Hardware Support for Fine-Grained Event-Driven Computation in Anton 2

J. P. Grossman,[1,†] Jeffrey S. Kuskin,[1] Joseph A. Bank,[1] Michael Theobald,[1]
Ron O. Dror,[1] Douglas J. Ierardi,[1] Richard H. Larson,[1] U. Ben Schafer,[1]
Brian Towles,[1] Cliff Young,[1] David E. Shaw[1,2,†]

[1] D. E. Shaw Research, New York, NY 10036, USA.
[2] Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032, USA.
[†] Correspondence: JP.Grossman@DEShawResearch.com and David.Shaw@DEShawResearch.com

Abstract

Exploiting parallelism to accelerate a computation typically involves dividing it into many small tasks that can be assigned to different processing elements. An efficient execution schedule for these tasks can be difficult or impossible to determine in advance, however, if there is uncertainty as to when each task's input data will be available. Ideally, each task would run in direct response to the arrival of its input data, thus allowing the computation to proceed in a fine-grained event-driven manner. Realizing this ideal is difficult in practice, and typically requires sacrificing flexibility for performance.

In Anton 2, a massively parallel special-purpose supercomputer for molecular dynamics simulations, we addressed this challenge by including a hardware block, called the dispatch unit, that provides flexible and efficient support for fine-grained event-driven computation. Its novel features include a many-to-many mapping from input data to a set of synchronization counters, and the ability to prioritize tasks based on their type. To solve the additional problem of using a fixed set of synchronization counters to track input data for a potentially large number of tasks, we created a software library that allows programmers to treat Anton 2 as an idealized machine with infinitely many synchronization counters. The dispatch unit, together with this library, made it possible to simplify our molecular dynamics software by expressing it as a collection of independent tasks, and the resulting fine-grained execution schedule improved overall performance by up to 16% relative to a coarse-grained schedule for precisely the same computation.

Categories and Subject Descriptors C.1.3 **[Processor Architectures]**: Other Architectural Styles—*data-flow architectures*; C.1.4 **[Processor Architectures]**: Parallel Architectures; C.3 **[Special-Purpose and Application-Based Systems]**; D.1.3 **[Programming Techniques]**: Concurrent Programming—*parallel programming*

General Terms Performance, Design

Keywords Event-driven, task scheduling, parallel, dispatch unit, Anton 2

1. Introduction

Many scientific computations naturally decompose into small tasks that can be assigned to different processors when the application is parallelized. Examples of such tasks include modifying individual positions and velocities within a particle simulation, or updating the local state variables of a Navier-Stokes fluid dynamics model. When the number of tasks per processor is large and the data is readily available, the tasks can be efficiently executed in a series of tight loops, each of which evaluates a single type of task for many different data inputs. The overhead of invoking these loops is small compared to the total compute time. When there are few tasks per processor, however, or when the tasks must wait for the arrival of data from other processors, the overheads of communication latency and synchronization can become a significant portion of the overall computation time, and it is much more challenging to keep the processors busy with useful work.

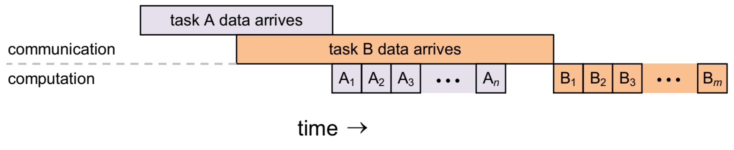
The problem of scheduling tasks to minimize these overheads can be divided into three parts: determining *when* a task is ready to run, deciding *where* a task should be executed, and choosing an *order* for the tasks that are ready. Previous work on task scheduling has primarily concentrated on software and hardware mechanisms for dynamically assigning tasks to cores within a chip multiprocessor (e.g., [7,12,14,17,19,24,27]). In a massively parallel machine with highly non-uniform memory access, however, tasks must be co-located with their data in order to maximize performance. As such, there is less flexibility in assigning tasks to cores, so the focus of task scheduling shifts to detecting and prioritizing tasks that are ready to run. Our work addresses these challenges.

Much of the difficulty arises from the often-unpredictable order in which data arrives at a processor. If an oracle were to provide this order in advance, then an optimal schedule for the tasks could be pre-determined. This would allow a processor to efficiently detect the next input datum (for example by polling a single memory location) and immediately begin the appropriate computation when it arrives. In reality, processors must be able to handle an unknown arrival order and dynamically schedule tasks accordingly. Coarse-grained scheduling—waiting for and processing large chunks of data at once—is the simplest approach, but reduces the overlap of communication with computation because it requires waiting for an entire data chunk to arrive before processing any of it.

In theory, we can improve processor utilization by allowing each task to run as soon as its input data is available, that is, by formulating the computation in a fine-grained event-driven manner. This is reminiscent of dataflow computing [4] and is similar to data-driven multithreading [28], but with scheduling performed

```
wait for data for task A
for i = 0 ... a.size() - 1
    a[i].eval()
wait for data for task B
for i = 0 ... b.size() - 1
    b[i].eval()
```

(a)



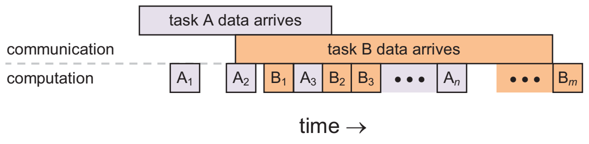
(b)

Figure 1. (a) Procedural implementation of an application with two types of computational tasks. (b) Execution schedule arising from this implementation.

```
void handleA (A *a)
    a->eval();

void handleB (B *b)
    b->eval();
```

(a)



(b)

Figure 2. (a) Event-driven implementation of an application with two types of computational tasks. (b) Possible execution schedule arising from this implementation.

at the task level rather than at the instruction or basic-block levels. This form of task scheduling is challenging in practice because it can be expensive to check for the presence of multiple smaller chunks of data and choose an execution order for the tasks whose data has arrived. Some amount of flexibility must generally be sacrificed in order to arrive at a practical implementation; message-driven architectures, for example, impose a one-to-one mapping from messages to tasks and schedule tasks in strict first-in-first-out order (e.g., [11,16,21]).

We faced these considerations in designing the hardware and software for Anton 2, a massively parallel special-purpose supercomputer for molecular dynamics simulations. Like its predecessor (referred to here as Anton 1) [26], Anton 2 maps part of the computation to hardwired pipelines and performs the remainder on programmable cores. This remaining computation has several characteristics that make it difficult to schedule: each core handles a small, heterogeneous set of fine-grained (tens to hundreds of instructions) tasks, the cores receive input data for these tasks via Anton 2's communication network, the arrival order of this data is unpredictable, and some of the input data is consumed by multiple tasks. Anton 1, which dramatically accelerated molecular dynamics relative to the previous state of the art, employed a batch processing model, using coarse-grained hardware synchronization counters to detect the arrival of input data for the next group of tasks. One of the ways in which Anton 2 seeks to further improve performance is by reformulating molecular dynamics as a fine-grained event-driven computation, resulting in a more aggressive task execution schedule.

To this end, we introduced the *dispatch unit*—a hardware block with support for flexible dependency tracking and task dispatching. The dispatch unit differs from previous hardware accelerators for task scheduling by offering an arbitrary many-to-many mapping from incoming data to a set of hardware synchronization counters, providing the ability to categorize tasks by type and query for a subset of these types, and supporting flexible prioritization of different task types. The dispatch unit's support for fine-grained event-driven computation arises from the ability to use a separate synchronization counter to track the arrival of input data for each task, but the fixed number of hardware counters raises the issue of how to support applications with arbitrarily many tasks. We address this problem with a software library that hides hardware resource limitations, allowing programmers to treat Anton 2 as an idealized machine containing infinitely many synchronization counters.

The dispatch unit, together with this software library, makes fine-grained event-driven computation practical from both an architectural and a software standpoint. It allowed us to achieve an improved task execution schedule for our molecular dynamics application, increasing overall performance by up to 16% relative to a coarse-grained schedule for the same computation. Equally important, the dispatch unit and associated library made it possible to simplify our application software by expressing it as a collection of independent tasks, with most of the task sequencing logic moved into the hardware.

2. Event-Driven Programming Model

Consider a parallel application with two different types of computational tasks—call them A and B—each of whose input data arrives over some communication network. A conventional batch-mode procedural implementation of the application waits for all task A data to arrive, evaluates all instances of task A, then repeats this process for task B. Pseudocode for this implementation, along with the resulting execution schedule, is shown in Figure 1.

While straightforward, this implementation makes suboptimal use of the processor, which sits idle waiting for all the task A data to arrive, and then again waiting for all the task B data to arrive. We can achieve better processor utilization by adopting an event-driven programming model in which each individual task runs in response to the arrival of its input data.[1] An event-driven implementation consists of a collection of event handlers, as shown in Figure 2 alongside a hypothetical execution schedule. Because individual tasks are allowed to execute as soon as their data arrives, this implementation achieves better overlap of communication with computation, resulting in improved processor utilization and faster overall execution.

When multiple events are waiting to be handled, some scheme is required to determine the order in which the handlers run. The simplest approach is to handle the events in the order that they occur, for example, by keeping track of events within a first-in, first-out queue. This may not be optimal, however. In the previous

[1] Here we use the term "event" to refer to the arrival of a task's input data; events are handled by executing the appropriate task.

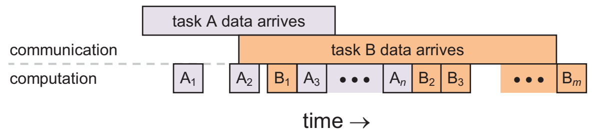


Figure 3. Modified execution schedule with task A statically prioritized over task B.

example, suppose that task A is in the critical path of the overall computation, whereas task B is not. In this case, performance can be improved by statically prioritizing task A over task B. Figure 3 shows how the execution schedule in Figure 2b changes with this prioritization. While the processor utilization stays the same, the last instance of task A finishes sooner, shortening the critical path.

3. Support for Event-Driven Programming in Anton 2

The previous section leaves some important questions unanswered. How are events detected? How are events mapped to their handlers? How are the event handlers scheduled? Anton 2 has been designed so that each of these activities can be performed within the dispatch unit. A full description of the Anton 2 architecture will be presented elsewhere; here we focus on the aspects related to event-driven computation.

Anton 2 comprises a set of identical application-specific integrated circuits (ASICs), which are directly connected to form a three-dimensional torus. The ASIC is a tiled architecture with two types of computational tiles connected by an on-chip mesh network: High Throughput Interaction Subsystem (HTIS) tiles, which contain special-purpose datapaths for computing particle interactions, and Flexible Subsystem (flex) tiles, which each contain several programmable cores called *Geometry Cores* (GCs), 256 KB of local SRAM, a network interface, and a dispatch unit (Figure 4). The fundamental unit of data in Anton 2 for all communication and memory accesses is a 128-bit quad word, which we will refer to as a *quad* for brevity.

Any flex tile can read or write the SRAM within any other flex tile. There is no memory hierarchy associated with this SRAM; in particular all addresses are physical and uncached, so Anton 2 is effectively a distributed, shared physical address machine. Anton 2 supports *counted remote writes* [10], which are remote writes of quads directly into another flex tile's SRAM that cause a synchronization counter to be incremented upon receipt. Counted remote writes form the basis of event detection: after a counted remote write is processed by SRAM, the address is forwarded to

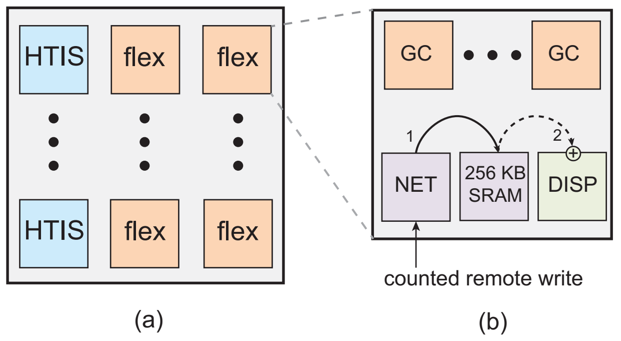


Figure 4. (a) The Anton 2 ASIC contains two types of computational tiles. (b) A flex tile comprises several GCs, local SRAM, a network interface, and a dispatch unit. Counted remote writes are first handled within SRAM (1), then forwarded to the dispatch unit to increment a synchronization counter (2).

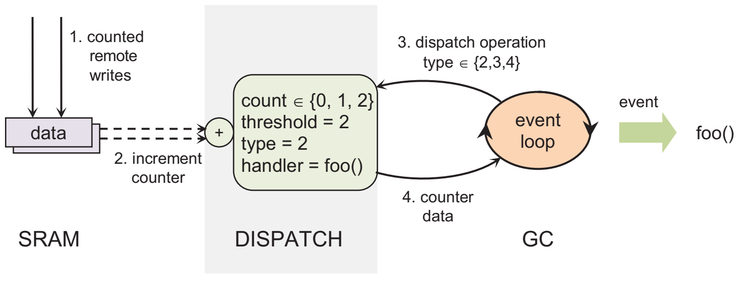
the dispatch unit so that it can be mapped to a synchronization counter (Figure 4b). The synchronization counter is incremented; an event occurs when a counter reaches its threshold, indicating that all data required for a task has arrived.

Each counter has a programmer-specified event type that can be used to dynamically select and prioritize events. When a GC queries the dispatch unit for work (this is referred to as a *dispatch operation*), it specifies the set of types that it is willing to accept in up to four priority levels. The dispatch unit selects a synchronization counter of an appropriate type that has reached its threshold (if one exists) and returns to the GC some programmer-specified data associated with the counter. This data is used to describe the event and includes a function pointer to the event handler, which is then invoked by the GC. The intended programming model is for each GC to run an event handling loop, alternately issuing a dispatch operation to fetch the next event and then handling the event (Figure 5). All of the GCs within a flex tile query the same dispatch unit for work, providing dynamic load-balancing across the flex tile.

To summarize, in Anton 2 events are detected using hardware synchronization counters, events are mapped to their handlers via function pointers associated with these counters, and event handers are scheduled by prioritizing event types within dispatch operations. Events must also be assigned to synchronization counters; the dispatch unit is primarily designed to support this assignment for applications in which the set of events is known in advance (so that counters can be initialized before the application

```
void event_loop ()
{
    while (1)
    {
        data = dispatch();
        (*data.handler)(data);
    }
}
```

(a)



(b)

Figure 5. (a) Simplified version of the event loop that runs on each GC. (b) Sequence of events leading to the execution of an event handler: (1) input data is received via counted remote writes; (2) a counter starts at zero and is incremented by one with each counted remote write until it reaches its threshold; (3) a GC's event loop issues a dispatch operation specifying a set of types that includes the counter's type; (4) The dispatch unit responds with the counter's data, which includes a pointer to the handler; (5) The GC invokes the handler.

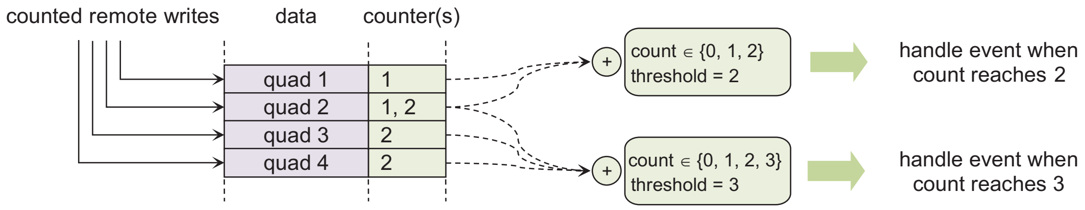


Figure 6. SRAM quads can be mapped to either individual counters or lists of counters.

runs) and repeats many times over the course of a computation (in particular, counters are automatically reset to zero upon reaching their threshold in preparation for the next occurrence of the same event). Dynamic events can be supported by introducing a level of indirection, for example by embedding a pointer within the counted write data to a secondary function that is called by the initial event handler. When necessary, it is also possible modify the counters and their SRAM associations at run time.

4. Dispatch Unit Implementation

The previous section outlined the general operation of the dispatch unit; here we provide some additional details related to its implementation within Anton 2.

4.1 Counters and counter lists

A dispatch unit contains 128 hardware synchronization counters, each of which has a 32-bit count, a 32-bit threshold, a 5-bit type, and a quad of programmer-specified counter data that forms the result of a dispatch operation. A key feature of the dispatch unit is its ability to arbitrarily map the SRAM quad addresses of counted remote writes to synchronization counters. This reduces the amount of global information that must be shared and communicated between tiles, since the sender of a counted remote write only needs to know the destination address and does not need to specify a counter ID.

It is possible for multiple tasks to depend on the same input data, in which case it becomes desirable to increment multiple counters in response to a single counted remote write. The dispatch unit provides efficient support for this operation via 128 counter lists. Each counter list specifies an arbitrary subset of the 128 counters; an SRAM quad can be mapped to either a single counter or a counter list (Figure 6). The mapping uses eight bits per SRAM quad—one bit to select a counter versus a counter list, and seven bits to specify one of the 128 counters or counter lists—representing a ~6% overhead for the flex tile SRAM.

4.2 Dispatch operations

When a counter reaches its threshold it is marked as "active" and becomes a candidate for selection when a dispatch operation is processed. A GC issues a dispatch operation by atomically exchanging a quad with a special memory address, so in particular no hardware support for dispatch operations is required within the GCs beyond the ability to perform 128-bit atomic exchanges with memory. The request quad sent from the GC to the dispatch unit specifies the event types that the GC can accept in up to four priority levels.

If a dispatch operation succeeds (i.e., at least one active counter's type matches the request), then the dispatch unit selects one of the highest-priority matching counters and returns that counter's data quad to the GC. If a dispatch operation fails, then it stalls until a matching counter becomes active, at which point the operation completes successfully. This blocking behavior obviates the need to run a polling loop on the GCs, and can cut the expected latency between counter activation and event handling roughly in half relative to a standard polling loop. Figure 7 illustrates the sequence of events involved in a blocking dispatch operation.

4.3 Error detection

There are a number of ways in which software errors and race conditions can lead to incorrect operation in the form of counters that become active prematurely or never at all. The dispatch unit contains three hardware error detectors designed to catch the majority of these errors, which would otherwise be extremely difficult to diagnose.

1. ***Counter overflow.*** This error is generated when a counter is incremented past its threshold, which can be caused by a misconfigured threshold, or an errant counted remote write (one that should have been omitted or that was directed to the wrong SRAM address).

2. ***Phase mismatch.*** Every counted remote write includes a software-generated single-bit *phase* that is generally expected to alternate in value between consecutive activations of a single counter. For example, a counter with threshold five expects five phase 0 counted remote writes, after which it becomes active and is reset to zero in preparation for five phase 1 counted remote writes. A *phase mismatch* error is generated when a counted remote write is received with the wrong phase, indicating a software race condition.

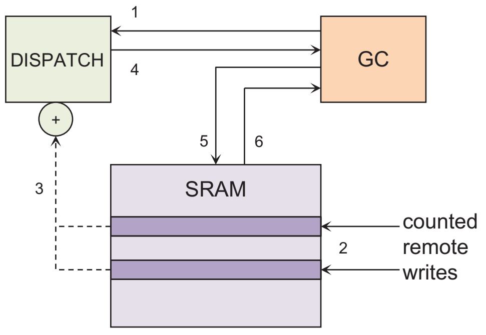


Figure 7. Typical blocking dispatch operation. (1) A GC issues a dispatch operation to the dispatch unit; no active counter is available so the operation stalls. (2) Counted remote writes are received from the network containing the data required for a task. (3) The SRAM addresses are forwarded to the dispatch unit. The task's counter is incremented, causing it to reach its threshold and become activated. (4) The dispatch unit returns the counter's data quad to the GC. (5,6) The GC reads the data from SRAM when it handles the event.

3. ***Repeated activation.*** An active counter must be processed by a dispatch operation before it can become active again, since the dispatch unit does not maintain a queue of pending events, but simply keeps track of which counters are active. If an active counter reaches its threshold again before being dispatched then a *repeated activation* error is generated, indicating a software race condition.

5. Software Support

The raw dispatch hardware presents a difficult resource allocation problem: software must manage cases where there are more tasks than counters, the counters must be partitioned among the various task types, and only 128 counter lists are available when in principle each of the 16,384 SRAM quads could be associated with a different set of counters. To ease the burden on the programmer, we implemented a higher-level software library that abstracts away the finite hardware resources, allowing arbitrarily many events and counter-SRAM associations to be defined when the application is initialized. In addition, we reduced the run-time overhead of event handler dispatching by prefetching data from SRAM and employing a form of tail recursion.

5.1 Prefetching task data

The counter data quad returned from a dispatch operation is passed to the event handler in order to describe the corresponding task. In particular, if the task requires some SRAM data (which is generally the case), then the quad must include a pointer to this data. A simple convention for event handlers would be to accept a single argument (the counter data quad), and begin by reading the required data from SRAM (Figure 8). The drawback to this approach is that SRAM reads are expensive, requiring over 20 cycles to complete, so most event handlers will immediately stall for a significant amount of time.

To reduce this overhead, we adopt the (slightly more complicated) convention of always prefetching the first data quad from SRAM *before* the event handler is called, and passing this "prefetch" quad as a second argument using a register calling convention (Figure 9). This allows some of the latency of the SRAM read to be hidden behind the latency of invoking the event handler (which is at least seven cycles). In some cases the prefetch quad is the only SRAM data that is required; in other cases additional quads must be read, but the latency of these reads can often be hidden behind computation involving the prefetch quad.

```
void some_handler (quad counter_data)
{
    task_data = sram_read(counter_data.task_data);
    process the task data
}
```

Figure 8. An event handler could take a single argument (the counter data quad) and begin by reading data from SRAM.

```
void some_handler (quad counter_data, quad prefetch_data)
{
    possibly read additional quads from SRAM
    process the prefetch quad
}
```

Figure 9. Our software convention for event handlers prefetches the first SRAM data quad so that the SRAM read latency can be partially hidden.

```
void event_loop ()
{
   while (1)
   {
      quad counter_data = dispatch();
      quad prefetch_data = sram_read(counter_data.task_data);
      (*counter_data.handler)(counter_data, prefetch_data);
   }
}
```

Figure 10. An event loop repeatedly fetches the next event and invokes the event handler.

5.2 Tail recursion for event handlers

Conceptually, the software runs an event loop that invokes the event handlers, as shown in Figure 10. In order to reduce the overhead between handlers, we introduced a compiler transformation that modifies the end of a handler to jump directly to the next handler. This is essentially tail recursion for event handlers, and it eliminates the overhead of returning to an event loop.

To enable this transformation, the programmer marks the handler with a special attribute and places a dispatch operation at the end of the handler. The compiler then inserts code after this dispatch operation to clean up the stack (as if it were about to return from the handler), load the prefetch quad from SRAM, and jump to the next handler, using registers to pass the counter data quad and the prefetch quad as arguments. In addition to obviating the need for an explicit event loop, placing the next dispatch operation directly in the handler gives the compiler the opportunity to schedule it earlier so that some of the dispatch latency can be hidden. In our molecular dynamics software, we observed that the compiler was able to schedule the dispatch operation 20 cycles before the end of some handlers, reducing the time between back-to-back event handlers from 46 cycles to as little as 26 cycles.

5.3 Allocating and initializing counters

The software library virtualizes the counter hardware, presenting the programmer with a seemingly unlimited number of counters and counter lists. The application software can allocate a separate virtual counter for each discrete task. A virtual counter is initialized with a software task type and the counter data quad, which contains a function pointer to the event handler, an SRAM pointer to the task data (of which the first quad is prefetched), and 64 bits of arbitrary user data (Figure 11).

The software task type is assigned by the programmer and is used by the library to categorize tasks. It is more general than the 5-bit hardware event type associated with each physical counter. Software task types are mapped to hardware event types (as specified by the programmer), but this mapping can be many-to-one, so there is no limit on the number of software task types. Our molecular dynamics software, for example, uses over 70 distinct software task types. Different hardware event types are only required for tasks that are prioritized differently within dispatch operations.

A counter can be associated with any data quad in SRAM by specifying the quad's address as well as the total expected remote write count from that quad (Anton 2 supports accumulation into

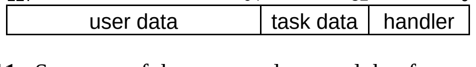


Figure 11. Structure of the counter data quad that forms the result of a dispatch operation. The first 32-bit word is a function pointer to the event handler. The next 32-bit word is a pointer to the task data in SRAM. The remaining 64 bits can contain arbitrary event-specific information.

```
// initialization time
void initialize_adder (sram_ptr pa, sram_ptr pb, sram_ptr psum)
{
    cntr = cntr_new(ADDER_EVENT, adder_handler, pa /*task data*/, make_user_data(pb,psum) /*user data*/);
    cntr->attach_to_sram(pa, 1 /*expected count*/);
    cntr->attach_to_sram(pb, 1 /*expected count*/);
}

// run time
void adder_handler (quad counter_data, quad prefetch_data /*=a*/)
{
    quad b = sram_read(counter_data.user_data[0]);
    quad sum = prefetch_data + b;
    sram_write(counter_data.user_data[1], sum);
}
```

Figure 12. Simple counter example for a two-input adder. The counter is incremented by one with the arrival of each input data quad, so its threshold is two. In the run-time handler the first input is prefetched; the pointers to the second input and the sum are stored in the user data portion of the counter data quad.

SRAM quads, so multiple counted writes may be directed to a single quad in order to add their values, resulting in multiple counter increments). The counter's threshold is the sum of these expected counts. Figure 12 shows a very simple example for a task that adds two quads. The counter data is initialized with three SRAM pointers: two for the source operands, and one for the sum.

5.4 Mapping virtual counters to physical counters

Once the virtual counters have been allocated, they must be assigned to physical counters. A virtual counter's flex tile is implicitly specified by the counter's task data pointer and SRAM associations (which must all resolve to the same flex tile). When at most 128 virtual counters are assigned to a flex tile, they can each be mapped to a separate physical counter. When a flex tile has more than 128 virtual counters, however, its physical counters must somehow be shared amongst them.

In some cases it is possible to time-multiplex a physical counter between multiple virtual counters, for which the software library does provide some support, but doing so safely requires very specific knowledge of the application's dataflow, as well as appropriately timed dynamic modifications to the counter state. More typically, time multiplexing is not an option because it is difficult or impossible to place restrictions on when a given task's input data will arrive. Instead, we allow a group of tasks of the same type to use a single physical counter, coarsening the event granularity: an event occurs when the input data for all tasks in the group has arrived.

Tasks can be grouped in this manner by assigning their virtual counters to the same physical counter. Their thresholds are added together, and the data for the individual tasks is stored in a linked list (Figure 13). The handler is responsible for iterating over the linked list to execute each of the tasks.

While it is possible to perform this counter assignment manually, doing so is tedious and forms a difficult optimization problem. The library automates the process using *counter pools*, which are collections of tasks that all have the same type. The programmer can explicitly create a counter pool of a given type and add arbitrarily many tasks to this pool. The library decides how many physical counters to allot for each counter pool (assigning more physical counters to pools with more tasks), then within each counter pool partitions the tasks among the physical counters.

Some care must be taken to avoid introducing deadlock. In particular, if two tasks are assigned to the same physical counter but one of them has an input dependency on the other, then the counter will never reach its threshold. The library prevents such deadlocks by imposing the restriction that any two tasks of the same type must be *independent*, that is, none of a task's input data can be generated by another task of the same type. The programmer is responsible for ensuring that this is the case, so that it is safe to assign multiple tasks from a counter pool to a single physical counter.

5.5 Counter lists

The programmer can arbitrarily associate virtual counters with SRAM quads without having to explicitly create or manage counter lists. Once the virtual counters have been mapped to physical counters, the library automatically creates a counter list for any SRAM quad associated with two or more physical counters. The number of such implicitly generated counter lists may exceed the hardware limit of 128, however, in which case the library auto-

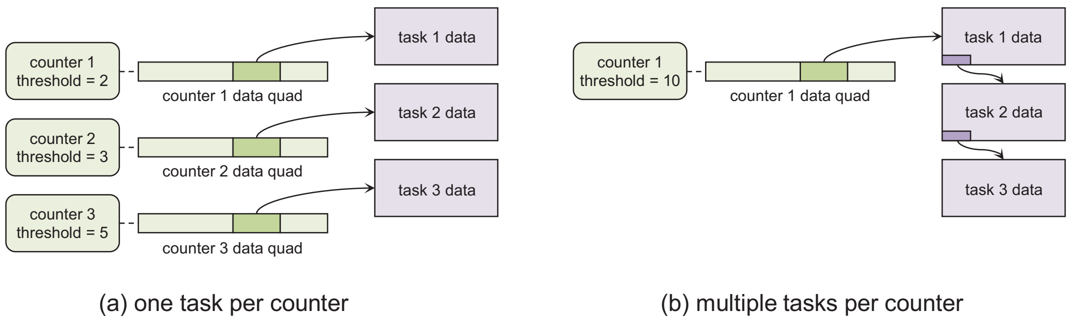


Figure 13. When multiple tasks are assigned to the same physical counter (b), the thresholds are added and the data for the tasks is stored as a linked list in SRAM.

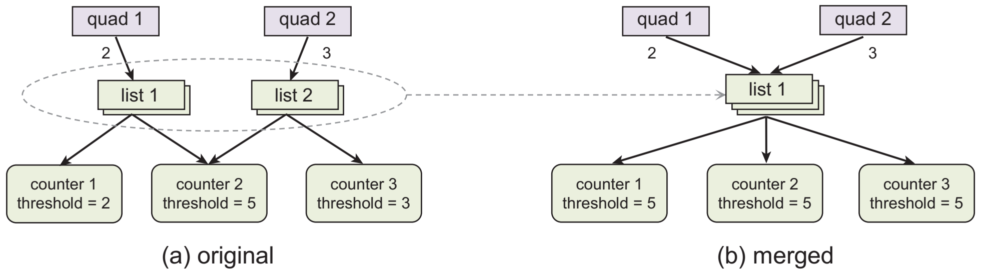


Figure 14. Counter lists are merged by replacing two lists with their union. (a) The programmer specifies that counters 1 and 2 will receive two increments from quad 1, while counters 2 and 3 will receive three increments from quad 2, which implicitly defines two counter lists. (b) If the two lists are merged, then all three counters will receive two increments from quad 1 and three increments from quad 2.

matically merges certain groups of counter lists so that a smaller number of "union lists" are required (Figure 14).

This operation preserves all original dependencies: a counter associated with an SRAM quad still depends on that quad for activation. It also introduces false dependencies that were not originally specified by the programmer. For example, the merged list in Figure 14b causes counter 1 to depend on quad 2 and counter 3 to depend on quad 1. The counter thresholds are automatically adjusted to account for the false dependencies, but there is another complication: false dependencies can lead to deadlock. In the same example, if the event triggered by counter 1 is responsible for sending data to quad 2, then merging the lists creates a circular dependency and none of the counters will ever reach their thresholds. To avoid this situation, the library only merges lists whose tasks are of the same type, again leveraging the restriction that any two tasks of the same type must be independent. In the extreme case, a single list is used for all counters of a given task type, so that none of the counters will be activated until the data for all the tasks of that type has arrived.

The algorithm for merging counters lists on a given flex tile begins by counting, for each event type t, the number N_t of distinct lists of type-t counters. The 128 physical counter lists are then partitioned among the types such that the number L_t of physical lists allotted for type t is proportional to N_t. The task of merging lists for type t until only L_t remains can be formulated as the following covering problem: given N_t lists and an integer $L_t < N_t$, find L_t "covering lists" such that each of the original lists is contained in one of the covering lists. We then merge each group of lists contained within a common covering list.

The goal is to find covering lists that approximate the original lists as closely as possible. To define this metric more precisely, observe that whenever a list is contained in a covering list with more counters, the "extra" counters in the covering list lead to false dependencies as described above. The covering lists are generated using a greedy heuristic that attempts to minimize the total number of false dependencies. For example, two possible 2-coverings of the lists {[0,1], [0,2], [3,4]} are {[0,1], [0,2,3,4]} and {[0,1,2], [3,4]}; of these, the latter is preferred as the covering lists contain fewer extra entries.

6. Event-Driven Implementation of Molecular Dynamics

A molecular dynamics (MD) simulation models the motions of individual atoms as a chemical system (e.g., a protein surrounded by water) evolves. The simulation steps through time, alternating between a force calculation phase that determines the force acting on each simulated atom, and an integration phase that advances the positions and velocities of the atoms according to the classical laws of motion. Each iteration through these phases is referred to as a *time step*. On Anton 2, the chemical system to be simulated is spatially divided into a regular grid of boxes, with each box assigned to one ASIC.

There are several reasons that MD is well-suited to an event-driven implementation in general, and the Anton 2 dispatch unit in particular:

- An MD computation naturally consists of a large number of fine-grained tasks, which are known in advance and repeated in each time step, and can therefore be statically mapped to counters within the dispatch unit.
- When a simulation runs on a large parallel machine, there may only be zero or one instances of a given task type executed on a given core, so a batch processing model (executing many tasks of the same type in a tight loop) is ineffective.
- Statically scheduling these tasks when MD is parallelized is challenging, because the order in which input data arrives is unpredictable.
- Prioritization is important when the tasks are dynamically scheduled because some affect the critical path more than others.

Figure 15 shows the dataflow of an MD simulation. The force calculation phase consumes atom positions and produces forces; the integration phase consumes forces and produces updated atom positions and velocities. The force calculation phase has three major components: various specific interactions that are computed by hardware within the HTIS tiles; bond terms, which involve small groups of atoms connected by one or more covalent bonds; and a three-dimensional fast Fourier transform (FFT), which forms part of an algorithm to efficiently compute long-range interactions. The bond terms, FFT, and position/velocity updates are computed within the flex tiles and are implemented in an event-driven manner, as described in the following sections.

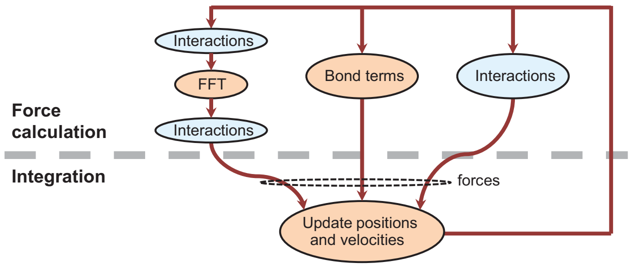


Figure 15. Dataflow of a molecular dynamics simulation.

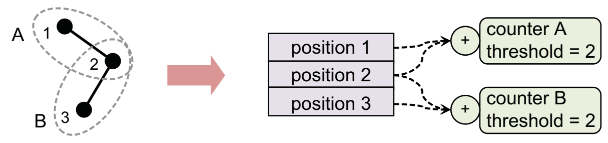


Figure 16. Two bond terms (A, B) involving three atoms (1, 2, 3) are assigned to a flex tile. Three SRAM quads are allocated to receive the atom positions; the two bond term counters are incremented as the positions arrive. Atom 2 participates in both bond terms, so a counter list is used to increment both counters when its position arrives.

6.1 Bond terms

The set of bond terms does not change during the course of a simulation: on each time step, positions for the same groups of atoms must be brought together to compute forces. On Anton 2, this is accomplished by statically mapping the computation for each bond term to a task on a fixed flex tile. Each atom has an associated list of flex tiles to which its position must be sent using counted remote writes; the dispatch units track the arrival of these positions to determine when each bond term can be evaluated. An atom can participate in multiple bond terms mapped to the same tile, so counter lists are used to update the appropriate set of counters as each atom position is received (Figure 16). Bond term tasks are created within a pool; the software library automatically partitions them among the available physical counters.

6.2 Three-dimensional FFT

The FFT comprises six "rounds" of computation. Rounds 1–3 perform one-dimensional FFTs in each dimension, round 3 also performs a Fourier-space computation with the transformed data, then rounds 4–6 perform one-dimensional inverse FFTs. There is a communication phase between pairs of consecutive rounds, in which the output from the previous round is scattered over Anton 2's communication network to the appropriate flex tiles for the next round. The FFT is mapped to exactly six counters per flex tile (one for each round); each round is triggered by the arrival of its input data (Figure 17).

The FFT tends to affect the critical path more than bond terms because additional HTIS work is required to convert the output of the FFT into forces, as shown in Figure 15. We therefore statically prioritize FFT events above bond events within dispatch operations. This allows the GCs to work on bond terms during the communication phases of the FFT, then resume FFT work once all data has arrived for the next round. Without hardware support for this type of fine-grained prioritized event interleaving, the easiest way to give the FFT absolute priority over bond terms

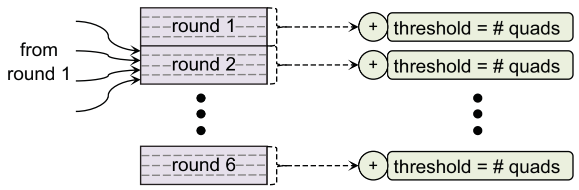


Figure 17. Six arrays of SRAM quads are allocated to receive the input data for the six FFT rounds. A single counter is used for each round, whose threshold is the number of quads in the corresponding array.

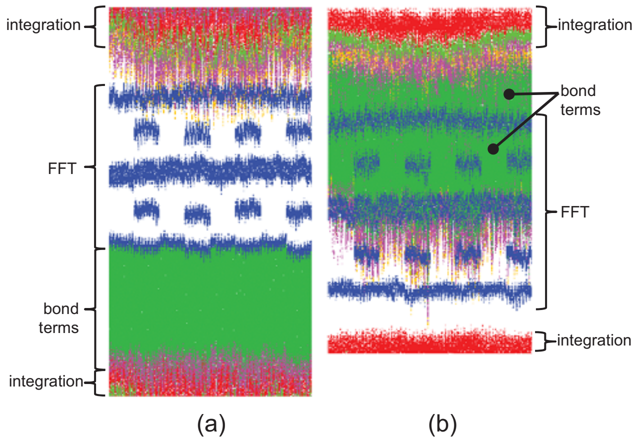


Figure 18. GC activity during force calculation. The communication phase between rounds 3 and 4 of the FFT is very short, causing these rounds to appear merged in the activity plots. (a) Bond events are delayed until the FFT has completed. (b) Prioritized event types are used to handle bond events in the background during the FFT.

would be to wait for all rounds to complete before starting the bond computations. This would place the bond terms on the critical path following the FFT, degrading performance.

Figure 18 shows GC activity for a single force calculation phase across all flex tiles that participate in the FFT. In Figure 18a, FFT events are given absolute priority over bond events by delaying the bond work until the FFT has completed. In Figure 18b, the bond event handlers are dynamically interleaved with the FFT event handlers using prioritized event types. Even though this interleaving causes the FFT to finish slightly later, the overall force calculation is accelerated as the majority of the bond work is removed from the critical path.

6.3 Integration

During the integration phase, small groups of 1–4 atoms (referred to as *constraint groups*) have their velocities and positions updated according to Newton's laws of motion and certain rigid-body constraints. Each constraint group is mapped to a task within a counter pool, and depends on the accumulation (in SRAM) of forces for all atoms within the group. When a constraint group counter is associated with a force quad in SRAM, the total expected count for that quad is greater than 1, because the total force for an atom is the sum of multiple partial forces from bond terms and HTIS tiles (Figure 19). Each partial force arrives as a separate counted remote accumulation with the same destination address. Constraint groups are disjoint, so no counter lists are required.

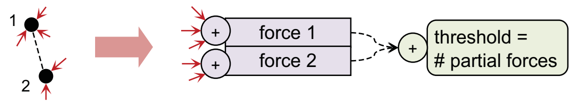


Figure 19. Two SRAM quads are allocated to accumulate the partial forces (small arrows) for a constraint group with two atoms (1, 2). The constraint group's counter is incremented as each partial force is accumulated, and its threshold is the total number of partial forces for both atoms.

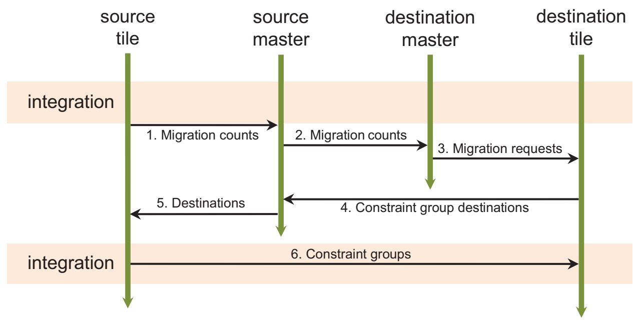


Figure 20. Constraint groups are migrated from a source tile on one ASIC to a destination tile on another ASIC using a six-stage algorithm involving migration "masters," which coordinate all migrations into and out of an ASIC.

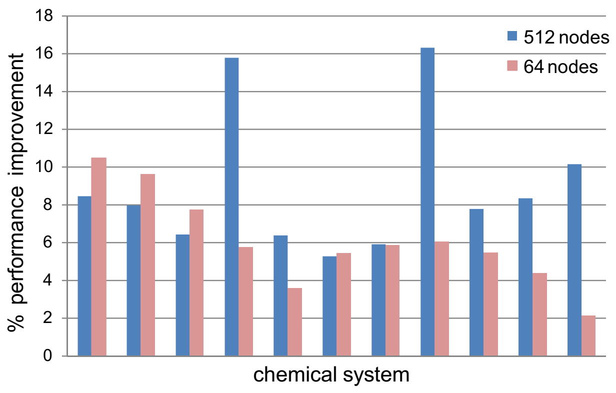


Figure 21. Percentage performance improvement relative to a coarse-grained task execution schedule.

6.4 Migration

Because the simulation is spatially subdivided among the ASICs, as the system evolves and atoms move through space it becomes necessary to periodically migrate constraint groups from one ASIC to another. We implement migration using a six-stage algorithm, shown in Figure 20. The first three stages propagate migration requests from source tiles to destination tiles via "master" tiles, which coordinate all migrations into and out of an ASIC. The destination tiles allocate SRAM storage and return destination addresses to the source tiles in stages four and five. Finally, the constraint groups are sent to these destination addresses in stage six. Each stage in the migration algorithm uses a distinct counter.

Migration is pipelined across two integration phases: the decision to migrate a constraint group is made during the first phase, but the actual migration does not occur until the second. This allows the majority of the migration communication and synchronization to be overlapped with force computation, removing it from the critical path. Migration events are given lower priority than the bond and FFT events so that they can be handled in the background. No software is required to schedule or sequence the stages; each event is simply handled once its data has arrived and there are no pending bond or FFT events. Without the dispatch unit, it would be necessary to explicitly schedule the migration stages in software, and it would be difficult to overlap the migration work with force computation.

7. Performance Evaluation

For our performance experiments, we used 11 sample chemical systems with varying characteristics: some are proteins surrounded by water while others are proteins embedded in membranes; the relative amount of bond and FFT work differs from system to system; and the overall sizes range from 23,558 atoms to 147,683 atoms (this represents a typical range for long-timescale molecular dynamics simulations; Anton 2 will also support much larger simulations). We measured the average number of cycles required to complete a single time step (one force phase and one integration phase) for each system on simulated 512-node and 64-node Anton 2 machines (these are two of the physical configurations that will be supported).

The performance advantages of the dispatch unit stem from its ability to dynamically generate an efficient execution schedule for a heterogeneous set of fine-grained tasks. To quantify this effect, we ran our molecular dynamics software using two different execution schedules: the more aggressive schedule enabled by the dispatch unit, and a coarse-grained schedule that fixes the order of task types within a time step, and executes all tasks of a given type as a single batch (by waiting for all of their input data to arrive before execution begins). This schedule is suitable for machines that provide a limited number of synchronization counters but no other hardware support for task scheduling.

Figure 21 shows the result of this comparison. The performance improvement afforded by the dispatch unit generally ranges from 2% to 10%, but is larger (~16%) on a 512-node machine for two systems in which bond terms constitute a higher fraction of the overall work. For these two systems, the primary advantage of the dispatch unit is the ability to dynamically interleave bond terms with FFT computation. On a 64-node machine this effect is much less pronounced because the bond terms represent a larger fraction of the total computation time (the FFT computation is communication-limited and takes about the same amount of time on 64- and 512-node machines), so there is less advantage to interleaving them with the FFT.

One of the important design parameters of the dispatch unit is the number of physical counters. A certain minimum number of counters is required to support the set of distinct task types within an application (which, for our molecular dynamics software, is around 48). Beyond this minimum number we generally expect performance to improve with additional counters, since this permits finer-grained task scheduling.

In Figure 22 we vary the number of counters per dispatch unit from 48 to 128, showing the performance improvement relative to 48 counters. Most of the performance gains occur in the first half of this range (48–88 counters); using more than 88 counters confers little additional performance advantage. This indicates that 128 counters are more than sufficient for our target application. More generally, it demonstrates that using only a limited number of counters, we can achieve the performance of a hypothetical machine with enough resources to assign each task to a distinct counter. Note that on a 512-node machine the smaller systems do not benefit at all from additional counters since few GCs have more than one task of any given type. For these systems, the primary benefit of the dispatch unit is the ability to dynamically schedule tasks of different types based on priority and the availability of input data.

One specific performance advantage of the dispatch unit's fine-grained scheduling is its support for efficiently processing background tasks using prioritized event types. The various migration events are given a lower priority, for example, and will be handled only when no higher-priority events are ready. Migration

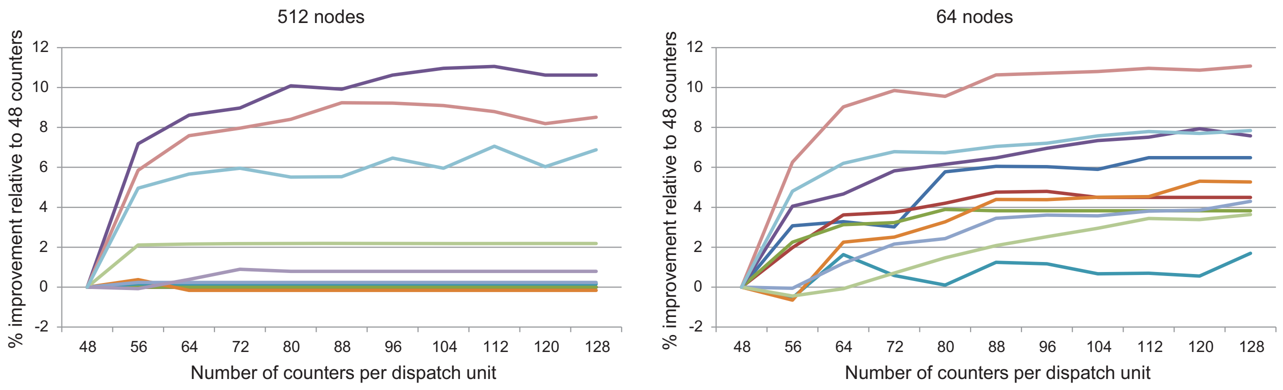


Figure 22. Percentage performance improvement relative to a dispatch unit with only 48 counters.

uses a complex multi-stage algorithm with many different events, so one might expect it to have a significant performance impact. In Figure 23, however, we see that the maximum performance overhead (measured by running a small number of time steps both with and without migration) is just over 4%, and is less than 1.4% on average. By contrast, the migration overhead in Anton 1—which does not have a dispatch unit and thus needs to explicitly schedule all migration activities in software—is typically closer to 10%.

8. Related work

At the extreme end of fine-grained event-driven computation are static and dynamic dataflow machines [4], in which every single instruction is triggered by the arrival of its operands. Scheduling every instruction independently incurs significant overhead [22], so hybrid architectures have been described in which small groups of instructions are executed together, combining the dynamic parallelism of a dataflow architecture with the efficiency of sequential execution [3,13,23,28]. These architectures are geared towards chip multiprocessors and leverage basic-block-level dependencies that can be discovered by the compiler; they are less suitable for large parallel machines with explicit messages used to initiate user-defined task.

Early work on hardware and software mechanisms for creating tasks in response to messages within a parallel machine, often referred to as *active messages*, dates back to the nineties [11,16,21]. These message-driven architectures assume a one-to-one mapping from messages to tasks and always schedule tasks in the order that the corresponding messages were received. By contrast, the dispatch unit supports many-to-many mappings from messages to tasks as well as dynamic prioritization of tasks, both of which are important for our event-driven implementation of molecular dynamics.

More recent work on task creation has focused on scheduling tasks among the cores of a chip multiprocessor using variations of *work stealing* [2,6,7,9,14,15,17]. This approach assigns a local task queue to each processor, but improves load-balancing of irregular algorithms by allowing processors to "steal work" from other queues once they complete their local tasks. Performing work stealing entirely in software is inefficient for fine-grained tasks because the overhead of task migration becomes comparable to the run time of each task, so several hardware mechanisms have been proposed to reduce this overhead [1,19,24,27]. The dispatch unit does not provide hardware support for task migration (which is undesirable in Anton 2 since each task's data resides in SRAM on a specific flex tile), but it does provide some dynamic load balancing by virtue of being shared among the GCs within a flex tile.

Software is typically responsible for determining when a task can be executed, although hardware mechanisms have also been proposed that perform this function by tracking inter-task dependencies [1,12,27]. While this approach is effective for tasks that communicate via shared memory in a cache-coherent single-chip multiprocessor, Anton 2 must directly track data dependencies across the nodes of a parallel machine, since most tasks receive their input data via counted remote writes.

The dispatch unit provides flexible support for tracking counted remote writes by allowing one or more hardware counters to be associated with every quad of SRAM. Word-level synchronization mechanisms have appeared previously in various architectures; examples include I-structures in the Tagged Token Dataflow Machine [5], presence tags in the J-Machine [21], message queue control words in the Cray T3E [25], and register-mapped hardware receive queues in the Tile Processor [29]. Using a separate set of counters introduces a level of indirection that supports more powerful synchronization primitives. Hamlyn, Anton 1 and Blue Gene/P each allow incoming packets to be mapped to receiver synchronization counters [8,18,20]. In these architectures, however, the receiver can only poll individual synchronization counters rather than poll for any event within a set of types, so they are less suitable for fine-grained event-driven computation.

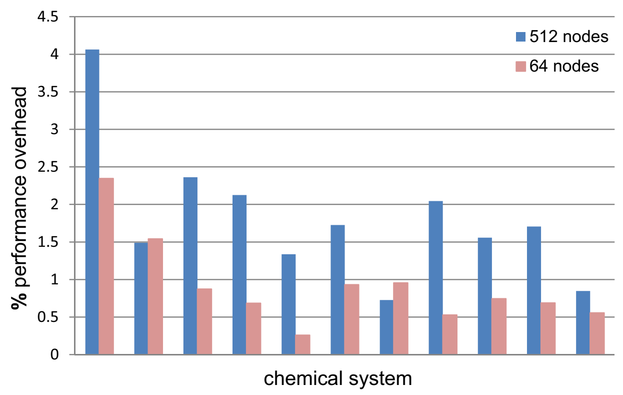


Figure 23. Percentage performance overhead of migration.

9. Conclusion

Event-driven computation is a powerful paradigm, but is difficult to realize efficiently due to the overheads of synchronization, scheduling, and task creation. Hardware support can reduce these overheads, but carries the additional challenge of mapping arbitrarily many tasks to a finite set of synchronization counters. The dispatch unit represents a practical event-driven architecture that improves both the performance and the programmability of Anton 2. Performance is improved by executing tasks as soon as their input data is available and scheduling non-critical-path tasks in the background with a lower priority. Programmability is enhanced by allowing applications to be formulated as a set of independent tasks, largely eliminating the need to perform task scheduling and prioritization in software, and enabling the use of algorithms with many fine-grained tasks and/or communication rounds (such as constraint group migration).

Our software library simplifies the development of event-driven applications and addresses the problem of supporting a potentially large number of tasks with a small set of synchronization counters. By virtualizing the counters, providing a counter pool mechanism and automatically generating counter lists, the library presents an abstraction with unbounded resources and insulates the programmer from low-level details of counter and counter list allocation. The programmer simply creates a virtual counter for each task, initialized with the appropriate SRAM associations.

Although the overheads of synchronization and task creation are much smaller in Anton 2 than in commodity processors, there is still room for improvement. More than 40 machine cycles are required to start a new task in the worst case. With typical task durations measured in hundreds of cycles, this overhead represents a significant fraction of the overall compute time. It may be possible to reduce this overhead by replacing the dispatch operation with a mechanism that pushes tasks directly to the GCs. We chose not to pursue this design in Anton 2 because of the additional complications that it introduces, such as deciding which GC should receive a task and what to do when a GC cannot process a task it receives.

The motivation for the dispatch unit was specifically to accelerate molecular dynamics simulations, but the design consists of general primitives for event-driven computation. Other applications with similar characteristics—namely those that comprise a large, static set of fine-grained tasks whose input data is received in an unpredictable order from other processors—may be able to benefit from similar hardware mechanisms.

Acknowledgments

We thank the anonymous reviewers for their helpful suggestions and Mollie Kirk for editorial assistance.

References

[1] Ghiath Al-Kadi and Andrei Sergeevich Terechko, "A hardware task scheduler for embedded video processing," *4th International Conference on High Performance Embedded Architectures and Compilers* (HiPEAC '09), Paphos, Cyprus, January 25–28, 2009, pp. 140–152.

[2] Nimar S. Arora, Robert D. Blumofe and C. Greg Plaxton, "Thread scheduling for multiprogrammed multiprocessors," *10th Annual ACM Symposium on Parallel Algorithms and Architectures* (SPAA '98), Puerto Vallarta, Mexico, June 28–July 2, 1998, pp. 119–129.

[3] Joseph M. Arul and Krishna M. Kavi, "Scalability of scheduled dataflow architecture (SDF) with register contexts," *5th International Conference on Algorithms and Architectures for Parallel Processing* (ICA3PP 2002), Beijing, China, October 23–25, 2002, pp. 214–221.

[4] Arvind and David E. Culler, "Dataflow architectures," *Annual Review of Computer Science*, Volume 1, June, 1986, pp. 225–253.

[5] Arvind and Rishiyur S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, Volume 39, Issue 3, March, 1990, pp. 300–318.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall and Yuli Zhou, "Cilk: an efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, Volume 37, Issue 1, August, 1996, pp. 55–69.

[7] Robert D. Blumofe and Charles E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, Volume 46, Number 5, September, 1999, pp. 720-748.

[8] Greg Buzzard, David Jacobson, Milon Mackay, Scott Marovich and John Wilkes, "An implementation of the Hamlyn sender-managed interface architecture," *2nd USENIX Symposium on Operating System Design and Implementation* (OSDI '96), Seattle, WA, October 28–31, 1996, pp. 245–259.

[9] David Chase and Yossi Lev, "Dynamic circular work-stealing deque," *17th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA 2005), Las Vegas, NV, July 18–20, 2005, pp. 21–28.

[10] Ron O. Dror, J.P. Grossman, Kenneth M. Mackenzie, Brian Towles, Edmond Chow, John K. Salmon, Cliff Young, Joseph A. Bank, Brannon Batson, Martin M. Deneroff, Jeffrey S. Kuskin, Richard H. Larson, Mark A. Moraes and David E. Shaw, "Exploiting 162-nanosecond end-to-end communication latency on Anton," *International Conference on High Performance Computing, Networking, Storage and Analysis* (SC10), New Orleans, LA, November 15–18, 2010.

[11] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein and Klaus Erik Schauser, "Active messages: a mechanism for integrated communication and computation," *19th International Symposium on Computer Architecture* (ISCA 1992), Gold Coast, Australia, May 19–21, 1992, pp. 430–440.

[12] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta and Mateo Valero, "Task superscalar: an out-of-order task pipeline," *43rd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '43), Atlanta, Georgia, December 4–8, 2010, pp. 89–100.

[13] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatimili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Berger and Kathryn S. McKinley, "An evaluation of the TRIPS computer system," *14th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2009), Washington, D.C., March 7–11, 2009, pp. 1–12.

[14] Danny Hendler and Nir Shavit, "Non-blocking steal-half work queues," *21st Annual ACM Symposium on Principles of Distributed Computing* (PODC 2002), Monterey, CA, July 21–24, 2002, pp. 280–289.

[15] Ralf Hoffmann, Matthias Korch and Thomas Rauber, "Performance evaluation of task pools based on hardware synchronization," *ACM/IEEE Conference on High Performance Networking and Computing* (SC04), Pittsburgh, PA, November 6–12, 2004.

[16] Laxmikant V. Kale and Sanjeev Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," *8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 1993), Washington, D.C., September 26–October 1, 1993, pp. 91–108.

[17] Matthias Korch and Thomas Rauber, "A comparison of task pools for dynamic load balancing of irregular algorithms," *Journal of Concurrency and Computation: Practice & Experience*, Volume 16, Issue 1, December, 2003, pp. 1–47.

[18] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Dong Chen, Mark E. Giampapa, Philip Heidelberger, Michael Blocksome, Ahmad Fa-

raj, Jeff Parker, Joseph Ratterman, Brian Smith and Charles Archer, “The deep computing messaging framework: Generalized scalable message passing on the Blue Gene/P supercomputer,” *22nd International Conference on Supercomputing* (ICS ’08), Island of Kos, Greece, June 7–12, 2008, pp. 94–103.

[19] Sanjeev Kumar, Christopher J. Hughes and Anthony Nguyen, “Carbon: architectural support for fine-grained parallelism on chip multiprocessors,” *34th International Symposium on Computer Architecture* (ISCA 2007), San Diego, CA, June 9–13, 2007, pp. 162–173.

[20] Jeffrey S. Kuskin, Cliff Young, J.P. Grossman, Brannon Batson, Martin M. Deneroff, Ron O. Dror and David E. Shaw, “Incorporating flexibility in Anton, a specialized machine for molecular dynamics simulation,” *14th International Symposium on High Performance Computer Architecture* (HPCA-14), Salt Lake City, UT, February 16–20, 2008, pp. 343–354.

[21] Michael D. Noakes, Deborah A. Wallach and William J. Dally, “The J-Machine multicomputer: an architectural evaluation,” *20th International Symposium on Computer Architecture* (ISCA 1993), San Diego, CA, May 16–19, 1993, pp. 224–235.

[22] Gregory M. Papadopoulos and Kenneth R. Traub, “Multithreading: a revisionist view of dataflow architectures,” *18th Annual International Symposium on Computer Architecture* (ISCA 1991), Toronto, Canada, May 27–30, 1991, pp. 342–251.

[23] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama and Toshitsugu Yuba, “An architecture of a dataflow single chip processor,” *16th Annual International Symposium on Computer Architecture* (ISCA 1989), Jerusalem, Israel, June, 1989, pp. 46–53.

[24] Daniel Sanchez, Richard M. Yoo and Christos Kozyrakis, “Flexible architectural support for fine-grain scheduling,” *15th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2010), Pittsburgh, PA, March 13–17, 2010, pp. 311–322.

[25] Steven L. Scott, “Synchronization and communication in the T3E multiprocessor,” *7th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 1996), Cambridge, MA, October 1–5, 1996, pp. 26–36.

[26] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J.P. Grossman, C. Richard Ho, Douglas J. Ierardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine McLeavey, Mark A. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles and Stanley C. Wang, “Anton, a special-purpose machine for molecular dynamics simulation,” *34th Annual International Symposium on Computer Architecture* (ISCA 2007), San Diego, CA, June 9–13, 2007, pp. 1–12.

[27] Magnus Själander, Andrei Terechko and Marc Duranton, “A look-ahead task management unit for embedded multi-core architectures,” *11th Euromicro Conference on Digital System Design Architectures, Methods and Tools* (DSD 2008), Parma, Italy, September 3–5, 2008, pp. 149–157.

[28] Kyriakos Stavrou, Costas Kyriacou, Paraskevas Evripidou and Pedro Trancoso, “Chip multiprocessor based on data-driven multithreading model,” *International Journal of High Performance Systems Architectures*, Volume 1, Number 1, 2007, pp. 24–43.

[29] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III and Anant Agarwal, “On-chip interconnection architecture of the Tile Processor,” *IEEE Micro*, Volume 27, Issue 5, September, 2007, pp. 15–31.