Decoupled Architectures for Complexity-Effective General Purpose Processors

Ronny Krashinsky and Mike Sung
Advanced VLSI Computer Architecture (6.893) Term Project
MIT Laboratory for Computer Science, Cambridge, MA 02139
{ronny,darkman}@mit.edu

12-12-2000

Abstract

Decoupled architectures have previously been investigated in the context of high performance scientific computing. For general purpose computing, however, superscalar processors have proven to be flexible in providing high performance across a wide range of applications. To achieve this goal, these architectures have incorporated enormous amounts of complexity to obtain modest performance improvements. Looking towards the future, complexity will be of primary importance, and these inefficient designs will not be tolerated. This work investigates decoupled architectures as a complexity-effective means of obtaining high performance for general purpose workloads. A general survey of decoupled architectures is presented, as well as some proposals for incorporating decoupled architectures into a general purpose computing environment.

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 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. However, the evolution of out-of-order superscalar designs has a tradition of incoporating unwarranted complexity.

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 exection, 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 [26]. Some superscalar processors [22] use a clustered organization to simplify issue logic, whereby instructions are directed to separate clusters with independent register files and functional units. Additionally some designs [18] are incorporating deep queues to decouple instruction fetch from execution. These designs are being forced to take on some of the attributes of decoupled architectures as complexity becomes unmanageable.

More traditional decoupled architectures have primarily been targeted at scientific codes. In this paper, we investigate the potential of using decoupled architectures for general purpose computing. We 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.

Section 2 introduces decoupled access execute architectures, and Section 3 describes extending these designs with simultaneous multithreading. The addition of decoupled control flow is presented in Section 4, along with some description of loss of decoupling events. Section 5 compares the merits of superscalar and decoupled architectures. Finally, a proposal for using decoupled architectures for general purpose computing is presented in Section 6.

2 Decoupled Access/Execute

The first major investigation of decoupled architectures was done by James Smith [35, 34] (although, [34] provides a survey of several similar designs [7, 31, 32, 6]), and led eventually to the Astronautics ZS-1 Processor [12, 33]. In his preliminary study, Smith introduces the concept of a decoupled access/execute (DAE) machine, as shown in Figure 1. The 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, out-of-order superscalars).

In the proposed DAE architecture, the access processor sends load addresses to memory, and the data is pushed onto the AEQ when it arrives (if the data is to be used by the EP). The value(s) at the head of this queue can be used directly as operands to arithmetic instructions in the EP which will block if the data is not yet available; this provides a seamless integration of the architectural queues. In this way, the slots in the architectural queues take the place of resource allocation with register renaming used in superscalar processors[33, 24]. Store addresses are produced by the AP, and they enter the WAQ to wait for the corresponding store data to arrive from the EP via the EAQ. A potential advantage of this design is that loads can bypass previous stores as long as the load address does not match any of those waiting in the WAQ. Program control flow is implemented by inserting corresponding conditional branch instructions in each instruction stream, and allowing branch conditions to be passed between the processors via the AEBQ and the EABQ. When possible, the AP will determine the branch outcomes; in this case, the branch condition latency is hidden from the EP and it effectively observes unconditional branches. The performance degradation which will occur if the AP depends on a branch condition which must be determined by the EP is not discussed in this work; we will further investigate such loss of decoupling events below. Smith also points out that deadlock may occur in a DAE architecture, and describes a simple hardware detection mechanism in addition to source code constraints which can ensure deadlock avoidance.

The preliminary study also describes using 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 [12, 33]. 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 in the instruction splitter.

Descendants of the ZS-1 DAE architecture include the PIPE project [13] which was followed in turn by MISC [38, 37] (Multiple Instruction Stream Computer). The proposed MISC design (Figure 2) 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. To execute a program, the compiler (human or otherwise) partitions the computation among the four PEs. 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. To minimize control overhead, the PEs can operate in a "vector loop" mode in which a loop counter is used to execute a basic block a specified number of times. Another interesting mechanism for control is the ability to execute an instruction until the value in an input data stream matches a sentinel value (e.g. a null pointer). The MISC architecture can be considered a less scalable predecessor to tiled architectures such as RAW [11].

The WM architecture [3, 4] is another variation on 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.

3 Simultaneous Multithreading and Decoupling

In their analyses of DAE architectures [28, 29, 30], 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. The proposed architecture is shown in Figure 3. The extension to a traditional DAE architecture is relatively straightforward; the fetch and dispatch stages and the register file and queues are replicated for each context, while the functional units and data cache, which is augmented to four ports,

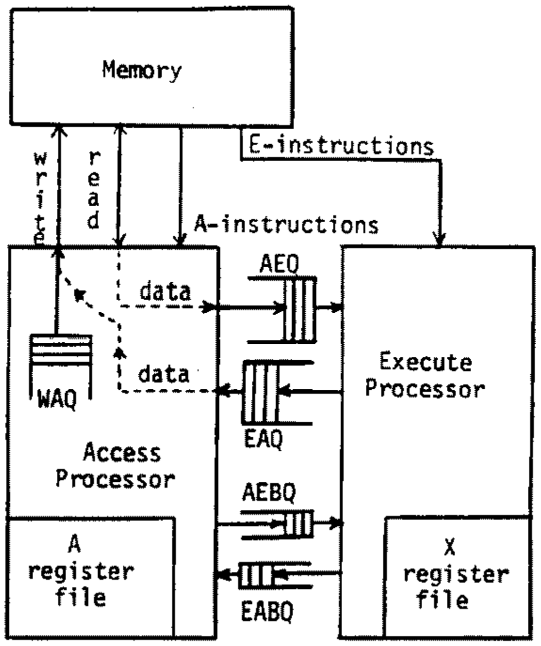

Figure 1: Decoupled access/execute architecture. Taken from [35].

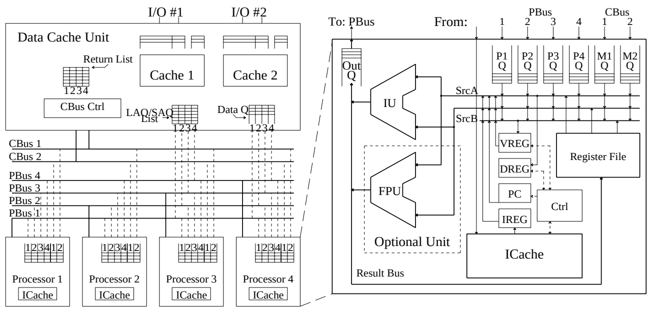

Figure 2: MISC architecture. Taken from [37].

are shared. 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 this latency.

4 Decoupled Control/Access/Execute

Another extension to the 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 [5] proposed an implementation of such an architecture, shown in Figure 4. 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 basic blocks (IFBs). These directives include the address and length of the 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 queues with ready-to-execute instructions. The address and execute processors operate on this stream of valid (non-speculative) instructions, and do not implement conditional branches. Since they process streams of valid instructions and data without using speculation, they can be considered *stream units* [27]. The ACRI description does however provide these engines 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.

In the ACRI proposal, 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 [36, 14]. In the planned ACRI implementation, the control processor

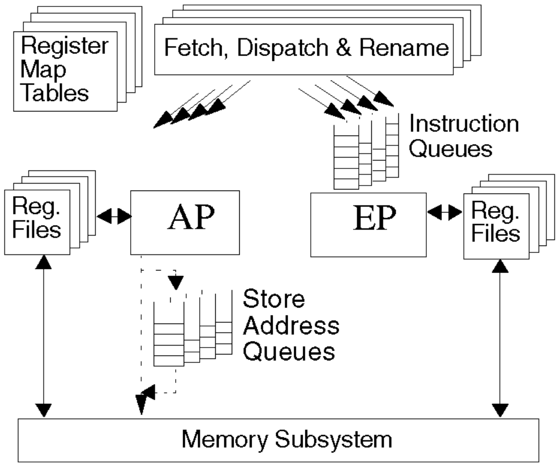

Figure 3: Multithreaded decoupled access/execute architecture. Taken from [30].

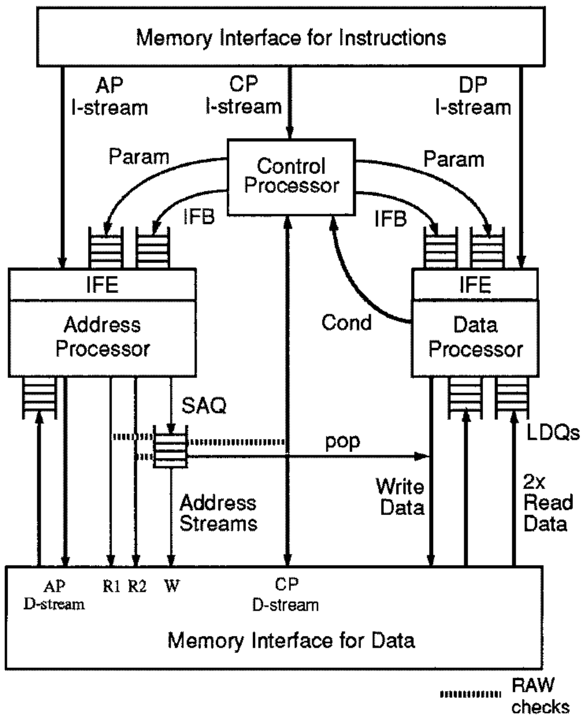

Figure 4: Decoupled control/access/execute architecture. Taken from [5].

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

Table 1: Description of LOD events of Figure 5.

was actually a DEC 21064 Alpha processor which would run the operating system as well [20]. Thus, the configuration bears some similarity to machines with a vector coprocessor that is controlled by a scalar unit.

The DCAE architecture performs best when there is full decoupling between the control, access, and execute instruction streams. Any dependencies that disrupt the decoupling and cause the instruction streams to resynchronize will adversely affect performance. There have been several studies that investigate these loss of decoupling events, termed LODs [14, 36, 17, 27]. The LODs can be characterized as shown in Figure 5. They are summarized in Table 1, and described further in [27]. Paths are provided in the architecture to resolve commonly encountered LODs, such as a condition bit sent from the EP to the CP; otherwise, the data must be communicated through memory.

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 the 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.

5 Decoupled versus Superscalar

Superscalar and decoupled architectures employ different mechanisms to obtain some of the same performance advantages. Studies comparing these architectures include the decoupled references above, as well as [24, 16, 21]. Comparisons can be made in how these architectures handle:

- Memory Latency

  Superscalar machines hide memory latency by maintaining large reorder windows to keep track of outstanding loads and the instructions which depend on them while other instructions are executed. Data prefetching can also be implemented either in hardware or software. Increasing the latency hiding requires more independent instructions [21]. Data value speculation is another technique targeted at hiding memory latency.

  Decoupled architectures hide memory latency by executing the access instruction stream in advance of the execution stream. Increasing the latency hiding can be accomplished by further decoupling these streams.

- 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.

  By determining control dependencies outside of the execution processor, decoupled architectures can effectively hide this latency. 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 general, all instructions use the same pool of resources. However, clustered superscalar architectures partition the resources.

  The queues in a decoupled machine provide a cheap form of register renaming, where the queue elements themselves provide the resources for outstanding instructions, and the architecturally visible queue head takes the place of complicated naming schemes. Additionally, resources are partitioned between the decoupled processors.

- 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. Additionally, the three separate instruction streams provide an immediate source of ILP.

Thus, to achieve performance, superscalar architectures must expend a great proportion of area and complexity on issue and decode logic, including the instruction window, reorder buffer, register renaming logic, and bypass logic. It is widely accepted that the issue logic of superscalar architectures is becoming increasingly expensive to implement in terms of area, delay, and power consumption [26, 8]. Large structures such as issue windows that require associative dependency-checking are rapidly becoming the limiting factor in scaling the issue rate of superscalar machines [1]. Additionally, superscalar architectures rely on speculation and prediction which incurs a large amount of overhead.

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. 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 [11]. This is a logical extension to the standard decoupled architecture in which independent decoupled streams can be run on separate tiles of the architecture.

Given the 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. Decoupled architectures have traditionally been targeted at scientific code which is not control intensive. As described earlier, the Achilles heel of decoupled architectures come from the LOD events, so programs with complicated control flow can suffer from severe performance degradation on decouple architectures. Another possible limitation of decoupled machines is the complexity involved in compiling programs to separated instruction streams. More research will be necessary in this area, although compilers have been developed for decoupled architectures [37, 14, 15].

6 Empty Decay

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. Thus, 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, this latency can be hidden if we have multiple threads executing concurrently.

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 [30]. We propose to extend this idea and provide multithreading on a DCAE architecture. A 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.

For the MT-DCAE architecture, we can leverage much of the design from the ACRI architecture [5] and the multithreaded DAE design [30]. Multithreading on the MT-DCAE processor can improve performance on two levels by handling both LOD dependencies and instruction dependencies within a thread. 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. The multithreading can be implemented by replicating the instruction fetch block (IFB) and parameter queues of the ACRI architecture (one for each thread). The address and execute processors themselves would have state for multiple contexts (i.e., instruction queues and register files) to allow for fast context switches between threads. These design issues can be leveraged from existing SMT studies. A block diagram of our proposed MT-DCAE architecture is shown in Figure 6.

6.1 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 relatively fast context switching capabilities, and have it switch control threads whenever it encounters a long latency event such as a LOD. In this way, the control processor can switch to processing another thread while the address and data processors continue to process instructions from the fist 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 these resources.

An important design issue is how the control processor should schedule threads to fully take advantage of the MT-DCAE architecture to hide LOD latency. We note that maximum performance occurs when the execute processor is fully utilized. The main function of the CP and AP is to allow prefetch of instructions and data such that the execution processor is constantly running. Since the control processor's instruction streams are ideally processed significantly faster than those of the execution processor, it can possibly afford not to have extremely efficient thread context switching. Thus, it may not be necessary to add ad-

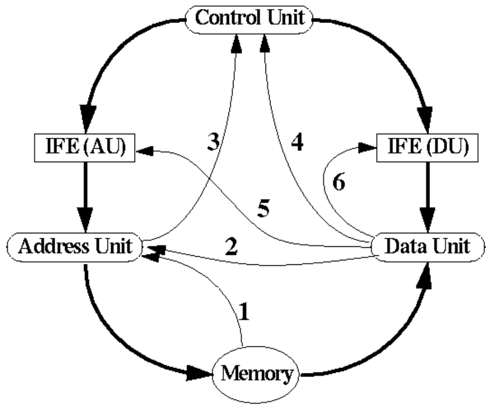

Figure 5: Loss of decoupling events corresponding to Figure 4. The bold arrows represent decoupled execution, while each numbered arrow depicts an LOD event. Taken from [27].

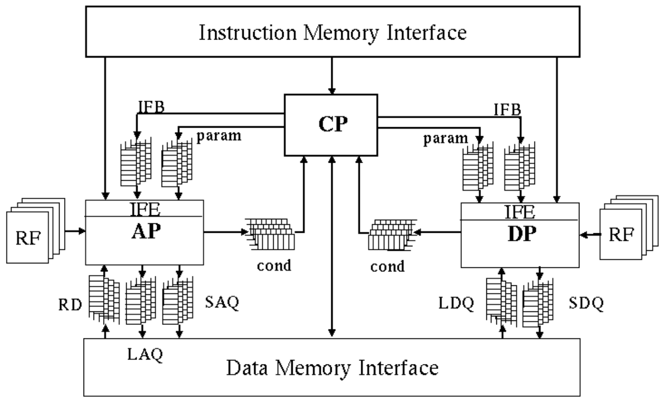

Figure 6: MT-DCAE multithreaded decoupled control/access/execute architecture.

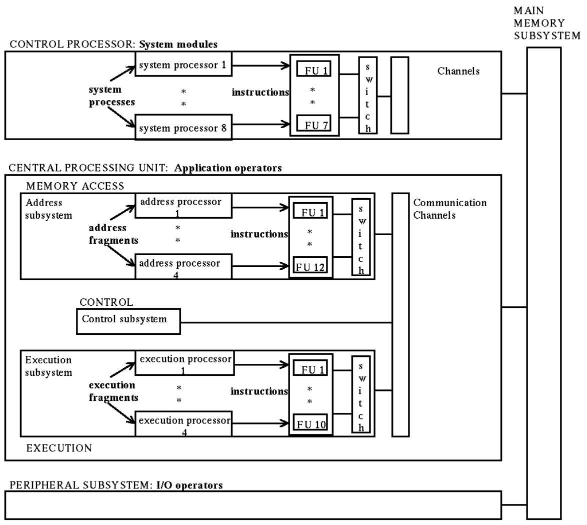

Figure 7: MARS multithreaded decoupled control/access/execute architecture. Taken from [39].

ditional 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.

6.2 Speculative Multithreading

Another technique that may improve performance on MT-DCAE is speculative multithreading [25]. 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 [27]. In this description, 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 discarded.

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.

6.3 DCAE with Superscalar

Superscalar processors will probably always be better than decoupled architectures at processing code which is extremely control intensive. To get the best out of both architectures, we can consider implementing the control processor in a DCAE design as a more capable high-performance microprocessor. Then, the compiler can choose to avoid decoupling 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. The decoupled architecture is a better fit for streaming code such as multimedia; but such code is often mixed with control intensive portions of computation, so the hybrid architecture can be a good choice. This type of computational model is similar to that used for vector or SIMD array coprocessors [2, 19]. Decoupled access/execute engines are more flexible than these alternatives, but a comparison of performance and complexity would be interesting.

6.4 More Related Work

It is noteworthy that Dorojevets, et. al. [10, 39] implemented a MT-DCAE architecture, the MARS-M (Modular, Asynchronous, Extensible Systems) computer, shown in Figure 7. This architecture supports simultaneous execution of up to four address, four data, and one control thread, with communication provided by various queues. The control processor is responsible for splitting a thread's unified instruction stream and passing code fragment calls (analogous to the IFBs of the ACRI) to either address or execution processors. These processors are implemented as VLIW processors which support simultaneous multithreading. Both the address and execution processors and the communication queues are dynamically assigned. In a follow up study, [9] makes the observation that control dependencies hinder parallel execution, and proposes speculative multithreading.

7 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. We do not attempt to propose any specific implementation as it is beyond the scope of this project to be able to provide any meaningful simulation results. 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] K. Asanović. Vectorizing SPECint95. Unpublished manuscript extracted from PhD Thesis, March 1998.

[3] Wm. A.Wulf. The WM computer architecture. *Computer Architecture News*, 16(1), March 1988.

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

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

[6] W. C. Brantley and J. Weiss. Organization and architecture tradeoffs in fom. In *IEEE Int. Workshop Comput. Syst. Organization*, New Orleans, LA, March 1983.

[7] E. U. Cohler and J. E. Storer. Functionally parallel architectures for array processors. *IEEE Computer*, 14(9):28–36, September 1.

[8] S. Cotofana and S. Vasiliadis. On the design complexity of the issue logic of superscalar machines, 1998.

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

[10] 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.

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

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

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

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

[15] W. Lee *et. al.* Space-time scheduling of instruction-level parallelism on a raw machine. In *ASPLOS 8*, pages 4–7, San Jose, CA, October 1998.

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

[17] 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.

[18] L. Gwennap. Mips r10000 uses decoupled architecture. *Microprocessor Report*, October 1994.

[19] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings FCCM*, pages 24–33, April 1997.

[20] http://www.paralogos.com/DeadSuper/ACRI/.

[21] 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.

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

[23] J. Kreuzinger and T. Ungerer. Context-switching techniques for decoupled multithreaded processors, 1999.

[24] W. Mangione-Smith, S.G. Abraham, E.S. Davidson, and J. E. Smith. A performance comparison of the ibm rs/6000 and the astronautics zs-1. *IEEE Computer*, 24(1):39–46, January 1991.

[25] P. Marcuello, A. Gonzales, and J. Tubella. Speculative multithreaded processors, July 1998.

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

[27] 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.

[28] 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.

[29] Joan-Manuel Parcerisa and Antonio Gonzalez. Improving latency tolerance of multithreading through decoupling. Technical Report UPC-DAC-1999-28, UPC-DAC Technical Reports, Universitat Politecnica de Catalunya, 1999.

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

[31] A. R. Pleszkun. A structured memory access architecture. Technical Report CSG-10, Coord. Sci. Lab., Univ. Illinois, Urbana, Comput. Syst. Group Rep, October 1982.

[32] R. R. Shively. Architecture of a programmable digital signal processor. *IEEE Trans. on Computers*, C-31, January 1982.

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

[34] J. E. Smith, S. Weiss, and N.Y. Pang. A simulation study of decoupled architecture computers. *IEEE Computer*, C-35(8):692–702, August 1986.

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

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

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

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

[39] Peter Wolcott. *Soviet Advanced Technology: The Case of High-Performance Computing*. University Microfilms, Inc., Ann Arbor, MI, 1993.