Data-triggered Multithreading for Near-Data Processing

Hung-Wei Tseng and Dean M. Tullsen
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA, U.S.A.

Abstract

Data-centric computing becomes increasingly important because of the rapid growth of application data. In this work, we introduce the DTM (Data-Triggered Multithreading) programming model that extends the DTT (Data-Triggered Thread) model and is fully compatible with existing C/C++ programs. The DTM model naturally attaches computation to data. Therefore, the runtime system can dynamically allocate the computing resource that provides affinity and locality. We demonstrate the potential of DTM model to improve response time and improve scalability over the traditional multithreaded programming model.

1. Introduction

The growing number of computing devices, social networking applications, online services, and business transactions has led us to an era of data explosion. As of 2012, we created an average of 2.5 exabytes of new data every day [1]. With conventional parallel architectures and programming models, the application working on a huge data set can create intensive data movement and perform inefficiently. To address the issues of processing huge data, data-centric computing which processes data in a data-oriented approach is increaingly important.

This work proposes the data-triggered multithreading (DTM) programming model, a data-centric programming model revised from the data-triggered threads (DTT) model to better address the need for data-centric computing. Similar to the predecessor, DTM model initiates parallel computation when the application changes memory content. The programmer uses C/C++ extensions of the DTM model to define computation that manipulates the changing data in the parallel thread. Initiating parallelism in this way brings several benefits. First, the application can avoid redundant computation with the same input data. Second, the application can exploit parallelism immediately after input data change. Third, the system can potentially improve the application performance using the memory address that triggers the computation. The previous DTT model [15, 17] mainly focuses on the first aspect – avoiding redundant computation by not triggering computation if the data is not changing. The previous work achieves 15% speedup over serial SPEC2000 benchmarks with an additional thread that executes the computation triggered by changing data. However, because of insufficient support for data-level parallelism, the previous work serializes the execution of data-triggered support threads and may suffer significant performance degradation if the programmer tries to generate many threads.

The DTM model inherits the power of elminating redundant computation from DTT but enhances the design of the programming model and runtime system to demonstrate the ability to support massive data-level parallelism. The DTM model provides a new type of data trigger declaration that allows programmers to more efficiently trigger computation. The DTM model also allows programmers to describe the ordering of triggered computation. The DTM model supports many threads running at the same time and executes threads in an out-of-order fashion. Based on the changing memory address that triggers computation, the runtime system of DTM can dynamically schedule computation to the most appropriate computing resource to reduce the amount of cache misses and data synchronization traffic. The runtime system can also balance the workload among processing units.

The DTM model is fully compatible with existing C/C++ applications. It does not require any special hardware or file system support. Therefore, the DTM model can be easily deployed to many types of multithreaded computing systems. Our current experimental result reveals the potential of using this model to overlap I/O latency with computation. We also find that the DTM model can achieve better scalability than conventional pthread parallelism by performing computation based on data locations.

In this paper, we make the following contributions:
(1) We introduce the DTM model, an extension of the DTT model, that targets data-centric computing but requires no support from hardware and file systems.
(2) We present the design philosophy of the runtime system for the DTM programming model and implement the prototype system.
(3) We provide some case studies that reveal the potential performance benefits of applying the DTM model.

The rest of the paper is organized as follows. Section 2 describes the DTM programming model. Section 3 details the design of our runtime system. Section 4 presents the preliminary experimental results. Section 5 discusses other related work. Section 6 concludes and depicts the future work.

2. Programming model

The DTM (Data-Triggered Multithreading) programming model defines a set of extensions of C/C++ programming languages. The user of the DTM model can express dataflow-like parallelism by declaring several data triggers and associ-

ating each data trigger with a support thread function. Upon any update to the data triggers, the system will execute the associated support thread function using a computing resource nearby the changing data.

In the DTM model, the user can declare (1) a variable, (2) a field within a data structure or (3) the left-hand (destination) value of an assignment as a data trigger. The previous DTT model only supports (1) and (2) to be a data trigger [15, 16], and can potentially create many unwanted threads, resulting in performance degradation, if the user declares a data trigger with a frequently changing element. The DTM model allows the user to declare a data trigger after an assignment statement, by which the system only triggers multithreaded execution when the destination value of the assignment changes. With this feature, the user can trigger computation only at the assignment closest to the consumer to alleviate the problem of generating unwanted threads.

The support thread function describes the computation to perform when the program changes the value of a data trigger. The support thread function is a special function that takes the address of the changing data trigger as the only argument. The support thread function can access other variables or data structures through global shared memory. The DTM model also provides a barrier where the user can attach a support thread function to. When the thread reaches the barrier, the thread will stop until all the support thread functions associated with the barrier finish.

Figure 1 uses simplified code from *Black-Scholes* to illustrate the DTM model. To trigger a support thread function, the programmer can choose one of the following as a data trigger: (1) the `sptprice` array (Figure 1(a)), (2) the `s` field in the OptionData structure (Figure 1(b)), or (3) when the assignment changes the value of `sptprice[i]` (Figure 1(c)). By using the array declaration or the data structure field declaration, the DTM model triggers support thread functions whenever there is a change to these data triggers. If the programmer only wants to trigger a support thread function after executing a specific assignment but not any other place in the program, the programmer should use the assignment declaration. The programmer can write the support thread function like a general C function that has only the triggering address as the argument (Figure 1(d)). With data triggers and the support thread function, the DTM model can run many support threads **bsInnerLoopDTTSptPrice** working on different triggering addresses at the same time. To make sure all the support threads finish before we need the result, the programmer should also attach the support thread function **bsInnerLoopDTTSptPrice** to the barrier shown in Figure 1(e).

There is no restriction on what types of variables or what kinds of functions can become data triggers and support thread functions. The DTM model also allows a support thread function to trigger another support thread. Because the DTM model triggers support thread functions asynchronously and can execute support thread functions out-of-order, users must consider potential data races and data synchronization issues as in a conventional multithreaded programming model. We also make the DTM model compatible with the original DTT model. If the programmer declares a support thread function with DTT pragmas instead of DTM pragmas, the DTM runtime system will serialize the execution of the support thread functions, as in the DTT model.

```
fptype sptprice[10000000];  #trigger bsInnerLoopDTTSptPrice()
```

(a) Array declaration

```
typedef struct OptionData_ {
        fptype s;           #trigger bsInnerLoopDTTSptPrice()
        fptype strike;
        fptype r;
        fptype divq;
             .
             .
             .
             .
             .
}
OptionData;
```

(b) Data structure declaration

```
sptprice[i] = data[i].s; #trigger bsInnerLoopDTTSptPrice()
```

(c) Assignment declaration

```
#DTM bsThreadInnerLoop
void *bsInnerLoopDTTSptPrice(fptype *ptr) {
    int i = ptr - &sptprice[0]; // Get the index value
    fptype price;
    price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
                                 rate[i], volatility[i], otime[i],
                                 otype[i], 0);
    price[i] = price;
    return 0;
}
```

(d) The support thread function

```
int bs_thread(void *tid_ptr) {
    int i, j;
    fptype price;
    fptype priceDelta;

    #DTM_Barrier bsThreadInnerLoop
    return 0;
}
```

(e) The barrier

Figure 1: Programmer-written extensions to *Black-Scholes* to exploit DTM. We simplified some source code for clarity. We present DTM pragmas in bold.

3. Design of DTM runtime system

The DTM model can work on existing systems and does not require any special type of file system support. In this section, we provide an overview of the runtime system that we use to prototype the model. We will also describe some details of the enhancements in our runtime system over the previous work.

3.1. Runtime system overview

The prototype of our DTM runtime system contains a runtime library and maintains several data structures to manage the execution of programs.

Figure 2 illustrates the execution flow of a DTM program

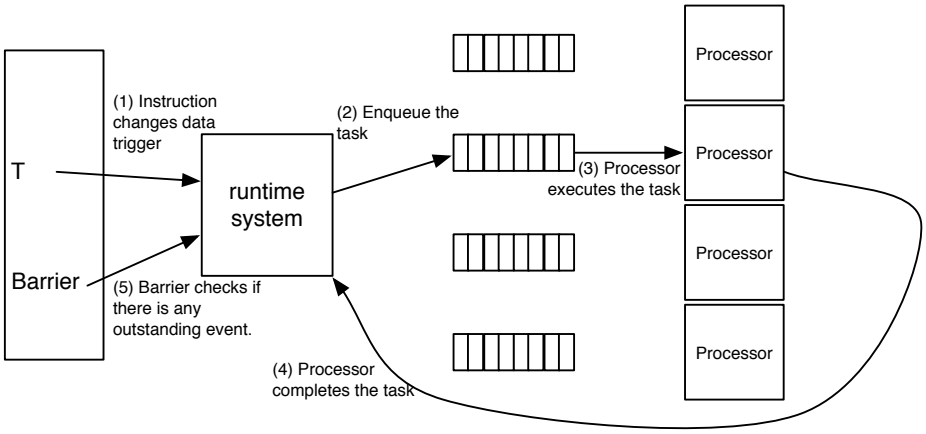


Figure 2: The execution flow of DTM model

running on the runtime system. The application in the figure contains an operation **T** that can potentially modify the memory content of a data trigger, and a DTM barrier that waits for the completion of all outstanding events of a certain support thread function. When the application executes **T**, the runtime system will take the following steps.

1. The runtime system checks if the operation writes a different value than the current version. If the runtime system detects a change, the runtime system will create a new support thread function event containing the writing address and the support thread function associated with the data trigger. The runtime system will also increase the counter of the barrier associated with the support thread function. If the runtime system detects no change to the memory content in operation **T**, the runtime system will not trigger any computation and skip steps 2.–4. to avoid redundant computation.
2. The system will analyze the writing address and then enqueue the event to the most appropriate queue.
3. When the hardware context is free and sees the queued event, the hardware context will execute the support thread function.
4. Once the support thread function finishes execution, the polling thread notifies the runtime system and releases the queue entry of the completed task. The runtime system will then decrease the counter of the barrier associated with the support thread function.
5. When the program reaches the barrier that is associated with the support thread function, the system stops and checks if there is any incomplete event associated with the barrier. The program can resume if there is no running or pending event associated with the barrier.

The runtime system of the DTM model is similar to software DTT [17] in several aspects but significantly enhances the support for massive data-level parallelism. In the rest of this section, we will discuss these designs.

3.2. Address-aware distributed event queues

In the beginning of the program execution, the DTM system spawns several threads on different hardware contexts. These threads monitor the queues and execute events from the queues until the end of an application. Unlike the prior DTT runtime system which uses a centralized queue, the DTM system allocates a unique queue for each polling thread.

One big advantage of the DTM execution model is that computation is always associated with an address. This knowledge, presented to the runtime system, or even the hardware (in a hardware-supported DTT system [15]), can be used to naturally provide data affinity and partition computation by data location.

Because DTM allows very fine-grained data-level parallelism among threads, this work uses the address attached to a support thread event to design address-aware distributed queues. Each polling thread in the DTM system is responsible for a different set of memory pages. When the runtime system detects a memory content change, the system will parse the writing address and enqueue the event to the queue of the polling thread responsible for the memory address. As a result, the runtime system will always enqueue threads triggered by the same memory address or memory addresses within the same memory page to the same queue. The support thread functions that manipulate the same data or neighboring data are likely to execute on the same hardware context to reduce cache misses and exploit data localities.

The hardware contexts running these polling threads can be any processor within the system. If the hardware system contains processors in different levels of the memory hierarchy, such as in-memory processors or storage processors, the DTM system can allow polling threads running on those heterogeneous processors. Alternatively, the runtime system could be responsible for analyzing the writing address to allocate the best computing resource for the changing data.

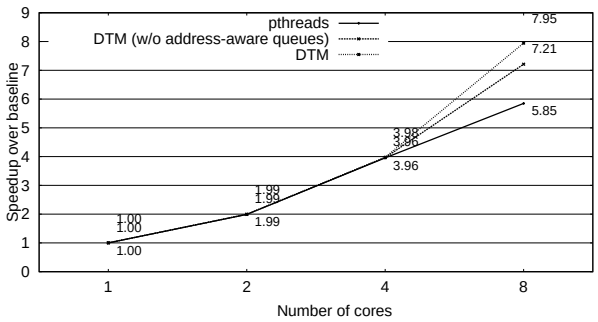


Figure 3: The speedup of swaptions using pthreads and the DTM model

3.3. Load balancing

The distribution of memory addresses of data triggers and the different computing speeds of processors can cause unequal workloads among computing resources.

To improve the hardware utilization under unequal workloads, the DTM runtime system allows an idle polling thread to fetch an event from the tail of the event queue owned by another thread. If a thread becomes idle because the thread reaches a barrier with incomplete events associated with the barrier, that thread can also fetch events from the tail of other threads so that it need not be idle.

If there are no free queue slots for the designated event queue, the program thread generating the new value has to wait for an available queue slot. In this case, the main thread will execute the support thread function in-place.

4. Case study

To investigate the performance of applications written in the DTM model, we use a computer system with dual Intel Xeon E5520 (Nehalem) processors as the experimental platform. Each processor has private L2 caches for each core but a shared L3 cache. The Nehalem processor also supports simultaneous multithreading, but we always schedule the polling threads on a distinct core in this work.

Evaluating the full power of the DTM model requires significant redesign of an application. In this paper, we present some case studies to demonstrate the current performance of our system. We select applications from PARSEC [3] benchmark suite to compare the performance with traditional parallel programming model using pthreads.

4.1. Swaptions

We first examine Swaptions from the PARSEC benchmark suite. Swaptions takes no input from file I/O which makes this application purely computation bound. The previous DTT model [17], which the DTM model extends, shows the effect of removing redundant computation is significant. In this work, we focus more on the performance of massive data-level parallelism. Therefore, when we modified the code, we avoid exploiting the redundant behavior, but target creating parallelism that performs the computation of `HJM_Swaption_Blocking` function on each element in the `swaptions` array.

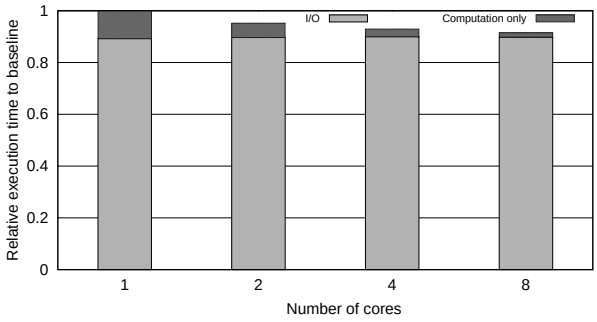
Figure 3 compares the performance of the DTM model compared with the pthread model. The baseline is the single-threaded version of swaptions. Both the DTM and pthread models exploit the same type of data-level parallelism; however, the DTM model slightly outperforms pthreads by initializing multithreaded computation earlier when the number of cores is less than 8. But when the number of cores equals 8, half of the threads run on a processor with a different socket, increasing the cost of communication between threads. The DTM model maintains the multithreaded scaling and achieves 7.95x speedup but the pthread model can only achieve 5.85x speedup. The DTM model reduces inefficient communications between sockets by triggering the computation according to the data locations.

To demonstrate the effect of the address-aware approach, we also implemented a runtime system without address-aware distributed queues (DTM w/o address-aware queues in Figure 3). In this version of DTM runtime system, we distribute the events to queues using a round-robin approach. We switch the queue for incoming events every *n* thread events. We present the result when *n* equals 8 in Figure 3 because it performs the best among all configurations we examined for this version of runtime system. The result indicates that this runtime system still outperforms pthreads with the help of load balancing features and achieves 7.21x speedup. However, the absence of address-aware distributed queues does hurt the performance of DTM runtime system.

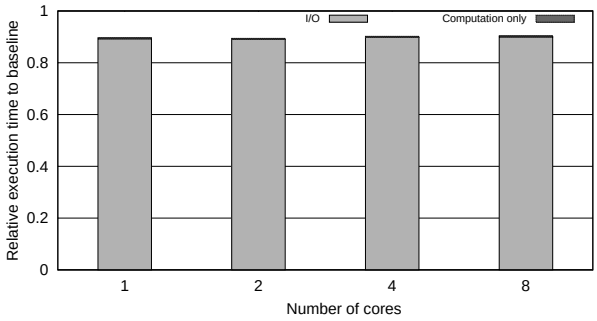
4.2. Black-Scholes

Another case we studied in this work is Black-Scholes from the PARSEC benchmark suite. When testing Black-Scholes, we set the number of runs within the application to 1 (removing repetitive execution that exists for benchmark timing purposes only) to avoid artificial redundant computation. The removal of artificial redundancy makes Black-Scholes an I/O-bound application. For the DTM version of Black-Scholes, we start a support thread when an element in the `data` array is initialized from the input file.

Figure 4 presents the normalized execution time for pthread and DTM versions of Black-Scholes. The execution time is normalized to the pthread version with one thread. The graph uses different colors to represent the time for scanning the input file and outputting the result (I/O) and the time that is CPU computation only. In terms of the pure computation time, the pthread version of Black-Scholes demonstrates scalability – 6.4x speedup with 8 threads. However, with the DTM model, Black-Scholes can start computation as soon as data is moved from the input file to memory and all the computation finishes at almost the same time the application finishes scanning the input file. As a result, we see that the application spends almost no time purely for computation with



(a) pthreads



(b) DTM model

Figure 4: The result of blackscholes using (a) pthreads and (b) the DTM model

the DTM model.

The experimental result of Black-Scholes demonstrates the potential of DTM to hide I/O time with computation. DTM also allows the computation to finish as soon as an I/O completes, the application can have better response time than traditional parallel programming techniques that expect data to be in memory before initializing computation. The DTM model makes it natural to express computation so it is performed as soon as data arrives. For example, we would expect the programmer to express the Black-Scholes computation in this way even if they did not realize this computation was I/O bound.

In all the above cases, we eliminated code that could result in redundant computation to simply test the effect of parallelism. The DTM model still has the ability to eliminate redundant computation as in the DTT model. We expect applications with parallel behavior, where some parallel computation is initiated on redundant computation, to provide much higher gains; however, this work focuses on maximizing performance available from parallelism.

5. Related work

The DTM model tries to achieve data-level parallelism by triggering computation somewhat similar to dataflow architectures [14, 2, 12, 7]. However, these architectures usually require pure dataflow programming languages [5, 13], which are incompatible with imperative programming languages, to describe programs as dataflow graphs. The hardware support for dataflow architecture can also be complex. The DTM model requires no hardware support and allows programmers to use existing C/C++ programs. For von Neumann machines with fine-grained multithreading availablity [4], the DTM model also provides an option for composing applications.

The DTM model shares the same spirit as Cilk [9] and CEAL [10] that extend the C/C++ programming language to support dataflow-like programming and execution models on conventional architectures. Cilk exploits dataflow parallelism like functional programming language. CEAL encourages programmers to use incremental algorithms on changing data to avoid redundant computation. The DTM model extends Data-Triggered Threads, which is designed to exploit both dataflow-like parallelism and reduce redundant computation. However, the previous DTT [15, 17] model benefits most from eliminating redundant computation, because of the limitation of language design and runtime system. Specifically, the prior system had no knowledge of legal orderings of DTT threads, and had to serialize the support threads (although they execute in parallel with the triggering thread). The DTM model also shares the goal of data-centric programming models [8, 11], which seek to perform computation where data is located. In contrast to these systems, the DTM model does not need any support from the underlying file system.

The runtime system design of DTM model is similar to Habanero [6] in triggering multithreaded computation asynchronously, allocating tasks using data locations and load balancing. However, with the DTM model, the program can avoid redundant computation that Habanero does not address. Qthreads [18] also proposed a location-aware resource allocation runtime system, but their programming model is still similar to the traditional pthread model and difficult to take advantages from triggering computation in dataflow fashion.

6. Conclusions and future work

As we move into the era of big data, computer architectures and programming languages need to handle more than terabytes of data efficiently. The partitioning of data becomes more important than the partitioning of the code. Because the DTM programming model naturally attaches computation to data, computation naturally moves to a location that provides affinity and locality.

The preliminary results are promising. The DTM model can potentially improve the application response time by starting computation as soon as part of the input data arrives in the system. DTM model can also achieve better scalability than the pthread model because of the data-centric execution model.

The future work of the DTM model will be applying the programming model and runtime system to real big data applications like genome sequence processing and warehouse applications.

References

[1] IBM Big Data – What is Big Data, http://www.ibm.com/big-data/.

[2] Arvind and R. S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39:300–318, March 1990.

[3] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[4] S. Brunett, J. Thornley, and M. Ellenbecker. An initial evaluation of the tera multithreaded architecture and programming system using the the c3i parallel benchmark suite. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC 1998)*, pages 1–19, 1998.

[5] D. C. Cann, J. T. Feo, A. D. W. Bohoem, and O. Oldehoeft. *SISAL Reference Manual: Language Version 2.0*, 1992.

[6] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, 2011.

[7] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, April 1991.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, June 1998.

[10] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a c-based language for self-adjusting computation. In *ACM SIGPLAN 2009 conference on Programming language design and implementation*, pages 25–37, June 2009.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, 2007.

[12] R. S. Nikhil. Can dataflow subsume von Neumann computing? In *16th Annual International Symposium on Computer Architecture*, pages 262–272, May 1989.

[13] R. S. Nikhil. Id reference manual, version 90.1. *CSG Memo 284-2*, September 1990.

[14] G. Papadopoulos and D. Culler. Monsoon: an explicit token-store architecture. In *17th Annual International Symposium on Computer Architecture*, pages 82–91, May 1990.

[15] H.-W. Tseng and D. M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *17th International Symposium on High Performance Computer Architecture*, pages 181–192, February 2011.

[16] H.-W. Tseng and D. M. Tullsen. Eliminating redundant computation and exposing parallelism through data-triggered threads. *IEEE Micro, Special Issue on the Top Picks from Computer Architecture Conferences*, 32:38–47, 2012.

[17] H.-W. Tseng and D. M. Tullsen. Software data-triggered threads. In *ACM SIGPLAN 2012 Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2012.

[18] K. Wheeler, R. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.