Efficient Execution of Memory Access Phases Using Dataflow Specialization

Chen-Han Ho   Sung Jin Kim   Karthikeyan Sankaralingam
University of Wisconsin-Madison
{chen-han,sung,karu}@cs.wisc.edu

Abstract

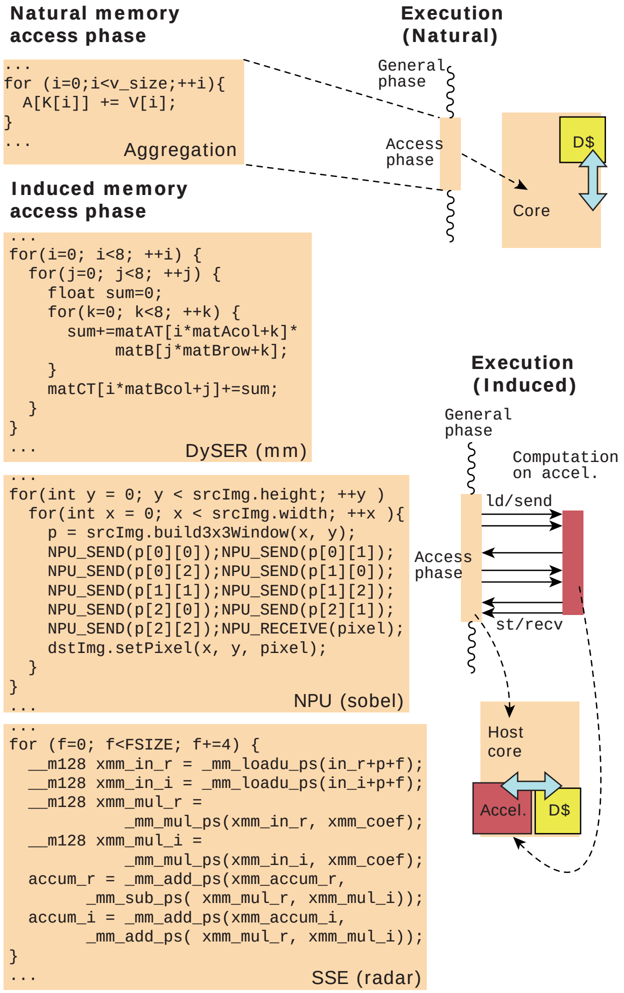
This paper identifies a new opportunity for improving the efficiency of a processor core: memory access phases of programs. *These are dynamic regions of programs where most of the instructions are devoted to memory access or address computation. These occur naturally in programs because of workload properties, or when employing an in-core accelerator, we get induced phases where the code execution on the core is access code. We observe such code requires an OOO core's dataflow and dynamism to run fast and does not execute well on an in-order processor. However, an OOO core consumes much power, effectively increasing energy consumption and reducing the energy efficiency of in-core accelerators.*

We develop an execution model called memory access dataflow (MAD) that encodes dataflow computation, event-condition-action rules, and explicit actions. Using it we build a specialized engine that provides an OOO core's performance but at a fraction of the power. Such an engine can serve as a general way for any accelerator to execute its respective induced phase, thus providing a common interface and implementation for current and future accelerators. We have designed and implemented MAD in RTL, and we demonstrate its generality and flexibility by integration with four diverse accelerators (SSE, DySER, NPU, and C-Cores). Our quantitative results show, relative to in-order, 2-wide OOO, and 4-wide OOO, MAD provides 2.4×, 1.4× *and equivalent performance respectively. It provides* 0.8×, 0.6× *and* 0.4× *lower energy.*

1. Introduction

This paper is a specialization technique targeted at a prevalent and growing category of program behavior: memory access phases. A memory access phase is a dynamic portion of a program where its instruction stream is predominantly for (as a heuristic say 90%) memory accesses and address generation.

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3402-0/15/06$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750390



```
...
for (i=0;i<v_size;++i){
  A[K[i]] += V[i];
}
...
```

```
...
for(i=0; i<8; ++i) {
  for(j=0; j<8; ++j) {
    float sum=0;
    for(k=0; k<8; ++k) {
      sum+=matAT[i*matAcol+k]*
           matB[j*matBrow+k];
    }
    matCT[i*matBcol+j]+=sum;
  }
}
...
```

```
...
for(int y = 0; y < srcImg.height; ++y )
  for(int x = 0; x < srcImg.width; ++x ){
    p = srcImg.build3x3Window(x, y);
    NPU_SEND(p[0][0]);NPU_SEND(p[0][1]);
    NPU_SEND(p[0][2]);NPU_SEND(p[1][0]);
    NPU_SEND(p[1][1]);NPU_SEND(p[1][2]);
    NPU_SEND(p[2][0]);NPU_SEND(p[2][1]);
    NPU_SEND(p[2][2]);NPU_RECEIVE(pixel);
    dstImg.setPixel(x, y, pixel);
  }
}
...
```

```
...
for (f=0; f<FSIZE; f+=4) {
  __m128 xmm_in_r = _mm_loadu_ps(in_r+p+f);
  __m128 xmm_in_i = _mm_loadu_ps(in_i+p+f);
  __m128 xmm_mul_r =
            _mm_mul_ps(xmm_in_r, xmm_coef);
  __m128 xmm_mul_i =
            _mm_mul_ps(xmm_in_i, xmm_coef);
  accum_r = _mm_add_ps(xmm_accum_r,
        _mm_sub_ps( xmm_mul_r, xmm_mul_i));
  accum_i = _mm_add_ps(xmm_accum_i,
        _mm_add_ps( xmm_mul_r, xmm_mul_i));
}
...
```

Figure 1: Natural and Induced Memory access phases

In compiler terms, the load- and store- back-slice contribute to 90% of dynamic instructions.

Observation We observe that memory access phases are naturally prevalent in many applications. Profiling the SPECINT2006 and Mediabench suite shows many such phases; Table 1 explains their qualitative roles. We call these *natural* memory access phases or simply natural-phases through the rest of this paper and they are sketched at the top of Figure 1. A second (and rapidly growing) category are program phases that include code that executes concurrently on an in-core accelerator and processor core. Here the code running on the main processor feeds values to the accelerator. Examples include code running with NPU [25], Convolution

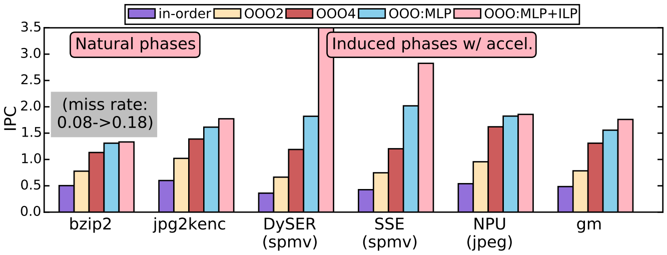


Figure 2: Performance on natural and induced phases

Engine [56], and DySER [32]. On compiler-generated or handwritten SSE/AVX code regions, the non-compute instructions form such a phase. We call these phases induced memory access phases (or *induced-phases*). Their characteristic of memory access behavior is induced by offloading the computation to an accelerator[1]. With more accelerators, there will be more and more induced phases increasing the opportunity for specializing the core itself for such phases. We first make the overarching observation that many accelerators can be viewed as executing under the decoupled access execute paradigm [62].

Our analysis of these phases, leads us to the following observations. The properties of natural and induced phases are similar and present an opportunity. They commonly have abundant instruction-level parallelism (ILP) and memory-level parallelism (MLP - ability to issue far away memory accesses), they have much dynamic behavior in the cache hits/misses, they have some control-flow, and they commonly have a small static-code footprint of 10 to 300 instructions.

Figure 2 presents quantitative measurements that support these observations by showing two representative natural phases and three induced phases on three different accelerators, and geometric mean across our entire suite (details in Section 6). We first compare performance on an in-order machine to two realistic out-of-order machines (OOO2 and OOO4). Across natural-phases and induced-phases, in-order cores are 2× to 4× worse than realistic OOO counterparts. While an OOO core with ILP, MLP, and dynamism tolerance is useful, it comes at an exorbitant power cost. Compared to the in-order core, OOO2 is almost a factor of three higher in power, and an OOO4 is another factor of three. Because of this, the *energy improvement* from very low-power accelerators like DySER, NPU, and even SSE (which consume on the order of 200 to 400 mWatts [32, 33, 25, 19]) is constrained by the 2-watt to 7-watt OOO core (details in Section 6.2). Integrating these to OOO cores provides, overall *higher energy* than integrating them with an in-order core (Table 3 in Results).

To examine whether there is the opportunity to exceed the performance of an OOO core, we look at two hypothetical configurations which isolate the benefits of MLP alone (an OOO8 with 64 cache ports) and MLP+ILP (64-wide issue OOO with 64 cache ports). Examining the last two bars, we see there is a potential for 1.5× *improvement* over an OOO4 and up to 4× in some cases.

Domain	Benchmark Suites	Examples
Database Analytics	Database kernels	Array aggregation(`aggre`), compare records to keys and update results(`filescan`).
Media Processing	MediaBench [42]	Convert samples to the output color space(`djpeg`), synthesis filter(`gsmdecode`).
Irregular Codes	SPECINT	Block sort(`bzip2`), check the cost of the arcs and update the results(`mcf`).
Accelerators (DySER, SSE, C-Cores)	Parboil, Rodinia, Throughput-Kernels	Fast Fourier Transform(`fft`), dense linear algebra(`kmeans`), convolution (`conv`, `radar`).
Accelerators (NPU)	NPU benchmarks [25]	Fast Fourier Transform(`fft`), Jpeg encoding (`jpeg`).

Table 1: Memory access regions

Problem statement This paper investigates the question of how to build a power efficient and high performance mechanism for executing memory access phases. Such a mechanism can serve as a general way for in-core accelerators to integrate with high-performance or low-performance cores without compromising performance yet running at low power by turning off the core during such phases.

Lessons from OOO An OOO core does the following for high performance: it dynamically unrolls the program, maintains it in the instruction window, repeatedly extracts the dataflow graph, and uses register renaming and a large physical register file to maintain multiple copies of an instruction's output across its various dynamic instances. Abstracting away microarchitecture implementation details, an OOO core's primitives are: (1) *dataflow computation* of address and control conditions using the extracted graph; (2) the outcome of these few dataflow computation patterns create *events* inside the core pipeline (like values returned from cache); and (3) based on these events, it performs *actions* like moving data between the register file and memory (and an accelerator on induced phases). Dataflow computation extracts ILP and concurrent events and actions exploits MLP and dynamism.

Overview Our execution model and hardware architecture called Memory Access Dataflow (MAD), expresses the aforementioned OOO core's primitives actions using the concept of event-condition-action (ECA) rules and named event queues for storage. ECA rules are borrowed from the databases and algebraic theory literature and comprise two parts: first it formally state a set of conditions as a boolean function of a set of events, and second a set of actions are triggered when the function evaluates to true. The idea of ECA rules elegantly and explicitly specifies the OOO core's primitives in the MAD ISA. To explain, named event queues store data values produced or consumed in a access phase. A programmable hardware substrate implements the conditions and triggering logic to

[1] Offloading computation to off-core accelerators makes induced-phases more prominent in the rest of the original program - there are not yet that many benchmarks/applications of this type. Hence we don't quantitatively consider this. Regardless our technique should work there as well.

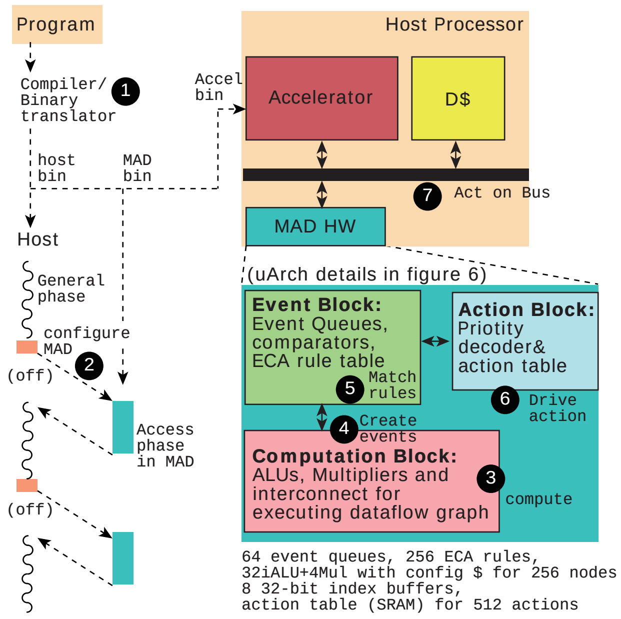


Figure 3: MAD Overview

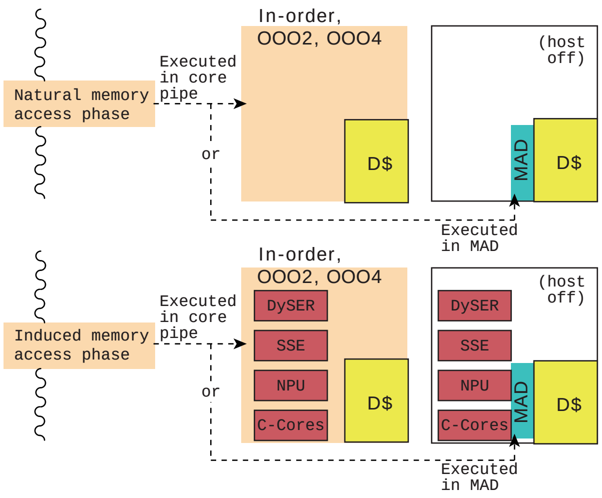


Figure 4: Integrating MAD with Accelerators

initiate actions based on the event queues. A dataflow substrate computes addresses and values that are deposited into the queues.

Compared to an OOO, MAD extracts ILP by explicit dataflow computation on the event queue's values. It extracts MLP and achieves dynamism tolerance by triggering actions using ECA rules avoiding any instruction-by-instruction serialization. It achieves power benefits over an OOO by avoiding the overheads of dataflow extraction and per-instruction overheads of fetch, register renaming, and buffering.

Microarchitecture and Hardware implementation Our hardware design comprises of three high-level blocks. Each block is simple and with well defined roles as shown in Figure 3. The MAD hardware engine is integrated into a core and interfaces with its load-store unit (LSU). The *computation block* includes a dataflow computation substrate and produces values and events. The *Event block* includes named event queues using FIFOs, a programmable boolean logic array which applies boolean functions on events to trigger actions, and the rules on how to trigger actions based on the events. The *Action block* includes an action look-up table and a decoder that asserts control signals on communication buses to move values as defined by the actions. The dataflow computation substrate eliminates the need for repeated instruction fetch and rediscovery of dataflow. The named event queues explicitly maintain multiple temporally ordered values of variables instead of renaming as accomplished in an OOO using register-renaming and a big physical register file. And finally, instead of control signals and dependence/control stalls, explicitly specified actions are triggered by applying boolean functions on the status of multiple event queues as and when they get populated. There are no large or associative structures like in an OOO. Our implementation occupies 0.93 mm^2 in 55nm technology, consumes typically 700 mwatts, most of which is address computation power.

Integration with accelerators and target systems with MAD The MAD execution model and hardware can serve as a general means for any accelerator to run its induced-phase as shown in Figure 4 outlining four diverse accelerators. The core is turned off with MAD taking on the role of core in these phases. We emphasize that MAD is useful for *both* accelerator-free cores and for the emerging class of chips which integrate accelerators into cores. For natural phases (Figure 4 top-half), a microprocessor vendor who has interest in workloads that are memory intensive (database workloads for example) can put MAD into the core. MAD is a lightweight and simpler (less effective) alternative to an entire specialized database-accelerator for example. For induced phases, the workload is run on an accelerator with MAD integrated on the chip (Figure 4 bottom-half). MAD serves as the engine to feed memory values to the accelerator turning off the host core.

It is instructive to consider whether sophisticated prefetching combined with a simple inorder core is sufficient. Our results show the MAD engine is lower-power than an in-order core. Furthermore MAD's OOO capability provide it linked-data-structure (LDS) prefetching and other irregular prefetching-like benefits. Conceptually MAD can be viewed as a sophisticated, programmable, yet very power-efficient prefetcher that powers off the core.

Overview of execution Below, we describe an end-to-end overview of MAD's execution, revisting Figure 3, with a generic accelerator that uses MAD to execute the induced memory access phase. First, a compiler creates MAD regions and encodes them in the program binary ❶. For induced phases, the compiler uses the accelerator compiler's output, while on natural phases it works on the IR. At run-time, upon entering a memory access phase, the core first configures MAD by sending it configuration information created by the compiler ❷. The core also sends initial events to start MAD execution. The core is then turned off except for its load-store unit and MAD (for natural phases), or MAD+accelerator (for induced phases), that execute. The initial events trigger the computations in the computation block ❸; the execution model of the computation block is pure static dataflow, which means that whenever a ready data appears at the input queues of the computation block, computation is triggered. We assume any accelerator integrated with MAD has a similar interface. Based on the outcome of the computation, a new set of events are created in parallel and arrive at the event block ❹. The event block applies boolean algebra on the events (as configured for that region with Event-Condition-Action (ECA) rules) to generate any actions which are delivered as action indices to the action block ❺. The action block takes the indices, selects the ones that could be issued in the current cycle, and controls the data bus to move the data values to the accelerator or between event queues ❻ ❼. When executing *natural-phases*, actions are exclusively data movement between event queues and the core's load-store unit. This entire cycle repeats itself and stops when the end-of-program action is triggered. At that point, the MAD hardware wakes up the core and through memory passes architectural state changes.

Results We evaluate MAD on natural-phases in the SPECINT2006 and MediaBench suite, and induced-phases of four diverse accelerators (DySER, SSE, NPU, and C-cores). Our results show MAD is a high-performance and energy efficient access engine: providing almost same performance as a 4-wide OOO while consuming 0.4× energy, or 2.4× performance of an in-order, consuming 0.8× of its energy.

Paper organization Section 2 and Section 3 describes the MAD ISA and microarchitecture. Section 4 describes how to integrate MAD to a conventional core and other accelerators to MAD, and Section 5 discusses some complex access scenarios to demonstrate its versatility. Section 6 presents results, Section 7 discusses related work and Section 8 concludes.

2. ISA

The MAD ISA has two components: computations specified using named targets, and events/actions specified using formal event-condition-action rules. We define these formally and explain with a running example in this section. The next section shows these abstractions are a natural fit for an efficient hardware implementation of memory access phases.

Preliminaries We use a pseudo code snippet for an induced-phase with a hypothetical coarse-grained reconfigurable architecture (CGRA) accelerator as our running example. For the purpose of this discussion, the only relevant detail is the CGRA uses a FIFO interface and decoupled access-execute to receive/send values from the core and performs computation on them. Figure 5(a) shows a pseudo memory access region, which has a loop that sends the data in array A and B into the CGRA. This pseudo program is re-written into a stylized generic RISC ISA as shown in 5b. In this RISC ISA, the base address of array A and B is mapped to register `$r0` and `$r1`; the CGRA interface port is mapped to `acc0`, `acc1`, and `acc2`; and the induction variable `i` and iteration number `n` is mapped to register `$r2` and `$r3`. The RISC program contains five instructions: two loads to the accelerator, one store to the cache, a loop counter increment instruction and a branch. This code could also be a *natural* memory access region if the inner-loop was simply `C[i] = A[i] + B[i]`. Readers uninterested in the formal description of MAD's ISA can skip ahead to an example (Section 2.3).

2.1. Encoding the Computations

A memory access region's computation is specified using a single dataflow graph[2]. The input of dataflow graph nodes can be (1) region invariants which do not change during an application region; (2) intermediate computation results of simple operations; and (3) external dynamic inputs (like from memory) which we call dataflow *events*. Region invariants are encoded as invariant constants (similar to immediate values in a RISC ISA), intermediates are encoded with node names, and events are assigned to named *event queues*. These event queues communicate values within an access phase between address-generation/control instructions and memory, or values between an induced-phase and off-loaded computation. MAD ISA encoding is shown in Figure 5(c) encoded for our example's two load instructions, one store, and one `addi` for control (induction variable), encoded as as `N0, N1` and `N3`.

2.2. Event-Condition-Action in MAD

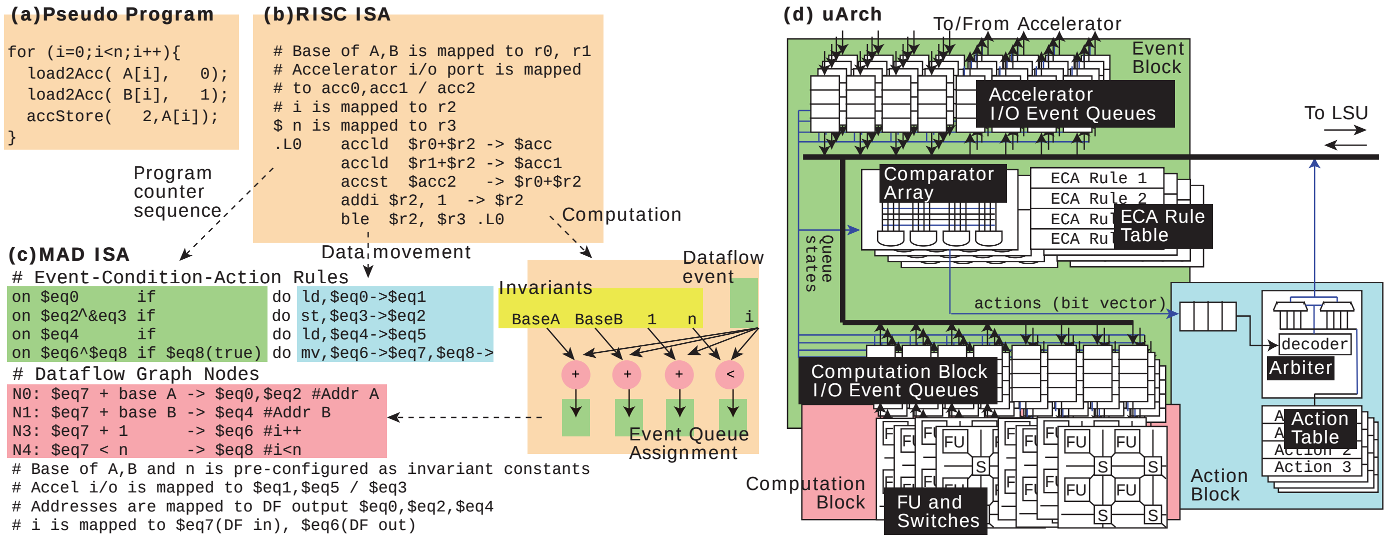
The concept of event-condition-action (ECA) is related to production rules [11] and formalized in the active database literature [26, 48, 64, 47] and used in other domains [55]. The MAD ISA leverages the same concept to encode the dataflow events and data-movement actions. Compared to VLIW ISAs, this eliminates the scheduling complexity for the compiler and the hardware. Compared to dataflow ISAs, this provides more efficient and rich control-flow support. An ECA rule in the MAD ISA follows traditional semantics and syntax:

on *event* **if** *conditions* **do** *actions*

The *event* defines when a rule has to be triggered; a triggered rule examines the current state and matches it with the *condition*; and *actions* are fired if the condition holds. Executing a rule may fire the action that in turn triggers another rule. In the MAD ISA, there is an end of program rule that triggers an action that finishes the MAD execution and gives the control back to the host processor to continue the application.

Events The event in a MAD ECA rule is a combination of dataflow *primitive-events* through event algebra [29, 7]. While there are many common operators in a traditional event algebra, the MAD ISA only adopts conjunction operator ($\wedge$),

[2] Recall from the definition of the access phase that this computation is predominantly for address generation and related control-flow.



(a)Pseudo Program

```
for (i=0;i<n;i++){
  load2Acc( A[i],   0);
  load2Acc( B[i],   1);
  accStore(   2,A[i]);
}
```

(b)RISC ISA

```
# Base of A,B is mapped to r0, r1
# Accelerator i/o port is mapped
# to acc0,acc1 / acc2
# i is mapped to r2
$ n is mapped to r3
.L0    accld  $r0+$r2 -> $acc
       accld  $r1+$r2 -> $acc1
       accst  $acc2   -> $r0+$r2
       addi $r2, 1  -> $r2
       ble  $r2, $r3 .L0
```

(c)MAD ISA

```
# Event-Condition-Action Rules
on $eq0       if                do ld,$eq0->$eq1
on $eq2^&eq3  if                do st,$eq3->$eq2
on $eq4       if                do ld,$eq4->$eq5
on $eq6^$eq8  if $eq8(true)     do mv,$eq6->$eq7,$eq8->
# Dataflow Graph Nodes
N0: $eq7 + base A -> $eq0,$eq2 #Addr A
N1: $eq7 + base B -> $eq4 #Addr B
N3: $eq7 + 1      -> $eq6 #i++
N4: $eq7 < n      -> $eq8 #i<n
# Base of A,B and n is pre-configured as invariant constants
# Accel i/o is mapped to $eq1,$eq5 / $eq3
# Addresses are mapped to DF output $eq0,$eq2,$eq4
# i is mapped to $eq7(DF in), $eq6(DF out)
```

Figure 5: The MAD ISA and microarchitecture

which can be naturally implemented with AND gates in hardware. A dataflow primitive-event is *the arrival of data* resulting from computation (computation block or accelerator) or memory access. Named event queues hold primitive events holding both the data and two pieces of state-information indicating *data arrived* and *branch condition*. Event algebra for event evaluation operates only on the data arrived state.

Conditions The conditions in a MAD ECA rule specify additional states for the primitive triggering events that have to be satisfied to drive an action. In the MAD ISA, this is used to steer tasks based on conditional branches by specifying conditions on the branch condition state in the event queues. A MAD ECA rule may not need to examine the state of the event; the primitive event itself fulfills the driving condition. In such a case, the condition part of the rule is empty.

Actions The action part in a MAD ECA rule specifies data-movement actions, which includes loads, stores, or moves between event queues. These actions pop or push data from event queues and create new data-arrived primitive events.

2.3. MAD ECA Rules Example

As shown in Figure 5(c), parts of the RISC instructions are translated into ECA rules. The Event-Condition and Action are separately color coded in green and blue, respectively. Events and conditions are both described with named event queues, where in the event part the named event queues indicate the data arrival state, and in the condition part they indicates the true/false of the branch condition state of the queue. The ECA rules of load and store actions are often triggered by the dataflow primitive events on memory addresses (and data in cases of stores). The branch instruction (`ble $r2, $r3, .L0`) in the program, however, includes a non-empty condition part; this condition part examines the brand condition state of the 8th event queue (`$eq8`), and drives the action when the state is true. The action moves the induction variable for the next loop condition check and pops the 8th event queue (discards it) since the value is no longer needed.

2.4. Compilation and Generating the MAD Configuration

Generating the MAD configuration bits in the MAD ISA can be done in a co-designed compiler or in a binary translator (and thus virtualized) in a straight-forward way. The compiler can produce the MAD configuration in the machine code generation pass using its internal representation of the memory access region (after the accelerator-specific code generation region in the case of induced regions). For this work, we use a binary translator, which performs the following steps. First, it scans the instructions and constructs the program dependence graph from the instructions, which are the nodes in the graph. Second, it breaks the graph into sub-graphs and map computations into the computation block. Third it assigns the leaf nodes in the subgraphs to either a load/store action, a move action (moving data between event queues), or a branch output that triggers different actions based on the outcome of the branch. Finally, it prioritizes the above actions based on the program order, creating the ECA rules for actions[3].

3. The MAD Hardware

Figure 5(d) presents the detailed microarchitecture of the MAD hardware comprising three main blocks. Through the action block, MAD is interfaced to core's Load-store unit (LSU) (details in Section 4). In our description below, we include the description of MAD's interface to an accelerator to execute induced-phases. In natural-phases, that part of MAD is simply clock-gated and unused.

[3] We have completed an LLVM-based compiler for MAD for natural phases (there isn't anything particular novel in our implementation). It is not used in this paper's evaluation because integrating it with DySER, NPU, and C-cores compilers is logistically overwhelming and this exercise is orthogonal to goals of this paper. And we wanted to use a single framework for evaluation of natural and induced phases.

3.1. Computation Block

The responsibility of the computation block is to perform dataflow execution consuming values from event queues and writing to event queues. For low power and area-efficient execution, many alternatives are possible spanning coarse-grained spatial CGRAs [32, 30, 18], FPGAs [38, 24, 70], and compound functional units [35]. In practice we found a small number of recurring computation patterns in the memory access phases, and the regions are themselves small. The common patterns are stack pointer, base address, offset, and index. While few patterns dominate, they are not trivial either because of the computations of these primitives may vary. The hardware must be able to tolerate and handle non-deterministic delays in arrival of inputs. Hence we choose coarse-grain reconfigurable and circuit-switched clusters.

The computation block computes the pre-configured dataflow graph statically with functional units (responsible for compute operators) and switches (responsible for forming dataflow graph edges). Figure 5(d) outlines a 16-functional unit computation block. In this design, four function units and four switches construct a cluster, and the clusters are interfaced with event queues. The entire cluster is pipelined and is implemented with a data-driven flow control mechanism using a ready state bit in the reconfigurable datapath. Thus, these circuit-switched clusters, with the ready signals in the datapath, require no dynamic scheduling of operands or functional units and can synchronize data in the datapath if input operands to the same node are ready in different cycles. To sum up, the configuration information establishes a static dataflow path from the event queues to the cluster ports, the data from the event queues to the clusters is dynamically used when possible, and the functional units consumes the data from previous stage whenever both of the operands are ready.

3.2. Event Block

The event block connects the datapath between the Load-store-unit, the computation block and the execute accelerator. As shown in Figure 5(d), it has the following sub-blocks.

Computation I/O event queues (and accelerator I/O event queues): These event queues hold inputs and outputs for the computation block and are implemented as FIFOs with data and a few meta bits. Identical to the above, the block includes accelerator I/O event queues which allow an accelerator to interface to a core using MAD for running its induced phase (elaborate discussion follows in Section 4).

ECA rule table: The ECA rules in the translated program are stored in the ECA rule table, and are compared to the state of event queues to trigger actions.

Comparator array: The event block employs a comparator array to match the dataflow events and states from event queues to pre-configured ECA rules. The comparator array takes three types of inputs: (1) the pre-configured ECA rules from the ECA rule table; (2) the data-arrived state of the event queues; (3) the branch-control state of the event queue. It applies the event algebra, check against conditions, and produces a trigger bit-vector indicating which actions are triggered. The comparator array is a tree of AND gates that compares the state-information from the event queues with pre-configured ECA-rules. The state-information (data-arrival and branch-control) uses two bits, and our implementation employs an array of 2 (states) x 4 (limiting event algebra and conditions to 4 event queues each) x 256 AND and NOT gates for 256 ECA rules. We use mux for each gate for the event algebra functions, and configure the mux and the rule table before entering the phase. The output of these gates are a bit-vector that indicates which actions are triggered. Comparing to an OOO, this module entirely performs the role of wakeup, select, and instruction control flow.

The triggering of the event-condition and the execution of the action is decoupled; the triggering event queues in a satisfied ECA rule turn into an inactive state before the values are popped by actions. This strategy of deferring the driving of the actions reduces the timing and bandwidth requirement over matching events and driving actions in a single cycle.

3.3. Action Block

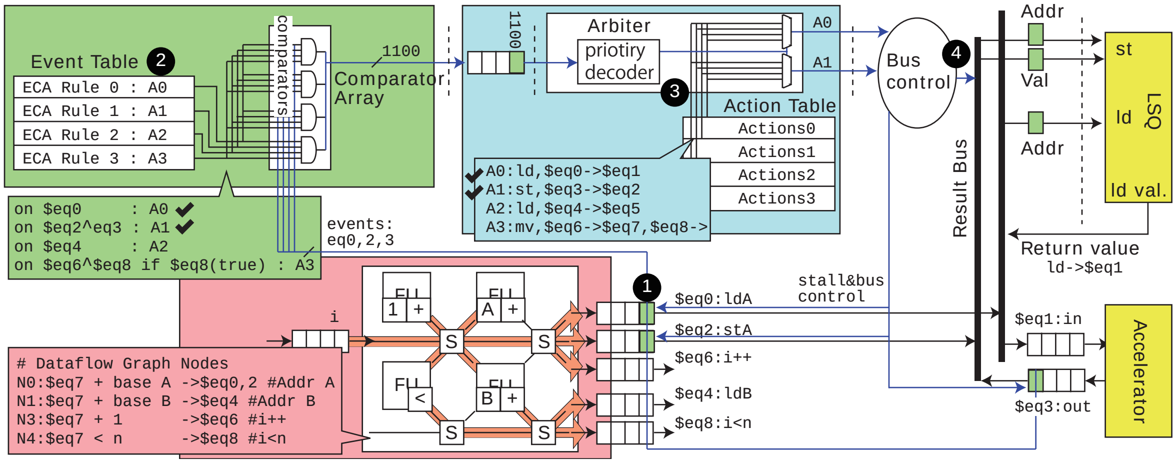
Finally, the action block is responsible for controlling the data movement between event queues and the load-store unit of the core MAD is embedded into. The actions in MAD ECA rules are stored in an action table (a dual-ported SRAM); they are indexed by the action index bit vector (received from the event block). When the triggered actions arrive as a bit vector, the action block buffers them and decodes them with a priority decoder, which arbitrates the buffered actions. If there are more triggered actions than the bandwidth of the data bus, actions with higher priority are issued first.

3.4. Microarchitectural Execution Flow

Carrying the same pseudo program, Figure 6 shows the MAD execution of the occurrence of two events, the matching two ECA rules, and the driving of two actions. Before the execution, the processor configures the 3 blocks of the MAD hardware and fills configurations as shown in the colored dialogue boxes. For illustration purposes, we simplify the drawing to present only the activated microarchitecture logic. The example execution flow begins with three primitive dataflow events ❶: there exists data arrival event in queue 0, 2, and 3. Since there no conditions need to be evaluated in these two rules, these events directly triggers two actions, action 0 and 1, as in the action index vector❷. The priority decoder decodes the bit-vector from a buffer, and outputs the two actions, `ld,$eq0->$eq1` and `st,$eq3->$eq2`, to the bus controller ❸. Finally the bus controller pops the event queues and moves the data values to LSU ❹.

3.5. Implementation and Design Decisions

We implemented the MAD microarchitecture in Verilog RTL and synthesized it with a TSMC 55nm library and it occupies



```
on $eq0       : A0 ✓
on $eq2^eq3 : A1 ✓
on $eq4       : A2
on $eq6^$eq8 if $eq8(true) : A3
```

```
✓A0:ld,$eq0->$eq1
✓A1:st,$eq3->$eq2
 A2:ld,$eq4->$eq5
 A3:mv,$eq6->$eq7,$eq8->
```

```
# Dataflow Graph Nodes
N0:$eq7 + base A ->$eq0,2 #Addr A
N1:$eq7 + base B ->$eq4 #Addr B
N3:$eq7 + 1      ->$eq6 #i++
N4:$eq7 < n      ->$eq8 #i<n
```

Figure 6: Detailed MAD execution with a simple code

an area of 0.93 mm^2. This section discusses several design decisions. First, one can intuitively implement the interconnections between event queues and comparator array as a fully connected network; this implementation, however, may cause timing problems and is over-designed. We observed that, in a typical program, the ECA rules can be divided into disjoint sets, and the hardware (event queues and comparators) can be clustered. Second, the number of functional units in the computation block determines the maximum available computational parallelism in the MAD hardware. Our final implementation uses 32 int ALU and 4 int multipliers, with a 4-entry configuration buffer, allowing regions with 256 dataflow graph nodes. The event block implements 64 (computation + accelerator) event queues clustered into 8 groups, supporting 256 ECA rules each. The total configuration bits is 1.5KB per phase.

4. Integration

We now describe MAD's integration with a core and we demonstrate its hardware generality by discussing how four diverse in-core accelerators can interface with the core using MAD instead of their current ad-hoc mechanisms.

4.1. MAD integration with core

The modifications to the core are simple and few. First, we need some datapath extensions to write into the event-queues to enable starting an access phase. This is used sparingly, so is not performance critical. Second, the load-store interface of the core should be muxed to communicate with MAD's memory ports. When MAD is integrated into an in-order core (which lack load-store queues typically), for performance reasons, integrating an LSQ inside MAD, will leverage it best.

4.2. Integrating other accelerators to MAD

In induced-phases wherein a compute accelerator is executing concurrently, MAD takes on the role of the core and hence provides the interface of the event queues to integrate with accelerators. The accelerators themselves are oblivious to *how* MAD works and can become oblivious to whether the core allows memory reordering, its cache hierarchy etc. And MAD is oblivious to what accelerator it is feeding and *how* the accelerator works. To be integrable with MAD, an accelerator must use a queue interface and standard queuing semantics: pop values on arrival, and push values into the queues to return values to MAD (Note: MAD supports multiple logical and physical named queues). Almost all in-core accelerators adhere to these semantics within a subtle design space of implicitly-decoupled, explicitly-decoupled, and loosely-combined.

Implicitly-decoupled (SSE/AVX): An *implicitly-decoupled* accelerator has code that is abstractly different for access (induced) and execute regions, but both execute on a tightly integrated substrate communicating through shared architectural registers - SSE/AVX execution is an example and is of industry relevance today. In such cases, the accelerator's implementation must be slightly modified to communicate with event queues instead of reading from a register file. Specifically for SSE/AVX, two modifications have to be made: (1) the input operands of the SIMD unit have to be changed from specialized vector register to the accelerator I/O event queues in MAD; and (2) the front-end processor pipeline that decodes SSE instructions for driving the SSE unit should be augmented with a region cache interfaced to the event queue. This is conceptually similar to loop cache and decoded μop cache [9] (and on cores where this exists [19, 60], it needs to be only slightly modified). The region cache checks the operand readiness from event queues to issue SSE instructions to the SSE unit. The SSE/AVX compilation's final code generation step should be extended to convert the x86 access instructions into a MAD configuration.

Explicitly-decoupled (DySER/NPU): An *explicitly-decoupled* accelerator already uses decoupled-access-execute principles to communicate with the core to receive and send memory values. DySER [32] and NPU [25] strictly adhere to this, and Convolution Engine to some extent as well. In such cases, no microarchitectures changes are necessary.

Loosely-combined (C-Cores): A *loosely-combined* accelerator, performs work for the access (induced) phase in a substrate that is different from the core and is loosely combined with the computation work it does. C-Cores [68] is an example, wherein, hardware is synthesized for address generation of each basic block's load and store as part of the c-cores. CAMEL is another [18]. Using MAD instead, can reduce area, make the C-cores paradigm more general, and allow higher throughput delivery of memory compared to the serialized single ld/store per cycle in the original C-cores design. To integrate with MAD, we can remove all memory related datapath (including address computation) and replace them with accelerator I/O queues; this replacement divides C-Cores into small computation operator clusters, where each of them is attached to some subset of accelerator I/O event queues for I/O operands. In the interest of extreme energy efficiency, C-Cores allows one basic block to be active. Since MAD provides a high delivery rate from memory, we can relax this to more active basic blocks (we evaluate with two).

In general, when changes are required they are small. MAD's versatility frees future accelerator designers from having to redesign and rethink interfacing to the core and provides a general and efficient mechanism, freeing them to focus on computation mechanisms alone. Furthermore integration with high-performance and in-order cores becomes possible.

5. Complex Scenarios

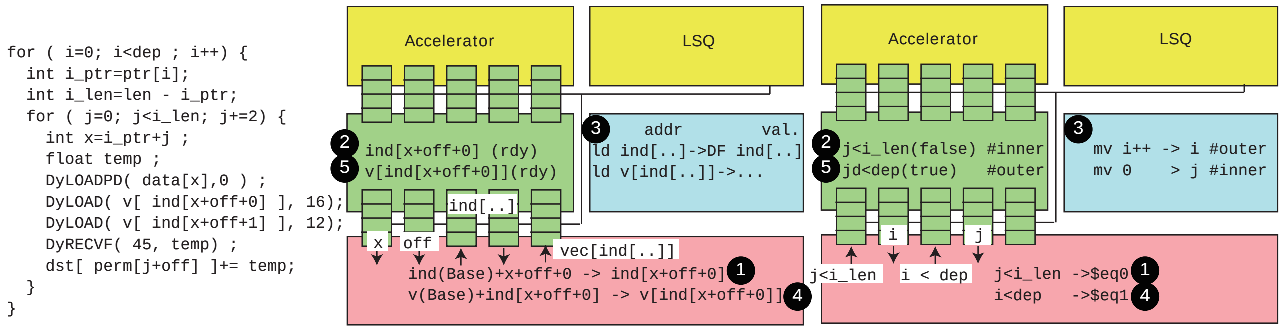
To demonstrate MAD's application generality, several complex scenarios common in memory access phases are discussed here: (1) indirect memory access, which creates irregular offset addresses for a base; (2) nested loops and control-flow; and (3) need for memory disambiguation. We explain using Sparse Matrix-Dense Vector Multiplication (`spmv`) from the Parboil suite [2] executing on DySER and its relevant *induced phase* (with code cleaned up for readability). The only relevant DySER detail is that it is a coarse-grained reconfigurable compute engine. In terms of syntax, `DyLOADPD` and `DyRECVF` are the DySER instructions ISA extensions for the core, which load from the data cache to DySER ports and receive from DySER ports to the MAD hardware, respectively.

Indirect Access: The MAD ISA (and hardware) elegantly supports indirect access using named event queues. To do `ld [ ld [x] ]`, the inner load's output is written to a named queue, which is set as the input queue for the outer load through an action. This can be naturally extended for many levels, and using register fill/spill like techniques when names are exhausted (in practice we never encountered this). In Figure 7(b), we show an example with MAD code and execution steps on the hardware for `DyLOAD(v[ind[x+off+0]],16).`

Nested loops, control-flow: In general, event-algebra and the ECA rules allow the encoding of the equivalent of any particular branch being taken, control-serialization, and many dynamic invocations. Every branch in a region creates its own *primitive event* whose value is the condition being taken or not-taken. Consider a two-level nested loop with events e_{outer} and e_{inner} for the branches. The condition $\neg e_{inner} \wedge e_{outer}$ triggers the next iteration of the outer loop, and the condition e_{inner} triggers the next iteration of the inner loop. Figure 7(c) shows the execution of the nested loop indices `i` and `j`, with simplified encoding and execution steps. Exploring a hybrid of also using predicated dataflow execution [61], with ϕ function support in the computation block introduces interesting tradeoffs which are future work.

Memory disambiguation: A final issue is ordering of loads and stores without explicit data-dependence edges, deemed may-alias at compile time. To explain how MAD supports this, we revisit MAD's interface to the core's load-store unit. Abstracting away the exact encodings, a load-store unit module in a processor's core, expects an address and a time-stamp for all loads and stores. It may be serialized (stores held until all prior stores), speculative and aggressive (return all loads immediately, check violations on stores), or employ a memory dependence predictor. There is rich literature on this topic and one representative paper is [65]. If it employs speculation, it also outputs a flush signal on a mis-speculation. MAD does not have (or need) its own LSQ and uses the same LSQ as the host. In our design, we assume this LSQ is part of the core's load-store-unit. For our quantitative evaluation, we assume the load-store-unit uses the store set memory dependence predictor to reduce the number of LSQ searches. This reduces the core's (baseline) load-store serialization and LSQ power and reduces MAD's serialization and power as well.

To support memory disambiguation, MAD must provide time-stamp information and capability to handle the flush signal. The time-stamp information is readily available in the hardware: it is action index with the iteration number from the event queues providing an ordering of all loads and stores. To handle a flush signal, we essentially use the dataflow encoding to flush all dependents with a simple implementation avoiding any need for checkpoints. We add state bits in each event queue and ECA rule table to indicate if a value is speculative. If the MAD execution reaches a load that may-alias with a previous store, it sets the destination event queue entry into speculative state. The speculative state propagates to triggered ECA rules and computation block. On receipt of a flush signal, the MAD hardware flushes all the incorrect values in the event queues (and also the computation block that consumes this mis-speculated value) by walking through the event block. An LSU that has such prediction implements store buffers, and will flush relevant entries using the time-stamp information. If a speculative action targets an event queue that already contains speculative data, the execution stalls; this simplifies the hardware design by preventing overlapped speculation. In essence, MAD handles mis-speculation by rolling-back in dataflow order. Multiple speculation and deadlock on the same MAD instruction can be detected and avoided using a physical hardware index and iteration number (age). This information allows earlier instructions to always make forward progress in



```
for ( i=0; i<dep ; i++) {
  int i_ptr=ptr[i];
  int i_len=len - i_ptr;
  for ( j=0; j<i_len; j+=2) {
    int x=i_ptr+j ;
    float temp ;
    DyLOADPD( data[x],0 ) ;
    DyLOAD( v[ ind[x+off+0] ], 16);
    DyLOAD( v[ ind[x+off+1] ], 12);
    DyRECVF( 45, temp) ;
    dst[ perm[j+off] ]+= temp;
  }
}
```

(a) Code Listing

(b) Indirect Access

(c) Nested Loop

Figure 7: MAD execution with a complex example

the dataflow fabric and in triggering events. More aggressive speculative execution in MAD could potentially increase the energy efficiency and is future work.

Memory models: MAD execution semantics with the triggering of ECA rules, is dynamic dataflow execution following the dataflow order in the program (enforced by ordered queues), and can result in a weak memory consistency model for multi-processor execution of access regions. It may reorder the reads and writes to *different* memory address locations and thus break the memory consistency model of the core's ISA.

6. Evaluation

The goal of our evaluation is to understand MAD's performance, power and energy on natural phases and on diverse accelerators in induced phases in comparison to in-order and out-of-order cores. We consider the previously mentioned four accelerators in Section 4 which demonstrated MAD's generality qualitatively. MAD is expected to have significantly (integer factors) lower power than an OOO, while for performance it should match or come close to OOO cores, and combining these two should provide significant energy reduction. Hence, we describe our results in terms of power breakdown first and them performance improvement and energy.

6.1. Experimental Setup

Performance models We use gem5 [10] for modeling of three baseline x86 cores: single-issue in-order, a low power dual-issue out-of-order core(OOO2) configured to resemble Intel Silvermont or AMD Bobcat [3, 37], and a 4-issue out-of-order core(OOO4) representing big cores such as Intel Sandy Bridge [1]. We also developed a detailed performance model for MAD, and developed (or downloaded) performance models for the four accelerators we study. We then integrated them with the core models (DySER we downloaded, SSE is part of gem5, NPU and C-cores we developed in-house based on their papers). Our DySER configuration is 64 functional units [34], we consider SSE/SSE2/SSE3, our NPU configuration is 8 neurons and identical to [25], and for C-cores we implemented benchmarks in RTL by hand. Our MAD configuration is detailed in Section 3.5.

Parameters	**Dual-issue OoO**	**4-issue OoO**
Fetch, Decode, Issue, WB width	2	4
Branch Predictor	Tournament predictor w/ 4K BTB	
ROB Size	40	168
Scheduler	32	54
RF (Int/FP)	56/56	160/144
LSQ (ld/st)	10/16	64/36
DCache Ports	1(r/w)	2(r/w)
L1 Caches	I-Cache: 32 KB, 2 way, 64B lines D-Cache: 64 KB, 2 way, 64B lines	
L2 Caches	2 MB, 8-way unified, 64B lines	

Table 2: General purpose host processor models

Configuration and design space We evaluated three MAD integrations: MAD+inorder, MAD+OOO2, and MAD+OOO4. In all evaluations of MAD, upon starting a memory access region, the core is turned off with only its load-store unit (LSU) remaining on. The load-store unit of our in-order and OOO2 cores are identical (by design to simplify the presentation results), and hence performance and power of MAD+inorder is almost identical to MAD+OOO2 and this configuration is simply called MAD2 [4]. Our MAD+OOO4 we call MAD4. When considering induced phases, we compare core X running induced phase + accelerator Y, to MAD (integrated into core X, with core turned off) running induced phase + accelerator Y. Our design space exploration is outlined in Figure 3 (page 3). In total we examine 3 processor configs, 2 MAD configs, four accelerators, and 38 benchmarks .

Power Modeling We use McPAT's [43] power model for the host processors and the SSE unit, DySER's RTL model [23], and our own RTL models for NPU and C-cores and estimate power from Synopsys Design Compiler; *we perform best-effort to match the published results*. We compare and report core+MAD+accelerator power only (i.e. L2, L3, memory system excluded since they are orthogonal and our focus is the core), and only dynamic power. Note that, if we had included static power, MAD's power reduction could only be *higher* since it is much smaller than the core.

[4] Our MAD design is equipped to issue two memory actions, but in this configuration is always constrained to one by the core's LSU.

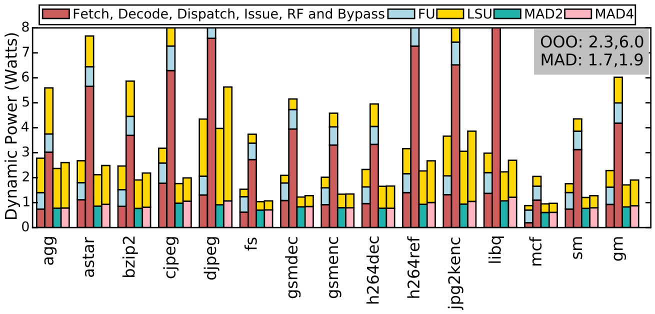


Figure 8: Power breakdown

Benchmarks To study natural phases, we consider three dominant database analytics kernels: aggregate, filescan(`fs`), and sortmerge (`sm`) [44, 69], the SPECINT2006 [63] and MediaBench [42] suite. On the latter two, we are reporting *only* benchmarks that have dominant memory access phases in them i.e. they have regions where 90% of the instruction stream of ld/st address generation (we perform this selection using a Pin [45] tool we developed) and the static code fits in the 256-node MAD configuration. We exclude benchmarks whose representative region is not a memory access region or if the region's static size exceeds MAD's size (DFG nodes or ECA rules).

To study induced phases (which are arguably more important), we largely hold benchmarks constant across the accelerators; we use benchmarks in Parboil [2], Rodinia [14] and throughput kernels from [59] to match the DySER and SSE evaluation in [34]. We implemented RTL versions of these for C-Cores. Note that our C-Cores model relaxes the single basic block active and single load/store rule to get a better tradeoff of performance vs. power, compared to the original paper which was extreme power efficiency only. For NPU, we used its benchmarks [25].

6.2. Power Breakdown

To isolate the power consumption of the core compared to MAD, we report power breakdown for natural phases (there is no concurrently running compute accelerator) as shown in Figure 8. We combine the power of fetch, decode, dispatch, issue, RF and bypass logic as the red bar and show the functional unit power and load-store unit power separately[5]. Within MAD2, the computation block's power is 639 mW (closely matching and slightly less than FU power of OOO2), with the rest consuming 112 mW; the LSU is 800 mW. Numbers are in similar range for MAD4. Considering these, MAD's ILP, MLP, dynamism overhead is 6.5% to 9%. In comparison, for OOO2 and OOO4 it is 38% and 67% respectively.

Observation-1: *Overall, MAD2 is* 1.3× *lower power than OOO2 and* 3.4× *lower power than an OOO4. Excluding power of the LSU, MAD is* 1.8× *and* 5.7× *lower power. MAD can extract dataflow with much less power than OOO.*

	Speedup (Higher is better)				Relative Energy (Lower is better)			
	O2	O4	MAD (M4)		O2	O4	MAD (M4)	
Natural	1.5	2.2	2.0	(2.3)	1.5	2.7	0.8	(0.8)
Ind-DySER	1.5	2.7	2.3	(2.6)	1.2	1.7	0.8	(0.7)
Ind-SSE	1.7	2.9	2.9	(3.6)	1.3	2.1	0.7	(0.7)
Ind-NPU	1.6	2.2	2.1	(2.5)	1.1	1.6	0.7	(0.6)
Ind-C-Cores	2.5	*	2.6	(*)	1.2	*	0.8	(*)
GM all	1.7	2.5	2.4	(2.7)	1.2	2.0	0.8	(0.7)

O2 is 2-wide OOO; O4 is 4-wide; M4 represents MAD4 config.

Table 3: Performance & Energy normalized to in-order

6.3. Performance and Energy Results

Detailed graphs are in Figure 9 and Table 3 summarizes the results. We report speedup relative to the in-order baseline (higher is better) and relative energy (lower is better). So, to compare OOO2 to MAD2, or OOO4 to MAD2, one can divide the numbers in the relevant columns. Our high level findings are below with analysis to follow.

Observation-2: *Compared to an in-order, MAD provides typically* $\geq 2\times$ *performance improvement and lower energy.*

Observation-3: *MAD2 improves performance compared to an OOO2 by* 1.4× *because it can extract more ILP, and can match an OOO4. Since it is much lower power than an OOO, MAD2's energy is* 0.6× *of OOO2, and* 0.4× *of OOO4.*

Observation-4: *For natural-phases MAD is better than in-order, OOO2, and OOO4 for* ***performance and energy****.*

Observation-5: *For induced-phases (which use an accelerator), MAD successfully eliminates the core as a performance or power bottleneck.* ***Accelerators+MAD perform at (or exceed) Accelerator+OOO performance at lower than Accelerator+inorder energy.***

Natural phases In general, MAD2 and MAD4's ILP, MLP, and dynamism mechanism are superior to an OOO2 (because of more functional units and larger window effectively) and similar to an OOO4. Comparing OOO4 to MAD4 reveals insights that point to MAD's capabilities: In some cases MAD is *more control tolerant*, wherein data-dependent branches and high mis-prediction rate hurts OOO4 but not MAD - this use case shows the benefit of ECA rules combined with dataflow. For example in `astar` and `fs`. In some cases, like `mcf`, MAD is *control restricted*, when there are parallel control-dependent memory accesses where the control branch requires 4 computations and 2 loads to be executed. OOO4 can speculatively execute the memory access before the branch is resolved.

Induced phases: Explicitly decoupled (DySER, NPU) and loosely combined accelerators (C-Cores): Here the induced and compute phase are nicely decoupled. Hence MAD2 easily and significantly outperforms in-order and OOO2 (because of more FUs and larger window it finishes the access

[5] Two clarifications are in order: LSU power consumption when using MAD2 compared to an OOO2 is higher because of higher activity (which results in more performance as we will show shortly). The OOO4's LSU power is much higher than an OOO2, partly because of more ports, and also because it has higher activity.

phase faster, feeding the accelerators faster). For benchmarks with high ILP and no memory dependencies in their induced phases (e.g., `cutcp`, `sad`, `needle`, and `nbody`), MAD4 even exceeds OOO4. Specifically in `needle`, the program is software-pipelined and the dependencies are removed because of the rearranging of the loads. This happens consistently for NPU workloads, and hence MAD appreciably outperforms OOO4+NPU (we examined the code and confirmed the reasons is not the *control tolerance* property).

Behavior with C-Cores is similar, with the only difference being OOO4 integration is omitted. The idea of C-Cores is extreme efficiency with modest performance goals and integration/comparison to an OOO4 is unrealistic.

Induced phases: Tightly integrated accelerator (SSE) For SSE execution, access and the compute (SSE) instructions stress the front-end and issue logic since both execute on same core. With MAD based execution, this front-end pressure is relieved and allows higher-throughput SSE execution. Here MAD2 and MAD4 exceed OOO2 and OOO4 for two reasons: they make the access phase faster for the aforementioned reasons *and* they free up compute resources on the core. Note that in this execution, the core is *not* turned off, its SSE units and a modeled region cache remain on and accounted for in our power modeling.

Other sensitivity studies: We have completed sensitivity studies to MAD's queue sizes, amount of compute resources, number of memory ports, various core parameters (larger instruction window, larger issue width etc.), and cache access latency [39]. In particular, we comment on the role of parallelism. Increasing parallelism can be isolated by keeping ports the same but more MAD resources. And our results show MAD2 with doubled computation fabric improves performance slightly (2%). Doubled parallelism (bigger fabric) on MAD4 improves it by a further 10% to 20%. Recall that MAD is already a highly parallel fabric.

7. Related Work

MAD is a novel confluence of concepts in disparate categories - accelerators/heterogeneity, the decoupled access/execute model, dataflow, and event-condition-action rules. Precomputation and loop accelerators share MAD's end goal of improving the memory behavior.

Decoupled access-execute The classic DAE model was introduced in the early 1980s [62, 31], and is incarnated in classic vector processors, conditional vectors, and the more recent VT and Maven architectures which are decoupled vector clustered microarchitectures [41, 8] and in very recent work, in a fragment processor design [5] allowing/exploiting extremely regular addresses in graphics workloads. In a sense, accelerators can be viewed as vector cores with arbitrary vector chaining. Their FIFOs provide a highly irregular vector register file to vector functional unit mapping. In that regard, dynamic vectorization conceptually seeks to achieve the same as specializing the access-code [52, 67].

Dataflow Our dataflow execution model resembles the classic dataflow machines from 70s to 90s [20, 36, 40, 6, 53], and more recent incarnations [58, 66, 12]: we also specify explicit targets and dataflow based computation of work. Triggered Instructions(TI) [54] which combined action/events with dataflow is most similar to MAD. None of they use low-level dataflow to address memory access. Nowatzki et al. argue for a paradigm of fine-grained hybrid dataflow execution to get benefits of OOO and dataflow [51], and MAD is one such instance.

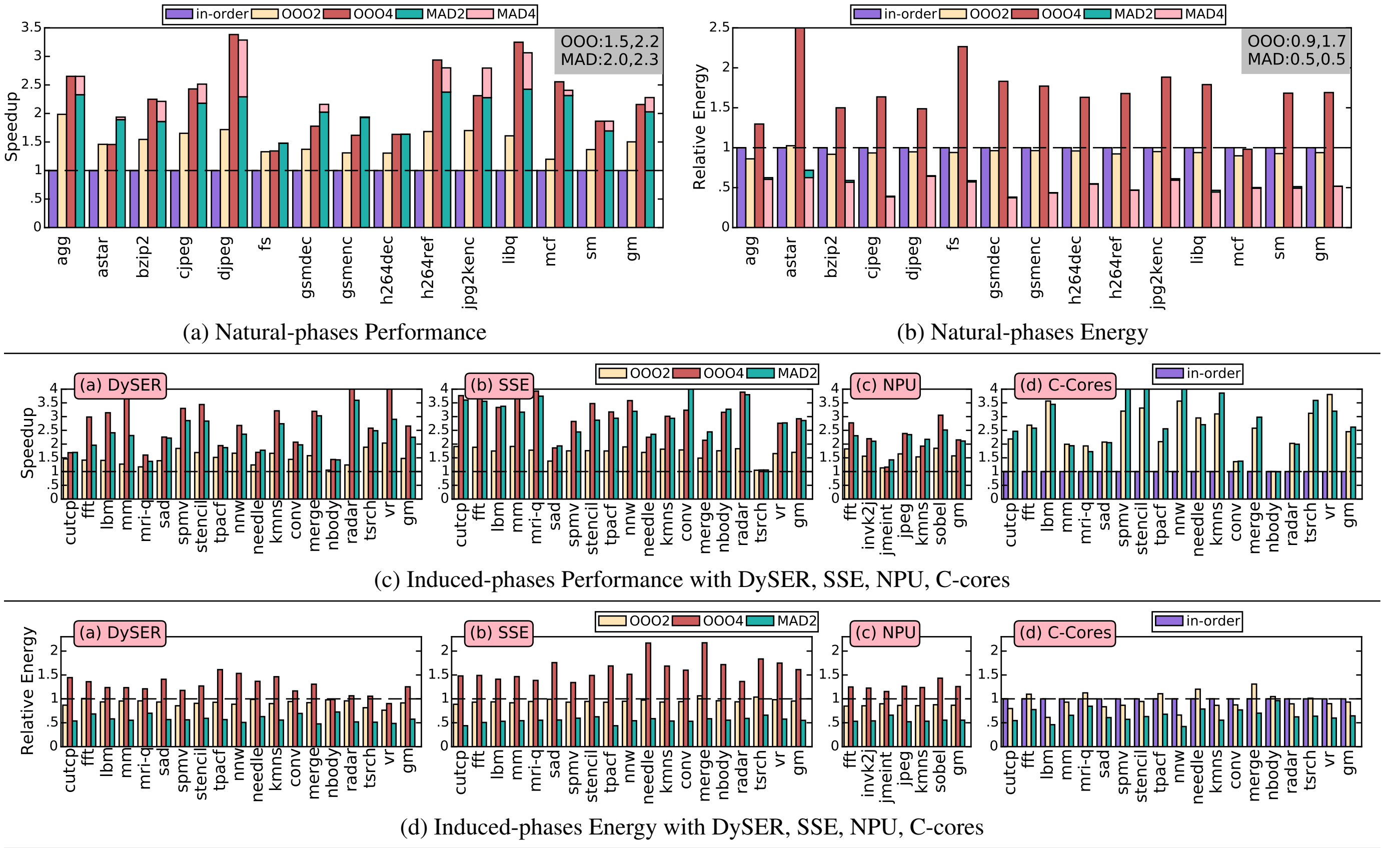
Event-condition-action rules The concept of event-condition-action (ECA) is related to production rules [11] and formalized in the active database literature [26, 48, 64, 47]. The MAD ISA leverages the same concept to encode the dataflow events and data-movement actions. Compared to VLIW ISAs, this eliminates the scheduling complexity for the compiler and the hardware. Compared to dataflow ISAs, this provides more efficient and rich control-flow support.

Pre-computation, pre-computation with reintegration, runahead In pre-computation, the idea is to execute a reduced version of a program in the hope of triggering cache accesses early for costly cache misses [72, 71, 4, 49] or for branch mispredictions [27]. Assisted execution [21] and SSMT [13] are general paradigms for such pre-computation, and lookahead combines pre-computation and DAE [57, 28]. Pre-computation with register integration as developed in the SDDM work when viewed in the accelerator context, creates a data-driven thread for each load and store. This is conceptually similar to MAD's execution model, but with very different implementation targets dictated by different end goals. Runahead execution is in this spectrum and has been proposed to exploit MLP [22, 50, 15]. From a power perspective it does not address von-Neumann inefficiencies and hence it does not improve power efficiency when applied to big-cores or in-order cores, and its ILP extraction is less than MAD when applied to in-order cores.

Loop Accelerators (LA) Typically LAs include interrelated compute and memory access techniques that are not separable to be applied as a memory engine for other accelerators [46, 16, 17].

8. Conclusion

In this paper, we have developed the Memory Access Dataflow execution model and hardware architecture which combines principles of decoupled access/execution, dataflow computation and event-condition-action rules. The MAD hardware engine applies these concepts to re-develop the main primitives of an OOO core in a power-efficient way targeted at memory accesses which naturally occur in programs or get induced when some work is offloaded to an accelerator. We described our RTL implementation of MAD, qualitative and



(a) Natural-phases Performance

(b) Natural-phases Energy

(c) Induced-phases Performance with DySER, SSE, NPU, C-cores

(d) Induced-phases Energy with DySER, SSE, NPU, C-cores

Note MAD4 bars omitted in (c) and (d) for readability.

Figure 9: Performance and Energy: Detailed results

quantitative results showing MAD can integrate with four diverse accelerators. On natural and induced memory access phases, MAD provides higher performance and lower energy than using in-order or big out-of-order cores.

MAD has promising potential to be a general mechanism for future accelerators to tightly integrate with cores. Conceptually MAD can be viewed as a sophisticated, programmable, yet very power-efficient prefetcher that powers off the core. With computations off-loaded to accelerators, and memory accesses off-loaded to MAD, whether a high-performance core is required at all is an open question. Another question for future work is whether MAD can itself be simplified further - in particular is it overly general? An alternative worthy of consideration is whether the decoupled access/execute paradigm can be combined with coarse-grained logic blocks that implement commonly used memory access patterns.

Acknowledgments

We thank the anonymous reviewers, Greg Wright, Milo Martin, and the Vertical Research Group for their thoughtful comments. Support for this research was provided by NSF under the following grants CCF-1162215, CNS-1228782, CNS-1218432.

References

[1] "Intel's Sandy Bridge Microarchitecture," http://www.realworldtech.com/sandy-bridge/, accessed: 2014-08-14.

[2] *Parboil Benchmark Suite*. http://impact.crhc.illinois.edu/parboil.php.

[3] "Silvermont, Intel's Low Power Architecture," http://www.realworldtech.com/silvermont/, accessed: 2014-08-14.

[4] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *ISCA '01*.

[5] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile gpu performance with a decoupled access/execute fragment processor," in *ISCA '12*.

[6] K. Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Trans. Comput.*, vol. 39, no. 3, pp. 300–318, Mar. 1990.

[7] E. Bach, "The algebra of events," *Linguistics and Philosophy*, vol. 9, no. 1, pp. 5–16, 1986.

[8] C. F. Batten, "Simplified vector-thread architectures for flexible and efficient data-parallel accelerators," Ph.D. dissertation, Cambridge, MA, USA, 2010, AAI0822514.

[9] N. Bellas, I. N. Hajj, C. D. Polychronopoulos, and G. D. Stamoulis, "Energy and performance improvements in microprocessor design using a loop cache," in *ICCD '99*.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[11] L. Brownston, R. Farrell, and E. Kant, *Programming Expert Systems in Ops5: An Introduction to Rule-Based Programming*. Addison-Wesley, 1985.

[12] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial Computation," in *ASPLOS XI*.

[13] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," in *ISCA '99*.

[14] S. Che, M. Boyer, M. anoyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing.”

[15] Y. Chou, B. Fahs, and S. Abraham, “Microarchitecture optimizations for exploiting memory-level parallelism,” in *ISCA ’04*.

[16] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, “The reconfigurable streaming vector processor,” in *MICRO ’03*.

[17] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, “Application-specific processing on a general-purpose core via transparent instruction set customization,” in *MICRO ’04*.

[18] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, “Composable accelerator-rich microprocessor enhanced for adaptivity and longevity,” in *ISLPED ’13*.

[19] K. Czechowski, V. Lee, E. Grochowski, R. Ronen, R. Singhal, R. Vuduc, and P. Dubey, “Improving the energy efficiency of big cores,” in *ISCA ’14*.

[20] J. B. Dennis and D. P. Misunas, “A preliminary architecture for a basic data-flow processor,” in *ISCA ’75*.

[21] M. Dubois and Y. H. Song, “Assisted execution,” Department of EE-Systems, University of Southern California, Tech. Rep. #CENG 98-25, 1998.

[22] J. Dundas and T. Mudge, “Improving data cache performance by pre-executing instructions under a cache miss,” in *ICS ’97*.

[23] “Hardware specialization with dyser.” [Online]. Available: research.cs.wisc.edu/vertical/DySER

[24] C. Ebeling, D. C. Cronquist, and P. Franklin, “Rapid - reconfigurable pipelined datapath,” in *FPL ’96*.

[25] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *MICRO ’12*.

[26] K. P. Eswaran, “Aspects of a trigger subsystem in an integrated database system,” in *ICSE ’76*.

[27] A. Farcy, O. Temam, R. Espasa, and T. Juan, “Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes,” in *MICRO ’98*.

[28] A. Garg and M. C. Huang, “A performance-correctness explicitly-decoupled architecture,” in *MICRO ’08*.

[29] N. H. Gehani, H. V. Jagadish, and O. Shmueli, “Composite event specification in active databases: Model &amp; implementation,” in *VLDB ’92*.

[30] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, “PipeRench: A Reconfigurable Architecture and Compiler,” *IEEE Computer*, vol. 33, no. 4, pp. 70–77, April 2000.

[31] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, “Pipe: A vlsi decoupled architecture,” in *ISCA ’85*.

[32] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, “Dynamically specialized datapaths for energy efficient computing,” in *HPCA ’11*.

[33] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, “Dyser: Unifying functionality and parallelism specialization for energy efficient computing,” *IEEE Micro*, vol. 33, no. 5, 2012.

[34] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, “Breaking simd shackles: Liberating accelerators by exposing flexible microarchitectural mechanisms,” in *PACT ‘13*.

[35] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, “Bundled execution of recurring traces for energy-efficient general purpose processing,” in *MICRO-44 ’11*.

[36] J. R. Gurd, C. C. Kirkham, and I. Watson, “The manchester prototype dataflow computer,” *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985. [Online]. Available: http://doi.acm.org/10.1145/2465.2468

[37] T. R. Halfill, “AMD Bobcat snarls at Atom,” *Microprocessor Report*, August 2010.

[38] J. R. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Coprocessor,” in *FPCC ‘97*.

[39] C.-H. Ho, “Mechanisms Towards Energy-Efficient Dynamic Hardware Specialization,” PhD Dissertation, Unversity of Wisconsin-Madison, 2014.

[40] R. A. Iannucci, “Toward a dataflow/von neumann hybrid architecture,” in *ISCA ’88*.

[41] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, “The vector-thread architecture,” *Micro, IEEE*, vol. 24, no. 6, pp. 84–90, 2004.

[42] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems,” in *MICRO ‘97*.

[43] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *MICRO ‘09*.

[44] Y. Li and J. M. Patel, “Bitweaving: Fast scans for main memory data processing,” in *SIGMOD ’13*.

[45] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *PLDI ’05*.

[46] B. Mathew and A. Davis, “A loop accelerator for low power embedded vliw processors,” in *CODES + ISSS 2004*.

[47] D. McCarthy and U. Dayal, “The architecture of an active database management system,” in *SIGMOD ’89*.

[48] M. Morgenstern, “Active databases as a paradigm for enhanced computing environments,” in *VLDB ’83*.

[49] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi, “Slice-processors: An implementation of operation-based prediction,” in *ICS ’01*.

[50] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, “Runahead execution: an alternative to very large instruction windows for out-of-order processors,” in *HPCA ’03*, pp. 129–140.

[51] A. Nowatzi, V. Gangadhar, and K. Sankaralingam, “Exploring the potential of heterogeneous von neumann/dataflow execution models,” in *ISCA ‘15*.

[52] A. Pajuelo, A. González, and M. Valero, “Speculative dynamic vectorization,” in *ISCA ’02*.

[53] G. Papadopoulos and D. Culler, “Monsoon: an explicit token-store architecture,” in *ISCA ‘90*.

[54] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, “Triggered instructions: A control paradigm for spatially-programmed architectures,” in *ISCA ’13*.

[55] A. Poulovassilis, G. Papamarkos, and P. T. Wood, “Event-condition-action rule languages for the semantic web,” in *EDBT’06*.

[56] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, “Convolution engine: Balancing efficiency & flexibility in specialized computing,” in *ISCA ’13*.

[57] W. Ro, S. Crago, A. Despain, and J.-L. Gaudiot, “Design and evaluation of a hierarchical decoupled architecture,” *The Journal of Supercomputing*, vol. 38, no. 3, pp. 237–259, 2006.

[58] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, S. W. Keckler, D. Burger, and C. R. Moore, “Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture,” in *ISCA ’03*.

[59] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, “Can traditional programming bridge the ninja performance gap for parallel computing applications?” in *ISCA ’12*.

[60] R. Singhal, “"inside intel next generation nehalem microarchitecture",” in *Hot Chips*, 2008.

[61] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley, “Dataflow Predication,” in *MICRO 39*.

[62] J. E. Smith, “Decoupled access/execute computer architectures,” in *ISCA ’82*.

[63] *SPEC CPU2006*. Standard Performance Evaluation Corporation, 2006.

[64] M. Stonebraker, “A rules system for relational database management system,” in *International Conference on Databases*, 1982.

[65] S. Subramaniam and G. H. Loh, “Fire-and-forget: Load/store scheduling with no store queue at all,” in *MICRO ‘06*.

[66] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, “Wavescalar,” in *MICRO ‘03*.

[67] S. Vajapeyam, P. J. Joseph, and T. Mitra, “Dynamic vectorization: A mechanism for exploiting far-flung ilp in ordinary programs,” in *ISCA ’99*.

[68] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation Cores: Reducing the Energy of Mature Computations,” in *ASPLOS ’10*.

[69] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, “Navigating big data with high-throughput, energy-efficient data partitioning,” in *ISCA ’13*.

[70] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, “Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit,” in *ISCA ’00*.

[71] C. Zilles and G. Sohi, “Execution-based prediction using speculative slices,” in *ISCA ’01*.

[72] C. B. Zilles and G. S. Sohi, “Understanding the backward slices of performance degrading instructions,” in *ISCA ’00*.