Disintermediated Active Communication

Anne Bracy[*], Kshitij Doshi, Quinn Jacobson,
Intel Corporation

***Abstract*— Disintermediated Active Communication (DAC) is a new paradigm of communication in which a sending thread actively engages a receiving thread when sending it a message via shared memory. DAC is different than existing approaches that use passive communication through shared-memory—based on intermittently checking for messages—or that use preemptive communication but must rely on intermediaries such as the operating system or dedicated interrupt channels.**

An implementation of DAC builds on existing cache coherency support and exploits light-weight user-level interrupts. Inter-thread communication occurs via monitored memory locations where the receiver thread responds to invalidations of monitored addresses with a light-weight user-level software-defined handler. Address monitoring is supported by Cache Line User-bits, or CLUbits. CLUbits reside in the cache next to the coherence state, are private per thread, and maintain user-defined per-cache-line state. A light weight software library can demultiplex asynchronous notifications and handle exceptional cases.

In DAC-based programs threads coordinate with one another by explicit signaling and implicit resource monitoring. With the simple and direct communication primitives of DAC, multi-threaded workloads synchronize at a finer granularity and more efficiently utilize the hardware of upcoming multi-core designs.

This paper introduces DAC, presents several signaling models for DAC-based programs, and describes a simple memory-based framework that supports DAC by leveraging existing cache-coherency models. Our framework is general enough to support uses beyond DAC.

I. DAC Concept

With increased hardware concurrency, efficient use of thread level parallel (TLP) computation is necessary for achieving high performance. Whereas in instruction level parallel (ILP) computation the processor microarchitecture educes data or control dependences for efficient scheduling, in TLP the burden for efficient conveyance of dependences falls on software. This burden consists of both relaying the changes of state from one thread to another efficiently, and in recognizing and reacting to them quickly. Coming years will see the maturing of new languages (such as CHAPEL [1]) to achieve easy and abstract expression of concurrencies and data-flows at flexible granularities and to automate the composition of parallel execution on a range of hardware. This places us at the threshold of an exciting opportunity: creating the hardware and runtime enhancements that provide seamless continuity between the data/control flows of dynamically scheduled single processor pipelines and the event flows in thread parallel computation. Dependences between thread level computations made explicit by the programmer or elicited by the language could be handled by timely propagation of nudges between such computations, and the management of scheduling handled transparently to the programmer. To actuate such event-coupled parallel thread execution, inter-thread communication needs to be flexible, instruction granular, and reactive.

Efficient fine-grained communication is achieved today in a model such as MPI [5] through a message channel: the sender of communication places a message in the channel, while the receiver polls the channel at some regular or irregular interval. The channel can be implemented as an efficient shared memory structure enabling communication that is both flexible and fine grained. However, it lacks reactivity: the receiver must choose how frequently to poll and what to do while waiting for a message to arrive.

For the sender of a message to cause the receiver to react quickly, a baseline mechanism that can be employed today is to force a hardware exception at the receiver, using a mechanism such as the inter-processor interrupt (IPI), so that the receiver's control can be switched from whatever it was doing, to a message receiving handler. The overhead for such an approach is not trivial, ranging from a thousand cycles to tens of thousands of cycles, depending upon the complexity of the operating system-mediated interrupt delivery. Special purpose hardware can reduce the need for such operating system intervention, such that a signal sent from one processor to another can cause the receiver to enter a very lightweight exception handler preemptively. The DAC (*Disintermediated Active Communication*) approach proposed in this paper incorporates this method in the abstract. However, instead of implementing new inter-processor signaling hardware, DAC builds upon signaling already provided for cache coherence, and thereby unifies the act of storing to a memory location with that of active messaging via the same location.

DAC uses active, memory-based store signaling. Conceptually, a receiver thread asks the processor to monitor some memory variable on its behalf, while the thread performs computation. The processor notifies the thread when the monitored variable is the target of a store by automatically invoking a designated application space software handler – although the thread has the option to receive such notifications through passive polling of status as well. The monitor-notify interaction in DAC can be intra-thread or inter-thread, or inter-process via shared memory.

II. Abstract Application of DAC

One idiomatic use of the monitor-notify approach of DAC is for a thread to designate a special memory location it agrees

[*] Anne Bracy is also a student at the University of Pennsylvania.

Manuscript submitted: 23 Aug. 2006. Manuscript accepted: 23 Oct. 2006. Final manuscript received: 1 Nov. 2006.

Posted to IEEE & CSDL on 11/09/2006
DOI 10.1109/L-CA.2006.15

to monitor, and advertise this location to all other threads. This memory location can be called the thread's doorbell. The thread then does its assigned work, until some other thread gets its attention by storing into the doorbell location. How a particular thread responds to its doorbell is defined entirely by the application programmer. Threads can use doorbells to distribute, coordinate and synchronize parallel computations among each other. They can proactively ask each other to produce or consume data, resources, and conditions.

Fundamentally, DAC enables the push-based model of delivering notifications instead of the traditional pull-based model in which receivers check whether an event has transpired.* Instead of requiring all threads of computation to repeatedly check for a rare but critical condition, one thread can monitor the condition and when it develops, and alert all others through their doorbells. Liberated from having to poll, a thread can pick up a large grain of work and proceed to complete it in a streamlined manner, safe in the knowledge that it will be preempted quickly if a critical event arises. Conditions that require global or barrier synchronization for brief periods of time (*e.g.,* a new mapping appears in the program address space) can be easily accommodated by storing into everyone's doorbells with fanned broadcasts.

A thread that announces a doorbell for itself effectively uses the DAC mechanism to check continually if anyone stores into the doorbell, even as the thread continues to do some other work. We call this behavior *virtual polling* by the thread. Virtual polling is not limited to explicitly advertised doorbells; it arises whenever a thread uses DAC to perform observation in the background while proceeding with work in the foreground. A powerful use of virtual polling is in waiting for releases of contended locks. For example a thread that fails to acquire a user level lock [2] can monitor the lock variable's memory address for a store. Until it is notified of the store, it can pursue other work, engage in speculative computation in absence of the lock, enter itself into a power saving state, or pre-fetch data it might need elsewhere. Although a thread has the option to do any of these even without virtual polling, virtual polling allows the aggregation of such "foreground work" so that it can be transacted flexibly in the shadow of a lock miss whose duration is difficult to predict. More generally, virtual polling can be used to maximize the duty cycle of any critical resource by ensuring that the contenders for the resource acquire that resource as early as possible, but without busy-waiting for the resource. A user-level software library can be used to wrap the DAC mechanism and provide a simple consistent interface. The library can keep a log of all memory locations being monitored and their previous value. In response to resource constraints the library can transparently revert to physical polling at an appropriate fidelity, exploiting the user-level interrupt mechanism as a timer. In response to disruptive events, such as a context switch, the library can poll all the monitored memory locations and reactivate hardware monitoring. The library can also be used to demultiplex notification so that a program can have specific handlers invoked in response to specific variables or even specific values. This means that through successive generations of DAC capable hardware, the exact capacity of the DAC mechanism in a single generation does not dictate the structuring of software meant to benefit from the availability of the monitor-notify mechanism.

* Operating system supported condition variables or events provide a push-based communication model to programmers, but the overhead of these features limits their usefulness for fine-grain thread coordination.

III. DAC Framework

The architecture that supports DAC contains three main components: a small extension to the architectural state of the local cache lines, a user-level control transfer mechanism, and three new instructions that enable software to manipulate and observe the new state in the cache.

A. *Cache Line User bits (CLUbits)*

Our DAC framework is shown in Figure 1. We start with a simple CMP design where each core has a single hardware thread context and a private data cache. Lower level caches may be shared between cores. Cache coherency is maintained on a per cache line basis, with a MESI protocol.

Central to the architecture is the addition of a set of Cache Line User bits, or **CLUbits**, to augment the cache line state in the memory hierarchy. The concept of adding extra state bits to cache lines for specific purposes has been proposed in numerous prior works. For example, a Speculative Versioning Cache (SVC) [3] contains a speculative load bit, which is set by particular types of load instructions and triggers actions if another thread invalidates the line. Like many TLS proposals [8], our work derives from this concept. Unlike SVC and its derivatives, our framework distills the concept of added cache state down to a general building block for multiple usages.

From the program's point of view, each cache line has a few CLUbits that can be read, written, and interpreted exclusively by the program. We arbitrarily assume four CLUbits residing next to the tag and coherency state information for each cache line (see Figure 1). The meaning of each of the 4 different flavors of CLUbits is defined by the program. For example, whereas one CLUbit might be used to monitor a particular resource, another CLUbit can be used to receive "stop-the-world" signals from other threads. The program is responsible for determining not only what each CLUbit flavor means, but also which cache lines' CLUbits should be set or tested.

CLUbits can be supported through any layer of the cache hierarchy. Without loss of generality, we consider only the first-level data cache. When a cache line enters the first-level cache, CLUbits for that line are set to the default value 0. CLUbits are cleared when a cache line is invalidated or evicted, and at a context switch.

The CLUbits can be easily extended to different cache organizations. If CLUbits are supported in a shared cache, which would include the first-level cache in the case of SMT, there must either be a set of CLUbits for each hardware thread sharing the cache or an ownership field (the latter saves bits but limits the ability for multiple threads to monitor the same address). For SMT, or any other case of shared caches, writes from another hardware thread sharing a cache must appear like an invalidate to a thread monitoring an address. In a write update protocol, the update clears the CLUbits.

B. User-Level Control Transfer Mechanism

The next component of our framework is a user-level asynchronous control transfer mechanism similar to the YIELD-CONDITIONAL [9]. Like YIELD-CONDITONAL, a special instruction allows software to register both its interest in a class of events and the beginning address of its own handler. If a registered event occurs, the mechanism triggers, transferring control of the program to the user-supplied handler procedure. The architectural state for the trigger response mapping is saved and restored by the OS across thread context switches. Handlers are defined by the application and execute at the application's privilege level. Because handlers are activated as asynchronous function calls, they must save and restore the state of any registers they touch.

When software registers its interest in monitoring an address, it must also indicate the CLUbit flavor. For each flavor the software can specify the handler it wishes to be asynchronously invoked. A single handler can be used for all flavors or different handlers can be used. If more than one handler is used the priority for invocation is programmable.

Figure 1 shows a limited view of one possible state of the control-transfer mechanism. The software has registered its interest in the second of 4 CLUbit flavors and has specified the address 0xA as the handler address. Should any monitored lines *of this flavor* be invalidated or evicted from the cache, control will transfer to this handler in response.

C. ISA Extensions for CLUbit Manipulation

Three new instructions are added to the existing ISA to provide software an interface for CLUbit management. To begin monitoring, software can set a CLUbit via the **ld_set_CLUbit** instruction. In addition to the destination register, this instruction takes the following arguments: (1) the address on which it performs a normal load, (2) the CLUbit flavor, and (3) the value to which the CLUbit should be set. For example, ld_set_CLUbit([addr], 0100, 1) loads the value at address [addr], and sets the second flavor CLUbit to 1. The result of executing this instruction is shown in Figure 1.

When a cache line has at least one CLUbit set, the line is being monitored. Setting a CLUbit tells the processor "watch this location for me." The program decides how many cache lines to monitor and can discriminate between different usage models by using different CLUbit flavors.

Whenever the coherency state of a line in the local cache of a core changes to INVALID, the core checks to see whether this line has any CLUbits currently set and whether this flavor has a handler registered to it. If this is the case, the handler will be asynchronously invoked.

A program may choose not to register a handler for a particular flavor but instead check the value of a CLUbit directly. This is supported by the **ld_check_CLUbit** instruction. This instruction is identical to ld_set_CLUbit but rather than setting a particular CLUbit, it checks its value. The value of the CLUbit can be placed in an appropriate location based on the ISA, such as recording the value in a flag register.

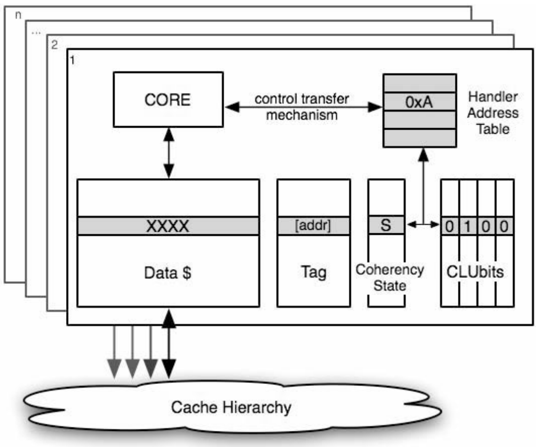

Fig. 1. Microarchitecture Supporting DAC. CMP design with private data cache and MESI cache coherence protocol. Cache Line User bits (**CLUbits**) extend the state of each cache line by 4 bits.

With the added ability to check the value of a CLUbit, our framework becomes a generalization of many instruction instrumentation techniques. CLUbits can serve as the filter bits used to avoid read barriers in hardware accelerated STM [6]. Informing memory operations [4] are memory instructions followed by a software check for a cache miss. CLUbits could implement these operations, but can trigger notifications for software-specified events far beyond cache misses. CLUbits could also be used to control memory accesses much like Blizzard-S's software-managed fine-grain access control [7]. Finally, CLUbits would be useful in implementing debuggers such as iWatcher [10], which uses tag bits in L1 and L2 caches, and triggers program specified monitoring functions upon detection of access. Although our framework supports these uses, they are not discussed further because they are beyond the scope DAC and therefore this paper.

When monitoring is no longer required, the software invokes the instruction **clear_CLUES** to cease all monitoring on a particular flavor of CLUbit. This instruction takes just one argument: the mask of the CLUbit flavors it should clear. For example, clear_CLUES(1010) flash clears all CLUbits in the first and third flavors of all cache lines.

IV. Examples

We show two examples of DAC use.

K-Compare Single-Swap (KCSS). KCSS is a powerful atomic primitive used to compare k variables to their expected values and set a final variable to a new value if all match. Pseudocode might look something like this:

```
atomic ( if (($a_1$ == $a_2$)&&($b_1$ == $b_2$)&& … ($k_1$ == $k_2$))
         { temp = $x_1$; $x_1$ = $x_2$; $x_2$ = temp; } )
```

KCSS can be used to perform safe non-blocking updates to structures. When modifying a linked list pointer, for example, all neighboring pointers must remain unchanged, else elements can be orphaned. Previously proposed KCSS implementations either require prohibitively many load-lock instructions or weaken semantics. DAC makes a complete and efficient KCSS implementation possible, as sketched below:

```
kcss(…) {
#define dacld(addr) ld_set_CLUbit(&addr,1000,1)
  if ((tmp = dacld(a1) !=a2) goto fail;
  if ((tmp = dacld(b1) !=b2) goto fail;
  …
  if ((tmp = dacld(k1) !=k2) goto fail;
  swap (x1, x2);
  fail: clear_CLUES;
}
kcss_dac_handler(): { clear_CLUES; ...  }
```

If any of the comparisons fail, this version of KCSS does not perform the swap; but it achieves atomicity without enforcing a costly barrier. Because the locations of $\mathbf{a}_1 \ldots \mathbf{k}_1$ are monitored by DAC, any change to their value will cause a preemptive yield to the kcss_dac_handler, which simply clears the current monitoring and terminates the swap. The example also shows the benefit of multiple CLUbit flavors (attribute bits): KCSS registers and uses or clears one series of monitor bits (1000 in this example) while other DAC uses are operative in the background.

Message queuing. Message queuing is a common form of communication between threads. The receiver frequently checks a channel for messages. Unless communication is at regular and predictable intervals, the receiver will often check for messages when there are none pending or receive new messages a while after they were ready. With DAC, pairs of threads use doorbells to notify receivers of message delivery.

We compare three possible message queuing techniques: spin-wait, DAC doorbells, and IPI wakeups. The scenario is as follows: four threads on a CMP sit in a logical ring configuration. A token is passed between the threads in a round-robin fashion for some number of rounds. Upon acquiring the lock, each thread increments a counter and passes the token to the next thread.

The spin-wait case is as follows:

```
while (locked_counter < NUM_ROUNDS)  {
  if (permission[my_id] != 0) {
    permission[my_id] = 0;
    locked_counter++;
    permission[next_id] = 1; } }
```

Threads repeatedly check a permission variable and proceed when it has been set. Threads quickly recognize permission, but spend all their wait time performing the check.

The DAC doorbell case is as follows:

```
value = ld_set_CLUbit(&doorbell[my_id], 1);
while (locked_counter < NUM_ROUNDS)
   // do whatever I like
Invalidation_handler () {
  if (!(value =ld_set_CLUbit(&doorbell[my_id]))
    return; // false alarm (maybe eviction?)
  doorbell[my_id] = 0;
  locked_counter++;
  doorbell[next_id]) = 1;
}
```

Rather than explicitly checking to see whether permission is granted, a thread simply sets the CLUbit on its doorbell and registers its interest in that flavor's invalidations (not shown). When the doorbell is rung (written to) by another thread, the original thread's local copy is invalidated, and its Invalidation_handler is invoked. This suffers the penalty of flushing the pipeline to invoke the user-level asynchronous control transfer mechanism. The thread then recognizes that it has been given permission to increment the counter, does so, and rings the doorbell of the next thread.

Our DAC framework is modeled by a cycle-based simulator of a modern x86 multi-core processor, roughly based on Intel's Core Duo, with a 4-way issue processor core and a 64 entry ROB. There are 32KB level one instruction and data caches. All test cases fit in the level two cache. The level one and two data caches have a 3- and 16-cycle load-to-use latency respectively. We compile with gcc version 3.3.2 (i686-pc-linux-gnu) with -O2 optimization.

We ran this scenario for one thousand rounds and measured the average per-hop cycle latency for communication. The average per-hop latency was 199 cycles for the spin-wait case and 383 cycles for the DAC case. DAC has higher overhead than the basic spin-wait mechanism because DAC invokes a handler, which has more instructions than a simple while loop. This overhead, however, is less than twice the overhead of the spin-wait implementation. The final technique (OS-supported IPI communication) achieves the same functionality by going through the OS and invoking IPIs between cores. We estimate the cost to be on the order of 10,000 cycles.

Although the spin-wait mechanism has less overhead, DAC enables a thread to respond to lock availability as it accomplishes useful work. With DAC, a thread never needs to check to see whether the lock is ready; it responds consistently independently of the work it is doing in the background. This allows a DAC-based program to be more responsive to the availability of permissions.

References

[1] D. Callahan, B. C. Chamberlain, H.P. Zima, The Cascade High Productivity Language, 9th Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) April 2004.

[2] H. Franke, R. Russel, and M. Kirkwood. Fuss, futexes, and furwocks: Fast user level locking in Linux. Ottawa Linux Symposium, 2002.

[3] S. Gopal, T. N. Vijaykumar, J. E. Smith, G. S. Sohi. Speculative Versioning Cache, In HPCA 1998.

[4] M. Horowitz, M. Martonosi, T. C. Mowry, M.D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In ISCA 1996.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Intl. Journal of Supercomputer Application, 8(3/4). 1994.

[6] B. Saha, A. Adl-Tabatabai, Q. Jacobson. Architectural Support for Software Transactional Memory. In ISCA 2006.

[7] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In ASPLOS 1994.

[8] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In HPCA 1998.

[9] P. Wang, J. Collins, H. Wang, D. Kim, B. Greene, K. Chan, A. Yunus, T. Sych, S. Moore, and J. Shen. Helper Threads via Virtual Multithreading On An Experimental Itanium 2 Processor-based Platform. In ASPLOS 2004.

[10] P. Zhou, F. Qin, W. Liu, Y. Zhou, J. Torrellas. iWatcher: Efficient Architectural Support For Software Debugging. In ISCA 2004.