Multithreading Decoupled Architectures for Complexity-Effective General Purpose Computing

Michael Sung, Ronny Krashinsky, and Krste Asanović
MIT Laboratory for Computer Science, Cambridge, MA 02139
{darkman|ronny|krste}@mit.edu

Abstract

Decoupled architectures have not traditionally been used in the context of general purpose computing because of their inability to tolerate control-intensive code that exists across a wide range of applications. This work investigates the possibility of using multithreading to overcome the loss of decoupling dependencies that represent the cause of this main limitation in decoupled architectures. A proposal for a multithreaded decoupled control/access/execute architecture is presented as a platform for achieving high performance on general purpose workloads. It is argued that such a decoupled architecture is more complexity-effective and scalable than comparable superscalar processors, which incorporate enormous amounts of complexity for modest performance gains.

1 Introduction

Complexity-effective design is taking on new importance in modern general purpose architectures. A limitation in these architectures is the cost of accessing large centralized resources as global communication delays continue to increase relative to computation delays [1]. This impacts the scalability of superscalar designs, as they depend on large multi-ported and highly-associative structures. Additionally, with energy consumption playing a major role in the cost and performance of designs, it is no longer feasible to greatly increase complexity to obtain diminishing performance gains.

Decoupled architectures have been explored as a more efficient means of achieving some of the same benefits as out-of-order superscalar execution. In these architectures, decoupling can provide memory and control latency hiding, parallel instruction execution, dynamic scheduling, and efficient resource utilization with a minimal amount of complexity. Additionally, the decentralized nature of decoupled designs makes them inherently scalable.

Due to the increased demands of scaling, superscalar architectures are beginning to use more complexity-effective designs [2]. Some superscalar processors use a clustered organization to simplify issue logic, [3] whereby instructions are directed to separate clusters with independent register files and functional units. Additionally some designs are incorporating deep queues to decouple instruction fetch from execution [4]. These designs are being forced to take on some of the attributes of decoupled architectures as complexity becomes unmanageable.

Historically, decoupled architectures have not gained wide appeal due to the difficulties associated with being able to decouple programs effectively. In this paper, we investigate the potential of augmenting traditional decoupled architectures with multithreading for general purpose computing. We first present a survey of various decoupled architectures, and follow up with a design proposal for a multithreaded control-decoupled architecture as a more complexity-effective alternative to superscalars.

2 Background of Decoupled Architectures

2.1 Decoupled Access/Execute

The first major investigation of decoupled architectures was done by Smith [5], which eventually led to the Astronautics ZS-1 Processor [6]. In his preliminary study, Smith introduces the concept of a decoupled access/execute (DAE) machine. An access processor (AP) and an execute processor (EP) work on separate instruction streams, communicating data values via queues. In this design, not only is instruction level parallelism exploited by processing two instruction streams at the same time, but these decoupled processing units can *slip* with respect to each other. This allows the access processor to run further ahead in the program to fetch values from memory, effectively providing a large amount of memory latency hiding. Smith argues that this design provides a more complexity-effective way of obtaining the benefits of dynamic scheduling than the methods used in designs with more complex issuing methods (namely, those used by out-of-order superscalars).

The preliminary study also describes a single interleaved instruction stream which is then split into separate streams for the AP and EP. This is the approach adopted in the Astronautics ZS-1 [7]. In this case, decoupling is accomplished through the use of instruction queues which feed the AP and EP; the queue for the EP is significantly longer since it usually runs behind the AP. All control flow instructions are executed by an instruction splitter in the front-end that precedes the AP/EP cores.

Descendants of the ZS-1 DAE architecture include the PIPE project [8] which was followed in turn by MISC (Multiple Instruction Stream Computer) [9]. The proposed MISC design consists of four generic processing elements (PEs) which collaborate to complete a common task. Dedicated communication channels connect each PE to every other PE, and each PE has four input queues to receive data from the other PEs as well as two input queues from memory. In a typical configuration, two PEs could operate as access processors, fetching data which is then sent to the two other PEs which perform computation on the data in a pipelined manner. The MISC architecture can be considered a less scalable predecessor to tiled architectures such as RAW [10].

The WM architecture [11] is another variation of an access/execute decoupled architecture. In this design, a single instruction stream controls a collection of decoupled components which communicate via architecturally visible queues. Parallelism and memory latency hiding are achieved through the use of decoupled data units which can process vector load and store instructions. The instruction fetch unit is also decoupled from the functional units and control unit.

2.2 Simultaneous Multithreading and Decoupling

In their analyses of DAE architectures [12, 13], Parcerisa and Gonzalez make the observation that although decoupled machines effectively hide memory latency, they suffer from functional unit latencies when there are true (RAW) data dependencies. They propose a synergy between simultaneous multithreading and access/execute decoupling in order to uncover more ILP and better utilize the functional units; this is the same motivation that prompted the development of SMT for superscalar processors. Parcerisa and Gonzalez find that SMT designs can effectively hide functional unit latencies, but do a poor job at hiding long memory latencies. The access/execute decoupling provides an effective means of hiding these memory latencies.

2.3 Decoupled Control/Access/Execute

An extension to the basic DAE architecture is to augment it with decoupled control flow (DCAE). This decoupling represents a further separation of basic program functionalities; control, memory access, and computation are partitioned into three instruction streams. The ACRI project [14] proposed an implementation of such an architecture. The different control, access, and execute instruction streams are each handled by separate processors (the CP, AP, and EP respectively). Decoupling the control flow into a separate instruction stream allows it to be processed ahead of the access and execute streams and potentially eliminates control overhead from these streams.

The control processor (CP) executes the control flow graph of the program, sending directives to the AP and EP to execute instruction fetch blocks (IFBs). These directives specify basic blocks and include the starting address and length of a block. The actual code for these basic blocks are in separate instruction streams, and instruction fetch engines (IFEs) in the AP and EP process the IFBs and fill the corresponding queues with ready-to-execute instructions. The designers of the ACRI also include a separate parameter queue as an additional input queue to the AP and DP in order to efficiently pass parameters (such as function arguments) without the need of going through memory. In order to avoid memory inconsistencies, the addresses of the CP memory accesses are compared with those in a pending store address queue (SAQ). The compiler is responsible for ensuring that the CP doesn't slip ahead and access memory before the AP puts potentially conflicting addresses in this queue.

The address and execute processors operate on the streams of non-speculative instructions generated by the IFEs (these instruction streams are free of control instructions such as conditional branches). Since they process streams of valid instructions and data without using speculation, they can be considered *stream units* [15]. In the ACRI description, the AP and EP engines are provided with limited control capabilities; an IFB can include a loop count specifying the number of iterations of the basic block to perform, and the processors can be augmented with support for predicated instruction execution to enable larger basic blocks.

The instruction set architecture for the individual processors in the DCAE architecture can be optimized for their particular task and capabilities. For example, the ISA for the AP can include specialized instructions such as auto-increment loads and stores. Additional parallelism can be achieved with little overhead by providing the AP and EP with VLIW instructions to match their mix of functional units. The CP is actually a fully functional processor with the capability of performing memory operations. One reason for this is if the CP requires a value from memory to determine control flow, the AP can be of little help since it usually trails the CP in program execution. Additionally, this allows the CP to implement procedure calls by directly manipulating the stack frame which is shared between the processing units [16, 17].

The DCAE architecture performs best when there is full decoupling between the control, access, and execute instruction streams. Any dependencies between these streams that disrupt the decoupling and cause the instruction streams to re-synchronize will adversely affect performance. For example, the control stream might be dependent on a value calculated by the EP. In this case, there is a performance penalty as the CP must stall until the EP catches up and provides the necessary value, followed by a period for the CP to decouple again. There have been several studies that investigate these loss of decoupling events, termed LODs [15, 16, 17, 18]. These performance-limiting LOD events are summarized in Table 1.

LOD	Description	Example cause
1	AP must wait for memory	indirect memory reference
2	AP must wait for EP	computed memory address
3	CP must wait for AP	read-after-write hazard
4	CP must wait for EP	computed branch condition
5	AP must wait for EP	conditional basic-block nullification
6	EP must wait for EP	conditional basic-block nullification

Table 1: Description of LOD events for a DCAE machine

3 Decoupled vs. Superscalar Machines

Superscalar and decoupled architectures employ different mechanisms to obtain some of the same performance advantages. Studies comparing these architectures include the decoupled references mentioned previously, as well as [19, 20]. Comparisons can be made in how these architectures handle:

- Memory Latency

  Superscalar machines hide memory latency by maintaining large issue/reorder windows to keep track of outstanding loads and the instructions which depend on them while other instructions are executed. Data prefetching and value speculation can also be implemented to help tolerate memory latency.

  For decoupled architectures, memory latency is hidden by executing the access instruction stream in advance of the execution stream through the use of simple queues, which is effectively a form of prefetching [20].

- Control Latency

  Because control instructions are resolved many cycles after instruction fetch, superscalar processors must employ accurate branch prediction combined with speculation to maintain performance.

  Decoupled architectures can effectively hide this latency by determining control dependencies outside of the execution processor. This decoupling essentially allows dynamic loop unrolling when loop conditions can be determined ahead of time.

- Resource Allocation

  Superscalar architectures allocate explicit resources for every outstanding instruction through the use of complex register renaming structures.

  In a decoupled architecture, the queues provide a cheap form of register renaming, where the queue elements themselves provide the resources for outstanding instructions, and the architecturally visible queue heads take the place of complicated naming schemes.

- Dynamic Scheduling and ILP

  A superscalar architecture uses dynamic scheduling to extract ILP by implementing dependency analysis in hardware.

  Decoupled machines achieve dynamic scheduling when the separate instruction streams slip with respect to each other. In addition, the separate instruction streams of a decoupled machine provide an immediate source of ILP.

To achieve sufficient performance, superscalar architectures must expend a great proportion of area and complexity on issue and decode logic, in the form of the instruction window, reorder buffer, register renaming logic, branch prediction, data prefetching, etc. It is widely accepted that the issue logic of superscalar architectures is rapidly becoming difficult and expensive to implement in terms of area, delay, and power consumption [2, 21]. Large structures such as issue windows that require associative dependency-checking are the limiting factors in scaling the issue rate of superscalar machines [1]. Additionally, superscalar architectures rely on speculation and prediction which incurs a large amount of overhead (both in terms of hardware and misprediction penalties).

Decoupled architectures provide mechanisms for forms of dynamic out-of-order execution, loop unrolling, and register renaming without the associated complexity as implemented in superscalar processors. Thus, in addition to the inherent memory latency toleration that decoupled architectures provide, they can also exploit ILP with much simpler issue logic than superscalar processors. All these benefits are achieved using very simple queues, which are both easily implemented and cheap in terms of area, speed, and power.

Since a decoupled machine alleviates the need for centralized resources, it is inherently more scalable than corresponding superscalar processors. Additionally, the queue-based designs of decoupled architectures are amenable to being incorporated in scalable tiled architectures with on-chip networks [10]. This is a logical extension to the standard decoupled architecture whereby independent decoupled streams can be run on separate tiles of such a machine.

Given the potential performance advantages of decoupled architectures, a relevant question to ask is why they have not come into the mainstream for general purpose computing. One main reason is that decoupling may not always be possible in general, especially given control-intensive code. As described earlier, the Achilles' heel of decoupled architectures come from LOD events, so programs with complicated control flow can suffer severe performance degradation on decoupled architectures.

Another limiting feature of decoupled machines is the complexity involved in effectively compiling programs into separate instruction streams. Although there have been a few compilers that have been developed for decoupled architectures [17, 22], it is often difficult for the compiler to create a decoupled program that avoids LOD events.

4 Multithreading DCAE Architectures

It seems evident that aside from the performance degradation resulting from LOD events, decoupled architectures can be a viable general purpose architectural platform that is both complexity-effective and scalable. Since LODs represent the primary performance limitation for decoupled architectures, it is crucial to remove the effects of these dependencies in order to enable general purpose computing. LODs trigger synchronization events between program streams in a single thread of control. Potentially, these synchronization latencies can be hidden if multiple threads are available to execute concurrently on a decoupled machine. Such a multithreaded decoupled architecture would hide runtime LOD events by simply switching threads whenever a LOD event occurs.

Traditionally, multithreading is used to hide long latency events such as memory accesses, as well as increase ILP by avoiding instruction dependencies. As discussed earlier, Parcerisa and Gonzalez proposed that multithreading be used in conjunction with a DAE architecture in order to hide functional unit latencies within the execute processor [13].

We propose to extend this idea and provide multithreading on a DCAE architecture (MT-DCAE). The motivation however, is not so much to hide functional unit latencies as it is to provide LOD event toleration within a particular instruction stream. The MT-DCAE architecture realizes the benefits of full control/access/execute decoupling while hiding the effects of LOD events that must necessarily occur in programs and is especially troublesome for a control-decoupled machine. Also, it is possible to allow the compiler more opportunity to decouple a given program by utilizing multithreading compiler techniques. For the MT-DCAE architecture, we leverage much of the design from the ACRI architecture [14] and the multithreaded DAE design by Parcerisa, et. al. [13]. A block diagram is shown in Figure 1.

All three processors (CP/AP/EP) of the MT-DCAE design are multithreaded. Multithreading is implemented by replicating all the queues of the original ACRI design, including the instruction fetch block (IFB) and parameter queues, once for each thread. The address and execute processors have state for multiple contexts (i.e., instruction queues and register files) to allow for fast context switches between threads. Thus, multithreading on the MT-DCAE architecture can improve performance on two levels by handling both LOD dependencies and instruction dependencies within a single thread. However, since the primary function of the multithreading is not to hide memory latency (provided by the decoupling itself) but to hide LOD events, we anticipate that a few threads will be adequate to achieve large performance gains.

4.1 Enabling Multithreading

To enable multithreading, the AP and EP must be supplied with instruction streams from multiple threads. One implementation of this is to provide the CP with context switching capabilities, and have it switch control threads whenever it encounters a long latency event such as a LOD. The control processor can switch to processing another thread while the address and data processors continue to process instructions from the first one. Later when the LOD event from the original thread is resolved, it again becomes a ready thread for the CP. In this way, contexts can be pipelined through the CP, AP, and EP to efficiently utilize the processors' resources.

An important design issue is how the control processor should schedule threads to fully take advantage of the MT-DCAE architecture and hide LOD latencies. The main function of the CP is to prefetch instructions to keep the access and execute processors busy. Since each IFB can represent a large amount of work and the control processor ideally runs significantly ahead of the access and execute processors, the CP can possibly afford not to have extremely efficient thread context switching. Thus, it may not be necessary to add additional context state on a per thread basis to the CP. This is in contrast to the multithreading on the AP and EP, which must be able to switch between threads quickly in order to maintain performance. Another design issue is keeping multiple threads' fetch blocks available in the input queues for the AP and EP, so that there is always at least one thread available to run. Accomplishing this might require thread execution balancing on the CP to make sure that the AP and EP are provided with ample thread parallelism. [23] discusses various context switching techniques for multithreaded decoupled architectures.

4.2 Speculative Multithreading

Another technique that may improve performance on MT-DCAE is speculative multithreading [24]. This allows the multithreading hardware to be beneficial even when only a single thread of control is available. This can be implemented as an extension to the special *conditional* or *speculative* execution modes for dispatch blocks discussed in [15]. Using this technique, blocks conditionally dispatched to the AP wait for a condition from the EP to determine whether or not they should be discarded. Blocks can also execute in the AP speculatively and if it is later determined that they are invalid, the updates to the queues from that block are dis-

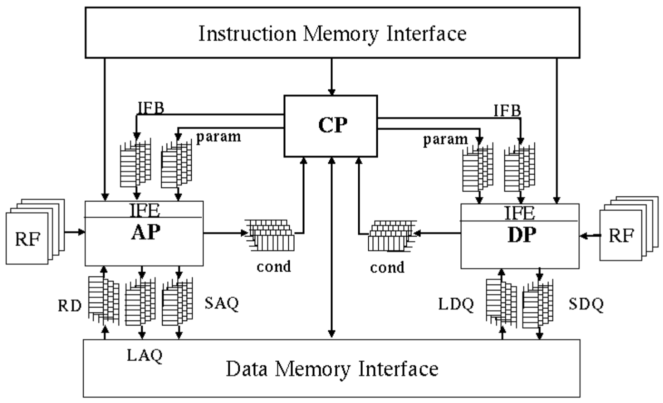

Figure 1: MT-DCAE multithreaded decoupled control/access/execute architecture.

carded.

With multithreading, the control processor can spawn a speculative thread at predictable branches such as procedure calls. In this case the speculative thread would keep executing code beyond the procedure call with the hope that there will be no dependencies. This sort of speculation is a natural fit for the decoupled architectures because all the memory addresses are put in queues. Therefore, misspeculation detection can be accomplished by dynamically comparing the addresses accessed by the two threads, and a speculative thread can be nullified by flushing its addresses from the queues and destroying its context. Speculative multithreading can also be used to hide the latency of predictable LOD events.

4.3 Scaling MT-DCAE architectures

A major consideration for an architecture's viability is its ability to scale in performance with a manageable amount of design complexity. The base MT-DCAE architecture described in the previous section can be readily extended in a variety of ways to allow for higher performance. For example, we can simply scale the input queue lengths to allow greater decoupling potential. As discussed earlier, these queues are much more readily scaled than the data structures of superscalar machines. We can also replace the control, access, and execute processing units with wider-issue processors to scale performance as well.

The MT-DCAE architecture can also be scaled by adding more discrete processing nodes (either control, access, or execute). In the most generalized form, decoupling begins to look very similar to a multiprocessor tile architecture, with each processor accessing input and output queues for communication with other nodes. This begins to blur the lines of what a decoupled architecture is, as one can view the nodes of a generalized multiprocessor array as being decoupled elements capable of exacting specific control, access, or execute functions.

4.4 Using DCAE as a Coprocessor

Superscalar processors will perhaps always be better than decoupled architectures at processing code which is extremely control-intensive. To exploit the advantages out of both superscalar and decoupled architectures, we can consider implementing the control processor in a DCAE design as a more capable high-performance microprocessor. This would allow the CP to make progress faster as it can access memory without going through the AP. Also, the compiler can choose to avoid decoupling and only use the CP when running control-intensive code that would result in an overabundance of LOD events. When decoupling is possible, the decoupled access/execute hardware provides a high-performance and complexity-effective computation engine. For example, a decoupled architecture is a better fit for streaming code such as multimedia; however, such code is often mixed with control-intensive portions of computation, so in this situation a hybrid architecture may be a good design point. This type of coprocessing computational model is similar to that used for vector or SIMD array coprocessors as well as more modern microprocessors with hardware multimedia support. Decoupled access/execute engines are more flexible than these alternatives, and a comparison of performance and complexity would be interesting.

4.5 More Related Work

It is noteworthy that Dorojevets, et. al. [25] implemented a decoupled machine that shares features of a MT-DCAE architecture and a general tiled machine, called the MARS-M (Modular, Asynchronous, Extensible Systems) computer. This architecture supports simultaneous execution of up to four address, four data, and one control thread, with communication provided by various queues. Both the address and execution processors and the communication queues are dynamically assigned. The individual processors are implemented as VLIW processors which support simultaneous multithreading. In a follow up study [26], Dorojevets et. al. also make the observation that control dependencies hinder parallel execution, and proposes speculative multithreading.

5 Summary

It seems that in recent years, interest in decoupled machines has waned. Still, there exist some attractive features of decoupled machines that for the most part have not been exploited or utilized. Since their inception, decoupled architectures have been touted as a complexity-effective and scalable way to provide provide memory and control latency hiding, parallel instruction exection, dynamic scheduling, and efficient resource utilization.

In the paper, we have attempted to present a comprehensive survey of the major research and industrial work on decoupled architectures. From our survey, we can conclude that the limitations of using decoupled architectures in general purpose processing are derived primarily from the inability for these machines to tolerate LOD latencies. To enable decoupled architectures to effectively perform general purpose computation, we propose to augment decoupling with multithreading to hide the latency of LODs assuming that sufficient thread level parallelism exists. Although previous work proposed the use of multithreading in conjunction with decoupling, they do so only in the context of hiding functional unit latencies and not as a general solution to the problem of LOD latencies on decoupled machines.

Of course, much more research is needed to determine whether such a scheme is viable for general-purpose computing. It may not be trivial to efficiently decouple programs or find enough thread-level parallelism to be able to cover LOD events. Also, there is a large design space of possible multithreading implementations that can fit into a MT-

DCAE framework. We simply provide the high-level concept of using multithreading as a potential solution to the LODs that can hinder decoupled architectures. Thus, it remains an open-ended question as to what is the best combination of techniques to fully unlock the synergy between decoupling and multithreading for general purpose computer architectures.

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *ISCA 27*, May 2000.

[2] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *ISCA 24*, pages 206–218, Denver, CO, 1997.

[3] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[4] L. Gwennap. Mips R10000 uses decoupled architecture. *Microprocessor Report*, October 1994.

[5] James E. Smith. Decoupled access/execute computer architecture. In *ISCA 9*, 1982.

[6] James E. Smith *et. al.* The astronautics ZS-1 processor. In *1988 IEEE International Conference on Computer Design*, pages 307–310, October 1988.

[7] J. E. Smith. Dynamic instruction scheduling and the astronautics ZS-1. *IEEE Computer*, 22(7):21–35, July 1989.

[8] J.R. Goodman *et. al.* PIPE: A VLSI decoupled architecture. In *ISCA 12*, pages 20–27, Boston, MA, June 1985.

[9] G. Tyson, M. Farrens, and A. R. Pleszkun. MISC: A multiple instruction stream computer. In *Micro-25*, pages 193–196, Portland, Oregon, December 1992.

[10] E. Waingold *et. al.* Baring it all to software: RAW machines. *IEEE Computer*, pages 86–93, September 1997.

[11] Wm. A.Wulf. Evaluation of the WM computer architecture. In *ISCA 19*, pages 382–390, Gold Coast, Australia, May 1992.

[12] Joan-Manuel Parcerisa and Antonio Gonzalez. Multithreaded decoupled access/execute processors. Technical Report UPC-DAC-1997-83, UPC-DAC Technical Reports, Universitat Politecnica de Catalunya, 1997.

[13] Joan-Manuel Parcerisa and Antonio Gonzalez. The synergy of multithreading and access/execute decoupling. In *HPCA 5*, pages 59–63, January 1999.

[14] P. Bird, A. Rawsthorne, and N. Topham. The effectiveness of decoupling. In *Int. Conf. on Supercomputing*, pages 47–56, 1993.

[15] Joan M. Parcerisa, Antonio Gonzalez, Josep Llosa, Toni Jerez, and Mateo Valero. The performance of decoupled architectures. Technical Report UPC-DAC-1996-23, UPC-DAC Technical Reports, Universitat Politecnica de Catalunya, 1996.

[16] N. Topham and K. McDougall. Performance of the ACRI decoupled architecture: The perfect club. In *HPCN - Europe*, pages 472–480, May 1995.

[17] N. Topham *et. al.* Compiling and optimizing for decoupled architectures. In *1995 ACM/IEEE Supercomputing Conference*, San Diego, CA, December 1995.

[18] A. Gonzalez, T. Jerez, J. Llosa, J.M. Parcerisa, and M. Valero. Performance diagnostics of the ACRI-1. Technical Report UPC-DAC-1996-1, UPC-DAC Technical Reports, Universitat Politecnica de Catalunya, 1996.

[19] M. Farrens, P. Ng, and P. Nico. A comparison of superscalar and decoupled access/execute architectures. In *Micro-26*, Austin, Texas, December 1993.

[20] G.P. Jones and N.P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *Micro-30*, pages 65–70, December 1997.

[21] S. Cotofana and S. Vasiliadis. On the design complexity of the issue logic of superscalar machines. In *Euromicro '98*, pages 277–284, 1998.

[22] G. Tyson and M. Farrens. Code scheduling for multiple instruction stream architectures. *International Journal of Parallel Processing*, 22(3), 1994.

[23] J. Kreuzinger and T. Ungerer. Context-switching techniques for decoupled multithreaded processors. In *Euromicro '99*, pages 248–251, 1999.

[24] P. Marcuello, A. Gonzales, and J. Tubella. Speculative multithreaded processors. In *International Conference on Supercomputing*, July 1998.

[25] M. N. Dorozhevets and Peter Wolcott. The el'brus-3 and MARS-M: Recent advances in russian high-performance computing. *The Journal of Supercomputing*, 6(1), March 1992.

[26] M. N. Dorojevets and V. Oklobdzija. Multithreaded decoupled architecture. *Int. J. High Speed Computing*, 7(3):465–480, 1995.