Despite more than 50 years of research, parallel computing has been accepted in only a few specialized areas, such as scientific computing and network servers. It has become increasingly clear that the main barrier to wider acceptance is the difficulty inherent in parallel programming. Computations and data must be distributed not only in time but also in space. A system that could automatically distribute computations without requiring explicitly parallel programming would make parallel computers practical in a much wider range of application domain.

Warren and Haridi introduced an approach for automatic data distribution within a distributed shared memory (DSM). In their approach, as Figure 1 shows, a cache-only memory architecture (COMA) consists of several interconnected processors. The processors’ local main memories have been augmented with cache controllers so that they function as large caches. To avoid terminological confusion, Warren and Haridi call these caches attraction memories. A cache-miss resolution mechanism implements the COMA's DSM, resolving an attraction memory miss in one processor of a COMA ensemble by issuing requests to the other processors’ attraction memories. Special cache coherence protocols serve to avoid conflicts. Thus, COMAs provide a comfortable migration mechanism for data. Data blocks (cache lines) migrate to processors where they are needed.

Despite automatic data migration, activity module programming remains explicitly parallel. The programmer is responsible for partitioning and distributing activity modules among the processors.

The self-distributing associative architecture (SDAARC) that we describe in this article is based on the COMA concept but extends the data migration mechanisms with migrating microthreads and a scheduling concept. A microthread is a sequential piece of code with a computable maximum execution time. The SDAARC scheduler treats microthreads as nodes in a coarse-grain dataflow graph. The results of one microthread represent the arguments of other microthreads. The exchange of these values between microthreads, along with the use of random-access data containers, causes microframes and data blocks to migrate into momentarily stable configurations that balance the competing needs of locality and parallelism.

This code and data migration mechanism requires an automatically parallelizing compiler that translates legacy C programs into migratory populations of microthreads and data blocks. This approach to automatic parallelization is applicable to tightly coupled...
multiprocessors, locally connected computer clusters, and globally connected computers (for example, the Internet).

SDAARC

While other discussions of SDAARC appear in the literature, here we present the architecture in a top-down fashion. We consider it first as a complete system consisting of compiler and runtime technology, then we examine the compiler technology and the runtime system in greater detail. The “Related work” sidebar identifies other work in this field.

Hybrid scheduling approach

Our main goal with SDAARC is to balance the competing demands of parallelism and locality. Spreading a program across all available processors maximizes parallelism. Assigning strongly interdependent portions of a program to the same processor maximizes locality, thus minimizing the need for communication. Optimizing performance requires

Figure 1. A cache-only memory architecture consisting of interconnected processors.

Related work

The basic scheduling terminology (microthreads, micro-frames, and so on) used in this article comes from the multithreaded architectures, in particular from P-RISC and TAM. COMA-based scheduling appears unique to SDAARC. Other COMAs don’t address program partitioning; they simply provide a distributed shared memory, relying on the (static or dynamic) scheduling in explicitly parallel programs. Warren and Haridi provide a good example. Mueller, Stallard, and Warren use multithreading with a COMA, but again only to accelerate programs that have already been partitioned.

SDAARC can also be compared with other systems for automatic scheduling. Some automatic distribution schemes concentrate on scheduling loops and arrays. In contrast, our proposal can also handle irregular problems and dynamic data structures. Moreover, it’s better at adapting to dynamically changing loads on parallel machines shared by multiple users and/or multiple tasks.

The scheduler closest to SDAARC is probably that used with Cilk. Indeed, our scheduling of the microthreads at each site is based in part on Cilk. However, Cilk is an explicitly parallel language.

An excellent review of various activities in and around Internet computing appears in The Grid: Blueprint for a New Computing Infrastructure.

References

finding a compromise between parallelism and locality. SDAARC addresses this problem with a two-stage approach. First, at compile time, static analysis identifies strongly connected computations, which the compiler backend then packs together into microthreads. (Our use of the term microthread is inspired by, but not identical to, the microthread concept defined by Nikhil.4) Second, at runtime, dynamic analysis maps the microthreads onto the available processors so that clusters of related microthreads remain together, thus minimizing communication costs.

The microthreads represent the finest level of granularity available for distribution across the processors. SDAARC exploits only interthread (not intrathread) parallelism. In other words, fine-grain scheduling decisions occur at compile time, while coarser granularities emerge at runtime.

Compiler technology

SDAARC doesn’t require (or for that matter allow) explicitly parallel constructs in the source program. The SDAARC compiler analysis starts with conventionally programmed sequential source code in established languages such as C or Fortran. Standard compiler front ends5 convert the source code into control dataflow graphs, as shown in Figure 2.

The dataflow graphs express all the implicit parallelism the compiler front ends could find, without regard to communication costs. As a first step toward balancing locality and parallelism requirements, SDAARC partitions the dataflow graphs into coarser grains—that is, microthreads consisting of several instructions. A first partitioning occurs along control structures. Each loop iteration and each function call, for example, represents at least one microthread. All but the smallest if-then-else constructs also induce microthread partitioning. Furthermore, accesses to global random-access data structures (arrays, and so on) as well as input and output induce microthread partitioning.

When a value is loaded (from an array, for example), the load is initiated in one microthread; another microthread starts once the value arrives from the data memory.

Thus, after this first partitioning, all microthreads are represented by acyclic subgraphs, and each microthread has a well-defined maximum runtime. Optionally, a bottom-up greedy algorithm described by Moore et al.6 can further partition these microthreads.

After compilation, the program is represented as a population of migratory microthreads and container objects. These objects have not yet been mapped onto the available resources. (Indeed, the compiler does not yet know how many processors are available.) This distribution occurs dynamically at runtime.

Runtime system

Limited hardware assumptions, adaptive scheduling, and specially designed protocols are key to SDAARC’s operation.

Hardware assumptions. SDAARC makes very few assumptions about the available hardware so that it can target as large an application field as possible. A SDAARC installation includes a certain number of sites and a network that allows direct or indirect messaging between any two sites. Each site consists of a processor with local memory, and a cache coherence controller. The controller can be implemented in hardware, software, or both. This description applies to all COMAs (see Figure 1).

Because the cache controllers have more responsibilities in SDAARC than in other COMAs, we call them brokers; this emphasizes their role in choosing, or shopping for, computation to be performed at their sites.
Adaptive scheduling. A COMA distributes data by moving copies of data blocks to the sites that access those blocks. If we could associate a data object with each microthread, a COMA could automatically distribute those data objects as well.

Data objects aren't hard to find. In the original literature on multithreaded architectures, we see that each microthread is conventionally associated with a microframe.\textsuperscript{4,7} Usually, several microthreads can share one microframe, but research suggests that letting each microthread have its own microframe has significant advantages. A microthread’s microframe stores its arguments and the addresses of the microthreads that consume its results. A Harvard architecture separates the instruction memory from the data memory, but SDAARC introduces a third memory partition optimized for microframes. One COMA cache coherency protocol governs the content migration of all three memories.

Finally, we stipulate that microthreads execute at the sites where their microframes reside (when the microframe finally contains all the data the microthread requires). Thus, microframe migration determines the mapping of the computation (the microthreads) onto the available processors. A conventional COMA can only move data to the computations that use it, but SDAARC also moves computations to the data they use and—because of data dependencies—to each other.

Ecoli protocols. Microframe migration drives scheduling in SDAARC. It remains to be seen how the brokers and their cache coherency protocols can implement microframe migration. Moore, Klauser, and Waldschmidt describe a snooping cache coherency protocol for SDAARC with a hierarchical bus network,\textsuperscript{5} and a directory-based protocol for SDAARC with point-to-point networks.\textsuperscript{2} These two protocols have much in common. For example, both allow copies of microframes and data containers to be in one of five states: exclusive, clone, original, leaving, and invalid (jointly called Ecoli). These states are similar to, but somewhat simpler than, states in other COMA protocols.\textsuperscript{8} An object is in the exclusive state in a certain attraction memory only if no copy of this object exists in any other attraction memory. If another site needs the object, that site gets a copy of the object. The copy is then in the clone state, and the former exclusive instance changes into the original state. Differentiating originals and clones is necessary to avoid accidental removal of all copies of an object. When a microframe has received all needed arguments, its state becomes leaving, because it is ready to be executed. The invalid state identifies objects having no valid information and means not present or removable.

The main difference between SDAARC and other COMAs lies in the transactions the protocol allows. While other protocols allow load and store transactions (either blocking or split phase), SDAARC allows only apply transactions, where results from one object (microframe or data container) are sent as arguments to another.

The simplest form of application in SDAARC is one microframe receiving the results of another microframe’s microthread. Specialized apply transactions provide random access to data containers. For example, two split-phase operations replace a load operation. In the first phase, a microframe sends a data container an offset, the addresses of the microframes that are waiting for the value loaded from that offset, and the addresses of any microframes waiting for the load to complete before proceeding (for synchronization purposes). In the second phase, the data container sends the value at the offset to the recipient microframes and sends synchronization tokens to any waiting microframes. Similarly, a store operation has a first phase in which the data container receives an offset, a new value, and the addresses of any microframes waiting for the store to complete before proceeding, and a second phase in which the container object sends synchronization tokens to the waiting microframes.

The protocols also let migration piggyback on these basic transactions. Just as a conventional cache obtains an entire cache line from the backing memory on a cache miss—even though the CPU has only requested a word—SDAARC sometimes sends an entire microframe or container object block to the transaction origination sites. In this way, related microframes and container blocks converge. This approach optimizes locality at the expense of parallelism.

Conversely, objects can move from overloaded to underloaded sites, causing overpop-
ulations of objects to diverge and optimizing parallelism at the expense of locality. To do this, all sites must have relatively up-to-date load information from at least some of their neighbors. The sites communicate by appending load information to every transaction on the network and by exchanging “idle” messages.

Implementation

We suggest three implementation paradigms for SDAARC: a monolithic system on a chip, an intranet solution, and an Internet solution.

System on a chip

Proper clock signal distribution is increasingly difficult in modern chip designs. One approach is to construct a multiprocessor on a chip, allowing each small processor its own clock. Thus, it’s possible to implement several smaller processors with attraction memory and a cache controller—interacting asynchronously with high bandwidth—on a single chip.

The concept of associativity is the greatest beneficiary of SDAARC on a chip. In this implementation, we define the network as a set of nodes containing information in the catalog memories about all data objects of the corresponding processor (PE, in Figure 3a). Using associative lookups, the nodes can determine whether their corresponding PE hosts the

![Figure 3. SDAARC on a chip (a) enhances associativity and permits scalability (b).](image-url)
requested data. If it doesn’t, they can route the request to a node that does contain the requested data item. Scalability results from assigning one of the several processor-nodes on a chip to communicate with other chips. Larger SDAARC clusters can then be hierarchically constructed as shown in Figure 3b.

Although a SDAARC implementation on a chip has attracted interest, especially in the field of parallel or cooperating embedded systems, research in this area lags because of the need for special equipment and specially trained personnel.

Intranet implementation

SDAARC is most easily implemented on an intranet. Because of the assumption that all the machines are available for use, we ignore here the subject of authorization. Also, if we assume a common (virtual) file system, such as SunSoft’s Network File System, we needn’t implement the DSM’s instruction partition, because presumably all processors have equal access to the executable code. The basic COMAs’ attraction memories are, in this case, the machines’ main memories. The brokers (cache coherency controllers) are implemented in software.

More precisely, all COMA extensions concerning the microthreads are implemented as a standard SDAARC kernel containing both a scheduler and the broker. The SDAARC compiler first translates the application program into a set of microthreads, and the linker links together the executable code for these microthreads with the standard kernel to create the finished application. Because this approach resembles the concept of a compiled-in simulation, we call it a compiled-in application. Figure 4 illustrates the complete flow.

The user starts the first copy of the application on any machine in the intranet. The program then starts copies of itself on all the other machines. Next, the kernel distributes the microthreads and container objects across the network as necessary. Small applications use only a portion of the available intranet.

The intranet version is currently the focus of our investigations, and the SDAARC kernel is under test.

Internet implementation

The possibility of SDAARC on an intranet leads naturally to the question of whether the architecture could be used on the Internet, the largest available concurrent computing resource. Because SDAARC allows for a hierarchical organization of the participating computing resources, this concept could extend to a coupling of the intranet and Internet implementations.

However, moving to the Internet raises several new problems. For example, with an Internet-related implementation, we can no longer assume a common virtual file system. Furthermore, owners of computing resources will accept Internet-based SDAARC clusters only if they can add and remove their machines to and from those clusters at any time. Security would be essential so that cluster components wouldn’t be subject to abuse or unauthorized use. Fortunately, researchers have already addressed most of these issues, and many of the solutions can be adapted for SDAARC.

Another important area will be fault tolerance, where research so far has focused on checkpointing.10 Suitably redesigned versions of the SDAARC Ecoli coherency protocols could offer checkpointing, redundant computation, and redundant storage.
Empirical results

Figures 5 and 6 present the first empirical measurements of SDAARC’s efficiency. The hardware platform for these tests was a cluster of four identical PCs, each with a 350-MHz Intel Pentium II CPU and a 64-Mbyte main memory, running Linux. A 100-megabit-per-second Fast Ethernet connected the PCs to a simple hub.

For the test program, consisting essentially of the C code shown in Figure 7, the system multiplied two square matrices of a given dimension and stored the results in a third matrix, producing the measurements shown in Figures 5 and 6.

We repeated the program three times for each of the matrix dimensions, 40, 44, 48, ..., 64. The measured times appear as a scatter plot in Figure 5. The lines connect the averages for the different matrix dimensions. The test program operates at a very fine-grained level—with no optimizations by, for example, loop unrolling.

The runtime decreases with an increasing number of sites. Figure 6a shows the speedup, which exceeds 3.0 with four sites and matrices of dimension 60 or greater.

Figure 6b shows the efficiency, which decreases with an increasing number of sites (as expected) but stays consistently between 62 and 87 percent. The smallest measured speedup with four sites was 2.37 (for matrix dimension 40), for an efficiency of 59 percent. The largest measured speedup with four sites was 3.11 (for matrix dimension 64), for an efficiency of 78 percent.

The results presented here do not yet prove that SDAARC is better than a conventional COMA. We believe SDAARC
can demonstrate its superiority, but not with the test applications this article discusses; matrix multiplication is exactly the kind of application that conventional COMA does well. We didn’t intend these tests for qualitative comparison but rather for studying SDAARC’s behavior on a simple, well-understood, yet not entirely artificial application. We remain confident that new test applications—using, for example, standardized benchmarks like SPEC2000—will produce empirical results that quantify the qualitative differences between SDAARC and other parallel computing systems. However, much work remains before such benchmarks will compile and run with speedup on SDAARC.

Moreover, we programmed this SDAARC simulator as a research vehicle, not as a production-quality runtime system. Making the system parameterizable, transparent, and flexible was a greater priority than raw performance. We expect to demonstrate much better efficiency once SDAARC is optimized and fine-tuned. For example, optimizing microthread granularity and using more-intelligent scheduling algorithms hold great potential.

We are testing the intranet-oriented version of SDAARC to increase stability and investigate how machine speed, communication latency, and microthread granularity relate to each other. A new compiler for automatically generating the SDAARC-oriented threaded code, using the front ends reported by Hall et al., is running and has passed its first tests.

We are also developing an Internet version of SDAARC, but we don’t expect testing in the near future.

Acknowledgment

The Deutsche Forschungsgemeinschaft supported parts of this work.

References

Frank Eschmann is a research assistant in the Technical Computer Sciences Department at J.W. Goethe University in Frankfurt, Germany. His research interests include parallel computer architectures and automatic parallelization methods. Eschmann has an MSc in computer science from J.W. Goethe University.

Bernd Klauer holds a postdoctoral position in the Technical Computer Sciences Department at J.W. Goethe University. His research interests include parallel computer architectures and automatic parallelization methods. Klauer has an MSc and a PhD in computer science from J.W. Goethe University. He is a member of the German National Association for Computer Sciences and the German Electrical Engineering Council.

Ronald Moore is an assistant to the CEO/CTO at IS Innovative Software. His research interests include parallel computing and complex adaptive systems. Moore has a BSc from Michigan State University and an MSc and a PhD in computer science from J.W. Goethe University.

Klaus Waldschmidt heads the Technical Computer Sciences Department at J.W. Goethe University. His research interests include parallel computer architectures and CAD for mixed analog and digital circuits. Waldschmidt has a Dipl.-Ing. and a Dr.-Ing. in electrical engineering from the Technical University of Berlin. He is a member of the Society for Information Technologies, the German National Association for Computer Sciences, and Euromicro.

Direct questions or comments about this article to Frank Eschmann, Box 11 19 32, D-60054 Frankfurt, Germany; eschmann@ti.informatik.uni-frankfurt.de.