The Illinois Aggressive Coma Multiprocessor Project (I-ACOMA)[1]

Josep Torrellas and **David Padua**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
torrella,padua@csrd.uiuc.edu
http://www.csrd.uiuc.edu/iacoma

Abstract

While scalable shared-memory multiprocessors with hardware-assisted cache coherence are relatively easy to program, if truly high-performance is desired, they still require substantial programmer effort. For example, data must be allocated close to the processors that will use them and the application must be tuned so that the working set fits in the caches. This is unfortunate because the most important obstacle to widespread use of parallel computing is the hardship of programming parallel machines.

The goal of the I-ACOMA project is to explore how to design a highly programmable high-performance multiprocessor. We focus on a flat-coma scalable multiprocessor supported by a parallelizing compiler. The main issues that we are studying are advanced processor organizations, techniques to handle long memory access latencies, and support for important classes of workloads like databases and scientific applications with loops that cannot be compiler-analyzed. The project also involves building a prototype that includes some of the features discussed.

1 Introduction

Scalable shared-memory multiprocessors promise to deliver large-scale computing power without surrendering much programmability. These machines are relatively easy to program because they eliminate the need to cross physical address spaces to perform interprocessor communication. A subclass of these machines, namely multiprocessors with hardware-assisted cache coherence, further simplify the programmer's task by guaranteeing the coherence of different cached copies of a variable without software involvement.

It can be argued, however, that if truly high performance is desired, hardware-coherent shared-memory machines still need substantial programmer effort. This is a result of the increasing complexity and physical distribution of the memory hierarchy. For example, in a shared-memory machine, it is important that data that will be accessed mostly by a given processor be allocated in memory modules close to that processor. In this way, if the processor's caches overflow, there is no need to issue a remote request to reload the data; a local memory access suffices. Similarly, it is important to tune the application so that the working set fits in the caches. This will also minimize the chances of having to issue requests to remote memory modules. The overall consequence is that, after an application is considered to be fully debugged, programmers are still likely to spend a considerable effort tuning the code for performance.

This is unfortunate since, in our opinion, the most important obstacle to the widespread use of parallel computing is the hardship of programming parallel machines. If scalable, truly programmable parallel machines became available, science and society would change in ways that we can only begin to imagine.

The goal of the I-ACOMA project is to explore design alternatives for a highly programmable high-performance multiprocessor. Given the chief importance of programmability, we focus on cache-only memory architectures (COMAs) and their use by a parallelizing compiler. Roughly speaking, comas [5, 6, 11, 21, 22] have a more complicated hardware than traditional cache-coherent shared-memory machines, but are easier to program. Indeed, in a coma, data automatically migrates to the local memory of the processor that is using it. Such memory is called *attraction memory* in coma machines. As a result, the actual initial placement of the data does not have much impact on the execution time of an application and there is not much need to worry about it. In fact, ideally, we should be able to take a program written for a bus-based machine, run it on a scalable coma machine without changes in the data layout, and still achieve high performance [5]. Similarly, it can be argued that the need to tune the size of an application's working set to the caches of the machine is not as pressing. This is because in coma, even if the caches overflow, the data is still likely to be in the local attraction memory. All these issues make coma architectures more programmable.

Traditional coma machines like KSR1 [6] and DDM1 [5], however, had at least one shortcoming. To help a processor find a memory line efficiently, the networks of these machines were organized in a hierarchy. Specifically, KSR1 had a hierarchy of rings while DDM1 had a hierarchy of busses. Such organization, called Hierarchical Coma [22] made it possible to search for a memory line efficiently, but it slowed down remote accesses relative to non-coma machines and complicated the machine. A second problem of the KSR1 was that memory was allocated in page-sized chunks in the attraction memories. For example, when a node missed on a memory line that existed somewhere else in the machine and no other line of the same page was present in the node's attraction memory, a page-sized chunk of physical memory was allocated in the node's attraction memory. Then, only the requested line was actually loaded into the node's attraction memory; the rest of the page was left unused. This physical memory fragmentation caused the displacement of many lines from the attraction memory and, as a result, eventually increased the number of attraction memory conflicts.

These two problems do not exist in I-ACOMA because the machine is organized as a Flat Coma [22]. Flat coma is different from hierarchical coma in that each memory line has a node that acts as its home. The home contains the line's directory entry. Unlike in non-coma machines, however, the home for a line is not guaranteed to keep a copy of the line or even space for it. The line may have migrated

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and RIA MIP 93-08098; ARPA Contract No. DABT63-95-C-0097; and NASA Contract No. NAG-1-613.

from the home and another line may have been brought in to use the space. The advantage of flat coma is that, while memory lines are allowed to migrate freely, they can still be located quickly without a slow and complicated hierarchical network. Indeed, the directory in the home node of the line keeps a record of where the line is. Consequently, transactions that want the line only need to check the directory in the home node. Finally, the memory fragmentation problem described above is not present in flat comas because memory space is allocated on a line by line basis.

This paper is organized as follows. Section 2 describes the I-ACOMA architecture with emphasis on the main issues, Section 3 describes the compiler efforts, and Section 4 discusses the implementation of the prototype.

2 The I-ACOMA Architecture

I-ACOMA is a scalable shared-memory multiprocessor organized as a flat coma [22]. Each node in the machine is envisioned to be a small bus-based multiprocessor with a portion of the memory of the machine. Such design in clusters helps keep the cost of the machine down. The nodes are interconnected in a 2-D mesh for ease of implementation. Caches are kept coherent in hardware with a vanilla directory-based cache coherence protocol. This approach significantly simplifies the programming of the machine in most cases. The directory-based cache coherence protocol is based on limited bit vectors and is distributed among the different nodes. The main issues in the machine are the processor organization, techniques to handle long memory access latencies, and support for important classes of workloads like scientific applications with loops that cannot be compiler-analyzed and databases. We consider each issue in turn.

2.1 Processor Organization

The processors used in I-ACOMA are simultaneous multithreaded (SMT) processors [24]. A SMT processor is a wide-issue superscalar processor with multiple threads running at the same time such that, in a given cycle, the processor can issue instructions from different threads. This approach delivers higher performance than aggressive implementations of single-threaded superscalar processors. Indeed, the performance of single-threaded superscalars is limited by insufficient instruction-level parallelism extracted by the compiler and by frequent processor stalls due to long-latency memory accesses. SMT processors are also likely to deliver higher performance than chips with multiple processors per chip. This is because in the latter organization, a single sequential application is not able to utilize the full issue capabilities of the chip. In addition, the performance of systems with multiple processors per chip often suffers in parallel applications that are memory-intensive or have load imbalances.

The processors have 8 threads and are capable of issuing up to 8 instructions per cycle. To simplify the architecture, we use an in-order issue processor. Each thread has a separate program counter, instruction buffer, register file, and scoreboard to keep track of outstanding results. In each cycle, a primary thread is selected in a round-robin fashion among the threads. This thread issues as many instructions as possible from its instruction buffer provided all dependences are satisfied and the appropriate functional units are available. If less than 8 instructions are issued, we proceed to the next thread and try to fill the remaining slots by issuing instructions from its buffer. This continues until 8 instructions are issued or all threads have been inspected. With this approach, a sequential application executing on an unloaded processor can still issue up to 8 instructions per cycle.

It is expected that, in a few years' time, state-of-the-art microprocessors will be cycling at several GHz. Under such frequences, it is important for the CPU to have a high-bandwidth low-latency path to the memory subsystem. Current system designs usually include one or more levels of caches on chip, a large off-chip cache and, finally, DRAM memory connected to a bus. While the resulting slow and narrow paths between the CPU and the memory subsystem may be tolerable for current processors, they are unlikely to be so for future ones. It is therefore important to increase the integration between processor and memory. Ultimately, this integration will lead to putting processor and memory in the same chip [8, 20].

It is currently impossible to integrate in the same chip a processor as complicated as the one used for I-ACOMA with enough DRAM memory to be useful. However, as DRAM technology advances and denser DRAMs become available [1], this will change. Since it is especially interesting to integrate processor and memory in a chip in the context of a coma machine, we are actively evaluating such a configuration for a possible future I-ACOMA machine. Indeed, there are two reasons why chips with integrated processor and memory are more useful as nodes for coma multiprocessors than for non-coma multiprocessors. First, in coma, when a request issued by the processor misses in the cache, it directly accesses the local attraction memory. In non-coma multiprocessors, instead, after the cache miss, the request must be sent to the home node of the requested data for a directory lookup. Consequently, non-comas usually waste the benefits of a higher integration. The second reason has to do with the volume of traffic to local memory. A larger fraction of the cache misses are satisfied by the local attraction memory in coma than in non-coma multiprocessors. This is a consequence of the fact that data in a coma machine automatically migrates to the attraction memory of the node that is using it. As processors and memories become more integrated, therefore, coma machines stand to gain more.

2.2 Handling Memory Access Latencies

Using such powerful processors for I-ACOMA will not be productive if they often stall waiting for the memory system to supply data and instructions. Typically, this problem is alleviated by the presence of the coherent caches. However, coherent caches alone are not likely to be enough for processors running at multi-GHz frequences. For this reason, I-ACOMA provides two supports that address this problem. The first one is the coma property that data migrates to the processors that use it, therefore minimizing the cost of conflict-induced cache misses. The second and most important support is a set of primitives targeted to the hiding of the latency of coherence misses. These primitives are data prefetching and several flavors of writes, namely write-forward, write-broadcast, write-update, and write-through. They can also hide other types of misses beyond coherence misses.

When a processor issues a write-forward to a word, the word is updated in the processor's cache and a copy of the updated cache line is sent to the caches of a set of processors specified as a bit map in the forwarding register [15]. The forwarding register is set by the compiler or by the programmer prior to the write-forward instruction. It identifies the processors that are likely to be consumers of the write. As a result, when the consumer processors finally access the word, they are likely to find it in their caches. Note that these consumer processors may or may not be current sharers. The current sharer processors that are not specified in the forwarding register are invalidated. Write-forwards are used when it is possible to determine with high confidence which processors will consume the write. In previous experiments [9], we found that, in many cases, data is write-forwarded to all processors. In this case, we use a special primitive called write-broadcast. No forwarding register is necessary here.

A write-update primitive updates all current sharers of the data, like in update-based cache coherence protocols. Our compiler algorithm uses this type of write only when there is a strong indication that past sharers of the data will use the data next. This occurs in loops where each iteration has the same producer and consumer processors. A write-through primitive sends the updated line to the home node. This reduces the latency of the next read to the line from a three network-hop transaction to a two network-hop transaction. Our compiler algorithm uses this type of write when the compiler determines that some other processor will read the data but it cannot determine which processor will. Finally, prefetching is sometimes used before the consumer reads. In particular, our compiler uses a prefetch before the consumer read instead of a write-forward in the producer write when the time separation between the write and read is expected to be very high. In such case, a write-forward in the producer write could send the data to the consumer cache so early that the data would be displaced from the consumer cache before being used.

Overall, these primitives can be used directly by the programmer in codes that the programmer knows well or by our parallelizing compiler in compiler-analyzable codes. However, even if the programmer or the compiler uses one of these primitives in the wrong place, the hardware cache coherence protocol will ensure that the application executes correctly. The performance, of course, will suffer.

For pointer-based, irregular applications, the compiler analysis may be very limited and the programmer may not have the time to fully understand the code and manually insert invocations to the primitives discussed. Examples of such applications are engineering programs in areas like graph problem-solving, event-driven simulation, VLSI design, or tree-handling. Often, the dominant data structures in these applications are small records no more than several tens of bytes in size interconnected and accessed via pointers. A processor often accesses several fields in a record and then dereferences a pointer. For these applications, I-ACOMA has support for what we call *Group Prefetching via Memory Binding* [27].

The idea is to bind together in hardware a set of variables that are strongly related to each other and from then on prefetch them together. The scheme works as follows. The compiler or the programmer tries to identify a group or set of variables that are laid out largely contiguously in memory and are accessed at similar points in the program. A typical group that the compiler can sometimes identify is the set of fields in a record or the set of elements in a matrix column. The compiler can identify these groups from variable declarations and an analysis of the source code. In this case, the compiler asks for confirmation from the programmer. Sometimes, however, the group is identified by the programmer. In any case, if the programmer confirms the group, the compiler inserts the *Group* system call in the code (Figure 1).

Figure 1 shows that, in the I-ACOMA machine, each memory line has two extra bits, namely the *next* (N) and the *boundary* (B) bits. These bits are actually in the directory. At run time, when the *Group* system call is executed, the kernel sets the N and B bits for the memory lines that contain the group. The N bit is set if the next line contains data belonging to the same group, while the B bit is set if this line contains at least one group boundary. Once these bits are set, every time that a processor misses on one of the lines of the group, the whole group will be supplied. This will be accomplished by a simple finite state machine in the directory that interprets the N and B bits of the line being read and, if necessary, forces a memory read for the next line. This scheme is specially targeted to migratory shared data [25].

Finally, I-ACOMA also has a very cheap yet effective support for instruction prefetching. This scheme is called *Guarded Sequential Prefetching* [26]. It is useful for systems codes like operating systems, which suffer many instruction misses because they have few tight loops and a large instruction working set. For our scheme to work, we need a two step approach. In the first step, we profile the code to determine the execution frequency of its basic blocks and paths among them, and place the basic blocks in memory to minimize cache conflicts [23]. This profiling step is motivated by the observation that operating system traces often contain fairly-deterministic repeated sequences of hundreds of instructions, often spanning several routines, that are not part of any obvious loop. These sequences we call *instruction sequences*. In this step, we identify the most important instruction sequences and place their basic blocks in memory in the sequence in which they are frequently executed. As a result, we expose much spatial locality and many conflict misses in the cache disappear.

The second step is to exploit the resulting code layout to perform a specialized type of prefetching. Indeed, we observe that, with the new layout, it is very likely that, once a processor starts executing an instruction sequence, it continues executing instructions that are contiguous in memory until the end of the sequence. At that point, the processor is likely to branch off. Consequently, we encode one bit in the last instruction of each sequence and set the bit to 1. This bit we call *Guard Bit*. We then design a sequential prefetch engine that, when a processor starts executing an instruction sequence, races ahead of the program counter, prefetching in sequence until it finds a guard bit set. At that point, the prefetcher stops.

This scheme is better than traditional sequential prefetching schemes that prefetch a fixed number of sequential lines on reference or on miss. This is because our scheme prefetches only what is likely to be used, namely a whole sequence – no more, no less. Furthermore, it does so as soon as possible, namely when the processor enters the sequence. If the processor jumps off before reaching the end of the sequence, the prefetch engine detects the new range of addresses issued and jumps to the new sequence too [26]. Interestingly, to implement our scheme we only need to make a small modification to existing next-line sequential prefetchers: the addition of a module to detect the guard bit. Such module can introduce as little delay as one gate delay. Finally, we use one bit in each TLB entry to enable or disable the prefetcher at run-time based on whether or not the page being accessed contains code with optimized layout. Overall, with all the latency-hiding schemes presented in this section, we hope to eliminate most of the processor stall time that caches do not eliminate.

2.3 Support for Non-Analyzable Loops

While these latency-hiding schemes will hopefully speed-up all workloads, I-ACOMA includes hardware support to enhance the performance of two very important classes of work-

```
struct foo_type {
    int a;
    int b;
     .
     .
    int f;
}
struct foo_type foo;
Group ( &foo.a, &foo.f);
```

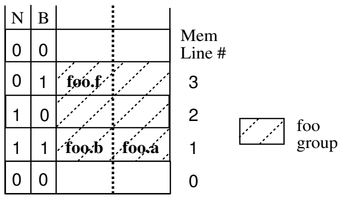

Figure 1: Group prefetching via memory binding. The figure shows how a group is formed.

loads, namely scientific applications with loops that cannot be compiler-analyzed and databases. We examine the former in this section, while we analyze databases in the next section.

Scientific applications in the areas of sparse matrix computations, domain decomposition, molecular dynamics, molecular biology, and image processing often have loops with data dependencies that are impossible to analyze by the compiler. An example of such loops is shown in Figure 2, where an array is read and written via a subscripted subscript. More complex loops are common in real codes, for instance involving conditionally executed instructions or multiple loop nesting levels. In general, these loops must be run serially because the compiler is unable to ascertain whether or not they are parallel. It turns out, however, that these loops often are fully parallel.

```
DO i=1,N
    ...
    A(f(i)) = ...            (S1)
    ...
    ... = A(g(i)) + ...      (S2)
    ...
ENDDO
```

Figure 2: Typical loop that, in most cases, cannot be analyzed at compile time.

One possibility is to parallelize these loops with costly run-time parallelization algorithms [4, 10, 12, 13, 16, 17, 19, 28]. In most of these proposals, the compiler inserts code that, at run time, determines what iterations do not depend on each other (inspector phase) and then executes these iterations in parallel (executor phase). However, these algorithms often have high overhead that precludes any speedup from parallelism. One of these algorithms [17] makes a backup copy of the arrays involved (array A in Figure 2) and then executes the loop as if it were completely parallel, while saving some information on what array elements are being accessed during execution. Finally, at the end of the loop, the algorithm checks if any array element has been accessed by more than one processor and written at least by one. In the hopefully rare case where this condition is true, the loop is re-executed serially. Otherwise, the loop execution has been sped up several times.

I-ACOMA includes some extra state in its coma tags to implement an improved version of this algorithm in hardware. Therefore, any loop that cannot be parallelized at compile time like the one in Figure 2 is executed by I-ACOMA completely in parallel. Hopefully, most of the times the loop will indeed be parallel. If the loop is not really parallel and a data dependence is illegally ignored, the event is immediately detected and the loop execution is stopped. What allows us to detect the event immediately and not have to wait until the end of the loop is the cache coherence protocol. We use the fact that the cache coherence protocol keeps the cached copies of any data (and its coma tags) coherent at all times.

More specifically, the coma tag for a given memory line keeps two fields for each array element $A(i)$ that is allocated in the line. The two fields contain the number of the iteration when the element was last read and the number of the iteration when it was last written respectively. An execution will be illegal if a processor is about to read an array element in an iteration whose number is lower than the number currently stored in the last-written tag of the element. Similarly, an execution will be illegal if a processor is about to write an array element in an iteration whose number is lower than the number currently stored in the last-read tag of the element. If any of these two accesses was allowed to proceed, it is obvious that an incorrect execution of the loop would result. All these comparisons and operations on the tags are done by special hardware in I-ACOMA. Overall, therefore, we see that, if the loop is parallel, we will run it in parallel without problems. Furthermore, even if the loop has inter-iteration dependencies, if the iterations are executed by the different processors in an order that, by chance, keeps the dependences, the parallel execution is legal. With this support, we hope to speed up these codes significantly.

2.4 Support for Database Workloads

Databases are an important class of workloads used in many domains. While some databases are I/O bound and, therefore, the only way to improve their performance is to optimize their I/O activity, some other databases are designed to be memory resident. These are the ones which the I-ACOMA project focuses on. In these databases, it is important that the memory hierarchy is utilized effectively. Unfortunately, little is known about how these databases use the memory hierarchy.

One important characteristic of the memory access patterns of databases is that working sets are so large that they overflow multimegabyte secondary caches. As a result, processors often have to re-fetch the data from remote locations. In addition, in queries where data is accessed via indexes, the data access patterns are often hard to predict at compile time. As a result, it is likely to be hard for the compiler to place the data in memory modules close to the processor that will access that data.

It is hoped that the coma support in our machine solves these two problems, at least partially. In a coma machine, data automatically migrates to the attraction memory of the node that uses it. As a result, a node is able to keep a much larger working set close by. There is less need to access the data from a remote node. In addition, since the data automatically migrates to where it is being used, the initial placement of the data is largely unimportant. There is no need to try to predict the access patterns at compile time. These arguments, which agree with the thoughts of other researchers [7, 18], need to be tested thoroughly.

3 Compiler Support

While the architectural organization of a machine affects how programmable the machine is, the software infrastructure available to write programs for the machine is a key determinant of end-user programming ease. For this reason, a sizable effort in the I-ACOMA project is devoted to providing software support for the machine. This support is provided by the Polaris parallelizing compiler.

Polaris is a parallelizing compiler recently developed at the University of Illinois [3]. The compiler takes scientific programs written in Fortran and performs transformations to (i) expose parallelism, (ii) enhance locality, and (iii) reduce communication overhead. As a result, several of the applications in the Perfect suite [2] are parallelized automatically to a greater degree than by any other translator, experimental or commercial, that has been tested to date. The compiler currently generates parallel code for several shared-memory machines, including the SGI Power Challenge, the Cray T3D, the Convex C-3, the Convex Exemplar, and multiprocessor Sparc workstations. In addition, it currently supports several of the specialized I-ACOMA features. In particular, it performs the analysis to invoke most of the latency hiding primitives described in Section 2.2.

The success of Polaris at parallelizing Fortran applications is largely based on innovations in four compiler techniques, namely dependence analysis, privatization, induction variable recognition, and reduction variable recognition. Polaris can also support traditional transformations to enhance the utilization of caches, including loop reversal, loop interchange, and blocking. There are, however, two challenges related to generating code for the I-ACOMA machine that are currently being addressed. The first challenge is to map the parallel code to the several levels of parallelism of I-ACOMA efficiently. Recall that there are three levels of parallelism in I-ACOMA, namely parallelism among the processors, among the threads in a processor, and among the functional units in a processor. Each of these levels is best equipped to exploit a different grain size of parallelism: the coarsest grain among the processors and the finest grain among the functional units of a processor. However, the same parallelism can be exploited by the different levels with different efficiency.

A second challenge is to generate parallel code that exploits the deep memory hierarchy of a modern scalable shared-memory machine like I-ACOMA effectively. This may involve re-writing traditional algorithms to enhance the locality of the data and computation at the expense of load imbalance or at the expense of replicating some operations across processors. Similarly, it may involve re-writing the algorithms to increase the aggregation of the interprocessor communication to make communication more efficient at the expense of load imbalance or replicated operations.

4 Building a Prototype

The architectural features described in this paper and some others are currently being evaluated using software simulations under Augmint. Augmint [14] is a public-domain multiprocessor tracing and simulation package that takes as input explicitly parallel codes and runs on any Intel x86-based uniprocessor PC. However, while software simulations provide valuable insights into many of these features, they are sometimes limited in the information that they provide.

For this reason, we are in the process of building a hardware prototype of I-ACOMA to evaluate some of the ideas presented. This prototype is expected to be built in cooperation with an industrial partner and will necessarily be based on off-the-shelf processors. It will be partly programmable to enable us to study more issues. With the prototype, we will evaluate the effectiveness of some of the memory-hiding primitives described in Section 2.2. The prototype will also support the off-chip hardware necessary for the parallelization of non-analyzable loops as described in Section 2.3; however, the on-chip hardware of the scheme will be emulated in software. Finally, the prototype will allow us to perform the long runs necessary to study the behavior of databases in coma machines, perform research on operating system issues for coma machines, and evaluate the effectiveness of the parallelizing compiler.

5 Conclusions

While scalable shared-memory multiprocessors with hardware-supported cache coherence are relatively easy to program, if truly high-performance is desired, they still require substantial programmer effort. Indeed, data must be allocated close to the processors that will use it and the working sets must be tuned so that they fit in the caches. This is unfortunate because the most important obstacle to the popularization of parallel computing is the lack of programmability of parallel machines.

The goal of the I-ACOMA project is to explore how to design a highly-programmable, high-performance multiprocessor. Given our emphasis on programmability, we focus on a flat coma multiprocessor supported by the Polaris parallelizing compiler. Ideally, in such a machine, the actual initial layout of the data in memory is not important, and there is not as pressing a need to tune the working sets to fit in the caches. In this project, the main issues that we are addressing are advanced processor organizations, techniques to handle long memory access latencies, and support for two classes of workloads: scientific applications with loops that cannot be compiler-analyzed and databases.

We are in the process of building a prototype that will include some of the features described. It is expected that it will be built in cooperation with an industrial partner.

6 The I-ACOMA and Polaris Teams

The graduate students currently working in the I-ACOMA project are: Sujoy Basu (virtual memory issues), David Koufaty (compiler algorithms to insert the different types of writes and the prefetches), Venkata Krishnan (simultaneous multithreaded processors), Jose Martinez (interconnection networks), Jovan Mitrevski (update cache coherence protocols), Anthony Nguyen (operating systems for coma), Arun Sharma (coma reliability), Edgar Torres (coma cache coherence protocols), Pedro Trancoso (databases), Liuxi Yang (hardware design and processor organizations), Ye Zhang (support to parallelize non-analyzable loops, together with Prof. Lawrence Rauchwerger from Texas A&M University), and Zheng Zhang (coma architectures and hardware issues). The graduate students currently working in the Polaris project are: John Grout (interprocedural analysis), Jaejin Lee (loop optimizations), Yunheung Paek (data dependence analysis and code generation for the Cray T3D), William Pottenger (idiom recognition and multiple levels of parallelism), and Carlos Velez-Rivera (program restructuring using algebraic information). Other members of the Polaris group are senior software engineer Jay Hoeflinger

(Polaris infrastructure) and Prof. Rudolf Eigenmann from Purdue University (loop transformations for parallel workstations, with his students Michael Voss and Insung Park). The faculty associated with the I-ACOMA project are Josep Torrellas and David Padua.

Acknowledgments

We thank Z. Zhang, S. Basu, and D. Koufaty for their comments on the paper. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors.* SIA, 1994.

[2] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[3] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Effective Automatic Parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.

[4] D. K. Chen, J. Torrellas, and P. C. Yew. An Efficient Algorithm for the Run-Time Parallelization of Do-Across Loops. In *Supercomputing '94*, pages 518–527, November 1994.

[5] E. Hagersten, A. Landin, and S. Haridi. DDM – a Cache-Only Memory Architecture. *IEEE Computer*, pages 44–54, September 1992.

[6] H. Burkhardt III et al. Overview of the KSR1 Computer System. Technical Report 9202001, Kendall Square Research, Waltham, MA, February 1992.

[7] Kendall Square Research. *KSR1 Technical Summary.* Waltham, MA, 1992.

[8] P. Kogge. Execube – A New Architecture for Scaleable MPPs. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages I77–I84, August 1995.

[9] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, to appear 1996. A shorter version appeared in Proceedings of the 9th International Conference on Supercomputing, pages 255-264, July 1995.

[10] V. Krothapalli and P. Sadayappan. An Approach to Synchronization of Parallel Computing. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 573–581, June 1988.

[11] A. Landin and F. Dahlgren. Bus-Based COMA – Reducing Traffic in Shared-Bus Multiprocessors. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 95–105, February 1996.

[12] S.-T. Leung and J. Zahorjan. Improving the Performance of Runtime Parallelization. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, May 1993.

[13] S. Midkiff and D. Padua. Compiler Algorithms for Synchronization. *IEEE Trans. on Computers*, C-36(12), December 1987.

[14] A. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. In *IEEE International Conference on Computer Design*, October 1996.

[15] D. Poulsen and P. Yew. Data Prefetching and Data Forwarding in Shared Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume II, pages 276–280, August 1994.

[16] L. Rauchwerger and D. Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 33–43, July 1994.

[17] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, June 1995.

[18] J. Robinson, D. Baxter, and J. Gray. Advantages of COMA. Unpublished manuscript, 1995.

[19] J. Saltz, R. Mirchandaney, and K. Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–611, May 1991.

[20] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[21] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 276–285, January 1995.

[22] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.

[23] J. Torrellas, C. Xia, and R. Daigle. Optimizing Instruction Cache Performance for Operating System Intensive Workloads. In *IEEE Trans. on Computers*, to appear 1996. A shorter version appeared in Proceedings of the 1st International Symposium on High-Performance Computer Architecture, pages 360-369, January 1995.

[24] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[25] W. Weber and A. Gupta. Cache Invalidation Patterns in Shared-Memory Multiprocessors. In *IEEE Transactions on Computers*, pages 794–811, July 1992.

[26] C. Xia and J. Torrellas. Instruction Prefetching of Systems Codes With Layout Optimized for Reduced Cache Misses. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 271–282, May 1996.

[27] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–199, June 1995.

[28] C. Q. Zhu and P. C. Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. In *IEEE Transactions on Software Engineering*, pages 726–739, June 1987.