Software Data-Triggered Threads

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

The data-triggered threads (DTT) programming and execution model can increase parallelism and eliminate redundant computation. However, the initial proposal requires significant architecture support, which impedes existing applications and architectures from taking advantage of this model.

This work proposes a pure software solution that supports the DTT model without any hardware support. This research uses a prototype compiler and runtime libraries running on top of existing machines. Several enhancements to the initial software implementation are presented, which further improve the performance.

The software runtime system improves the performance of serial C SPEC benchmarks by 15% on a Nehalem processor, but by over 7X over the full suite of single-thread applications. It is shown that the DTT model can work in conjunction with traditional parallelism. The DTT model provides up to 64X speedup over parallel applications exploiting traditional parallelism.

1. Introduction

The data-triggered threads (DTT) programming and execution model [16, 17] has been shown to enable applications to exploit parallelism and eliminate redundant computation. Unlike traditional execution models, which generate parallelism based on control flow, the DTT model initiates a thread when the program changes particular memory contents. This has two primary advantages. First, computation depending on the changed data can execute immediately, often exposing parallelism earlier. Second, untouched or unchanged data do not generate unnecessary computation. The latter effect, eliminating unnecessary, redundant computation, is shown to be particularly powerful. The DTT model is implemented as extensions to an imperative programming language, allowing the programmer to declare particular functions as support thread functions, and attach them to variables or structure fields in the code. However, the DTT model, as proposed, requires changes to the instruction set architecture and the addition of hardware tables to the processor, making it unavailable on existing processors.

This paper introduces a software implementation of data-triggered threads, enabling programmers to utilize the DTT model on existing architectures without any hardware support. Software DTT experiences overheads (primarily the high cost of spawning threads in current architectures) compared to the simulated hardware approach, yet still provides opportunity for significant gains. This paper proposes techniques for mitigating the software overheads of DTT. The thresholding mechanism, for example, ad-

dresses the scenario where excessive threads are being generated and cause large slowdowns, a case that the prior proposal was vulnerable to. Thus, we significantly increase the generality of data-triggered threads in two ways: (1) we make it available today on real hardware, and (2) we free the programmer to use DTT for any computation that lends itself to the DTT model, without having to think as carefully about the performance implications.

To demonstrate the feasibility of supporting the DTT model without architectural support, we build a prototype compiler and user-space runtime libraries. The compiler accepts programs written in C and C++ with DTT extensions and produces code using runtime libraries. The runtime libraries schedule and execute support thread functions when they detect memory content change to a marked variable. A key challenge in designing the runtime system for the DTT model is the high multithreading overhead. We introduce a fast thread spawning mechanism to avoid the thread spawning costs. To avoid performance degradations due to multithreading hardware, software, or communication overheads, or poorly written data-triggered threads programs, we also design a simple threshold mechanism that automatically and transparently disables the usage of the DTT model dynamically.

Software data-triggered threads can be used to either add a new type of parallelism to serial code, or to augment traditionally parallel code. This paper demonstrates both; thus, it is the first work to demonstrate the DTT model on existing parallel code. We run serial SPEC benchmarks and both serial and parallel versions of the PARSEC applications on several existing machines. Despite the overheads of our software approach, we achieve 1.15X average performance improvement for the SPEC benchmarks, but an average gain of 7.3X over single-thread execution overall (including the PARSEC results). For just the parallel applications, our speedups over traditional parallel execution with an equivalent number of cores vary, but achieve up to 64X (2-thread DTT compared to 2-thread traditional).

This paper makes the following contributions: (1) It presents a pure software framework that allows applications to take advantage of the DTT model. (2) It proposes improvements to the data-triggered threads implementation that mitigates thread spawning overheads. (3) It proposes further improvements to the framework to avoid slowdowns from ill-behaved data-triggered threads, enabling effective use of the DTT model on a wider variety of applications. (4) It demonstrates that DTT parallelism can be complementary to traditional parallelism, and the combination can dramatically outperform traditional parallelism alone.

This paper is organized as follows. Section 2 provides an overview of the DTT model and discusses the related work. Section 3 details the design of our runtime system that supports the DTT model. Section 4 describes our experimental methodology. Section 5 presents and discusses the experimental results. Section 6 concludes.

2. Background and Related Work

This section provides some background on the original data-triggered threads programming model that is implemented in soft-

Figure 1. The data-triggered thread execution model

ware in this work, some details on the originally proposed hardware implementation, and other work related to this project.

2.1 Data-triggered threads programming and execution model

Data-triggered threads [16, 17] presents a new model of parallelism that has the potential to expose parallelism more explicitly while avoiding redundant and unnecessary computation. Where conventional programming models generate parallelism based on control flow (the program counter reaches a fork instruction, for example), in the DTT model a thread is generated when a particular memory location is changed. The DTT model has two advantages. First, it exposes parallelism immediately, as soon as the program modifies the source data. Second, it eliminates redundant computation – if the data is not changed, we do not do the computation. The original work demonstrated that about 80% of all loads are redundant, meaning that they load the same value that the same load had fetched the last time it accessed this same address. The redundant load often feeds redundant computation that depends on that load, typically leading to a store that finally writes a value into memory that has not changed. With DTTs, the programmer can express the redundant computation in a support thread, which is only triggered when the data has actually been changed. This results in as much as a 6X speedup in one case where the primary gain comes from the ability of DTT to avoid redundant computation.

Figure 1 details the operation of the DTT model. The application in Figure 1(a) consists of code regions A, B, and C where the execution of B only depends on the memory contents written in A. In the DTT model (Figure 1(b)) the computation of B in the original application (or possibly an incremental form of section B) becomes available for parallel execution, in support thread S, immediately once the store producing the data executes. Additionally, DTT only performs the dependent computation when the store in A modifies the memory contents upon which B (from the original code) depends. The DTT model will skip the execution of B in the original code and immediately execute C after the support thread S finishes execution. If the main thread finishes executing A before S completes, the DTT model will stall the main thread until completion of S, before going on to C. Therefore, we consider the original location of code region B as an implicit barrier in the DTT model.

When the store instruction in A generates the same memory content, or if the program does not execute the store instruction at all, the inputs of B remain the same as the last execution, which means the prior execution of S is still valid. The DTT model will skip the computation and memory accesses of B of the original application without spawning any support thread. Because the DTT model skips the execution of code section B if the result of the latest support thread computation is still valid, the DTT model calls the code section B from the original application the “skippable region”. The DTT model still leaves the code of the skippable region in the main thread because the support thread may fail to execute.

```

int  otype[10000000];    #trigger bsInnerLoopDTTotype()
fptype sptprice[10000000]; #trigger bsInnerLoopDTTSptPrice()
fptype strike[10000000];  #trigger bsInnerLoopDTTStrike()
fptype rate[10000000];    #trigger bsInnerLoopDTTRate()
fptype volatility[10000000]; #trigger bsInnerLoopDTTVolatility()
fptype otime[10000000];   #trigger bsInnerLoopDTTotime()

#DTT bsThreadInnerLoop
int 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);
    prices[i] = price;
    return 0;
}

#block bsThreadInnerLoop
for (i=0; i<numOptions; i++) {
    price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
                                rate[i], volatility[i], otime[i],
                                otype[i], 0);
    prices[i] = price;
}
#end_block
return 0;
}

```

(a) Array declarations

(b) The support thread function

(c) The original code with DTT pragmas

Figure 2. DTT extensions to blackscholes. This is a simplified version for clarity. We highlight DTT pragmas in bold. We show only one of the six nearly-identical support threads that all share the same skippable region.

2.1.1 Data-triggered threads programming model

The DTT model is accessed via extensions to the C programming language. These extensions include pragmas to declare data triggers, pragmas to identify support thread function, and pragmas to mark the skippable region.

Using these pragmas, the programmer attaches a support thread, defined as a function, to a variable or a field in a structure definition. The variable or field becomes a data trigger, and any change to that variable, or that field in any variable of that type, results in a thread being spawned to execute the support thread function. There is no restriction on which variables or data structures can be declared as data triggers. However, since the DTT model benefits primarily from parallelism and eliminating redundant computation, it is most effective when the programmer targets variables or data structure fields that help to exploit these two benefits.

The programmer also needs to compose the support thread function which describes the computation that depends on the data trigger. To simplify the design of the DTT execution model, the support thread function takes the address that triggers the support thread as the only argument. The support thread can communicate with the main thread using shared memory.

There are some restrictions when composing support thread functions. First, because threads may be executed an unknown number of times before the result is used, the support thread function should be *idempotent*, which means the result is not affected by how many times the code is executed. For example, a support thread that accumulates a count of some activity is not a good candidate. Second, the current DTT model does not allow a support thread function to trigger another support thread, although that will likely be added in future work.

After all support thread functions finish execution, the DTT execution model can skip computation rendered unnecessary by the support threads (the skippable region). When modifying existing

code, it is easy to identify this skippable region – it is the original computation that is being replaced by the support thread function. When writing new code, the programmer needs to write a conventional routine as “backup”. The skippable region is identified by pragmas which attach it to a particular support thread function.

In an example borrowed from the prior work [16], `mcf` spends significant time in the `refresh_potential` function. That function calculates the `potential` fields which depend on data in nodes throughout the structure. However, the fields that the computation depends on tend to be highly static, rendering most of the calculation redundant. Therefore, we attach support thread functions to the appropriate fields of the `node_t` data structure. When values in those fields are changed, `potential` is recalculated. When they do not change, the existing value in memory is still correct and no recalculation is necessary.

Figure 2 provides an example from `blackscholes`, with the code somewhat simplified for clarity. The inner loop of the `bs_thread` function (Figure 2(c)) depends on `sptprice`, `strike`, `rate`, `volatility`, `otime`, and `otype` arrays to update the `prices` array. In other words, a value in the `prices` array changes only if a value with the same index in arrays of `sptprice`, `strike`, `rate`, `volatility`, `otime`, and `otype` changes. To enable the DTT model on this application, we attach support thread functions to the declaration of the arrays as in Figure 2(a) and compose support threads for each data trigger. We present one of the support thread functions that we create in Figure 2(b). The others differ only in function name and the calculation of the index from the trigger address. We declare the inner loop in Figure 2(c) as the skippable region using the pragmas. Therefore, if the program changes an element in the `sptprice` array, for example, the DTT model executes the support thread function `bsInnerLoopDTTSptPrice` to compute and update only the corresponding element in the `prices` array, but still leaves all the other elements of the destination array untouched. The program can then skip the execution of the inner loop when it gets to the `block` pragma.

2.1.2 Architectural support for data-triggered threads execution model

The prior work supports the DTT model using modifications in both hardware and the instruction set architecture. On the hardware side, the processor requires additional hardware tables – the thread status table (TST) and the thread queue (TQ). The TST contains information associated with each skippable region to indicate whether or not the processor can bypass the execution of the skippable region. The TQ stores information to manage active support threads.

To support the DTT model, the prior work proposed several new instructions: `tstore`, `tspawn`, `tcancel`, and `treturn`. The `tstore` instruction stores a value into a data trigger and checks if the stored value is the same as the current data in the target memory. If a `tstore` instruction detects a change to memory contents, the following `tspawn` instruction will transfer the support thread information to the TQ. The support thread will execute on a spare hardware context until the support thread terminates at a `treturn` instruction that updates the TST, or a `tcancel` instruction that simply invalidates the corresponding TST entry when the program takes a path that causes the result to not be able to be reused. Note that the implementation of the `tstore` instruction also assumes changes to the load/store pipe and cache to determine whether the value has been changed in memory.

The DTT model only initiates a support thread after a `tstore` instruction commits. As a result, support threads are always non-speculative threads. Therefore, the DTT model does not require any architectural support for thread speculation, like speculative multithreading [10] or program demultiplexing [2].

2.2 Other related work

Pure dataflow programming models [5, 13] that execute programs based on the generation of data provide native support for fine-grain parallelism. To extract the full power of this execution model, these languages require hardware support for dataflow [1, 7, 12, 14], which can be quite complex.

Cilk [8] and CEAL [9] incorporate the concept of dataflow programming models into imperative programming languages to take advantage of dataflow constructs without completely rewriting programs. They each propose extensions to the C/C++ programming language to trigger computation when the program generates new data. DTT shares the idea of triggering parallel threads upon the generation of data with Cilk, but Cilk does not exploit the potential of removing redundant computation with this model like DTT. CEAL targets incremental recomputation and can also avoid redundant computation; however CEAL does not incorporate parallelism or threading into its solution.

Software memoization [6, 11] is an optimization technique that stores the input and output values of frequent operations or functions to reduce redundant computation. Software memoization requires the program to allocate additional memory to hold input and output values. Software memoization typically requires significant change in algorithms, and requires the runtime system to store prior inputs and outputs. The DTT model requires no such storage, as it relies on detecting changes rather than detecting sameness or redundancy. Therefore, our software framework works for arbitrary sizes of code regions and unlimited data structure sizes, with no significant storage overhead.

3. Design and implementation of software data-triggered threads

In this work, we design a compiler-assisted, software-only framework that accepts programs written using the DTT extensions and executes the compiled programs in the DTT model. To enable the DTT model in software, we need to (1) replace hardware tables with software data structures, (2) detect if an operation changes memory contents, (3) schedule the thread, and (4) skip over the skippable region, if appropriate. This section introduces the basic design of our implementation to achieve these goals.

3.1 Software data structures

The hardware framework to support the DTT model [16] uses the Thread Status Table and Thread Queue to manage execution. In our runtime system, we introduce state variables and a software thread queue to replace these hardware structures.

For each skippable region, the compiler allocates a state variable that contains information that determines if our runtime system can skip the execution of that skippable region. In the basic implementation, a state variable contains a valid bit, a pending support thread counter, and a cancellation bit (set by the `cancel` pragma) for a skippable region. The valid bit indicates if the result of the previous DTT execution is still correct. The pending support thread counter records the number of running and queued support thread events that can potentially affect the result of the skippable region. The valid bit tends to change often, as it is always zero when there are active support threads; the cancellation bit is set very rarely, and stays set until we execute the skippable region. When the cancellation bit is high, the valid bit is always low. The program can only skip the execution of a skippable region when the valid bit is set and the pending support thread counter is 0. If the counter is positive, the main thread will wait for all support threads to complete. After that, if the valid bit is set it will skip the skippable region, but if it is reset it will execute the skippable region in place.

We can add performance-monitoring counters to allow more advanced management of the DTT runtime, as described in Section 5.4. Even then, we use no more than 40 bytes of memory for each state variable. In this paper, we only declare one or two code sections as skippable regions in each benchmark, so we only use at most two state variables in each application. However, this is not a limitation of the model; the programmer can express more.

When the program needs to generate a support thread, we enqueue the necessary information to a software data structure, the thread queue (TQ). We statically allocate the TQ at the beginning of execution. For each TQ entry, we record (1) the current support thread status which indicates whether the event is executing or waiting to execute, (2) the memory address that triggered the support thread event, (3) the support thread function to execute, and (4) a pointer to the state variable of the corresponding skippable region. The size of a TQ entry is also 40 bytes. We allocate 256 entries for the TQ in this paper.

3.2 Detecting changes to memory contents

The DTT model spawns support threads when a memory operation changes a value in memory. Therefore, the runtime system must be able to detect a change to memory content.

To detect whether or not a memory operation writes a new value to an address, the compiler attaches `tstore()` functions to all assignments that may store a new value to the data triggers. These functions replace the conventional store instructions in those cases.

The `tstore()` function takes the following arguments: the writing memory address, the triggering address, the new value that we are writing, the pointer to the support thread function, and a pointer to the state variable of the skippable region. The triggering address is the only argument of the support thread function. Our framework uses the base address of the writing object as the triggering address if the data trigger is attached to a field of a data structure (this simplifies the code for the programmer). Otherwise, the triggering address is identical to the writing address. Because the DTT model allows programmers to attach data triggers to any variable or field of a data structure with arbitrary type, we also need to pass the size of the modifying variable or data field to the `tstore()` function so that the value is stored properly, and so that changes are detected at the right granularity.

When the `tstore()` function detects a memory change and if the cancellation bit is not set, it enqueues the support thread event with the triggering address, the state variable of the corresponding skippable region, and the support thread function pointer into the TQ. The DTT runtime system also sets the state of the new support thread event to pending. If the TQ does not contain a free entry (all the queued events are still running or pending) for storing this information, the runtime system can either (1) execute a queued event on the current processor core to free up a TQ entry or (2) force the main thread to stall until a running thread releases a TQ entry. In our preliminary experiments, we found that the TQ rarely becomes full, so we choose the latter to simplify the design.

After the `tstore()` function successfully transfers the support thread information to the TQ, it will first increase the pending support thread counter of the corresponding state variable by 1. It will then clear the valid bit of the corresponding state variable and return to the main thread.

When there is any pending support thread event in the TQ, our framework will try to schedule the execution of the support thread function. The system will select the first event in the queue that does not have any running support thread event pointing to the same state variable; that is, we serialize the execution of support thread functions that may affect the same skippable region. This restriction is due to our current definition of DTT, where the compiler knows the triggering address but not necessarily the outputs of the support

thread. Thus, it cannot distinguish between those threads that need to be serialized (e.g., each thread updates the same structure) and those that can be executed in parallel if their triggering addresses differ (e.g., if each writes a separate element, such as if a change to `A[i]` creates a thread that updates `C[i]`). This will likely be addressed in future work. For now, this limits our parallelism to two, unless we have multiple skippable regions.

The initial baseline implementation spawns a new thread, if possible, when data is modified. However, our experiments (Section 5.2) find that this strategy significantly degrades the performance due to the overhead of spawning new threads. Therefore, we develop a fast thread spawn (Section 5.3) technique, which uses polling threads to minimize the thread spawning latency.

Once the runtime system starts executing a support thread event, it will first change the support thread event state to running. The support thread will then run as a conventional thread until it reaches a `dttreturn()` call. The `dttreturn()` function, inserted by the compiler at all function exit points, will atomically decrease the pending support thread counter by 1 and update the valid bit in the state variable of the corresponding skippable region to valid if (1) no other event currently in the TQ will change the computation result, and (2) the cancellation bit is low. It will then release the occupied TQ entry for future events. In addition to the `dttreturn()` function, we also provide a `dttcancel()` function that terminates the support thread when the support thread executes a path that may lead to unwanted results. The `dttcancel()` function will invalidate the valid bit, clear all the queued events associated with the state variable, reset the pending support thread counter to 0, and set the cancellation bit. Once we set the cancellation bit, all later events for the skippable region will be discarded until the main thread executes the skippable region code, which will clear the cancellation bit and re-enable the use of DTT on the skippable region. The cancellation bit enables an important implementation feature — by always initializing these bits high, we ensure that the skippable region is executed the first time, and prevent a flurry of support threads being spawned while data structures are being initialized.

In the DTT model, the skippable region provides an implicit barrier since the main threads stall at the beginning of a skippable region if any corresponding thread event is executing. In our framework, the compiler inserts a `dtbbarrier()` function right before each skippable region to perform the implicit barrier. The `dtbbarrier()` function examines the number of pending threads recorded in the corresponding state variable. If any outstanding support thread of this skippable region is running, the `dtbbarrier()` function will stall the main thread until the support thread finishes execution. If there are no pending support threads associated with the skippable region, the `dtbbarrier()` function will return with the value of the current valid bit of the state variable and allow the main threads to continue execution. In this version of DTT, all active main threads (in parallel code) must share the barrier — i.e., they must each contain the skippable region. In this way, there is no possibility of a race condition between reading and updating the active thread count.

Just prior to returning from the `dtbbarrier()`, the valid bit is read. If the valid bit is set, the main thread will jump to the end of the skippable region. Otherwise, the main thread will execute the skippable region code in place. The DTT runtime system will then set the valid bit of the skippable region to valid after the program executes the skippable region.

The DTT model allows support thread functions to modify global data. As in many parallel programming paradigms, the programmer needs to ensure that those data updates in support thread functions do not incur unwanted data races.

SPEC2000 benchmarks		
App.	Data triggers	Optimized function(s)
ammp	last	a.number()
art	fl.layer[].P	train.match()
bzip2	ss	sortIt()
crafty	search	Evaluate()
eon	a_MR and a_VHR	ggMaterialRecord() and mrViewingHitRecord()
equake	time	main()
gcc	basic_block_live_at_end	propagate_block()
gzip	max_chain_length	longest_match()
mcf	node_t	refresh.potential()
mesa	img->Format	sample_ld_linear()
parser	randtable and inputs of count()	hash()
perlbnk	inputs of gv_stashpv()	gv_stashpv()
twolf	new_total of dimptr	new_dbox.a()
vortex	EmpTkn010 and PersonTkn	PersonObjs.FindIn()
vpr	rr_node	check_node()
PARSEC benchmarks		
App.	Data triggers	Optimized function(s)
blackscholes	data	bs_thread()
bodytrack	pf	Estimate()
canneal	a and b in annealer.thread::Run()	calculate_routing_cost()
facesim	threading_auxiliary_structures ->extended_tetrahedrons->m	Update_Position_Based_State_Parallel()
fluidanimate	cell.a	ProcessCollisions()
swaptions	swaption	HJM_Swaption_Blocking()
vips	t[12]	im_lintra_vec()
x264	h->fenc->i_type	x264_encoder.encode()

Table 1. Modifications to selected benchmarks

The only memory consistency issue we need to be careful about is to ensure that the store that happens in `tstore()` precedes the support thread, and that stores in the support thread precede the skippable region. In our software implementation, a memory consistency violation is either impossible (any strong consistency system) or highly unlikely (weak consistency). In the latter case, a few memory fences suffice.

4. Methodology

To study the performance of software DTTs, we selected three processor architectures and a variety of applications. We describe those processors and applications in this section. Like the prior work, we focus on modifications to existing, mature code rather than new programs, which enables better comparison between DTT and non-DTT code.

4.1 Processors

To investigate the performance of the software implementation of the DTT model, we select three widely available processors – Intel Xeon E5520 (Nehalem), AMD Opteron 2427 (Opteron), and Intel Xeon E5420 (Core 2 Quad) as the experimental platforms. The memory hierarchies of the processors vary significantly. The Core 2 Quad features a shared 6MB L2 cache on each die. In contrast, the Nehalem and Opteron have private L2 caches for each core but a shared L3 cache. The Opteron has an exclusive cache hierarchy while the others are inclusive. The memory latencies for these processors also differ significantly. Each of the processors features very different microarchitectures including the pipeline design, the branch predictor, etc., but all these processors can execute multiple threads on the same die. The Nehalem processor also supports simultaneous multithreading (SMT) [18] which can execute two threads within the same processor core to maximize the utilization of functional units. In this work, we also examine the performance of software data-triggered threads on the SMT configuration using the Nehalem processor.

4.2 Applications

We use gcc-4.1.2 to compile all 15 applications in SPEC 2000 that are written in C or C++ into x86-64 binaries. We use the older SPEC 2000 benchmarks to be able to compare with the prior work. For each benchmark, we use Pin [15] to profile the memory instructions that frequently incur redundant loads, as in prior work [16]. Unlike the prior work, we profile the whole program to determine the potential of applying the DTT model. The profiling results help identify the data structures incurring most of the redundant loads. We select very few data structures as data triggers, and copy the code that depends on the data triggers to compose support thread functions. Table 1 lists our modifications to the benchmarks. These modifications in some cases are identical or similar to those made in the prior work [16], but in several cases (equake, gcc, gzip, mesa, perlbnk, and vpr) we target other sections of code because we found that the code used in the prior work did not address whole-program execution as much as it impacted the short simulated regions in the prior experiments.

We also investigate the interaction between DTT and traditional parallelism for the first time in this research. We use the PARSEC 2.1 benchmarks [3]. We found that the level of redundant execution varied more widely in PARSEC than it does in SPEC. Therefore, we run the majority of the PARSEC benchmarks, but did not attempt to transform those where the profiled incidence of redundant loads was below 30%. In each case for the remaining benchmarks, we optimize the single function that contains the most redundancy in the benchmark, but retain the traditional parallelism for the rest of the program.

In most cases, the DTT parallelism was not redundant with the traditional parallelism, or targeted redundancy rather than parallelism. As a result, the effects of DTT on non-parallel PARSEC was also instructive, particularly in light of our better access to runtime statistics while executing the serial versions. Therefore, we also include the single-thread versions of the PARSEC benchmarks with our single-thread analysis and results.

Figure 3. The overhead of tracking changes to memory content and managing the internal data structures in our runtime system, with the support thread execution disabled.

For blackscholes, our support thread computes a single value of the `prices` array in `bs_thread()`, replacing code that recomputes the entire array. For bodytrack, the support thread calculates the `Estimate()` method only when a particle filter object gets updated. For canneal, we detect the change of element `a` and `b` to update the result of `calculate_routing_cost()`. For facesim, we only perform UPBS initialization when the number of elements changes. For fluidanimate, the support thread function computes `x`, `y`, and `z` values for an element in the `cell.a` array right after the program generates a new value for the element; this eliminates the need to execute `ProcessCollisions()` after all threads finish `ComputeForcesMT()`.

For swaptions, we recompute the `dSimSwaptionMeanPrice` and `dSimSwaptionStdError` field of an element that is in the `swaptions` array only when the program changes other fields in that element that may change those fields. This avoids the redundant execution of the `HJM_Swaption_Blocking()` function for the entire array. For vips, we trigger the linear transform computation as soon as the input images are ready. For x264, we examine the type of frame and only regenerate the headers once the type of the upcoming frame is different from the previous frame.

For the parallel experiments, we still use just an additional thread to execute support thread functions and disable traditional parallelism when the application is using DTT. Because we only target one code section in each application, and exploit redundancy heavily, we tend (in the current implementation) to get nearly all of our DTT gains with just one extra core. This mutually exclusive usage of the two models of parallelism (either DTT or traditional parallelism is active) is not a part of the DTT programming model – traditional parallel threads should be able to access DTT triggers. However, this simplification in the current implementation still allows us to achieve high speedups.

Since it is difficult to quantify programmer effort required to achieve performance results, we have restricted our changes, both for the serial and parallel applications, to relatively minor changes with few skippable regions and support functions. For most benchmarks, we only target at one support thread function and one skippable region. This allows us to demonstrate that in many cases very significant speedups are possible with low programmer effort.

5. Results

This section quantifies the software overheads of our data-triggered threads implementation, and shows performance results for the modified applications. We discuss results on three different hardware platforms, results for both a single-thread implementation and for a multithreaded implementation, results for three different hard-

ware parallelism scenarios – on an SMT core, across cores on a CMP, and across sockets, and results for both serial applications and parallel applications. We separate the serial application and parallel application results for two reasons. First, for comparison with the prior work on hardware DTTs, we run the same SPEC benchmarks in similar configurations. Second, our performance monitoring library allows us to collect statistics more accurately for the single-threaded case, so we present more comprehensive data for those results.

Software data-triggered threads incur overheads, both in runtime monitoring and especially in multithreading-related latencies, that the hardware-driven approach did not see in the simulated experiments. Even compared to conventional parallelism, the DTT model tends to create many short threads, and conventional architectures are not optimized to execute short threads well [4]. This section will quantify some of those costs and describe mechanisms that mitigate those overheads, but they do not go away completely. As a result, even more than the prior work, our code modifications that exploit DTTs (see Section 4) target redundant execution more than new opportunities for parallelism. This is because if code is redundant, not only do we avoid the redundant computation, we also skip the overheads of spawning and communication associated with that support thread.

To better exploit parallelism in the DTT model, we optimize the runtime system design to avoid thread spawning overhead and adapt to the overhead in the underlying systems. With the optimized runtime system, we also demonstrate that DTT can be highly complementary with traditional parallelism to further improve the performance of parallelized applications.

In our experiments, we find that the general performance trends are similar across all the platforms we used, so we only present the result of the Nehalem machine in figures to save space, but discuss the result of other platforms in text. We use single-thread unmodified benchmarks as the default baseline unless otherwise specified.

5.1 Runtime system overhead

Without hardware support, the runtime system needs to check if modifications change the memory contents of variables declared as data triggers. The runtime system also needs to transfer information to the TQ and manage the software structures after detecting a change. These are overheads that impact the main thread of execution. To examine these overheads, we designed a runtime system (just for this experiment) that executes the `tstore()` functions and queues the support thread events, but does not compute support thread functions.

Figure 4. The relative performance of the software DTTs implementation with both the main thread and the support thread functions running in a single thread on the Nehalem machine.

Figure 5. The relative performance of the base software DTTs implementation with the support threads running on separate cores on the Nehalem machine.

Figure 3 presents the overhead of our runtime system on the Nehalem machine. The average slowdowns for DTT overheads are 1.4%, 2.4% and 1.0% on the Nehalem, Opteron, and Core 2 Quad, respectively. For most benchmarks, the software overhead is less than 2% since we only replace the assignments associated with data triggers, which is typically a very small fraction (lower than 0.01%) of all store instructions. For parser, which is more heavily impacted, the program touches data triggers frequently; we observe that the dynamic instruction count increases by 9% in that program.

5.2 Base implementation

While we focus our code changes on regions that exhibit redundancy, we do exploit both redundancy and parallelism with our DTTs. Following the lead of the prior work, we can separate these effects by running both a single-thread version of our DTT runtime system and a multi-context version. The latter benefits from both effects, while the former only benefits from redundancy. In addition, for this research, this also isolates the thread spawning and communication overheads, since the single-thread version does not incur these. Therefore, for the initial experiments in this section, we create a DTT runtime system that runs in a single thread. In this configuration, when the `tstore()` function detects a memory modification, the program invokes the support thread function immediately in the same hardware context.

Figure 4 shows the data-triggered threads performance for a single-thread implementation compared with running unmodified (no DTTs) code on a single core of the Nehalem processor. The experiments show that even without the help of parallelism and

with no hardware support, our modification can still improve the performance for most serial benchmarks from SPEC2000, with averages of 6%, 7%, and 8% on the Nehalem, the Opteron, and the Core 2 Quad, respectively. For PARSEC benchmarks, our modification achieves significant performance gain, 14.8X and 128.5X for blackscholes and swaptions on the Nehalem machine. Both blackscholes and swaptions contain about 70% redundant loads, which is somewhat lower than the SPEC benchmarks that we investigated. However, the actual level of redundancy, especially for swaptions, is significantly higher. The vast majority of the non-redundant loads are to temporary local structures that the program re-initializes in function calls that depend on highly static data. Thus, they are not recorded by our tools as redundant, but in fact are unnecessary when the global data does not change. These regions are easily identified by the programmer due to the presence of the redundant loads of the global data.

Over all single-thread benchmarks, we achieve average performance improvements of 7.3X, 4.4X, and 7.2X on the Nehalem, the Opteron, and the Core 2 Quad, respectively.

For benchmarks eon, earthquake, mcf, blackscholes, canneal, fluidanimate, and swaptions, which obtain the most speedup in this configuration, the data-triggered threads model significantly reduces dynamic instruction counts. A phenomenon we see in a few applications is exhibited most strongly for mcf, in which the DTT model reduces the dynamic instruction count by 21%, yet achieves more than 100% speedup. This happens because the redundant computation that we eliminate in that application is also the code that incurs the most cache misses and dominates the execution time.

Figure 6. The relative performance of the software DTTs implementation with the fast thread spawning mechanism (DTT + fast spawn) using a Nehalem machine.

Figure 7. The relative execution time of DTT + fast spawn on the Nehalem machine.

However, for parser, because the reduction of redundant computation cannot compensate for the runtime system overhead, we still slow down the performance of this benchmark. In crafty, our modification still increases the dynamic instruction count by 6% and causes significant performance degradation. As expected, we see lower performance in general than the hardware approach [16]. Unfortunately, we cannot make more specific comparisons with the hardware results, due to differences in machine architectures and ISA, run length (whole program vs short interval), real machines vs simulations, and different usage of DTTs in some applications.

To benefit from parallelism, we extend our data-triggered threads framework to utilize multiple cores within a processor. In the baseline multi-threaded DTT framework, the `tstore()` function creates a new pthread when the system finds a support thread event is available for execution. The `dttreturn()` function terminates the thread.

Figure 5 shows the performance of our initial software implementation of multi-core DTT on the Nehalem machine. Despite achieving speedup on 9 of 15 SPEC2000 benchmarks and 3 out of 8 PARSEC benchmarks on Nehalem, this implementation significantly underperforms the single-thread case. For the SPEC benchmarks, our implementation degrades performance vs. the baseline an average of 9% on the Nehalem processor and 10% on Opteron and the Core 2 Quad. For PARSEC, the multi-core DTT on the Nehalem machine still shows 128.2x performance improvement for swaptions, and marginally better performance for facesim and vips. However, the others all drop off considerably, including blackscholes which achieved 14.8X speedup in single-thread mode. For benchmarks like twolf, crafty, parser, blackscholes, canneal, and

fluidanimate, which generate many short support threads, the overhead of multithreading is dominant.

These results imply that the thread spawning overheads, including operating system overhead, warming up the cache, and communicating data, have a large impact on the software DTT performance for several applications. In the following sections, we present optimizations to minimize the thread spawning overhead and allow software DTTs to take better advantage of the available parallelism.

5.3 Fast thread spawning

While the initial, naive implementation of software data-triggered threads achieves speedup on some applications, anywhere threads are actually being generated with high frequency we fail to amortize the high cost of generating and spawning these threads. To minimize the effect of these overheads, we introduce a fast user-level thread spawning mechanism in our runtime system.

In our fast spawning runtime system, we host a thread on each core that we plan to use for executing support threads. This thread, when idle, will monitor the thread queue. If the polling thread finds a queued event that has not executed, it will fetch the queued event and invoke the specified support thread function. When the support thread function finishes, the `dttreturn()` function updates the state variable of the corresponding skippable region and goes back to the polling function.

Figure 6 shows the performance of our runtime system with fast thread spawning on the Nehalem machine for all benchmarks we examined. Because we optimize only one skippable region for each benchmark and our current framework serializes the support

Figure 8. The relative performance of the software DTTs implementation with fast thread spawning and thresholding mechanism (DTT + fast spawn + threshold) on a Nehalem machine, for SPEC.

threads associated with the same skippable region, we need only create one polling thread on a separate processor core.

The fast spawn scheme significantly improves the software data-triggered threads implementation. For SPEC2000 benchmarks, the fast spawn scheme improves the performance over the baseline architecture (single thread without DTT) with an average improvement of 11.5%. The fast spawn scheme also improves the SPEC2000 performance over the baseline by 3.7% on the Opteron machine and 4.2% on the Core 2 Quad machine. For ammp, art, mcf, vortex, and vpr, the fast spawn mechanism allows those benchmarks to take advantage of parallelism and outperform the single-threaded runtime system. For PARSEC benchmarks, the fast spawn helps exploit the parallelism of the DTT model and achieves significant performance improvement on blackscholes, canneal, and fluidanimate over the original implementation.

We can see the importance of parallelism to each application in Figure 7. It presents the relative portion of time that each benchmark spends in running only the main thread (main thread only), running only the support thread (DTT only), and running both the main thread and the support thread in parallel (MT/DTT in parallel) — we only present the execution time breakdown on the Nehalem machine; the results for other machines are similar. In fact, we not only see the importance of parallelism (how often both threads are executing), but also the importance of lack of parallelism (how often the support thread is running alone — which implies the main thread is stalled, waiting for the support thread completion). In particular, for crafty and parser, where we still suffer slowdown in the fast spawn scheme, we find that these benchmarks spend a significant portion of time where only the support thread function is running. This problem is addressed in the next section.

5.4 Thresholding

The fast spawn scheme minimizes the overhead in spawning threads. However, we still see that some benchmarks experience significant performance degradation because the main thread stalls frequently. This is a result of two factors. First, the DTT computation is not highly redundant — if support threads do not execute, the main thread does not wait for them. Second, there is insufficient slack to hide the latency of the support thread. Both must be true for the main thread to stall. For example, it is still advantageous for the programmer to create support threads with no slack (i.e., no parallelism with the main code), as long as there is significant redundancy.

To avoid this case where we are frequently stalling, we introduce a simple thresholding scheme for our software runtime system. For each skippable region, we add two counters to keep track of how many times the program calls the barrier function (at the

skippable region) and how many times the main thread stalls. This accounts for both factors above — as long as the main thread does not have to wait, we don’t distinguish whether it was because the support thread(s) did not run or because they ran and completed in time. When the runtime system invokes the barrier function a certain number of times, it calculates the percentage of support thread executions that caused stalls over that period. If the percentage is lower than a threshold, the runtime system will reset the counters. Otherwise, the later invocations of the `tstore()` function will only invalidate the state variable without enqueueing any support thread events to the TQ. Thus, the main thread will execute the code in the skippable region instead of using the support threads to perform this computation. After a certain period, the runtime system will retry the DTT model.

Figure 8 shows the performance of software DTTs with the thresholding mechanism on the Nehalem machine. For these experiments, we calculate the percentage of stalls every 1000 calls to the barrier function of each skippable region. If the threshold mechanism turns off the usage of the DTT model, the system will retry using DTT every 10000 calls to the corresponding barrier function. We examined thresholds ranging from 10% to 90% at increments of 5%. The threshold percentage that achieves the best overall performance gain is different for each of our hardware platforms. This is to be expected, as this value will be a factor of the relative cost of communication (e.g., thread cold start effects) to computation, which will be impacted by memory latencies, core architectures, inclusive vs. exclusive caches, etc. However, we always find a threshold value in each architecture which outperforms all the prior software DTT implementations we have investigated. The runtime system performs best with 50%, 10%, and 15% threshold values for the Nehalem, the Opteron, and the Core 2 Quad processor, respectively.

For SPEC2000 benchmarks, the runtime system improves the average performance by 14.8% on the Nehalem machine. On the Opteron processor, we achieve 7.3% performance improvement. On the Core 2 Quad processor, we improve performance by 9.3%. Over all single-thread benchmarks, DTT achieves 7.3X speedup on Nehalem.

For the threshold values that achieve the best performance on each platform, the scheme successfully eliminates the performance loss of parser and mitigates the performance loss in crafty. However, it also degrades the performance in ammp by 5% and 2% on Opteron and Core 2 Quad machines. That is the only case where our threshold appears to be too aggressive and negates some opportunity.

For PARSEC benchmarks, the runtime system with thresholding mechanism and fast spawn achieves average performance gains

Figure 9. The relative performance of the software DTTs implementation with support threads running on a separate core (different cores), or the same core (SMT) or a different core from another physical processor (different sockets).

of 18.9X on the Nehalem machine, 10.9X on the Opteron machine, and 18.9X on the Core 2 Quad machine. We will discuss the equivalent (serial main thread plus DTT) PARSEC results in Section 5.6.

Since the thresholding mechanism (with fast spawn) works the best among all our implementations, we use it as the default software DTT implementation in the rest of the paper.

5.5 Adapting to different types of hardware parallelism

For the above multithreaded implementations, we execute support thread functions on a separate core within the same chip. However, our experimental platforms allow us to explore two more options to schedule the execution of the data-triggered support threads. One is to schedule the support thread functions in a different hardware context within the same core using simultaneous multithreading, and the other is to schedule support thread functions to another core located in a different socket. The former minimizes communication and cold-start effects, because caches are shared between SMT contexts, but maximizes potential interference between the main thread and the DTT (both in the caches and the execution resources). The latter has the opposite tradeoff.

Figure 9 shows the result of executing the support threads on the same core using SMT. Running a support thread within the same core can achieve 12.7% performance improvement for SPEC2000 and 18.7X performance improvement for PARSEC over the baseline. Thus, the software technique is still effective for multithreading contexts, but just a bit lower than the performance on separate cores. In this case, the improvements in communication do not fully compensate for the increased interference. Notice that the interference affects even applications that rarely spawn support threads, because the fast spawn optimization utilizes polling.

We also investigate the impact of running support threads on different sockets in Figure 9. This configuration eliminates nearly all resource sharing between the main thread and support thread, but it also increases the communication latencies between them. Compared with running the support threads on a different core within the same chip, the performance impact of increasing communication latencies is very insignificant in most benchmarks. For SPEC2000 benchmarks, the increased communication latency does hurt the performance of gcc, crafty, and parser on some architectures. The average performance loss of running support threads on different sockets is within 0.3% of our baseline DTT running on different cores on the Core 2 Quad machine. For Nehalem and Opteron, scheduling support threads to a separate chip is affected more, but still within 3%. For PARSEC benchmarks, running support threads on different sockets significantly hurts the performance of blackscholes, fluidanimate and swaptions on all the platforms,

but the speedups are still high. Compared with our baseline DTT, scheduling support threads on a separate chip reduces the average performance by 3.5%, 3.5%, and 1.8% on the Nehalem machine, the Opteron machine, and the Core 2 Quad machine.

In general, however, we see that our implementation makes the DTT programming model very tolerant of varying communication latency. This is in large part due to the thresholding optimization. Thresholding successfully disables DTT in the cases where the increased latency is enough to push the DTTs past the breakeven point and start hurting performance. We confirm this in separate experiments we ran on the Nehalem machine. Without thresholding, the cost of moving from same-socket to cross-socket parallelism is 8% for SPEC (compared to 3% with thresholding).

5.6 Data-triggered threads and multithreaded applications

Having DTT versions of parallel programs for the first time allows us to investigate a couple of interesting questions. First, is DTT parallelism redundant with traditional parallelism (exploiting the same phenomena) or is it complementary? Second, how do DTT/parallel programs scale with thread count?

Figure 10 shows the performance gain of the modified PARSEC benchmarks with and without DTT using different number of threads. Because the performance trends are similar across all our experimental platforms, we only show the result on the Nehalem machine.

The results for the parallel benchmarks fall into a few distinct categories. For blackscholes and swaptions, DTT speedups are still dramatic. We see in this case that DTT subsumes (and surpasses) traditional parallelism, instead of the other way around. Performance, however, does not scale with the number of threads, because what little code still gets executed has not been parallelized. In these cases, DTT achieves speedup of 8X and 64X, respectively, over the 4-core parallel version.

For fluidanimate, we are able to target enough redundant computation (from the `ProcessCollisions()` function) to get 2.4X speedup from DTT. However, we still see excellent scaling from fluidanimate, even better than traditional parallelism, because the code that remains (after the elimination of the redundant code) is effectively addressed by the traditional parallelism. Thus, even in this implementation of DTT which does not scale beyond two threads, DTT can still improve parallel scaling if it removes serial code or parallel bottlenecks, leaving the remaining code more highly parallel than the full original code.

For bodytrack, the application has 53% redundant loads, so DTT exploits some redundant computation, which improves performance by 23% without the help of parallelism. When using mul-

Figure 10. The performance of PARSEC applications with DTT (DTT) and traditional parallelism (pthread only). The DTT version uses a combination of DTT and the original parallelism.

multiple threads, that redundant code is not on the critical path, and the DTT version simply tracks the parallel version with two or more threads.

In canneal, we see the one case where DTT parallelism interferes with traditional parallelism. Our DTT implementation targets code with modest redundancy, which gives us 27% performance gain. However, this is the same region that is targeted by the traditional parallelism, and our current implementation does not allow us to exploit both in the same region. Thus, we lose the gains of the original parallelism. We expect this limitation to go away in future implementations.

For the remaining benchmarks, facesim, vips and x264, we target code that traditional techniques could not or did not parallelize. Although the gains were typically small, they were complementary with traditional parallelism, such that we always gain more (if only slightly) from the combination of DTT and parallelism than from parallelism alone.

We see from these results that even on parallel applications, data-triggered threads enable us to express a new type of parallelism that is often complementary to traditional parallelism, and in other cases we can use it to express traditional parallelism more efficiently.

Although DTT alone does not currently scale beyond two threads, most of these applications scale well with the combination of DTT and traditional parallelism. When DTT targets and eliminates serial code, it can improve the scaling of traditional parallelism overall.

6. Conclusions

This paper presents a pure software approach to support the data-triggered thread programming and execution model. Our work improves the generality and portability of the data-triggered thread model. Having a complete software solution allows the use of the DTT programming model on any existing parallel machine. The presented solutions to mitigate thread spawning costs and to eliminate the performance lost to runaway, serial DTTs frees the programmer to use the DTT constructs without worrying about potential performance loss. Our system allows a set of serial programs applications from SPEC 2000 to be sped up by 15%, with minor code modification and no hardware support. The complete set of serial applications (including single-thread PARSEC) were sped up by 7.3X (arithmetic mean) or 1.6X (geometric mean). We also show

that DTT can be highly complementary with traditional parallelism and achieve significant performance gain, as high as 64X, even over the original parallel version.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work was funded in part by NSF grant CCF-1018356 and support from AMD.

References

- [1] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39:300–318, March 1990.
- [2] S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *33rd Annual International Symposium on Computer Architecture*, volume 0, pages 302–313, June 2006.
- [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] J. Brown, L. Porter, and D. Tullsen. Fast thread migration via cache working set prediction. In *Proceedings of 17th International Symposium on High Performance Computer Architecture*, pages 193–204, February 2011.
- [5] D. C. Cann, J. T. Feo, A. D. W. Bohoem, and O. Oldehoeft. *SISAL Reference Manual: Language Version 2.0*, 1992.
- [6] D. Citron, D. Feitelson, and L. Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and division units. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–261, October 1998.
- [7] D. E. Culler, A. Sah, K. E. Schauer, 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] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.
- [9] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIG-*

PLAN conference on Programming language design and implementation, pages 25–37, New York, NY, USA, 2009. ACM.

- [10] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *Proceedings of the 12th international conference on Supercomputing*, pages 77–84, New York, NY, USA, 1998. ACM.
- [11] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.
- [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. *17th Annual International Symposium on Computer Architecture*, pages 82–91, May 1990.
- [15] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *34th International Symposium on Microarchitecture*, pages 81 – 92, Dec. 2004.
- [16] H.-W. Tseng and D. M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *Proceedings of 17th International Symposium on High Performance Computer Architecture*, pages 181–192, Feb. 2011.
- [17] H.-W. Tseng and D. M. Tullsen. Eliminating redundant computation and exposing parallelism through data-triggered threads. *IEEE Micro*, 32(3):38–47, May 2012.
- [18] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.