strictly separated. Multithreaded architectures had little to say about data distribution, and COMAs were content to provide a DSM and leave the problems of program partitioning and scheduling to the programmers. This division of labor simplified the development of each camp. However, a point has been reached where significant further development can be obtained by lifting the division and optimizing each component (scheduling and DSM), based on knowledge of the other.

We call this combination of COMA and multithreading SDAARC, for *Self Distributing Associative ARChitecture* (Moore et. al., 1998a and 1998b).

The rest of this paper is organized as follows: Section 2 provides an Overview,

describing first the overall SDAARC Strategy, and then the hardware and software (compiler) prerequisites of SDAARC. Section 3 discusses the compiler technology in more detail, and section 4 discusses the run-time distribution protocol. Section 5 discusses possible extensions for embedded systems. Section 6 presents conclusions.

2 Overview

2.1. Strategy

In SDAARC, we treat both data and computation as a *population of migratory objects.* The central idea is to first select objects of an appropriate granularity, and then distribute those objects adaptively, responding to conditions that arise at run-time. The distribution must balance the conflicting goals of *locality* (to reduce communication costs) and *parallelism* (to reduce total run-time).

This strategy divides the distribution problem into two subproblems: First, appropriately sized objects must be extracted at compile-time from a program's source code. Second, these objects must be mapped at run-time onto the available attraction memory sites. As such, SDAARC consists of two major technologies: compiler technology for the extraction of objects, and run-time technology for their distribution.

2.2. Hardware Requirements

SDAARC makes a small number of very basic demands on the hardware utilized to implement the system. As described above, the hardware is to be organized into a collection of *attraction memory sites*, where each site consists of one or more processors, a broker, a memory, and a network connection. Additionally, multiple sites can be clustered together, and represented by a *cluster broker*, to build hierarchical systems.

The brokers can be implemented in hardware, software, or both. Many different network topologies are possible. Examples include standard tree topologies and a novel crossbar topology described in (Moore et. al. 1998a).

The brokers together with the network are responsible for supporting the SDAARC strategy. Since the program together with its data are treated as a population of migratory objects, almost all messages sent on the network will be addressed to objects, and not to locations. Each message needs to be sent to all brokers whose sites *could* currently contain the recipient.

2.3. Compiler Requirements

SDAARC assumes that each program has been represented as a set of *microthreads.* Compiler technology for translating programs into threads is illustrated in figure 2.

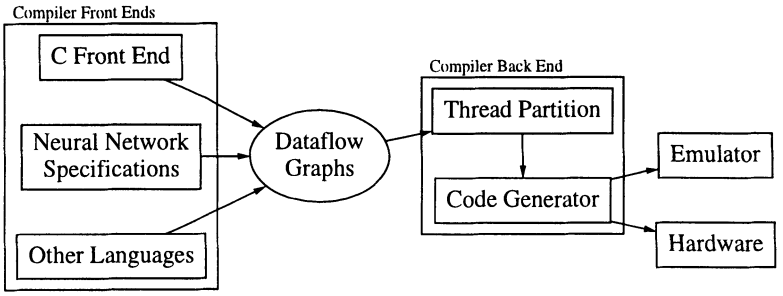

Figure 2: Compiler Technology for Representing Programs as Threads.

The compiler technology can be split into two groups: the front ends and the back ends. The fronts ends are responsible for translating a program into a dataflow graph. The choice of source language is left open to the programmer, and can be chosen according to which language best facilitates problem solving for a given application. Front ends for a subset of ANSI C and for a declarative neural network specification language are described in (Moore et. al. 1996 and 1997).

Dataflow graphs are chosen as an intermediate format for two reasons: first, to facilitate source language independence, and second, to represent all the parallelism implicit in the program.

The back end consists of two steps: First, the dataflow graph is partitioned into microthreads. Second, code is generated for each thread separately. While the code generation is done largely with conventional techniques, the graph partitioning receives closer examination below in section 3. Before closing this section however, we define the meaning given the term *thread* in this paper (since the term "thread" is unfortunately used with various, often contradictory meanings in the literature). We use the terminology from (Nikhil, 1993), where a *microthread* is an atomic subset of a dataflow graph, which never waits for data or an event. Should a *process* have to wait for some form of input, then that process is divided into (at least) two microthreads: one which performs the calculations up to the point where the input is necessary, and a second microthread which is triggered by the arrival of the input.

3. The Compiler Technology

3.1. Graph Partitioning Algorithm

The authors have implemented a prototype graph partitioning tool in order to evaluate the SDAARC approach empirically. The tool accepts dataflow graphs from the front ends described in (Moore et. al. 1996 and 1997), and assigns annotations to the vertices, assigning each vertex into a hierarchically organized tree of subgraphs. The algorithm proceeds in two passes:

In the first pass, nested control structures are identified and assigned to macrothreads. The result of this pass is a hierarchical graph partitioning tree, the leaves of which contain directed acyclic dataflow.

In the second pass, each directed acyclic subgraph is further partitioned into

microthreads. This is done, at the moment, with a greedy, bottom-up clustering algorithm which attempts (heuristically) to minimize a cost function. In the beginning of the second pass, each vertex is assigned to an individual microthread (each starting microthread thus has exactly one vertex). Pairs of microthreads are then selected and merged if and only if the merge reduces the cost, and introduces no cycles in the macrothread.

In the cost function, each vertex is assigned a static cost representing the time required for the computation, and each edge between two microthreads is assigned a cost representing an average of communication time and context-switch time. The cost of the subgraph is then the costs along the *critical path* within the subgraph.

The calculation of the critical path serves three purposes here: first, we select pairs of vertices to merge from the critical path. Second, having identified a pair for merging, we calculate a new critical path to obtain the new cost estimate. Finally, the critical path calculation determines whether the merge introduces cycles.

The partitioning tool is described in more detail in (Klang, 1997).

3.2 Initial Results

We show two examples of the results of the second pass: one regular and one irregular. For each example, we show two possible partitionings, one where we assume extremely optimistic (low) edge costs, and one where the edge costs are more realistic (higher).

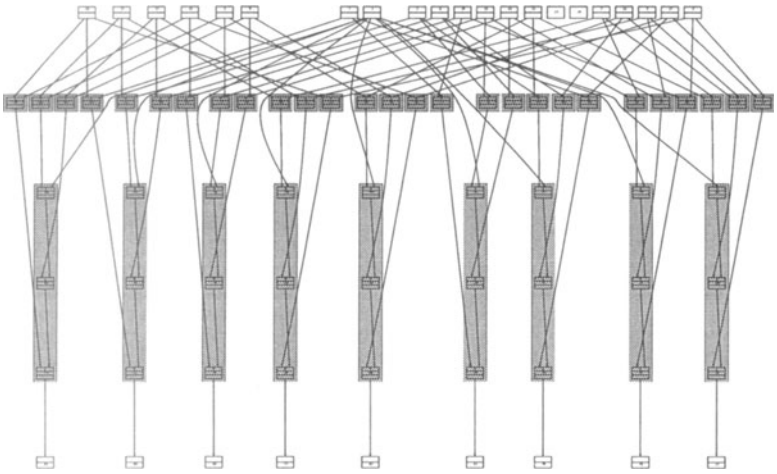

Figure 3: Partitioning a 3x3 matrix multiplication, assuming optimistic communication costs, results in 36 microthreads.

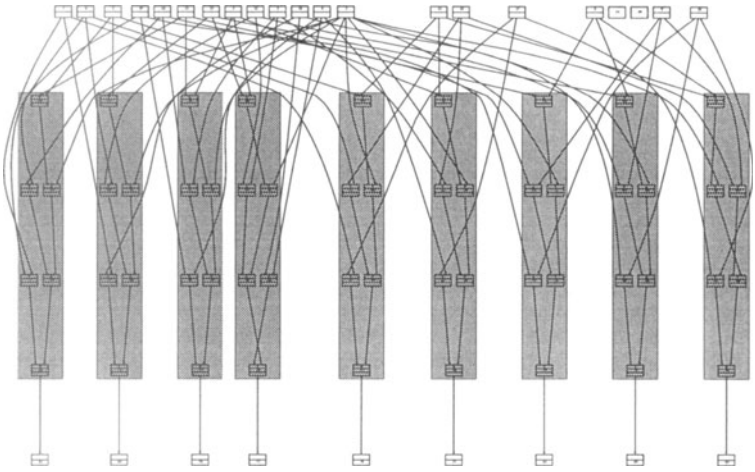

Figure 4: Partitioning a 3x3 matrix multiplication, assuming more realistic communication costs, results in 9 microthreads.

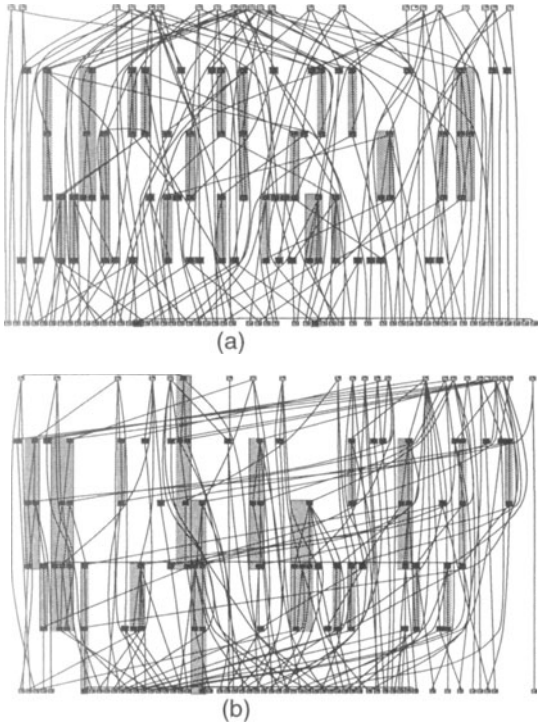

Figure 5: Partitioning a randomly generated dataflow graph with 160 vertices produces
(a) 52 microthreads when assuming optimistic costs, and
(b) 39 microthreads when assuming realistic communication costs.

The regular graph represents the calculations in a 3x3 matrix multiplication. This example has been chosen, not because it is necessarily typical, but rather because the structure is familiar, and illustrates the trade-off between parallelism and locality. When we assume optimistic edge costs, the graph is partitioned as shown in figure 3. Here, the partitioning algorithm has decided to obtain the maximum practical parallelism by performing all the multiplications in parallel, while the additions are merged into 9 microthreads.

In figure 4, the same graph has been partitioned under the more realistic communication assumptions. This time the algorithm has sacrificed some parallelism, and increased the locality. Here, each of the 9 results is calculated in a separate microthread.

Figures 3 and 4 demonstrate that the algorithm is capable of discovering and exploiting implicit structure in a regular dataflow graph. In order to see how the algorithm would deal with *irregular* graphs, we tested the partitioning algorithm with randomly generated dataflow graphs. In figure 5 we see the partitioning of a randomly generated 160 vertex dataflow graph. In figure 5(a) we again assumed optimistic communication costs, and the algorithm produced 52 microthreads. In figure 5(b) we assumed realistic communication costs, and the algorithms produced 39 microthreads.

4. Dynamic Distribution of Threads and Data

4.1 Adapting COMA for SDAARC

As discussed in Sections 1 and 2, SDAARC builds upon COMA technology. COMAs (e.g. Hagersten, 1992 or Saulsbury, 1996) implement a transparently distributed shared address space for parallel processors. Data blocks move from local memory to local memory in response to read and write operations. As in all cache technology, this strategy represents a gamble on locality: as long as the assumption of locality holds, the odds are good that the data needed at a site will be needed there again in the near future. When the assumption is not correct, data can be unnecessarily shuffled back and forth between sites (the so-called *ping pong* effect).

The first insight in SDAARC is that, if we could associate each thread with a data object, we could let the COMA distribute those objects amongst the attraction memory sites. We could then schedule tasks for each local processor corresponding to the thread-objects currently residing in its local memory. We do not have to search very far to find such objects: in the multithreaded architecture literature, for every thread there exists a *frame* which holds the arguments and return addresses for the thread. These frames closely resemble the *stack frames* in conventional (sequential) architectures, except that they are not organized in a stack. Instead, the frames are effectively contained in a distributed, common heap. Usually, several threads can share one frame (Nikhil 1993, Culler, 1993), but more recent literature suggests that performance enhancements can be obtained when each thread has a frame of its own, called a *framelet* (Annavaram and Najjar, 1996).

Having found a data object which represents each thread, and thus allowing the

COMA technology to be used to distribute *computation* as well as data, it soon becomes apparent that further optimizations are possible. We now have a system where not only data migrates to adapt to the program distribution, but where the program distribution simultaneously adapts to the data distribution. This provides a solution to the *ping pong* effect. In a conventional COMA, there is no effective way for data to adapt to a poorly distributed program: if a data block is utilized simultaneously in two sites, then the block needs to migrate back and forth between those sites. In SDAARC, it is possible for the competing portions of the computation to be redistributed to reside on the same site.

4.2. The Brokers and the ECOLI Cache Coherency Protocol

The brokers in SDAARC are an extension of the cache coherency controllers in the COMA literature. They are responsible for handling network traffic, for providing the processor with frames (threads) to execute, and with ensuring that the transparently distributed, shared address space remains consistent and coherent.

As in a standard COMA, the brokers enforce a *cache coherency protocol.* By combining computation and data, we have been able to significantly simplify the protocol compared to other COMAs. In our protocol (Moore et. al., 1988b), objects can be in one of five states: Exclusive, Cloned, Original, Leaving or Invalid (ECOLI). This can be compared to the standard SMP (non-COMA) MESI (Modified, Shared, Exclusive and Invalid) protocols, or the the 7 state COMA protocol in (Haridi and Hagersten, 1989).

The shared address space in SDAARC is separated into three partitions. As in a standard Harvard Architecture, there exists data and instruction partitions. However, we add a third partition for frames, to allow special optimized handling of these essential data objects. Further, as discussed in section 5, the instruction partition can also be extended to incorporate heterogeneous architectures.

4.3. Distribution Events

As explained above, the brokers must balance the conflicting demands of parallelism and locality (in order to reduce communication costs). To do this, they need to assemble collections of more or less connected threads and data objects. The goal is always to provide the shortest run-time, and not necessarily to provide an even load balance. Tasks should be distributed over only as many processors as they can efficiently exploit (especially in multi-tasking environments).

To accomplish this balancing act, we define a number of *forces* which drive the migration of objects and threads. The forces fall into two categories: *dissipative* forces drive objects apart, maximizing parallelism, potentially at the cost of locality; while *attractive* forces bring related objects together, maximizing locality, potentially at the cost of parallelism. These forces operate at precisely defined *distribution events.* When these events occur, the brokers make decisions about which objects migrate, and where they migrate to.

There are two dissipative distribution events:

1 We assume that the attraction memories are set-associative: Each object is statically mapped onto a *set* which holds a small, constant number (e.g. 4 or 8) of objects. When an object is inserted into a set, and that set is full, then some other object must migrate to make room

2 When the broker is idle, it can identify sets which are *almost* full, and spontaneously evict objects from these sets.

Similarly, there are two groups of attractive distribution events:

1 When one thread calculates an argument for another frame, and the receiving frame is resident on another site, then the receiving frame's broker can decide whether to simply accept the argument, or instead to send the receiving frame to the site sending the argument.

2 In SDAARC, data loads and stores always produce an acknowledgment, to enable barrier synchronization and mutex locking of data objects. When data is loaded or stored from the data partition, the broker can decide whether to send the entire data block to the site waiting for the acknowledgment.

5. Extensions for Embedded Systems

5.1. Extending Dynamically Reconfigurable Hardware

As mentioned above, the SDAARC address space contains a data partition, a frame partition and an instruction partition. We have until now tacitly assumed that the instructions can be used on all the processors. This need not be the case. If the architecture contains heterogeneous processors, the instruction partition can contain instructions for each processor type present. Of course, the compiler technology will have to be enriched with new code generators each time a new processor type is added to the system. Those programs which wish to exploit the new processor type will need to be recompiled. Recompilation will only be necessary when new processor types are added (changing the *number* of processors does *not* force recompilation), and only for those programs which need to exploit the new technology.

Perhaps the most interesting possibility is to incorporate not only instruction-driven processors, but also dynamically reconfigurable processors. Modern FPGAs store configurations in RAM, and FPGAs are now coming to market which can store multiple configurations on chip, and have configuration times on the order of (or less than) 1 millisecond. The best way to integrate this interesting new performance potential into conventional computing systems is the object of intensive research. Dynamically configurable logic has the same programmability problem as parallel processors, but in a much more extreme form. The performance potential is clear; what is unclear is the best way to bring applications to the hardware. Configuration times under 1 ms are too fast to be used only for rapid prototyping (with reconfiguration performed every few days or weeks), but still too slow to be performed, for example, at every function call in a conventional program (see

Villasenor and Mangione-Smith, 1997, Page, 1998, or Brebner, 1998).

A SDAARC system could be extended with sites consisting of a (specialized) broker, optional RAM and one or more FPGA's. We then view these FPGAs as both processors and, simultaneously, as *instruction caches.* The broker's decision to accept storage of a frame is thus equivalent to the decision to store a block of instructions in instruction cache. As with all cache logic, storing data (i.e. reconfiguring the FPGA) is a gamble on sufficient locality. We effectively bet that this configuration will also be useful for other frames in the immediate future. This provides an automatic, dynamic and adaptive way of deciding when to reconfigure and which reconfiguration to perform.

5.2. Extending SDAARC for Real Time Systems

Another fascinating potential for extending SDAARC lies with embedded systems. SDAARC could be useful either as a simulation platform or even as a target platform for embedded systems.

SDAARC as a simulation platform is attractive if we extend the concept to allow some objects (frames or data objects) to be *fixed,* that is, labeled as non-migratory. (This will be necessary in any case to allow the programming of device drivers). The designer of a distributed embedded system could then initially leave the distribution of the objects unspecified, use SDAARC to see which distributions arise naturally at run-time, and then tie objects to sites, arriving iteratively at a static distribution. Alternately, if SDAARC is also to be used as a target platform, the designer could leave some or all of the system migratory, depending on the criticality and performance of the simulation.

If SDAARC is to be used as a target platform for real time systems however, the scheduling decisions made in the brokers will need to respect *timing constraints.* To support these decisions, the compiler technology will need to be extended to produce *annotated* dataflow graphs, allowing the user to specify, and later the broker to respect, this timing information. Much research has been done in the field of real time operating systems (e.g. Ditze, 1995) and development environments (e.g. Kleinjohann et. al. 1997) which can be leveraged to extend SDAARC in this direction.

Of course, no target architecture can always guarantee that timing constraints will be met. Further since a certain degree of non-determinism is inherent to SDAARC's adaptivity, SDAARC will occasionally fail to meet timing constraints which other systems might have met. Nonetheless, many embedded systems include both critical and less critical subsystems, and the critical systems often leave significant resources unused, which can be assigned to the less critical systems in a dynamic and adaptive fashion. A similar approach was taken in, for example (Grewe et. al., 1998).

6. Conclusion

In this paper we have introduced a hybrid system for partitioning and distributing data and computational load in a parallel and distributed system. Fine-grain decisions are made at compile-time: A program is translated into a dataflow graph, from which

microthreads are extracted using a heuristic clustering algorithm. Coarse-grain decisions are made at run-time: Each processor is augmented with a *broker* whose job it is to: first, enforce a cache coherency protocol which implements a transparently distributed shared address space; second, select from the local portion of a shared collection of *frames* those frames which represents threads suitable for the local processor, based on dynamic load information.

This system, called SDAARC (Self-Distributing Associative ARChitecture), is an extension of COMA (Cache Only Memory Architecture) techniques. Whereas COMAs are content to offer a Distributed Shared Memory, SDAARC goes further and incorporates automatic distribution of computation.

SDAARC becomes interesting for embedded systems when extended in (one or both of) two directions: First, since the SDAARC concept is hardware-independent, it can be used to integrate heterogeneous processor technologies. In particular, dynamically reconfigurable logic can be incorporated, providing an adaptive automatic method for selecting the configurations for FPGAs. Second, if SDAARC's dataflow graphs are annotated with timing constraints, techniques from real time operating systems can be utilized to obtain a simulation architecture which evolves seamlessly into a target platform for distributed embedded systems.

Acknowledgments

This work was supported in part with funds of the Deutsche Forschungsgemeinschaft under reference number WA 357/11-2 within the priority program "System and Circuit Technology for Massively Parallel Computation"

References

Annavaram, Murali and Najjar, Walid (1996), *Comparison of Two Storage Models in Data-Driven Multithreaded Architectures,* Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP), New Orleans, LA., Oct. 1996, IEEE Computer Society Press, 122-129.

Brebner, Gordon (1998), *Field-Programmable Logic: Catalyst for New Computing Paradigms,* Proc. 8th International Workshop on Field-Programmable Logic and Applications (FPLA '98), Springer Verlag.

Culler, David E., Goldstein, Seth Copen, Schauser, Klaus Erik and von Eicken, Thorsten (1993) *TAM - A Compiler Controlled Threaded Abstract Machine*, Journal of Parallel and Distributed Computing, Special Issue on Dataflow, June, 1993.

Ditze, Carsten (1995), *DReaMS - Concepts of a distributed real-time management system*, Proc. 1995 IFIP/IFAC Workshop on Real-Time Programming.

Grewe, Claus, Obelöer, Wolfgang and Pals, Holger (1998), *Kombination von anwendungs- und systemintegrierter Lastverwaltung mit Mobilen Agenten*, Proc. 17. PARS Workshop, Karlsruhe, 16.-17. September, PARS Mitteilungen Nr. 17, 98-107.

Hagersten, E., Landin, A. and Haridi, S. (1992), *DDM - A Cache-Only Memory Architecture*, IEEE Computer, 25(9), 44-54.

Haridi, Seif and Hagersten, Erik (1989), *The Cache Coherence Protocol of the Data Diffusion Machine*, Proceedings of the PARLE 89, Vol. 1, Springer-Verlag, 1-18.

Klang, Melanie (1997), *Hierarchische Zerlegung von Datenflußgraphen für mehrfädige Architekturen,* Diplomarbeit, Fachbereich Informatik, J. W. Goethe-Universität, Oct. 1997.

Kleinjohann, Bernd, Tacken, Jürgen, and Tahedl, Christoph (1997), *A Design Environment Using High-Level Petri-Nets as Common Model for Specification, Analysis, and Validation of Hybrid Real-Time Systems,* 3. ITG/GI/GMM Workshop Hardwarebeschreibungssprachen und Modellierungsparadigmen, Holzau (Germany), Febuary 1997.

Moore, Ronald, Zickenheiner, Stefan, Klauer, Bernd, Henritzi, Frank, Bleck, Andreas, and Waldschmidt, Klaus (1996), *Neural Compiler Technology for a Parallel Architecture,* International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '96), Sunnyvale, CA, Aug. 1996.

Moore, Ronald, Klauer, Bernd and Waldschmidt, Klaus (1997), *Compiler Technology for Two Novel Computer Architectures*, 14th ITG/GI-Fachtagung Architektur von Rechensystemen (ARCHS '97), Rostock, Germany, Sept. 1997.

Moore, Ronald, Klauer, Bernd and Waldschmidt, Klaus (1998a), *A Combined Virtual Shared Memory and Network which Schedules*, International Journal of Parallel and Distributed Systems and Networks, 1(2), 51-56, 1998. Originally appeared at the International Conference on Parallel and Distributed Systems (Euro-PDS '97), Barcelona, June, 1997.

Moore, Ronald, Klauer, Bernd and Waldschmidt, Klaus (1998b), *Automatic Scheduling for Cache Only Memory Architectures*, Third International Conference on Massively Parallel Computing Systems (MPCS '98), Boulder, Colorado, April, 1998.

Nikhil, Rishiyur S. (1993), *A Multithreaded Implementation of Id using P-RISC Graphs*, Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon, Aug. 1993, Springer Verlag LNCS 768, 390-405.

Page, Ian (1998), *Design of Future Systems,* Proc. Design Automation and Test Europe (DATE '98), IEEE Press, 343-347.

Patterson, David A. (1997), *Microprocessors in 2020*, Scientific American Special Issue: The Solid-State Century, 8(1), 86-88.

Saulsbury, Ashley, Wilkinson, Tim, Carter, John and Landin, Anders (1996), *An Argument For Simple COMA*, First IEEE Symposium on High Performance Computer Architecture, Rayleigh, North Carolina 276-285.

Villasenor, John and Mangione-Smith, William H. (1997), *Configurable Computing,* Scientific American, June, 1997.